OST
Ostschweizer
Fachhochschule

VS Code Plugin: Node.js Package Metrics

Studienarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Herbstsemester 2020

Autoren:	Etienne Beyeler, Flavio Böni, Severin Hauser
Betreuer:	Silvan Gehrig
Projektpartner:	IFS, Institut für Software
Experte:	Mirko Stocker

Zusammenfassung

Node.js-Applikationen haben oft direkte und indirekte Abhängigkeiten zu hunderten von Paketen. Eingebunden werden diese meistens über die npm-Registry, welche über 1 Million Node.js-Paketen beherbergt. Die dort verfügbaren Paketen werden jedoch oft nur sporadisch gepflegt. Softwareingenieurinnen und -ingenieure stellt dies vor die Herausforderung, die Qualität verwendeter Paketen zu beurteilen.

Das Ziel der Arbeit besteht darin, aussagekräftige Metriken zur Beurteilung von Node.js-Paketen zu erarbeiten und diese zu visualisieren. Bei der Berechnung der Metriken eines Paketes sollen dessen Abhängigkeiten rekursiv miteinbezogen werden. Die Funktionalität soll als Erweiterung für Visual Studio Code implementiert werden.

Unter Einbezug einer Konkurrenzanalyse und Use Cases wurden Anforderungen erhoben und Metriken zur Bewertung von Node.js-Paketen evaluiert. Das Vorgehen beruhte auf einer Mischung von Scrum und Unified Process. Das Endprodukt besteht aus zwei Teil-Applikationen:

- Ein eigenständiges Node.js-Paket, welches die Kernfunktionalität zur Analyse eines Node.js-Projekts beinhaltet.
- Ein Erweiterung für Visual Studio Code, welche die Ergebnisse der obigen Kernfunktionalität in der Entwicklungsumgebung visualisiert.

Der Endanwendenden der Erweiterung erhalten so eine Übersicht aller verwendeten Pakete, sowie Informationen zu dessen Qualität, Schwachstellen und Lizenzen.

Management Summary

Ausgangslage

JavaScript und Node.js Als eine der populärsten Programmiersprachen und Kerntechnologie des World Wide Webs läuft JavaScript auf jedem modernen Webbrowser. Durch Verwendung der Laufzeitumgebung Node.js kann JavaScript unlängst auch ausserhalb des Browsers in Server-, Command-Line- und Desktop-Applikationen zum Einsatz kommen.

Pakete Node.js erlaubt die Wiederverwendung von Funktionalitäten anhand von Modulen, auch Pakete genannt. Dank Paketen kann der Entwicklungsaufwand für Applikationen stark reduziert werden. Statt bereits gelöste Probleme für jede Applikation neu zu implementieren, können diese durch die Installation eines darauf spezialisierten Paketes ersetzt werden.

npm Als De-Facto-Standard für die Suche und Installation von Node.js-Paketen hat sich npm etabliert. npm ist zum einen eine öffentliche Datenbank, in welcher Pakete registriert werden können, und zum anderen ein Paketmanager, mit dem diese installiert werden können. Die Anzahl der auf npm verfügbarer Pakete ist beachtenswert, Ende 2020 waren fast 1.5 Millionen Pakete registriert.

Aufwendige Bewertung von Paketen Für Entwickelnde ist die grosse Auswahl an Paketen sowohl Vor- und Nachteil zugleich. Während es die Entwicklung erleichtert, erschwert es die Beurteilung der Qualität der installierten Pakete. Da Pakete selbst auf anderen Paketen basieren können, kann bereits ein einfaches Programm Abhängigkeiten zu hunderten von Paketen haben. Deren manuelle Überprüfung auf Unterhalt, Aktualität, Lizenzen und Sicherheitsschwachstellen ist somit mit beträchtlichem Aufwand verbunden.

Ziele, Vorgehen und Technologien

Ziele Ziel dieser Arbeit ist das Erheben und Visualisieren von Metriken von installierten Paketen in einem Node.js-Projekt. Die Funktionalität soll in Form einer Erweiterung für die im Webumfeld populären Entwicklungsumgebung Visual Studio Code implementiert werden.

Vorgehen Die Studienarbeit wurde in zwei Teilen durchgeführt. Im ersten Teil wurden die Benutzerbedürfnisse erfasst und geeignete Attribute zur Bewertung der Qualität von Paketen bestimmt. Als Ergänzung wurde eine Konkurrenzanalyse durchgeführt, in welcher bestehende Lösungen untersucht wurden. Im zweiten Teil wurden die erfassten Anforderungen und Metriken in einer Erweiterung für Visual Studio Code umgesetzt.

Als Vorgehsmodell kam eine Mischung aus Scrum und Unified Process zum Einsatz.

Technologien Für die Entwicklung der Erweiterung ist die Programmiersprache TypeScript verwendet worden, ein Superset von JavaScript, welches Typensicherheit und den Einsatz erweiterter Konstrukte der objektorientierten Programmierung ermöglicht.

Ergebnisse

Nach über 670 Stunden Arbeit in 14 Wochen konnte eine Erweiterung für Visual Studio Code implementiert werden, welche die wichtigsten Ziele der Studienarbeit erfüllt.

Benutzende der Erweiterung können ein geöffnetes Projekt mit Node.js-Paketen analysieren lassen und die Ergebnisse erkunden. In der Analyse werden in einem ersten Schritt die installierten Pakete und deren Abhängigkeiten erfasst. In einem zweiten Schritt werden zu den gefundenen Paketen Metriken berechnet, bekannte Sicherheitsschwachstellen gesucht und anschliessend in der Erweiterung dargestellt.

Für die Darstellung der Ergebnisse wurde die Benutzeroberfläche von Visual Studio Code um mehrere Komponenten erweitert. Gefundenen Sicherheitsschwachstellen werden in der Problemliste hinzugefügt und bei der Deklaration im Texteditor hervorgehoben. In einer anderen Ansicht werden die gefundenen Pakete hierarchisch abgebildet, was den Benutzenden ermöglicht, diese zu durchsuchen. Bei Auswahl eines Pakets wird in einer anderen Ansicht ein Bericht dargestellt, welcher die erfassten Metriken, gefundene Sicherheitsschwachstellen und die im Paket verwendeten Lizenzen übersichtlich darstellt.

Aufgrund des begrenzten Zeitrahmens und der vielen verfügbaren Lösungen wurde auf die Erhebung von eigenen Metriken verzichtet; stattdessen wurden vorhandene Metriken verwendet. Diese eröffnen bereits einen guten Einblick in die Qualität der Pakete.

Ausblick

Das Resultat dieser Arbeit ist eine funktionierende Erweiterung für Visual Studio Code. Diese deckt bereits die wichtigsten Anforderungen der ursprünglichen Aufgabenstellung ab, kann jedoch noch um diverse Funktionalitäten erweitert werden.

Eine der aktuell grössten Limitierungen ist, dass die Analyse der Pakete lokal auf dem Rechner der Anwendenden ausgeführt wird. Grössere lokale Berechnungen werden daher nicht in angemessener Laufzeit durchführbar sein. Diese Limitierung könnte umgangen werden, indem die Berechnungen auf einem Server ausführt werden und Ergebnisse zu bereits analysierten Paketen zentral gespeichert und abrufbar sind. Die Anzahl der Netzwerkaufrufe und die Dauer der Analyse können durch eine dedizierte Schnittstelle reduziert werden und somit die lokale Applikation entlastet werden. Die Zentralisierung der Berechnung würde ebenfalls die Berechnung eigener Metriken und die Verwendung von zusätzlichen Quellen erlauben.

Durch die vorhandene Separation von Berechnungs- und Darstellungsogik ist die Adaption der Funktionalität für andere Anwendungen leicht machbar. Mögliche Szenarien wären etwa Erweiterungen für andere Entwicklungsumgebungen oder die Einbindung in automatisierte Test-Prozesse.

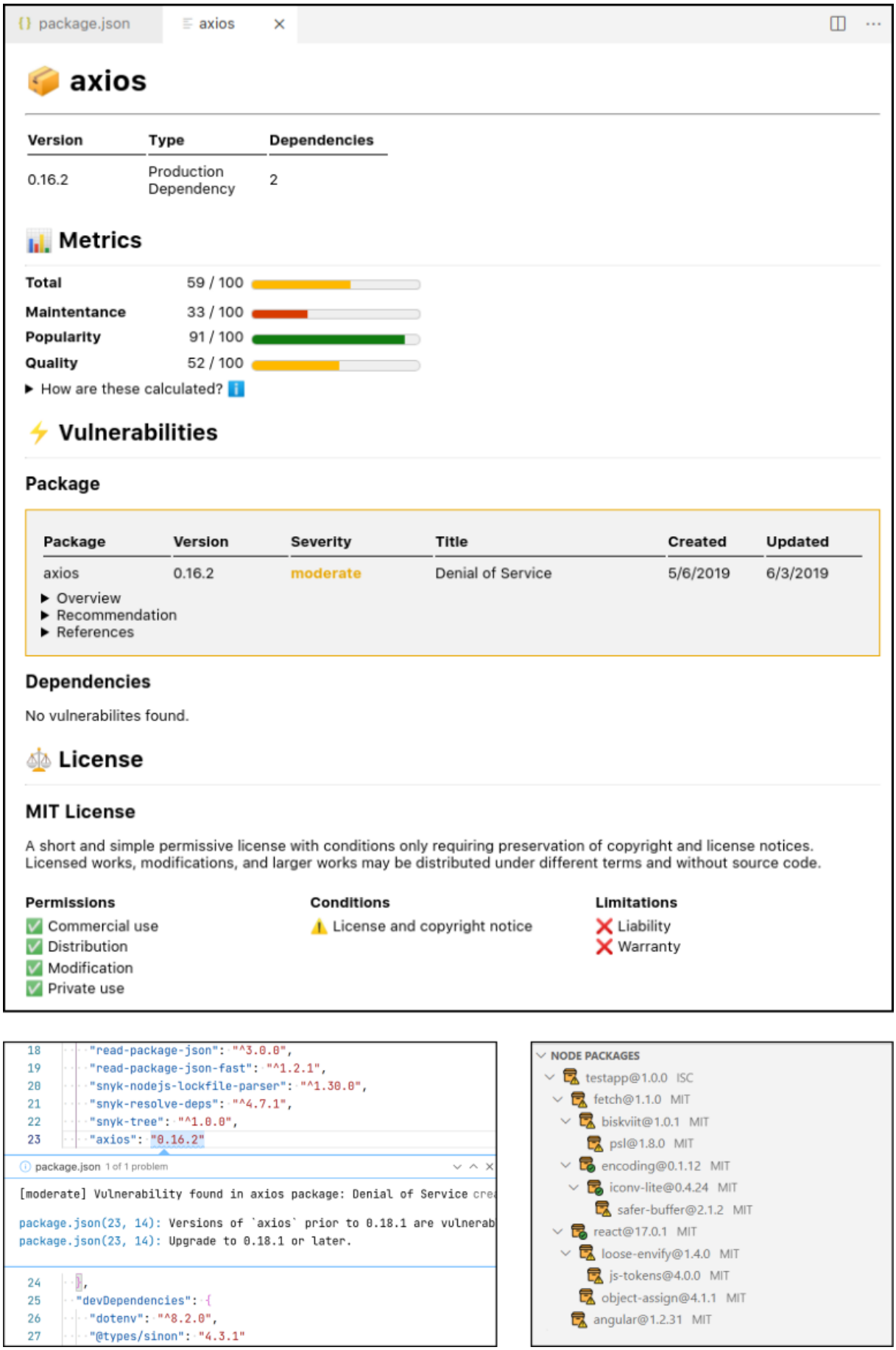

Abbildung 1: Screenshots der Visual-Studio-Code-Erweiterung

Inhaltsverzeichnis

Teil I

Technischer Bericht

Kapitel 1

Einführung

Als eine der populärsten Programmiersprachen und Kerntechnologie des World Wide Webs läuft JavaScript auf jedem modernen Webbrowser.[26] Durch die Verwendung der Laufzeitumgebung Node.js kann JavaScript unlängst auch ausserhalb des Browsers in Server-, Command-Line- und Desktop-Applikationen zum Einsatz kommen.[12]

Um bei der Entwicklung von JavaScript-Applikationen nicht ständig die selbe Funktionalität aufs Neue implementieren zu müssen, wird oft auf Softwarepakete und -module zurückgegriffen: Programme, die bestimmte Funktionen zur Wiederverwendung bereitstellen. Im Kontext von Node.js wird dabei folgendermassen zwischen Paketen und Modulen unterschieden: Ein Node.js-Modul ist eine beliebige im `node_modules`-Verzeichnis installierte Datei oder ein Verzeichnis, das von der Node.js-Funktion `require()` geladen werden kann. Ein Node.js-Paket ist eine Datei oder ein Verzeichnis, das durch eine `package.json`-Datei beschrieben wird.[8]

Node.js-Pakete (nachfolgend *Paket* genannt) können selbst wiederum auf Paketen basieren, wobei von einer Abhängigkeit gesprochen wird. Durch diese rekursive Struktur werden bei der Installation eines Pakets oft implizit diverse weitere Pakete mitinstalliert.

Um Pakete in Softwareprojekten einfacher verwalten zu können, bietet sich die Nutzung eines Paketmanagers an. Einer davon ist der Node Package Manager (npm), welcher standardmässig mit Node.js mitinstalliert wird. Genauer besteht npm aus drei verschiedene Komponenten: der Website, dem Command-Line Interface (CLI) und der Registry.[21] Die Website dient primär der Suche nach Paketen und dem Verwalten von Benutzerprofilen. Über die CLI, welche in einem Terminal ausgeführt wird, können Entwickelnde diese Pakete installieren oder eigene Pakete veröffentlichen.[21]

Bei der npm-Registry – der grössten Software-Registry der Welt – handelt es sich um eine öffentliche Sammlung von Paketen mit Open-Source-Code.[21] Pakete können zwar aus verschiedenen Quellen installiert werden, dennoch dient die npm-Registry als Hauptquelle für öffentlich verfügbarer Pakete.[8] Die npm-Website macht zwar keine genauen Angaben darüber, wie viele Pakete in der Registry vorhanden sind, gemäss dem *Third Party Tracker* Module Counts umfasst sie Ende 2020 jedoch fast 1.5 Millionen Pakete, wobei pro Tag 910 neue Pakete hinzukommen.[6] Durch die Vielfalt der Pakete in der Registry, welche unzählige Bedürfnisse der JavaScript-Community abdecken, hat sich npm zum De-Facto-Standard-Paketmanager für JavaScript etabliert.[21]

Für Entwickelnde ist die grosse Anzahl verfügbarer Pakete Vor- und Nachteil zugleich. Zum einen erleichtert es die Entwicklung, da eine Vielzahl an Teilproblemen bereits durch existierende Pakete gelöst wird. Anstatt die Funktionalität selbst implementieren und unterhalten zu müssen, kann einfach ein entsprechendes Paket eingebunden werden. Zum anderen wird so die Kontrolle über die Softwarequalität abgegeben: Entwickelnde sind darauf angewiesen, dass bei der Entwicklung des Paketes Sorge dafür getragen wird. Um eine Qualitätseinschätzung vorzunehmen, müsste jedes Paket und seine Abhängigkeiten einzeln beurteilt werden. Da in einem Softwareprojekt aber typischerweise zig Pakete zum Einsatz kommen, ist eine Überprüfung jedes einzelnen praktisch unmöglich.

Um Softwarequalität quantitativ messbar zu machen, können Softwarequalitätsmetriken eingesetzt werden. Eine Softwarequalitätsmetrik (nachfolgend *Metrik* genannt) ist eine Funktion, die eine Eigenschaft von Software in einen Zahlenwert abbildet. Dieser kann als Grad, in welchem die Software ein bestimmtes Qualitätsattribut besitzt, interpretiert werden.[16]

1.1 Aufgabenstellung

Der folgende Abschnitt basiert auf der zu Beginn der Studienarbeit definierten Aufgabenstellung, welche auch weitere Details zu Lieferumfang, Annahmen und Einschränkungen enthält.[14]

Das Ziel der Studienarbeit ist es, eine Erweiterung für Visual Studio Code zu erstellen, welche Metriken von Node.js-Paketen erhebt und visualisiert. Aus der Aufgabenstellung gehen drei Hauptaufgaben, welche nachfolgend ausgeführt werden.

1.1.1 Anforderungsanalyse

Anhand einer Konkurrenzanalyse sollen Anforderungen erarbeitet und definiert werden. Weiter soll die Benutzbarkeit durch Usability Tests sichergestellt werden.

1.1.2 Ausarbeiten von Metriken

Es sollen diverse Metriken evaluiert werden, welche aufgrund der Informationen einer `package.json`-Datei eruiert werden können. In diese Metriken sollen Abhängigkeiten und deren Folgeabhängigkeiten sowie bekannte Softwareschwachstellen einbezogen werden. Die Resultate sollen in einer von der Darstellung unabhängigen Form über eine Application Programming Interface (API) bereitgestellt werden. Historien über mehrere Versionen desselben Projektes sollen berechnet werden.

1.1.3 Integration in Visual Studio Code

Es soll eine Erweiterung für Visual Studio Code implementiert werden, welche die Metriken beim Öffnen einer `package.json`-Datei darstellt. Änderungen an der `package.json`-Datei sollen automatisch zu einer Anpassung der Metriken führen. Die Erweiterung soll so implementiert werden, dass eine Adaption für andere Entwicklungsumgebungen einfach umsetzbar ist.

1.1.4 Weitere Anforderungen

- Verwendung von TypeScript als Programmiersprache.
- Vorgehen nach Scrum und Unified Process (UP), Arbeit nach Projektplan.
- Ein lauffähige Erweiterung hat Priorität vor einer umfangreichen, aber nicht verwendbaren Erweiterung.
- Die Business- und Präsentationslogik sollen genügend voneinander abgekoppelt sein, um diese einfach in Erweiterungen für andere Entwicklungsumgebungen wiederverwenden zu können.
- Die Dokumentation des Projekts und der Software soll so umgesetzt werden, dass zukünftig Entwickelnde einfach darauf aufzubauen können.

1.2 Abgrenzungen

Bei der Durchführung einer Studienarbeit sind der Zeitrahmen und die verfügbare Ressourcen beschränkt. Um die Ziele unter diesen Voraussetzungen zu erreichen wurden folgende Abgrenzungen gesetzt:

- Die Funktionalität der Erweiterung wird rein lokal ausgeführt. Die Implementation eines unterstützenden Webservices liegt somit nicht im Rahmen dieser Studienarbeit.
- Die Funktionalität der Erweiterung soll nur auf frei zugänglichen Quellen basieren. Es soll also möglich sein, die Erweiterung ohne Anmeldung zu benutzen.
- Es werden nur Pakete berücksichtigt, welche in der npm-Registry registriert sind.

1.3 Vorgehen

Die Studienarbeit lässt sich in zwei Teile aufgliedern: den akademischen Teil, welcher sich mit der Auswahl der Metriken befasst, und den praktischen Teil, in welchem diese dann umgesetzt werden.

1.3.1 Akademischer Teil

Um später die Node.js-Pakete sinnvoll bewerten zu können, sollen geeignete Attribute bestimmt werden, mit welchen sich die Qualität von Node.js-Paketen bestimmen lässt. Dazu bietet sich eine Konkurrenzanalyse an: das Analysieren und Vergleichen von existierende Lösungen, die sich ebenfalls mit der Qualität von Softwarepaketen befassen.

Vorgängig ist es aber notwendig, die Bedürfnisse der Benutzenden richtig zu erkennen. Dies ist ebenfalls Bestandteil des akademischen Teiles.

Sollten keine passenden Metriken gefunden werden, werden eigene Metriken erstellt.

1.3.2 Praktischer Teil

Um die Metriken aus dem akademischen Teil angemessen umsetzen zu können, müssen die Kapazitäten der Visual Studio Code Extension API bekannt sein. Dazu ist es nötig, sich in die Dokumentation einzulesen und einen Prototypen zu erstellen. Der Prototyp kann parallel zum akademischen Teil entwickelt werden.

Sobald die Resultate aus dem ersten Teil und die Anforderungsspezifikationen vorhanden sind, kann mit der eigentlichen Umsetzung der Erweiterung für Visual Studio Code begonnen werden.

Kapitel 2

Evaluation von Metriken

2.1 Konkurrenzanalyse

Im Rahmen einer Konkurrenzanalyse werden bestehende Softwarepakete recherchiert und analysiert. Ziel der Analyse ist es, vorhandene Lösungsansätze festzuhalten und existierende oder fehlende Funktionalitäten zu erfassen. Die Resultate der Analyse fliessen direkt in die Spezifikation der Anforderungen mit ein und ermöglichen die Auswahl von möglichen Metriken.

Nachfolgend sind die wichtigsten Erkenntnisse der Analyse zusammengefasst; die ausführliche Konkurrenzanalyse ist im Anhang A.2 zu finden.

2.1.1 Erweiterungen für Visual Studio Code

Die zwei Erweiterungen Dependency Analytics[1] und Sonatype Nexus IQ[2] bieten vergleichbare Funktionen an. Sie analysieren installierte Pakete und präsentieren dem Benutzenden die Resultate in Form von Reporten. Der Fokus beider Erweiterungen liegt rein auf der Suche von Sicherheitsschwachstellen. Als Quelle für bekannte Schwachstellen werden dabei die Snyk Intel Vulnerability DB[3] bzw. die OSS Index Datenbank[4] verwendet. Beide Quellen verlangen für den Zugriff einen Benutzerkonto und erfordern somit eine Anmeldung des Benutzenden.

2.1.2 Softwarepakete

Die Recherche nach Softwarepaketen ergab nur einen Treffer: npq[5]. Dabei handelt es sich um ein CLI-Tool, welches Abhängigkeiten zum Zeitpunkt der Installation überprüft. Dabei wird das Alter des Pakete, die Anzahl Downloads sowie das Vorhandensein einer Lizenz und Readme-Datei berücksichtigt. Ist auf dem System das snyk-Package installiert und konfiguriert, werden Abhängigkeiten zusätzlich auf bekannte Sicherheitsschwachstellen in der Snyk Vulnerability DB überprüft.

[1] https://marketplace.visualstudio.com/items?itemName=redhat.fabric8-analytics
[2] https://marketplace.visualstudio.com/items?itemName=SonatypeCommunity.vscode-iq-plugin
[3] https://snyk.io/product/vulnerability-database
[4] https://ossindex.sonatype.org
[5] https://github.com/lirantal/npq

2.1.3 Webservices

Package Quality[6] und npms[7] sind zwei Webservices, welche Pakete analysieren und bewerten. npms ist eine Open-Source-Suche für Node.js-Pakete, welche diese Pakete ebenfalls analysiert und deren Qualität, Wartung und Popularität beurteilt und eine kostenlose API anbietet.[2]

2.1.4 Dienstleistungen von npm

Abfragemöglichkeiten von Sicherheitsschwachstellen und Metriken werden ebenfalls von npm selbst angeboten. Für die Metriken wird dabei das bereits beschriebene npms verwendet. Im Vergleich zu npms sind die Metriken jedoch nur über den API-Endpunkt für die Paketsuche verfügbar und reduziert sich auf die numerischen Werte, während npms die für die Berechnung relevanten Metadaten in den Suchresultaten mitliefert. Die Dienste sind über die npm-API frei und ohne Anmeldung verfügbar.

2.2 Sicherheitsschwachstellen

Um Sicherheitsschwachstellen in die Bewertung von Paketen einzubeziehen, sind zusätzlich zur Konkurrenzanalyse auch Anbieter von Informationen zu Sicherheitsschwachstellen überprüft worden.

Dabei wurde nach Lieferanten gesucht, welche die Schwachstellen öffentlich zugänglich zur Verfügung stellen. Wir mussten bei der Nachforschung feststellen, dass es bei vielen dieser Datenbanken Lizenzeinschränkungen gibt. Zusätzlich ist jeweils auch eine eindeutige Identifikation mittels Benutzeraccount nötig, um so etwaige Abfragekontingente einzuhalten. Die Schwachstellen können bei den meisten Anbietern gut über einen Webbrowser abgerufen werden. Das reicht für unsere Abfragen über die Metriken-API aber nicht, da wir die Daten auswerten müssen. Nach intensiver Suche haben wir herausgefunden, dass npm ebenfalls eine API für die Abfrage von Sicherheitsschwachstellendaten (nachfolgend *npm-Audit* genannt) anbietet. Diese API ist zwar nicht offiziell dokumentiert, funktioniert für unsere Zwecke aber einwandfrei. Es ist uns dadurch möglich, Schwachstellen zu jedem Paket einzeln abzufragen, ohne Kontingente oder sich vorgängig Authentifizieren zu müssen.[10]

Zusammenfassend kann gesagt werden, dass trotz der grossen Anzahl gefundener Anbieter und dem Umfang deren Angebot stellten nur wenige eine kostenfreie API zur Verfügung. Von den kostenfreien Angeboten war wiederum nur die API von npm-Audit ohne Authentifizierung verwendbar. Die ausführliche Analyse der Sicherheitsschwachstellen ist ebenfalls im Anhang A.3 zu finden.

2.3 Erkenntnisse

Auf Grund unserer Analysen haben wir erkannt, dass die Qualität eines Pakets nur bedingt objektiv beurteilbar ist. Ob der Wert einer Metrik gut oder schlecht ist, hängt oft

[6] https://packagequality.com
[7] https://npms.io

vom Kontext des Anwendenden ab. So ist zum Beispiel eine restriktive Lizenz eines verwendeten Pakete bei einem Hobby-Projekt nahezu irrelevant, bei einem kommerziellen Produkt hingegen äusserst wichtig. Auch Werte wie die Popularität, gemessen an der Anzahl Installationen, sind je nach Anwendung verschieden relevant.

In der Erweiterung sollen Metriken also vor allem zur Entscheidungsfindung von Endanwendenden beitragen. Dabei erachten wir eine Unterteilung der Daten in die folgende Kategorien für sinnvoll:

Metriken

- Quantitativ bestimmbare Werte zu einem Paket.
- Durch Anwendung einer einheitlichen Formel kann einem Paket ein Wert auf einer Skala zugewiesen werden (hoch $\Longleftrightarrow$ tief).
- Ob der Wert gut, oder schlecht ist, hängt vom Anwendenden ab (subjektiv).
- Können durch den Anwendenden nicht beeinflusst werden.

Lizenzen

- Rechtliche Nutzungsbedingungen eines Pakete.
- Ob die gewählte Lizenz gut, oder schlecht ist hängt vom Anwendenden ab (subjektiv).
- Können durch den Anwendenden nicht beeinflusst werden.

Sicherheitsschwachstellen

- Bekannte Sicherheitslücken eines Pakete.
- Beziehen sich oft nur auf bestimmte Versionen eines Pakete.
- Ob die Gewichtung gut oder schlecht ist, hängt nur bedingt vom Anwendenden ab (eher objektiv).
- Können indirekt durch den Anwendenden behoben werden (Installation einer nicht betroffenen Version).

Kapitel 3

Ergebnisse

3.1 Umsetzung

3.1.1 Herausforderungen

Bei der Ausarbeitung der Anforderungen und der Erstellung eines Prototypen haben sich schon früh einige grundlegende Probleme abgezeichnet. Die beiden wichtigsten werden im folgenden Kapitel beschrieben.

Auflösung von Abhängigkeiten

Die ursprüngliche Vision war, dass die Qualität eines Paketes rein aufgrund der Deklaration in der `package.json`-Datei bewertet wird. Da die Abhängigkeiten eines Paketes jedoch rekursiv in die Analyse miteinbezogen werden sollen, müssen diese zuerst hierarchisch aufgelöst werden, was eine komplexe Angelegenheit ist.

Bei der Installation eines Paketes wird diese Auflösung durch den Paketmanager übernommen. Dieser gleicht die deklarierten Abhängigkeiten mit der Quelle – meistens die npm-Registry – ab und installiert die passende Version. Dieser Vorgang wird für die Abhängigkeiten jedes installierten Paketes wiederholt. Dabei werden mehrere Optimierungen durchgeführt, wie etwa das Auslassen nicht benötigter Pakete und die Reduktion der Anzahl installierter Versionen eines Paketes.

Bei einer Auflösung rein auf Basis einer `package.json`-Datei müsste dieser Ablauf also durch die Erweiterung nachgestellt werden. Dieser Ansatz ermöglicht zwar die Analyse eins Paketes ohne dessen vorherige Installation, die Implementation wäre aber mit grossem Aufwand verbunden.

Eine Alternative zum vorherigen Ansatz wäre die Ermittlung der Abhängigkeiten auf Basis der lokal installierten Pakete. Dies wäre ohne Nachbau der Auflösung machbar, setzt jedoch voraus, dass die zu analysierenden Pakete bereits vor der Analyse installiert sind.

Die Vor- und Nachteile der beiden Ansätze sind also wie folgt:

1. Nur `package.json` verwenden, Abhängigkeiten über npm auflösen.
    - Vorteile

- Keine Installation nötig zur Analyse eines Paketes.
- Feedback an Benutzende bereits bei Änderung des Inhalts der `package.json`-Datei möglich.

- Nachteile
  - Langsamer Aufbau der Struktur, da Abhängigkeiten rekursiv von der Paketquelle geladen werden müssen.
  - Die Modulauflösung von npm muss nachgebaut werden, um die Auflösung von Abhängigkeiten gleich zu erhalten. Dies ist speziell für Sicherheitsschwachstellen relevant, da sich diese auf spezifische Paketversionen beziehen.

2. Aufbau aufgrund lokal installierter Abhängigkeiten.
   - Vorteile
     - Tiefere Komplexität für Implementation.
     - Versionen entsprechen exakt den verwendeten.
   - Nachteile
     - Benötigt Installation bevor ein Paket analysiert werden kann.
     - Je nach Plattform werden gewisse Abhängigkeiten nicht installiert. Dies bedeutet, dass diese nicht in der Analyse erscheinen.

Ressourceneffizienz

Die Analyse eines Paketes erfordert das Sammeln und Auswerten von Daten. Dies ist mit Ressourcenverbrauch aufgrund von Netzwerkaufrufen und Rechenleistung verbunden. Gerade die Abfrage von Quellen über das Netzwerk hat bereits bei einer kleinen Anzahl analysierter Pakete eine grosse Anzahl Netzwerkaufrufe zur Folge. Um die Laufzeit der Analyse niedrig zu halten, sollten also Netzwerkaufrufe möglichst effizient und nur wenn nötig gemacht werden.

Mögliche Lösungen für dieses Problem sind wie folgt:

- Reduktion der Tiefe der Analyse.
- Verwendung bereits aggregierter Metriken von Services wie npms.

3.1.2 Metriken und Sicherheitsschwachstellen

Aggregation und Aufbereitung der Daten

Die Anforderungen wurden anhand der Nachforschung nach bestehenden Lösungen konkretisiert. In Betracht gezogen wurden Lösungen, welche Daten von Node.js-Pakete sammeln und in geeigneter Weise auswerten. Im Detail wurde insbesondere auf die Sammlung von Sicherheitsschwachstellen, Lizenzinformationen und Metriken zur Messung von Qualität, Popularität und Wartbarkeit und deren Auswertung geachtet. Ein Gesamtpunktzahl über diese Daten für jedes analysierte Paket soll zudem einen schnellen Gesamteindruck

ermöglichen. Es konnten einige sehr interessante Architekturen und Vorgehensweisen ermittelt werden, welche bei der Umsetzung miteinbezogen wurden.

Auf Grund der existierenden Lösungen und dem begrenzten Zeitumfang haben wir auf die Umsetzung von eigenen Metriken verzichtet und uns dafür entschieden, die bestehenden Metriken der npm-Registry, welche intern auf npms beruhen, zu verwenden. Diese bewerten die drei Attribute Qualität, Popularität und Wartbarkeit. Aus diesen Zahlenwerten, welche zwischen 0 und 1 liegen, wird dann eine zusammengesetzte Gesamtwertung berechnet, welche wiederum zwischen 0 und 1 liegt; je näher bei 1, desto besser. Wir erachten einen Wert unter 0.5 als ungenügend und Kennzeichnen dies als Warnung in der Gesamtwertung. Da eine objektive Einstufung aber schwierig ist, werden zur besseren Nachvollziehbarkeit stets alle Werte dargestellt.

Aufgrund der Erkenntnisse der Konkurrenzanalyse haben wir uns ebenfalls dazu entschieden, die npm-Registry als Lieferanten für Informationen von Sicherheitsschwachstellen zu verwenden.

Die Aggregation und Zusammensetzung der oben erwähnten Daten wurde in einem eigenständigen Paket umgesetzt, welche nachfolgend Metriken-API genannt wird. An die Metriken-API kann die Abhängigkeitshierarchie des zu analysierenden Projektes sowie eine zusätzliche Konfiguration, welche beispielsweise die Analysetiefe festlegt, übergeben werden. Basierend darauf werden dann die Metriken zu den Paketen berechnet ausgewertet zurückgegeben. Die Metriken-API ist damit unabhängig von der Präsentation nutz- und erweiterbar und eignet sich für verschiedene Einsatzmöglichkeiten (Integration in verschiedene Entwicklungsumgebungen, Einbindung in CI-Pipeline etc.)

3.1.3 Lizenzen

Eine Softwarelizenz enthält die Bedingungen, welche regeln, wie eine Software genutzt werden darf.[22] Dabei gibt es unzählige verschiedene Lizenzen, welche teilweise nicht miteinander kompatibel sind. Damit Anwendende die Lizenzen korrekt verwenden können, soll ihnen diese Informationen auf einfache Art und Weise veranschaulicht werden.

Datenbeschaffung

Für die Beschaffung der Daten kommen mehrere Optionen in Frage: Lizenzinformationen können entweder lokal abgespeichert werden, per API abgefragt werden, via Web-Crawler von einer Seite extrahiert werden oder schlicht auf eine externe Webseite referenzieren.

Die Suche nach einer passenden API hat leider keine zufriedenstellenden Ergebnisse hervorgebracht. Mit der Open Source API gibt es zwar eine benutzbare API, die Daten werden von dieser jedoch in stark normalisierter Form zurückgegeben und müssten so auf aufwendige Weise wieder denormalisiert werden.[23] Ein Web-Crawler wurde ebenfalls als zu aufwändig und wartungsintensiv eingeschätzt. Da sich die Lizenzinformationen jedoch gut und einfach lokal (im JSON-Format) abspeichern lassen, wurde auf die Weiterleitung auf eine externe Webseite verzichtet – auch um zur User-Experience zu erhöhen.

Da es, bedingt durch die Vielzahl der existierenden Lizenzen, praktisch unmöglich ist, Informationen zu allen Lizenzen lokal abzuspeichern, werden in einem ersten Schritt folgenden Lizenzen berücksichtigt: *Apache License 2.0*, *BSD 3-Clause License*, *BSD 2-Clause*

License, *GNU General Public License (GPL)*, *GNU Lesser General Public License (LGPL)*, *MIT License*, *Mozilla Public License 2.0*

Diese Lizenzen sind einerseits in der Community sehr populär und sind anderseits von der Open Source Initiative (OSI) annerkannt.[24] Zu jeder Lizenz wird deren Name, eine kurze Beschreibung und die *Permissions*, *Limitations*, *Conditions* angezeigt. Die Daten dazu stammen von der Seite `https://choosealicense.com`, welche von GitHub Inc. betrieben wird.

Die Lizenzinformationen sind ebenfalls Teil der oben beschriebenen Metriken-API.

3.1.4 Darstellung

Die Darstellung der Daten der Metriken-API erfolgt in einer Erweiterung für Visual Studio Code, welche in einem separaten Node.js-Paket implementiert ist. Visual Studio Code wurde mit dem Hintergedanken der Erweiterbarkeit gebaut und bietet eine gut Dokumentierte Schnittstelle für Erweiterungen an, die Extension API. Durch diese können Komponenten im Editor erweitert werden ermöglichen eine geschmeidige Integration in die hellen und dunklen Farbthemen. Das Sicherstellen einer guten Bedienbarkeit war für uns von Anfang an ein wichtiges Kriterium. Aus eigener Erfahrung wissen wir, wie störend es sein kann, wenn sich Erweiterungen nicht wie gewohnt verhalten und sich schlecht in ein bestehendes Produkt integrieren.

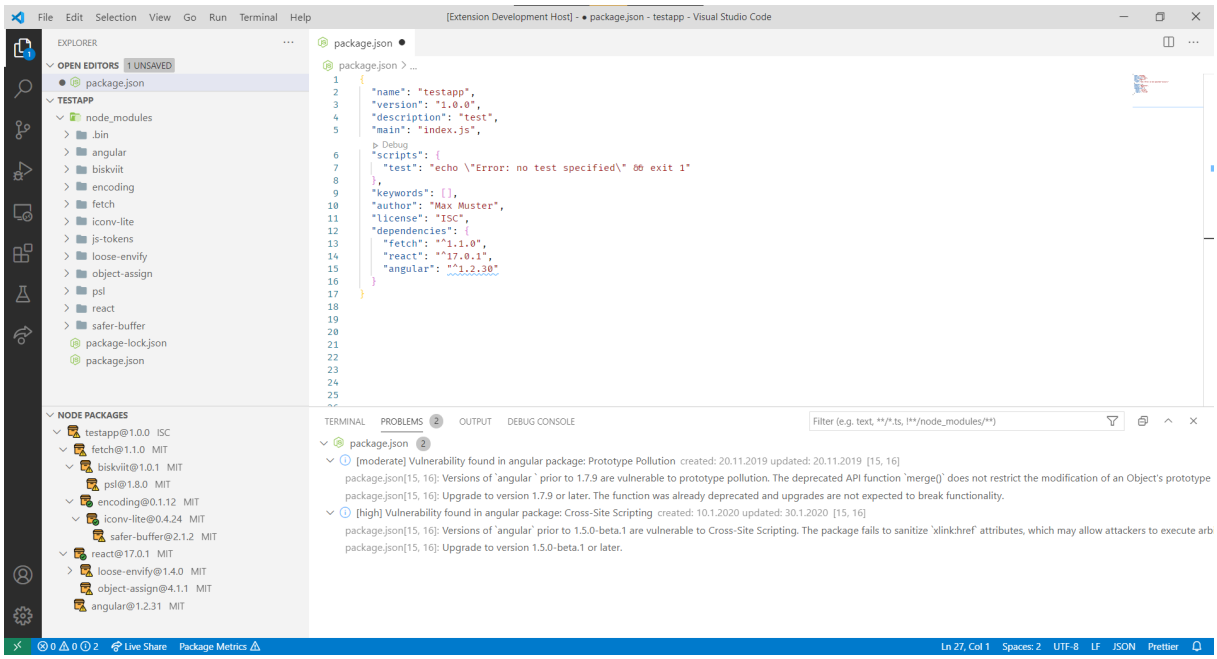

Abbildung 3.1: Die Erweiterung im hellen Visual Studio Code Theme

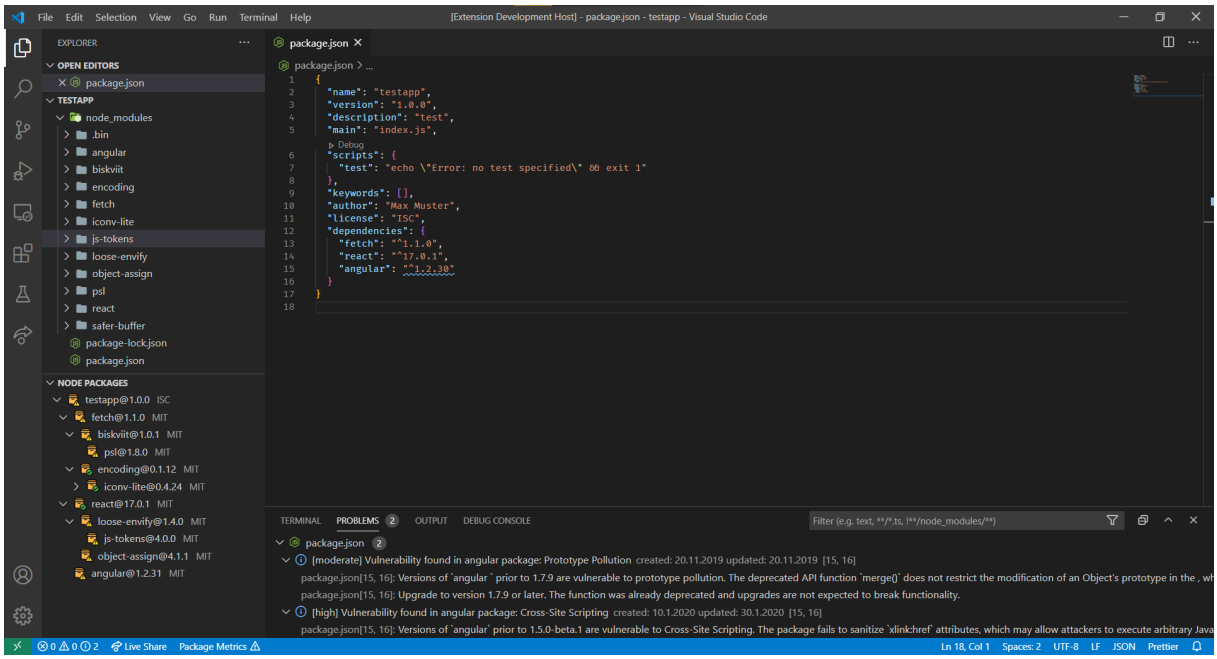

Abbildung 3.2: Die Erweiterung im dunklen Visual Studio Code Theme

Start der Erweiterung

Wenn eine `package.json`-Datei im Projekt vorhanden ist, wird die Erweiterung aktiviert. Es erscheint ein Hinweis, solange die keine `package.json`-Datei im Editor geöffnet wurde.

Sobald die `package.json`-Datei im Editor aktiv ist, wird ein Hinweis zur Vorbereitung und eine Schaltfläche für die Metrikberechnung angezeigt. Wie gesehen ist es zwingend notwendig, dass Node.js-Pakete vor der Berechnung installiert wurden. Die Metriken-API verwendet die interne Dateistruktur der Pakete, welche im Verzeichnis `node_modules` installiert sind.

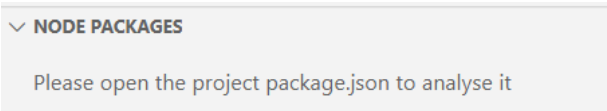

Abbildung 3.3: Hinweis bei Start der Erweiterung

3.1.5 Viusalisierung der Metriken

Wir haben uns bei der Darstellung der Paketabhängigkeiten für die Tree-View-Komponente entschieden. Eine Baumstruktur eignet sich gut, um hierarchische Abhängigkeiten aufzulisten. Detailliertere Informationen über ein Node.js-Paket können durch die Auswahl eines Eintrags in der Tree-View in einer Webview, welche sich in einem neuen Tab öffnet, übersichtlich dargestellt werden. Die Status-Bar-Item-Komponente wird verwendet, um den Gesamtzustand des Projektes anzuzeigen. Gefundene Sicherheitsschwachstellen von Paketen in der *package.json*-Datei werden direkt bei der Deklaration im Editor unterstrichen gekennzeichnet und im Problem-Panel mit einer kurzer Beschreibung aufgezeigt.

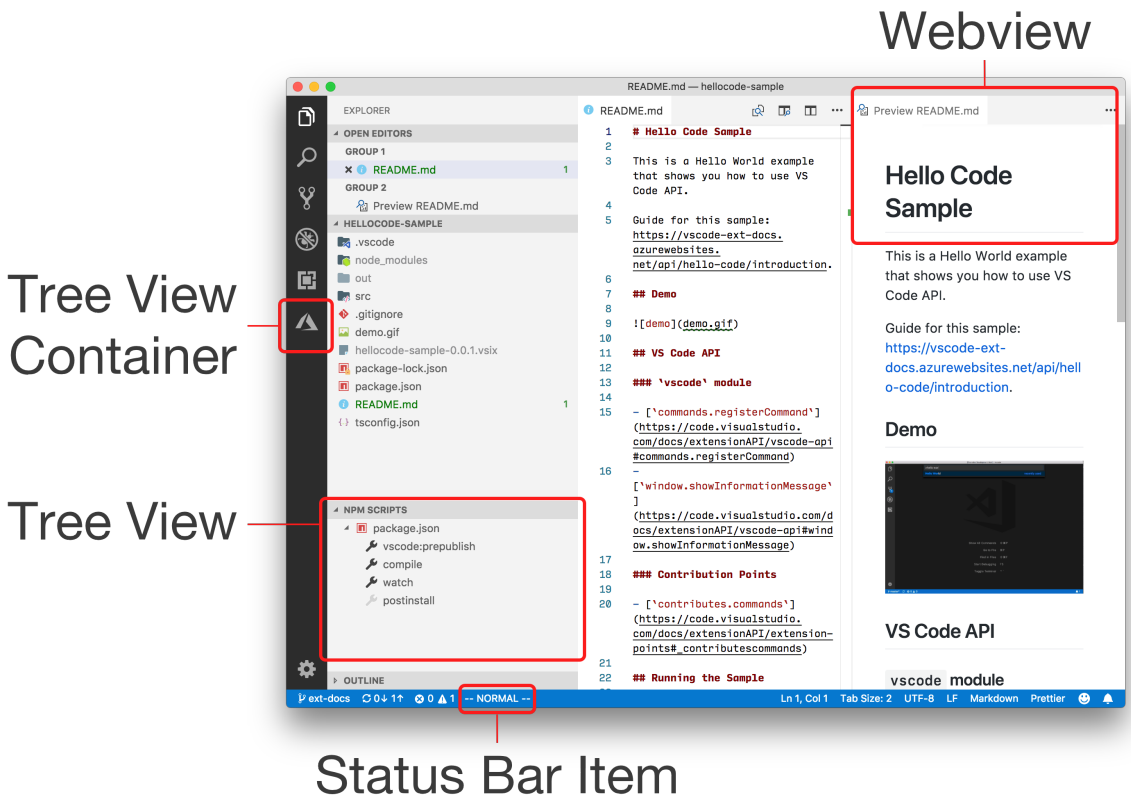

Abbildung 3.4: Extension API Komponenten[18]

Tree-View Das Projekt, in welchem die Erweiterung ausgeführt wird, erscheint nach erfolgreicher Berechnung selbst als Root-Paket in der Tree-View. Über diesen Einstiegspunkt lassen sich nun alle Abhängigkeiten aufklappen. Die Baumstruktur widerspiegelt die installierten Pakete im Projekt. Jeder Eintrag im der Tree-View besteht aus Paketname und -version. Falls vorhanden wird zusätzlich die verwendete Lizenz angezeigt. Damit sollen mögliche Lizenzinkompatibilitäten direkt ersichtlich sein. Ein Icon zeigt direkt den Zustand des Paketes an. Der Tooltip eines Eintrages bringt zusätzlich den berechneten Wert zum Vorschein.

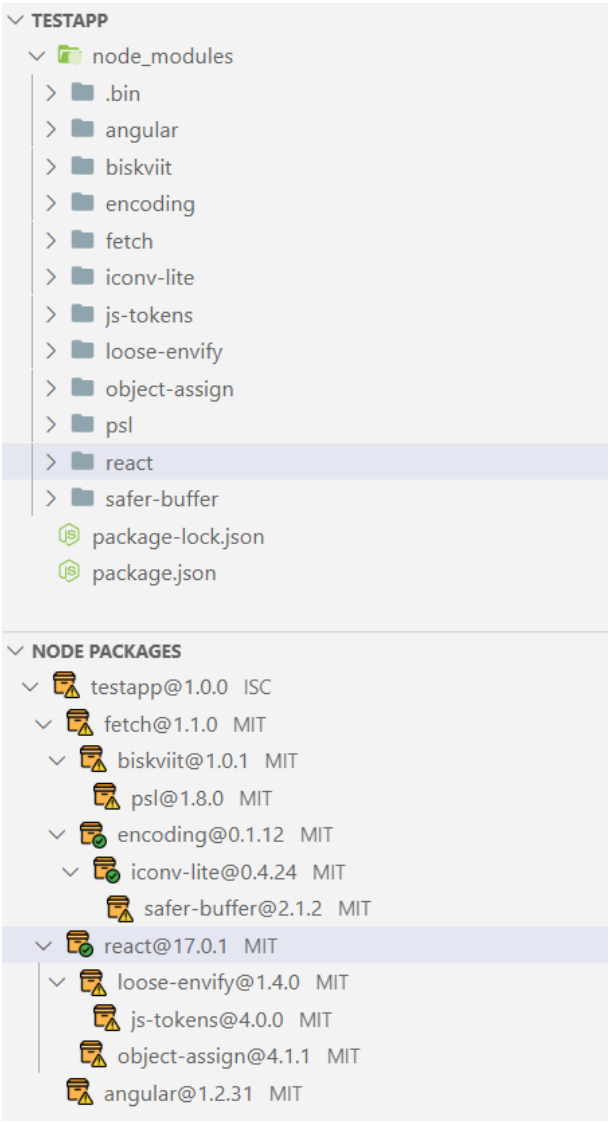

Abbildung 3.5: Tree-View in einem Beispielprojekt

Status Bar Item Wir verwenden die Status Bar um den Gesamtzustand über alle Abhängigkeiten der *package.json*-Datei direkt anhand einer Zustandsanzeige mittels eines Icons und einem Hinweis auf die Gesamtscore via Hover aufzuzeigen. Die Verwendung eines Icons mit den Zuständen für *Laden der Metriken*, *Keine Probleme gefunden*, *Probleme gefunden* und *Fehler ist augefteten*, zeigen wir kompakt den Gesamzustand an. Ein Problem kann eine tiefe Gesamtwertung des Node Packages, oder das Vorhandensein von Vulnerabilities von Abhängigkeiten bedeuten.

Webview Eine ausgewählte Dependency im Node Package Dependency Tree wird in einem neuen Tab als Webview geöffnet. In der Webview fassen wir alle gesammelten Daten über die Dependency kompakt zusammen. Der Tab hat den Namen des Node Packages und als erstes wird die Version, der Typ und die Anzahl direkter Abhängigkeiten angezeigt. Im nächsten Abschnitt wird die Metrikauswertung graphisch aufgewertet dargestellt. Zusätzlich wird die Quelle der Metriken und die Berchnung kurz beschrieben. Im mittleren Teil der Webview werden die gefundenen Vulnerabilities aufgelistet. Zu jeder Vulnerability werden Details tabellarisch aufgelistet. Zudem gibt es die Rubriken

Übersicht, Empfehlung und Referenzen welche für eine genauere Analyse aufgeklappt werden können. Am Schluss haben wir noch einige Lizenzinformationen bezüglich erlaubten Einsatz der Lizenz, Einschränkungen und Limitationen zusammengefasst.

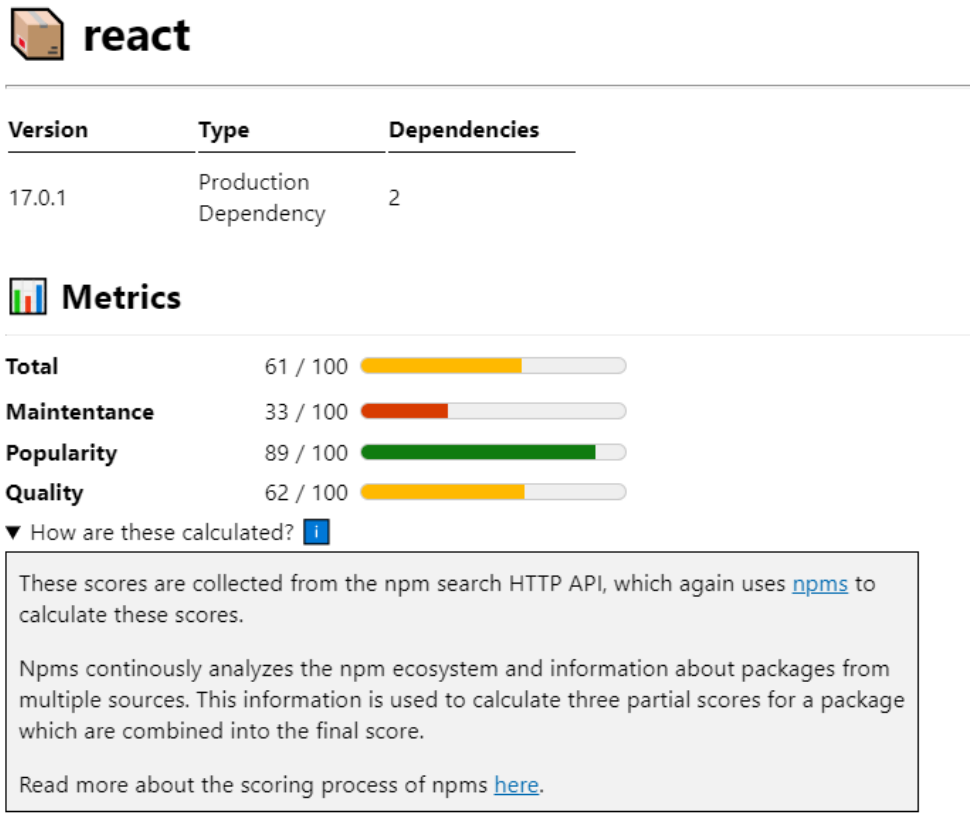

Abbildung 3.6: Metriken eines Node Packages in der Webview

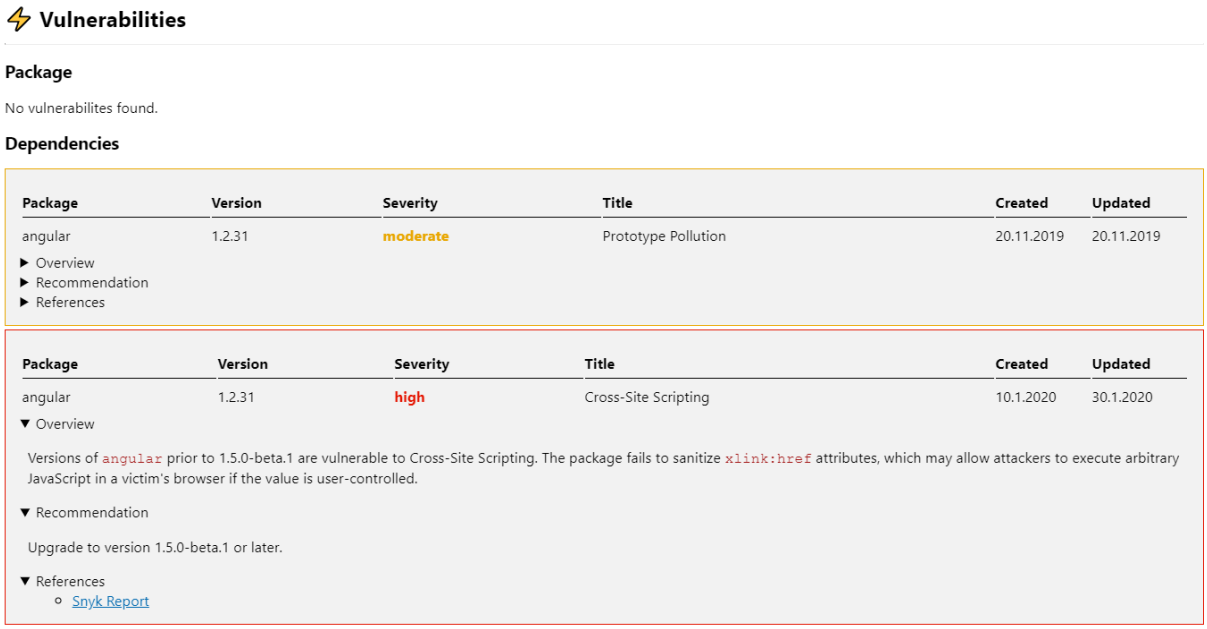

Abbildung 3.7: Vulnerabilites eines Node Packages in der Webview

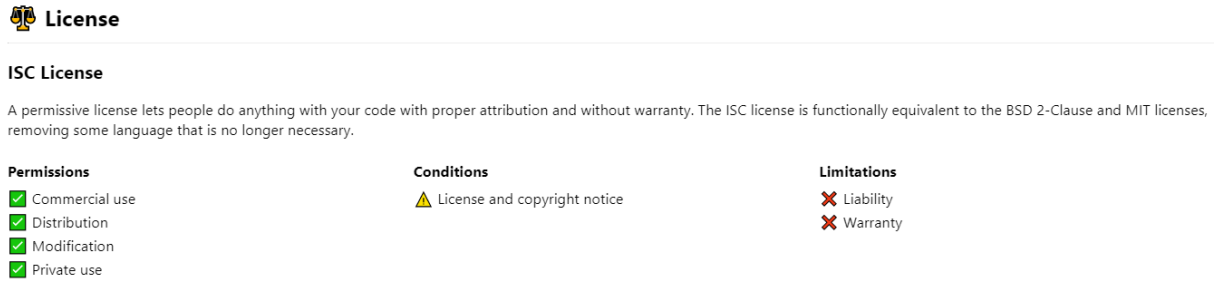

Abbildung 3.8: Lizenzinformationen eines Node Packages in der Webview

Problem Panel Das Problem Panel wird nur für die gefunden Vulnerabilities von direkten Abhängigkeiten verwendet. Wir mussten dies einschränken, weil es nur bei den sichtbaren Abhängigkeiten in der *packge.json*-Datei Sinn macht diese als Problem zu deklarieren. Probleme im Problem-Panel sind nur als Info Schweregrad gekennzeichnet um farblich nicht mit einem Schreibfehler im Editor in Konflikt zu geraten. Die Vulnerability selbst hat neben einem Schweregrad eine Beschreibung zur gefundenen Schwachstelle sowie welche Schritte zum beheben des Problems nötig sind. Dies ist in den meisten fällen, das erhöhen der Version des Node Packages in der die Schwachstelle behoben wurde.

Abbildung 3.9: Package.json Datei und Problem Panel mit gefundenen Vulnerabilites

Konfiguration

Das Plugin wird mit einer Standardkonfiguration ausgeliefert. Dabei sind die Debug-Log-Ausgaben standardmässig aktiviert, der Package Manager wird automatisch erkannt und die Tiefe der Analyse auf unbegrenzt gesetzt. Ein Anwender kann jederzeit seine eigene Benutzerkonfiguration erstellen, indem er die Werte unter den Einstellungen ändert.

Fehlerbehandlung

Auftretende Fehler werden als Benachrichtigung in einer gewohnten Visual Studio Code Benachrichtig dargestellt. Zusätzlich werden alle Meldungen in einen eigenen Output-Channel geschrieben. Falls aktive, sind in diesem Output-Channel alle Debug-Ausgaben und weitere Log-Einträge ersichtlich.

3.2 Zielerreichung

Wir haben das definierte Ziel der Arbeit, auch wenn nicht in vollen Umfang aller Use Cases erreicht. Es gab während dem Projektverlauf einige Herausforderungen, welche eine Kürzung des geplanten Umfangs erforderte. Es sind alle Kernfunktionen (*must* Use Cases) umgesetzt und es wurde ein solides Fundament für eine Weiterentwicklung geschaffen. Die Nachfolgenden Kapitel beschreiben die implementierten Ergebnisse im Detail.

Wir konnten in dieser SA die geplanten Ziele nicht in vollen Umfang erreichen. Das Hauptziel von aussagekräftigen Metriken über Node Packages anhand der package.json Datei visuell darzustellen, konnte erreicht werden. Wir mussten uns hauptsächlich auf die *must* Use Cases und User Stories beschränken. Wir kamen bei der Bearbeitung von Tasks nicht immer wie geplant vorwärts und mussten vielfach mehr kommunizieren als geplant. Durch den entstandenen hohen Kommunikationsaufwand im Team, musste auf einige der geplante Features verzichtet oder diese eingeschränkt werden. Erschwerend kam hinzu, dass wir im Team zu sehr unterschiedlichen Zeiten arbeiteten, was zu gewissen Redundanzen bei der Arbeit führte und die Aufteilung in einzelne Arbeitspakete erschwerte. Durch den Einsatz einer laufend aktualisierten Risikoanalyse konnte in vielen Fällen eine Lösung gefunden werden. Viel Aufwand wurde in eine mehrmals überarbeite Metrik-API Code-Umgebung gesteckt. Anhand verschiedenster Software-Engineering Techniken, wie dem Einsatz von Software-Pattern, strengem Linting des Codes in der CI-Pipeline, sowie Pull-Request für Code Review, wurde ein qualitativ hochstehender Code erschaffen.

3.3 Schlussfolgerungen

Nach 14 Wochen und über 670 Arbeitsstunden konnte das Projekt mit einem funktionsfähigen Visual Studio Code Plugin und einer unabhängigen Metrik-API abgeschlossen werden. Es konnten nicht alle Funktionalitäten im geplanten Umfang umgesetzt werden. Im folgenden Kapitel wird die erreichte Arbeit bewertet und geben eine Schlussfolgerung zum gesamten Projekt.

3.3.1 Bewertung der Ergebnisse

Ziele aus der Aufgabenstellung Wir habe wie geplant ein funktionsfähiges VS Code Plugin mit einer unabhängigen Metriken-API entwickelt. Die Metriken werden direkt im Plugin anhand der installierten Abhängigkeiten aus der package.json Datei kompakt in einer Baumstruktur visuell dargestellt. Die Metriken können auf Befehl neu berechnet werden und verwenden eine konfigurierbare Tiefe für die Analyse. Es wurden verschiedene Methoden aus dem Software-Engineering angewendet um eine hohe Code Qualität zu erreichen. Die Verwendung geeigneter Komponenten in VS Code stellten eine optimale Benutzbarkeit sicher. Es konnten nicht alle Aufgaben aus der Aufgabenstellung umgesetzt werden. So fehlen z.B. die Historien von Metriken und bei der Ausarbeitung von Metriken wurde der Umfang gekürzt. Es gibt zwar eine Metrikauswertung pro Dependency, aber eine rekursive Bewertung über alle Abhängigkeiten wird nur beim Gesamtzustand berücksichtigt. Eine Live-View Änderung ist nur indirekt möglich. Für eine erneute analyse nachdem Anpassungen an den Abhängigkeiten in der *package.json* Datei vorgenommen wurden, muss zuerst sichergestellt werden, dass alle Abhängigkeiten lokal installiert wurden.

Ziele aus der Aufgabenstellung	Ziel erreicht
Lauffähiges VS Code Plugin	Ja
Direkte Integration der Metriken im Plugin	Ja
Live-View Änderungen an der package.json Datei	Teilweise
Bereitstellung einer UI unabhängigen Metriken-API	Ja
Gute Adaptierbarkeit in andere Entwicklerumgebungen	Ja
Sicherstellung der Benutzbarkeit	Ja
Einbeziehen von Sub-Abhängigkeiten	Teilweise
Einbeziehen von Node Package Vulnerabilities	Ja
Einbeziehen von Node Package Lizenzinformationen	Ja
Visualisierung der Metriken	Ja
Ausarbeitung von Metriken	Teilweise
Errechnen von Historien	Nein
Vorgehen nach Regeln des Software-Engineering	Ja

Use Cases Unsere zu Beginn definierten Use Cases waren nicht sehr detailliert definiert worden. Um die Zuweisung an eine Person zu vereinfachen und den Umfang eines Tasks besser zu definieren haben wir User Stories zu den Use Cases geschrieben. Anhand dieser User Stories wurden dann spezifische Tasks an Personen zugewiesen. Der Umfang wurde so gewählt, dass er möglichst innerhalb einer Iteration (2 Wochen) fertig gestellt werden konnte. Mit der Zeit ist dadurch eine unabhängigere Arbeitsweise entstanden. Der Kommunikationsaufwand war aber trotz dieser Arbeitsweise sehr hoch. Wir mussten uns aus Zeitgründen auf die Umsetzung der Prio 1 Use Cases konzentrieren.

Use Cases	Ziel erreicht
UC1: Zustand von Abhängigkeiten visualisieren (Prio1)	Ja
UC2: Konfiguration anpassen (Prio 2)	Teilweise
UC3: Lizenzdetails anzeigen (Prio 1)	Ja
UC4: Sicherheitslücken anzeigen (Prio 1)	Ja
UC5: Zustand visualiseren (Prio 2)	Teilweise
UC6: Trendananlyse der Abhängigkeiten (Prio 3)	Nein
UC7: Report generieren (Prio 2)	Nein
UC8: Integration in CI-Pipeline anpassen (Prio 3)	Nein

Nicht funktionellen Anforderungen Es konnten alle berücksichtigten nicht funktionellen Anforderungen wie geplant eingehalten werden.

Berücksichtige nicht funktionellen Anforderungen	Ziel erreicht
NFR1: Unabhängige Erweiterbarkeit der Metriken-API	Ja
NFR2: Kein Interferenzen mit anderen Plugins	Ja
NFR3: Farbschema basiert auf Farbthemes und verständliches Fehlerhandling	Ja
NFR4: Erweiterung ist auf den drei Betriebssystemen macOS, Windowsund Linux nutzbar	Ja
NFR5: Nachvollziehbarkeit von Fehlern und wichtigen Operationen	Ja
NFR6: Plugin benötigt keine privilegierte Nutzer-Rechte	Ja
NFR7: Steuerung der Analyse	Ja

3.3.2 Risikomanagement

Das Risikomanagement hat sich als sehr nützlich herausgestellt. Die grössten Riskos waren im Zusammenhang mit der Teamkommunikation und den unterschiedlichen Arbeitszeiten sowie dem Wissensaufbau. Diese Risikos mit geigneten Massnahmen zu beheben war eine grosse Herausforderung. Trotz verschiedensten Massnahmen, musste dafür viel Zeit aufgewendet werden.

3.4 Technische Schuld

Während der Entwicklung wurden bewusst einige technische Schulden aufgenommen, um im Gegenzug schneller voranzukommen. Die wesentlichsten davon sind nachstehend festgehalten.

3.4.1 Visual-Studio-Code-Erweiterung

Die Dateien, welche den Code für die Webview enthalten, machen aktuell keine Trennung zwischen HTML, CSS und JavaScript. Auf den Einsatz einer Template-Engine wurde bewusst verzichtet, einerseits da das HTML nur einen marginalen Teil der Applikation ausmacht, und andererseits weil dies zusätzliche Zeit für die Evaluation in Anspruch genommen hätte. Sollten künftig noch weiter Inhalte für die Webviews hinzugefügt werden, macht der Einsatz einer leichtgewichtigen Template-Engine und die Trennung von HTML, CSS und JavaScript auf jeden Fall Sinn.

3.4.2 Metriken-API

Einige der Datenmodelle weisen eine geringe Kohäsion auf oder sind zu stark aneinander gekoppelt. Teilweise könnten diese vereinfacht werden oder durch Extraktion aus verschiedenen Modellen neue, kohäsivere Klassen erstellt werden.

3.5 Fazit und Ausblick

Der Umfang der Arbeit ist zu Beginn noch sehr optimistisch gewählt worden. Der Aufbau des nötigen Wissens, welches teilweise mit Pair-Programming erlangt wurde, brauchte viel Zeit. Ebenfalls ist mehr Zeit für Recherchen aufgewendet worden, als ursprünglich geplant. Der hohe Kommunikationsaufwand und die unterschiedlichen Arbeitszeiten im Team sind von uns zudem unterschätzt worden. Dafür half uns der Einsatz geeigneter Tools und Software-Engineering-Praktiken, den Überblick zu behalten und eine hohe Codequalität zu gewährleisten. Durch den Gebrauch von agilen Methoden konnten wir einen kontinuierlichen, iterativen Arbeitsfortschritt erreichen. Der durch den anfänglichen Mehraufwand eingeschränkte Umfang konnte dann aber ohne weitere technische Hürden umgesetzt worden.

Nach Abschluss der Studienarbeit ist keine Weiterentwicklung an der Erweiterung geplant. Die Erweiterung und die Metriken-API können durch die gewählte Architektur aber sehr gut erweitert werden. So könnte beispielsweise ein Erweiterung für eine andere Entwicklungsumgebung erschaffen werden oder die Metriken-API in einer CI-Pipeline eingebunden werden.

Teil II

Projektdokumentation

Kapitel 4

Projektmanagement

4.1 Projektteam

Das Team setzt sich aus Etienne Beyeler, Flavio Böni und Severin Hauser zusammen. Alle arbeiten in der Rolle als Entwickler.

Im Team existieren keine hierarchische Strukturen. Wesentliche Entscheidungen werden im Plenum getroffen, wobei alle Teammitglieder gleichgestellt sind.

4.1.1 Externe Schnittstellen

- Silvan Gehrig (Betreuer, *Product Owner*), Institut für Software (IFS)
- Mirko Stocker (*Product Owner*), Institut für Software (IFS)

4.2 Besprechungen

Aufgrund der aktuellen Situation bezüglich COVID-19 finden Besprechungen vor allem online via Microsoft Teams statt. Das Team trifft sich dabei mehrmals pro Woche, etwa um das Vorgehen festzulegen, sich auszutauschen oder zum zusammenarbeiten.
Einmal in der Woche, für gewöhnlich am Dienstag von 10 bis 11 Uhr, findet ein Review-Meeting mit Silvan Gehrig statt.

Es werden 15 h pro Woche für Besprechungen eingeplant (3 Teammitglieder * 5 h). Alle Review-Meetings sowie wesentliche Punkte der Teambesprechungen werden protokolliert.

4.3 Vorgehen

Das Vorgehen im Projekt ist ablauforientiert, d. h. das Projekt wird durch das Setzen von Meilensteinen in zeitliche Abschnitte (= Projektphasen) unterteilt. [15] Konkret wird als Vorgehensmodell eine Mischung aus Scrum und Unified Process (UP) verwendet.

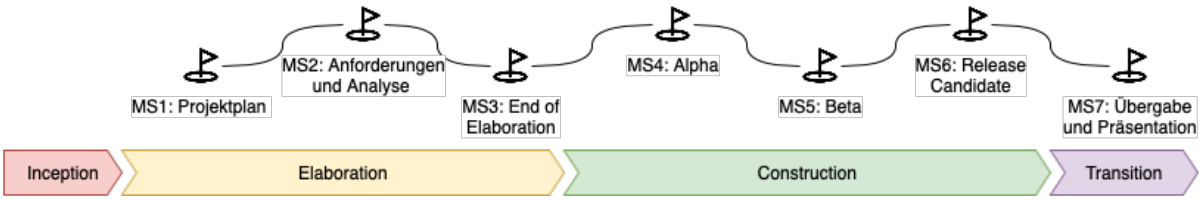

Abbildung 4.1: Übersicht über die Phasen und Meilensteine

4.3.1 Zeitaufwand

Die Studienarbeit erstreckt sich über eine Dauer von 15 Wochen. Der Zeitaufwand pro Teammitglied soll dabei gleichmässig auf die Projektdauer verteilt werden und lässt sich aus den acht ECTS, welche für das Modul ausgeschrieben sind, errechnen. Pro ECTS ist mit einem Aufwand von 30 Stunden zu rechnen.

$$8\,ECTS\ *\ 30\,h\ /\ 15\,Wochen\ =\ 16\,h\,pro\,Woche$$

Gestartet hat die Studienarbeit mit dem Kickoff-Meeting am 11. September und wird voraussichtlich mit der Abgabe am 18. Dezember enden.

4.3.2 Phasen

Aus UP werden die folgenden vier Phasen übernommen:

- Inception (11. bis 21. September)
- Elaboration (22. September bis 19. Oktober)
- Construction (20. Oktober bis 14. Dezember)
- Transistion (15. bis 18. Dezember)

4.3.3 Iterationen

Da sich die Phasen teilweise über mehrere Wochen erstrecken, wird innerhalb dieser in Sprints gearbeitet. Die Sprintlänge wurde auf zwei Wochen festgelegt, kann aber falls nötig auf eine Woche begrenzt werden. Ein Sprint beginnt jeweils mit der Sprintplanung und endet mit einem kurzen Sprintreview und, falls gewünscht, mit einer Retrospektive.

4.3.4 Meilensteine

Es wurden sieben Meilensteine festgelegt, welche jeweils in einer bestimmten UP-Phase angesiedelt sind. Diese können als Zwischenziele gesehen werden und sollen helfen, das Gesamtziel am Ende einer Phase zu erreichen. Ein Meilenstein enthält mehrere Scrum Sprints, die eine oder zwei Wochen dauern.

Jedem Meilenstein sind gewisse Tätigkeiten zugeordnet und ein Stichtag definiert.

- MS1 – Projektplan (28. September)
  - Projektplan
  - Risikoanalyse

- MS2 – Anforderungen und Analyse (12. Oktober)
  - Anforderungsspezifikationen (funktional und nichtfunktional)
  - Domänenanalyse
- MS3 – End of Elaboration (19. Oktober)
  - High-Level-Architektur
  - Prototyp
  - Tooling: Repository, CI/CD, Entwicklungsumgebung bereit
- MS4 – Alpha: Lauffähiges MVP (16. November)
  - Usability-Test
  - Detaillierte Softwarearchitektur
- MS5 – Beta: MVP um gewünschten Funktionsumfang erweitert (30. November)
  - Systemtest-Spezifikationen
  - Qualitätssicherung
- MS6 – Release Candidate: finale, getestete Version (17. Dezember)
  - Systemtest-Protokoll
  - Deployment-Plan
- MS7 – Übergabe (18. Dezember)
  - Schlussbericht

4.4 Anforderungserhebung

Um sich in die Lage von potenziellen Nutzenden zu versetzen und somit realistische Anforderungen zu erheben, welche deren Bedürfnisse befriedigen, werden zuerst Personas erstellt. Basierend auf diesen werden dann Use Cases ausgearbeitet und priorisiert. Während des Projektverlaufes werden aus den Use Cases dann sukzessive User Storys erstellt. Da zum entsprechenden Zeitpunkt gegebenenfalls bereits weitere Erkenntnisse vorliegen, kann es sein, dass sich die Use Cases und User Storys voneinander unterscheiden.

4.4.1 Arbeitspakete

Arbeitspakete werden im Backlog erfasst und sind stets einem Meilenstein zugeordnet. Der Backlog wird in ClickUp verwaltet und ist priorisiert. Arbeitspakete mit hoher Priorität sind dabei ausführlicher beschrieben als solche mit niedriger Priorität. Arbeitspakete können jederzeit dem Backlog hinzugefügt werden.

4.4.2 Zeitschätzungen

Der Zeitaufwand eines Arbeitspaketes in Stunden wird gemeinsam im Team geschätzt. Dabei werden Fibonacci-Zahlen verwendet. Durch die immer grösser werdenden Abstände

der Fibonacci-Zahlen ist so automatisch ein grösserer Puffer vorhanden.

4.4.3 Initiale Arbeitspakete

Basierend auf den Erfahrungen des Engineering-Projektes und den oben definierten abzugebenden Dokumenten haben sich initiale Arbeitspakete, welche im Anhang A.4 zu finden sind.

4.5 Infrastruktur

Für die Studienarbeit werden folgende physische und virtuelle Tools eingesetzt.

4.5.1 Physisch Tools

PC/Laptop Die Entwicklung findet ausschliesslich auf den persönlichen Geräten der Teammitglieder statt.

4.5.2 Virtuelle Tools

ClickUp Für die Projektplanung (Meilensteine, Sprints, Tasks etc.) wird ClickUp eingesetzt. Durch die Integration mit GitHub können ClickUp-Tasks in GitHub und Pull Requests aus GitHub in ClickUp referenziert werden. ClickUp dient als primäres Planungswerkzeug.

GitHub Zur Versionsverwaltung des Quellcodes, Code Revies und CI/CD wird GitHub verwendet. Alle Repositories werden in der GitHub-Organisation «ost-sa-node» eingegliedert.

Google Tabellen Für die Zeiterfassung wird eine Google-Tabelle verwendet. Die Teammitglieder erfassen darin ihre manuell ihre Zeiten auf die Aufgaben, welche aus ClickUp übernommen werden. Dies erlaubt eine unkomplizierte Auswertung von Soll/Ist nach Aufgaben und Teammitglied.

Zur Zeiterfassung existieren auch Softwarelösungen, wie z. B. Toggle Track, Clockodo oder Everhour. Leider bietet keine dieser Lösungen den gewünschten Funktionsumfang in einer kostenlosen Version an.

Microsoft Teams Für die Kommunikation im Team und Meetings mit externen Personen wird Microsoft Teams verwendet. Wichtige Dokumente sind ebenfalls in der Dateiablage von Microsoft Teams zu finden.

Notion Für erste Entwürfe von Dokumenten und als Wissensbasis wird das kollaborative Notiztool Notion verwendet. Durch seine universelle und einfache Verwendung ist es ideal geeignet, um Ressourcen zu sammeln oder gemeinsam an Entwürfe zu arbeiten.

Overleaf Overleaf ist ein einfach zu verwendender, webbasierter LaTeX-Editor und wird verwendet, um die finale Dokumentation zu erstellen.

Visual Studio Code Die Wahl der IDE, welche während der Studienarbeit verwendet wird, ist grundsätzlich frei. Empfohlen wird Visual Studio Code, was auch aufgrund des Zieles der Studienarbeit naheliegend ist. Des Weiteren verfügt Visual Studio Code über ein gut ausgebautes Ökosystem mit diversen (populären) Erweiterungen.

4.6 Qualitätsmassnahmen

4.6.1 Dokumentation

In der Dokumentation werden alle Informationen rund um die Studienarbeit strukturiert aufbereitet. Sie gliedert sich in drei Teile: die Projektdokumentation, den technischen Bericht und den Anhang.

Die Projektdokumentation umfasst alle Informationen, welche Planung, Administration und Organisation betreffen. Der technische Bericht beinhaltet Analysen, Details und Entscheidungen. Im Anhang sind alle weiteren Dokumente angehängt, wie z. B. Besprechungs- und Testprotokolle, Zeitauswertungen und die persönlichen Reflexionen.

Informationen, welche sich direkt auf den Code beziehen (Schnittstellenbeschreibungen, Abhängigkeiten, Setup etc.) werden direkt im Repository erfasst – entweder im Quellcode oder in einer entsprechenden Readme-Datei.

4.6.2 Systemtests

Abgeleitet von den umgesetzten Use Cases wurden verschiedene Testfälle festgelegt und durchgeführt. Die Testfälle und Testprotokolle mit Kommentare sind im Anhang dokumentiert.

4.6.3 Entwicklung

Der Quellcode wird in Git-Repositories eingecheckt und auf GitHub verwaltet. Dies bringt einerseits den Vorteil der Versionierung mit sich, anderseits ist durch die dezentrale Arbeitsweise von Git auch ein lokales Backup des Codes bei jedem Entwickler und auf dem GitHub-Server vorhanden.

Vorgehen

Das Vorgehen während der Entwicklung orientiert sich am GitHub-Flow. Dabei befindet sich der aktuelle Stand der Software auf dem *Main Branch*, die Entwicklung von Funktionen erfolgt auf separaten *Feature Branches*. [7] Das Vorgehen unterteilt sich in folgende Schritte:

1. Erstellen eines Feature Branches, basierend auf dem Main Branch.
2. Committen von Änderungen und Tests auf den Feature Branch. Bei jedem gepushten Commit werden automatisch die Tests in der GitHub Actions CI ausgeführt.
3. Erstellen eines *Draft Pull Requests*, sobald erstes Feedback gewünscht ist.
4. vornehmen von weiteren Änderungen.

5. Wechsel des Draft Pull Requests zu einem regulären Pull Request um den Request als «*ready to review*» zu markieren.
6. Review des Pull Requests durch Teammitglieder.
7. Bei positivem Review und erfolgreichem Durchlaufen der CI wird der Pull Request akzeptiert und somit die Änderungen des Feature Branches in den Main Branch gemerged.

Direkte Änderungen am Main Branch werden blockiert um das Einhalten dieses Ablaufs sicherzustellen.

4.6.4 Code Reviews

Um die Codequalität sicherzustellen und den Wissensstand des gesamten Teams zu verbessern werden Code Reviews durchgeführt. Dies geschieht bei jedem Pull Request, um so sicherzustellen, dass nur akzeptable Änderungen in den Main Branch gemerged werden.

Pull Requests werden jeweils durch mindestens ein, wenn möglich aber alle, Teammitglied durchgesehen.

Es gilt vor allem auf folgende Punkte zu achten:

- Ist die Funktionalität gemäss Arbeitspaket implementiert?
- Werden alle *Styleguides* eingehalten?
- Wurden entsprechende Tests erstellt bzw. ergänzt?
- Wurde der optimale Ansatz zur Lösung des Problemes gewählt? Sind keine Hacks enthalten?

Diese Punkte dienen als Richtlinie und sollten in Abwägung mit dem Aufwand abgestimmt werden. Damit soll die Effizienz hoch gehalten werden, ohne an Qualität zu verlieren.

4.6.5 Styleguides

Formatierung Zur Formatierung des Quellcodes wird Prettier verwendet, welcher bereits Voreinstellungen enthält. Abweichungen davon können in einer Konfigurationsdatei angepasst werden, wie z.B. das ausschliessen von generierten Ordnern. Die Verwendung der Prettier-Erweiterung für Visual Studio Code erlaubt es ausserdem, den Code automatische beim Speichern zu formatieren.

Linting Ergänzend zu Prettier wird ESLint, ein Werkzeug zur statischen Analyse von Quellcode, verwendet. ESLint wird mit der Airbnb-Basiskonfiguration initialisiert, welche ebenfalls angepasst werden kann.

Die Gesamtheit der in ESLint eingestellten Standard Regeln bildet somit den Code-Styleguide der Studienarbeit. Durch Verwendung der dazugehörigen ESLint-Erweiterung für Visual Studio Code können Verletzungen des Styleguides direkt im Editor angezeigt werden.

Conventional Commits Um eine einheitliche *Commit History* zu erreichen, werden *Commit Messages* gemäss der Spezifikation von Conventional Commits verfasst.

Die Einhaltung der Styleguides wird automatisch in der CI geprüft.

4.6.6 Definition of Done

Die Definition of Done besagt, wann ein Arbeitspaket als abgeschlossen gilt. Sie schafft Transparenz, indem sie allen Teammitgliedern ein gemeinsames Verständnis davon vermittelt.[17]

Die Definition of Done für die Studienarbeit wurde wie folgt festgelegt:

- Alle Akzeptanzkriterien sind erfüllt.
- Die Dokumentation ist aktualisiert.

Für Arbeitspakete mit Quellcode gilt zusätzlich:

- Der Code ist komplett und im Versionierungssystem eingespielt.
- Der Code wurde in einem Code Review für gut befunden oder im Pair Programming erstellt.
- Alle Styleguides sind eingehalten.
- Der Code ist durch Unit Tests abgedeckt, welche erfolgreich durchlaufen.

4.7 Softwaretests

Zu jeder Funktion werden Unit Tests und ggf. Integration Tests geschrieben. Sie befinden sich im selben Repository wie der Quellcode.

Für die automatisierten Tests werden folgende Frameworks und Bibliotheken verwendet:

- Mocha (Test Framework): Erweiterungen, welche durch den Extension Generator von Visual Studio Code erstellt werden, beinhalten standardmässig bereits ein vorkonfiguriertes Mocha.
- Chai (Assertion Library): Erweitert das standardmässig verwendete Node.js-Modul «Assert» um zusäztliche Funktionen. Chai wird oft in Kombination mit Mocha verwendet.
- Istanbul mit nyc (Testabdeckung): Istanbul ist ein populäres Tool zum Messen der Testabdeckung und kann gut mit Mocha zusammen verwendet werden kann.

4.7.1 Unit Tests

Um die Funktionalität des Programmes zu gewährleisten und Regressionen vorzubeugen werden Unit Tests durchgeführt. Diese sind Bestandteil von jeder Funktion und werden somit in der Zeitschätzung bereits eingerechnet. Die Unit Tests laufen sowohl lokal – etwa um nach einem Test-First-Development-Ansatz zu entwickeln – und werden in jedem Fall auch automatisch in der CI ausgeführt. Sie sind ein wichtiger Teil in der Qualitätssicherung, daher wurde eine Mindesttestabdeckung von 75 % festgelegt.

4.7.2 Integration Tests

Automatisierte Integrationstests sind an dieser Stelle nicht vorgesehen. In der Entwicklungsphase werden jedoch von Zeit zu Zeit manuelle Tests durchgeführt, welche das Zusammenspiel des Gesamtsystemes prüfen. Sollten während der Entwicklung drastische Qualitätsprobleme auftreten, welche auf die Absenz der Integrationstests zurückzuführen sind, wird eine Automatisierung nochmals berücksichtigt.

4.7.3 Usability Tests

Während einer frühen Entwicklungsphase wird mit einem einfachen Prototypen ein Usability Test durchgeführt. Dazu werden einige Benutzerszenarien ausgearbeitet und mit verschiedenen Benutzergruppen durchgespielt. Das Feedback soll schnell in die Entwicklung zurückfliessen und dazu dienen, Grundlegende Usability-Probleme früh zu erkennen und darauf reagieren zu können.

Kapitel 5

Risikomanagement

Im nachfolgenden Kapitel sind die möglichen Risiken, welche identifiziert wurden und während der Projektdauer auftreten können, sowie der damit verbundene Umgang festgehalten. Die Risiken wurden bei Erreichen jedes Meilensteins neu bewertet und neu aufgetretene Risiken erfasst.

Eine Übersicht der Revisionen im Verlauf des Projekts ist im Anhang A.6 zu finden.

5.1 Umgang mit Risiken

Die Ergebnisse aus der Risikoanalyse fliessen in die Planung und die Priorisierung von Funktionen mit ein. In der letzten Iteration der Construction-Phase wird ein Puffer eingeplant, um Verzögerungen durch das Eintreten solcher Risiken zu kompensieren. Tritt ein schwerwiegendes Risiko ein, dessen Auswirkungen nicht durch den Puffer kompensiert werden können, wird der geplante Funktionsumfang verringert, um die Fertigstellung sicherzustellen.

Eintrittswahrscheinlichkeit Die Eintrittswahrscheinlichkeiten unterteilen sich in die Quartile *Sehr gering (0–25 %)*, *Gering (25–50 %)*, *Mittel (50–75 %)* und *Hoch (75–100 %)*.

Schaden Das Ausmass der potenziellen Schäden unterteilt sich, aufsteigend nach Schwere, in die vier Bereiche *Sehr tief*, *Tief*, *Mittel* und *Hoch*.

5.2 Identifizierte Risiken

Die erkannten Risiken sind in nachstehenden Tabelle festgehalten und werden anschliessend genauer erläutert.

Nr.	Risiko	Eintritts-Ws.	Schaden
R1	Zwischenmenschliche Konflikte	mittel	mittel
R2	Ausfall von Teammitgliedern	sehr gering	hoch
R3	Mangelnde Qualität der Ergebnisses	gering	mittel

R4	Nicht erfüllbare Funktionalität	mittel	tief
R5	Datenverlust	sehr gering	mittel
R6	Lebensdauer von externen Abhängigkeiten	gering	tief
R7	Scope Creep	mittel	sehr tief
R8	Verzögerungen aufgrund fehlender gemeinsame Arbeitszeiten	hoch	tief
R9	Abhängigkeiten zu Drittparteien	gering	mittel
R10	Nichtfertige Kernfunktionalität blockiert restliche Funktionen	mittel	mittel

R1: Zwischenmenschliche Konflikte

Konflikte können die Zusammenarbeit im Team signifikant verschlechtern.

Vorbeugung Scrum-Retrospektiven am Ende jeder Projektphase oder bei Bedarf (bspw. falls von einem Teammitglied gewünscht).

R2: Ausfall von Teammitgliedern

Langfristiger Ausfall von Mitgliedern aufgrund von Krankheiten (bspw. Covid-19) oder Verletzungen.

Vorbeugung

- Einhalten der Empfehlungen und Vorschriften des Bundesamt für Gesundheit (BAG)
- Regelmässiges *pushen* von Quellcode auf GitHub
- Verwenden von Notion und Overleaf für schriftliche Artefakte

Verhalten beim Eintreten Reduktion von nicht-essentiellen Funktionen und Neuzuweisung von Aufgaben an die restliche Teammitglieder.

R3: Mangelnde Qualität der Ergebnisse

Lieferartefakte erfüllen nicht die gewünschte Qualität.

Vorbeugung

- Klaren Review-Prozess definieren
- Codeänderungen sind nur durch *Pull Requests* möglich
- *Definition of Done* für jedes Arbeitspaket definieren
- Zuteilen von Tasks, welche nicht direkt einer User-Story zugehörig sind

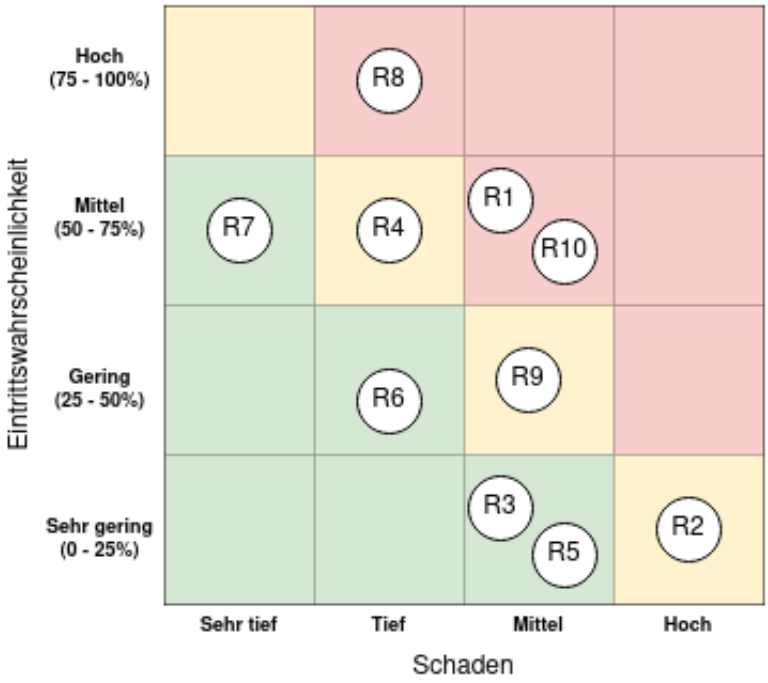

Abbildung 5.1: Risikomatrix

R4: Nicht erfüllbare Funktionalität

Die geplante Funktionalität ist mit Visual Studio Code / Node.js nicht oder nicht im gewünschten Rahmen umsetzbar.

Vorbeugung Frühzeitige Entwicklung von Prototypen, um die Kapazitäten der Umgebungen zu erproben.

Verhalten beim Eintreten Reevaluation der betroffenen Funktion, ggf. Ersatz mit einer vergleichbarer Funktionalität.

R5: Datenverlust

Das Repository wird gelöscht, lokale Änderungen gehen verloren oder werden überschrieben.

Vorbeugung

- Regelmässiges *pushen* von Quellcode auf GitHub
- Verwendung von Cloud-Diensten
- Regelmässige Backups

R6: Lebensdauer von externen Abhängigkeiten

Verwendete Bibliotheken oder Tools werden nicht mehr gewartet.

Vorbeugung

- Verwendung von etablierten Abhängigkeiten
- Evaluation von Abhängigkeiten vor deren Verwendung
- Implementationen mit möglichst tiefer Kopplung zu Abhängigkeiten

R7: Scope Creep

Der Aufwand für die geplante Funktionalität übersteigt das Projektbudget.

Vorbeugung Die Funktionalitäten nach Relevanz klassifizieren und entsprechend priorisieren.

R8: Verzögerungen aufgrund fehlender gemeinsame Arbeitszeiten

Aufgrund verschiedener Stundenpläne und beruflicher Tätigkeiten sind nur wenige Stunden pro Woche verfügbar in welchen alle Teammitglieder gleichzeitig Zeit für die Arbeit haben. Dies kann zu Verzögerungen führen falls wichtige Themen gemeinsam besprochen werden müssen.

Vorbeugung

- Freihalten der gemeinsamen Zeiten für Arbeit
- Gute Vorbereitung vor Besprechungen, um Zeit effektiv nutzen zu können
- Frühzeitiges Melden von möglichen Problemen
- Verschiebung des Sprint-Start-Tages von Dienstag auf Mittwoch um mehr Zeit für die Vorbereitung auf das Review und den Sprint-Abschluss zu erhalten

R9: Abhängigkeiten zu Drittparteien

Die verwendeten Drittparteien können *Rate Limits* einführen, die Nutzungsbedingungen und API ändern oder ihr Angebot einstellen. Dies kann dazu führen, dass gewisse Teile der Applikation nur noch beschränkt oder gar nicht mehr funktionieren.

Vorbeugung

- Evaluation der Drittpartei vor deren Einbindung
- Gewährleisten der Stabilität durch Auslegen der Architektur, sodass der Ausfall einer Drittpartei keinen Einfluss auf das Laufzeitverhalten der Erweiterung hat.

Verhalten beim Eintreten Reevaluierung der betroffenen Drittpartei, ggf. Ersetzen durch vergleichbare Drittpartei.

R10: Nichtfertige Kernfunktionalität blockiert restliche Funktionen

Ist eine zentrale Funktion noch nicht fertig während ein davon abhängige Funktion bereits implementiert werden sollte, kann dies zu Verzögerungen im Zeitplan kommen und Team-Mitglieder in ihrer Arbeit blockieren. Verzögerungen können auch dadurch entstehen, dass Wissen zuerst aufgebaut werden muss. Dies führt so zu einem grösseren Arbeitsaufwand als geplant und verursacht Verspätungen.

Vorbeugung Priorisierung der Implementation von Kernfunktionalitäten und frühzeitige Fixierung deren Interfaces. Pair Programming bei Unsicherheiten.

Kapitel 6

Anforderungsspezifikationen

6.1 Personas

Um die Herleitung der funktionalen Anforderungen und dessen Priorisierung zu vereinfachen, wird mit Personas gearbeitet. Personas sind (fiktive) Modelle, welche die konkreten Eigenschaften einer Person besitzen, dabei aber eine Gruppe von Anwendenden repräsentieren.

Für die Studienarbeit sind zwei Personas erstellt worden: Marina Tüftelus und Kurt Hässig. Ihre Eigenschaften basieren auf unseren Beobachtungen in der Arbeitswelt. Die Personas sind im Anhang A.1 zu finden.

6.2 Use Cases

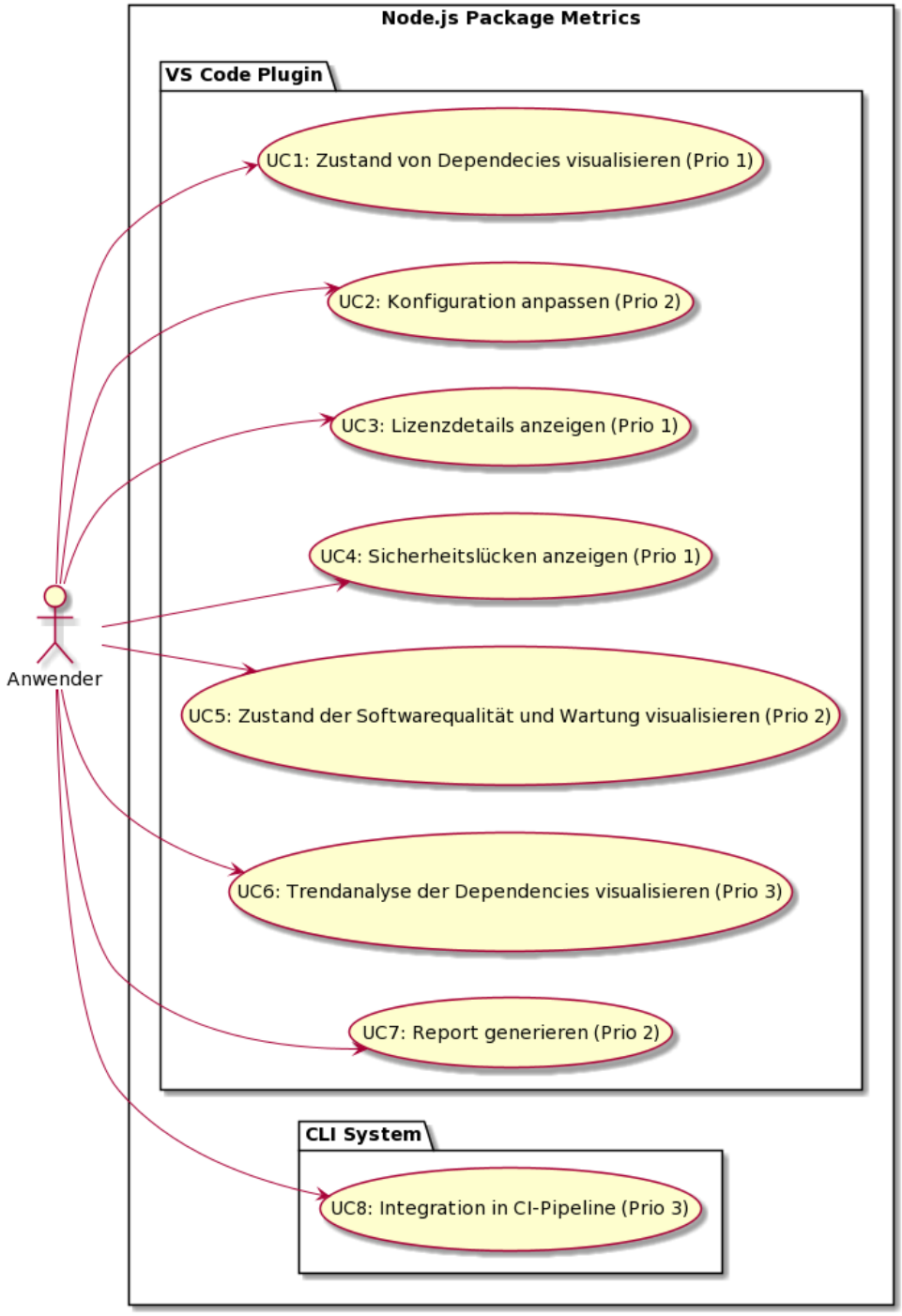

Abbildung 6.1: Use-Case-Diagramm

Priorisierung Da nicht alle Use Cases während der Dauer der Studienarbeit umgesetzt werden können, werden diese priorisiert. Dabei wird zwischen den folgenden drei Prioritäten unterscheiden: *must*, *should*, und *could*.

Use Cases mit der Priorität *must* werden als Erste bearbeitet.

User Stories Aus den Use Cases werden fortlaufend User Storys erstellt, beginnend bei den *must* Use Cases. Eine User Story entspricht dabei einem konkreten Arbeitspaket, beinhaltet eine Zeitschätzung und kann direkt einer Person für die Umsetzung zugewiesen werden. Essentiell ist hierfür, die optimale Grösse für eine User Story zu finden. Zu grosse User Storys sollen durch Story Splitting in kleinere aufgeteilt werden.

Zur Beschreibung der User Storys wird die folgende Vorlage verwendet:

User-Story-Vorlage

Als [Akteur] von [Anwendung] möchte ich [Zielfunktion], damit [Nutzen].

Akzeptanzkriterien

- .....

Technische Hinweise

- .....

Mittels Akzeptanzkriterien lässt sich eindeutig erkennen, ob eine User Story abgeschlossen ist. Die User Storys lassen sich ausserdem mit technischen Hinweisen für die Umsetzung ergänzen.

Die Use Cases wurden zu beginn der Arbeit erstellt und sind danach nicht mehr angepasst worden. Daher kann es zu geringfügigen Abweichungen zwischen den Use Cases und den User Stories kommen.

6.2.1 Akteure

Akteur	Beschreibung
Anwender	Der Anwender ist Hauptbenutzer des Systemes. Er hat die Erweiterung im Visual Studio Code installiert und benutzt sie. Es spielt keine Rolle, ob die Nutzung im privaten oder professional Umfeld (z. B. Hobbyentwickler, IT-Freelancerin etc.) geschieht.

6.2.2 Use-Case-Beschreibungen

In den folgenden Use Cases wird auf die erstellten User Stories verwiesen, welche im Anhang A.5 zu finden sind.

UC 1: Zustand von Abhängigkeiten

Kurzbeschreibung

Der Anwender sieht den Zustand der Abhängigkeiten, welche im aktuellen Projekt vorhanden sind. Der Zustand setzt sich aus verschiedenen Metriken zusammen, welche voreingestellt sind oder selbst festgelegt werden können (vgl. UC 2).

Akteur	Anwender

Trigger Der Anwender öffnet die `package.json` Datei im Editor.

Hauptszenario

1. Der Anwender öffnet die `package.json` Datei im Projekt
2. Die Extension wird gestartet und erkennt die offene `package.json` wodurch der Anwender die Metrikberechnung starten kann.
3. Dem Anwender wird nach der Metrikberechnung der Gesamtzustand der Dependencies es als Statusinformation und eine detailierte Dependency Auswertung in Form einer Baumstruktur dargestellt.
4. Neben jeder Abhängigkeiten (Felder `dependencies` und `devDependencies` in der Baumstruktur) zeigt ein Icon den aktuellen Zustand an. Icon-Zustände: schlecht, Warnung, gut, nicht überprüfbar.

Erweiterungen

2a. Der Anwender fügt eine Dependency zum `package.json` hinzu*.
2a1. Der Anwender speichert und installiert die neue Dependency.
2a2. Der Anwender startet die Neuberechnung der Metriken.
2a3. Der Zustand der neuen Dependency wird in der Baumstruktur ebenfalls dargestellt und der Gesamtzustand wird aktualisiert dargestellt.
2b. Der Anwender ändert die Version einer Dependency im `package.json`*.
2b1. Der Anwender speichert und installiert die neue Dependency.
2b2. Der Anwender startet die Neuberechnung der Metriken.
2b3. Der Zustand der neu spezifizierten Version der Dependency wird angezeigt.

* entweder manuell oder per CLI

User Stories

- US 1.1: Dependency Tree
- US 1.2: Gesamtzustand des Projektes
- US 1.3: Zustand von Dependencies in Tree ansehen
- US 1.4: Aktualisieren des Zustandes

UC 2: Konfiguration anpassen

Kurzbeschreibung

Der Anwender kann das Verhalten des Plugins auf seine Bedürfnisse anpassen. Dazu gehört das festlegen von:

- Metriken (*vgl. UC 1*)
- zugelassenen Lizenzen (*vgl. UC 3*)
- Quellen für (*vgl. UC 4*)
- Qualitätsattribute und Grenzwerte für (*vgl. UC 5*)
- Parameter für (*vgl. UC 7*)
- Grenzwerte (*vgl. UC8*)

Des Weiteren können Dependencies oder spezifische Versionen davon komplett von der Analyse ausgeschlossen werden.

Akteur	Anwender
Trigger	Der Anwender ändert das Verhalten des Plugins.
Hauptszenario	1. Der Anwender öffnet die Einstellungen des Plugins 2. Der Anwender ändert die Konfiguration basierend auf den vorhandenen Parametern
Erweiterungen	1a. Der Anwender öffnet die Konfigurationsdatei des Plugins
User Stories	• US 2.1: Pakettiefe eingrenzen

UC 3: Lizenzdetails anzeigen

Kurzbeschreibung

Der Benutzer sieht alle Lizenzen, welche in der Dependency vorhanden sind.

Akteur	Anwender
Trigger	Der Anwender betrachtet detaillierte Informationen zu einer Dependency an.

Hauptszenario

1. Der Anwender wählt eine Dependency in der Dependency-Baumstrukur aus
2. Eine Detailansicht öffnet sich
3. Der Anwender sieht die Lizenzen der Dependency. Ausserdem sieht er in der durch die Dependency-Baumstruktur, ob die Lizenz der Dependecy mit deren der Subdependencies kompatibel ist.

Erweiterungen

4. Der Anwender wählt eine Lizenz aus
5. Der Lizenztext und weitere Informationen (Open Source, Verbreitung usw.) werden angezeigt

User Stories

- US 3.1: License Tree
- US 3.2: Lizenzdetails

UC 4: Sicherheitslücken anzeigen

Kurzbeschreibung
Der Benutzer sieht alle Sicherheitslücken (*en. = vulnerabilities*), welche in einer Dependency vorhanden sind.

Akteur	Anwender
Trigger	Der Anwender betrachtet detaillierte Informationen zu einer Dependency an.

Hauptszenario

1. Der Anwender wählt den Eintrag "weiter Informationen anzeigenäus.
2. Eine Detailansicht öffnet sich.
3. Der Anwender sieht, ob in der Dependency bekannte Sicherheitslücken vorhanden sind und wie schwerwiegend diese sind.

Erweiterungen

4. Der Anwender wählt auf eine bekannte Sicherheitslücke aus
5. Die Webseite, auf welcher die Sicherheitslücke publiziert wurde, wird geöffnet

User Stories

- US 4.1: Zusammenfassung des Sicherheitszustandes
- US 4.2: Verlinkung zu einer Detailbeschreibung

UC 5: Zustand der Softwarequalität und Wartung visualisieren

Kurzbeschreibung

Der Anwender sieht den Zustand der Softwarequalität der Dependency. Die Qualität setzt sich aus verschiedenen Attributen zusammen, welche voreingestellt sind oder selbst festgelegt werden können (*vgl. UC 2*). Zur Qualität gehört auch die Wartung (Menge, Häufigkeit usw.) der Software. Jedem Attribut sind drei mögliche Zustände zugeordnet, dessen Grenzwerte auch eingestellt werden können.

Akteur	Anwender
Trigger	Der Anwender betrachtet detaillierte Informationen zu einer Dependency an.
Hauptszenario	1. Der Anwender klickt auf "weiter Informationen anzeigen" 2. Eine Detailansicht öffnet sich 3. Die Qualitätszustand der festgelegten Attribute wird analog zu *UC 1* angezeigt

UC 6: Trendanalyse der Dependencies visualisieren

Kurzbeschreibung

Der Anwender sieht den Zustand der Dependency als Trend über mehrere Versionen. Die Bewertung des Zustandes geschieht analog zu *UC 1*.

Akteur	Anwender
Trigger	Der Anwender betrachtet detaillierte Informationen zu einer Dependency an.
Hauptszenario	1. Der Anwender klickt auf "weiter Informationen anzeigen" 2. Eine Detailansicht öffnet sich 3. Der Anwender sieht den Zustandsverlauf der Dependency.
Erweiterungen	4. Der Anwender filtert nach spezifischen Versionen 4a. Der Zustandsverlauf wird basierend auf den Einstellungen dargestellt

UC 7: Report generieren

Kurzbeschreibung
Der Anwender generiert einen Report, auf dem alle Dependencies und deren Zustand ersichtlich ist. Welche Eigenschaften in welcher Tiefe im Report berücksichtigt werden, ist konfigurierbar. Das Format und der Speicherort des Reports kann eingestellt werden.

Akteur	Anwender
Trigger	Der Anwender erstellt einen Report.
Hauptszenario	1. Der Anwender wählt 'Report generieren' aus 2. Der Report wird generiert und im Dateisystem des Anwenders gespeichert
Erweiterungen	1a. Der Anwender führt den Befehl 'Report generieren' aus 2a. Der Report wird direkt in einen vom Anwender verknüpften Cloudspeicher (OneDrive, Google Drive, Dropbox) geladen

UC 8: Integration in CI-Pipeline

Kurzbeschreibung
Der Anwender möchte sich während der Entwicklung nicht direkt um die Qualität der Dependencies kümmern. Das Repository / Projekt enthält aber Qualitätsrichtlinien, welche eingehalten werden müssen. Durch eine CI-Pipeline kann die automatische Überprüfung des `package.json` mit den Dependencies erzwungen werden. Ein Grenzwert soll kritische Dependencies erkennen und die CI-Pipeline stoppen. Dieses Vorgehen erlaubt auch eine hohe Qualität während des ganzen Entwicklungszeitraums.

Akteur	Anwender
Trigger	Der Anwender löst die CI-Pipeline aus (z. B. durch `git push`).
Hauptszenario	1. Der Anwender committed seine neu erstellte oder angepasste package.json in sein Repository. 2. Die CI-Pipeline prüft über eine vorkonfigurierte Schnittstelle die Dependency Metriken der gefundenen `package.json` im jeweiligen Node-Projektordner des Repositories. 3. Die CI-Pipeline schlägt fehlt wenn die Dependencies einen Grenzwert überschritten haben.

Erweiterungen

1. Der Anwender erstellt einen Pull-Request
2. Es wird das komplette Repository abgesucht nach `package.json` Dateien.

6.3 Nichtfunktionale Anforderungen

Zum besseren Verständnis wird das FURPS-Modell, welches im Modul «Application Architecture» vorgestellt wurde, für die Gruppierung der nichtfunktionalen Anforderungen verwendet. FURPS ist ein Akronym und fasst verschiedene Qualitätsmerkmale von Software zusammen. [1]

- **F**unctionality (Kapazität, Wiederverwendbarkeit, Sicherheit)
- **U**sability (Erlernbarkeit/Bedienbarkeit, Verständlichkeit)
- **R**eliability (Verfügbarkeit, Fehlertoleranz, Wiederherstellbarkeit)
- **P**erformance (Geschwindigkeit, Ressourcenverwendung, Skalierbarkeit)
- **S**upportability (Wartbarkeit, Erweiterbarkeit, Testbarkeit, Installierbarkeit)

6.3.1 Berücksichtigte Anforderungen

1. (F/S) Der Core-Service soll separat von der Erweiterung verwendbar sein. Dies ermöglicht deren Wiedervendbarkeit für CLI Tools, Web API, oder Plugins für andere IDEs. Somit dürfen keine Visual Studio Code (VS-Code) Dependencies in den Dependencies des Core-Services verwendet werden.
   Akzeptanzkriterium: Architektur reflektiert Separation, wird überprüft in Architektur-Review. VS-Code Dependencies sind nur in den Dependencies der Erweiterung vorhanden, wird überprüft in Code-Reviews.

2. (R) Die Erweiterung soll ohne Interferenzen mit anderen installierten Erweiterungen nutzbar sein.
   Akzeptanzkriterium: Wird überprüft anhand Systemtests und Sichergestellt durch die Verwendung unabhängiger VSC-Komponenten.

3. (U) Das Farbschema berücksichtigt individuelle Farbthemen. Fehlermeldungen sind verständlich und werden dem Anwender unter Berücksichtigung des Farbthemas angezeigt.
   Akzeptanzkriterium: Es werden für VSC-Komponenten keine hard-codierten Farbwerte verwendet, sondern nur bereits vordefiniere VS Code CSS-sFarbvariablen. Wird überprüft in Code-Reviews.

4. (S) Die Erweiterung ist auf den drei Betriebssystemen macOS, Windows und Linux nutzbar.
   Akzeptanzkriterium: Integration Tests auf allen drei Betriebssysteme, Verwendung von unterschiedlichen Betriebssysteme während der Entwicklung

5. (S) Zur Nachvollziehbarkeit von Fehlern und wichtigen Operationen werden diese von der Erweiterung in einen Output-Channel geschrieben und bei schwerwiegenden

Fehlern wird der Benutzer benachrichtigt.
Akzeptanzkriterium: Keine Verwendung von `console` in der Erweiterung, sondern Verwendung eines Loggers mit Output-Channels und Notifications für schwerwiegende Fehler. Wird überprüft in Code-Reviews.

6. (S) Alle Funktionen der Erweiterung können ohne privilegierte Nutzer-Rechte verwendet werden. Die Erweiterung benötigt somit keine Administratoren- (Windows), bzw. Super-User-Rechte (macOS / Linux).
   Akzeptanzkriterium: Integration-Tests durchlaufen ohne privilegierte Rechte.

7. (P) Es muss für den Benutzer möglich sein, den Umfang der Analyse zu steuern. Das Abfragen von Metriken und deren Berechnung ist rechenintensiv, test- und messbar, jedoch stark von dem Betriebssystem und dem Gerät abhängig auf welchem die Erweiterung läuft.
   Akzeptanzkriterium: Konfigurierbarkeit vorhanden (UC 2). Analyse von Referenz-Package auf Referenz-System mit einer Analyse-Tiefe von 3 Stufen durchläuft in unter 5 Sekunden.

   - Referenz-System: Windows, Intel i5 CPU mit 4 Cores, 16 GB Arbeitsspeicher
   - Referenz-Package: koa (23 Dependencies, 11 Dev-Dependencies)

6.3.2 Nicht berücksichtigt Anforderungen

Die folgenden Nichtfunktionalen Anforderungen wurden erkannt, werden aber nicht umgesetzt. Dabei handelt es sich um Anforderungen, welche aufgrund der verfügbaren Zeit im Rahmen der Semesterarbeit zu aufwendig wären. Ob die Anforderung komplett ausgelassen wird, oder trotzdem in die Entwicklung einfliesst wird bei jedem dieser NFA einzeln erklärt.

1. (U) Die Verwendung des Plugins benötigt keine detaillierte Anleitung und wird (wo nötig) dem Anwender per Benachrichtigung oder sonstigem Hinweis direkt mitgeteilt.
   Abgrenzung: Wird nicht überprüft, aber die grundlegende Funktionen werden im Readme / dem Marketplace-Eintrag der Erweiterung ausgelegt.

2. (F) Die Erweiterung soll ohne zusätzliche Authentifizierung benutzbar sein. Die Erweiterung soll somit direkt nach Installation vollumfänglich nutzbar sein.
   Abgrenzung: Wird während der Entwicklung berücksichtigt, aber nicht explizit getestet.

3. (U) Es werden aussagekräftige Farben und Formen verwendet um den Zustand einer Dependency eindeutig zu Kennzeichnen.
   Abgrenzung: Befragung von Benutzern in Usability Tests

4. (R) Fehlerhafte Eingaben führen nicht zu inkonsistenten Zuständen oder Programmabstürzen.
   Abgrenzung: Wird nicht explizit getestet, aber Error-Handling und Umgang mit invaliden Daten ist Teil von Unit- und Integration-Tests.

5. (P) Für Daten, welche über das Netzwerk übertragen werden, wird eine ressourcenschonender Ansatz gewählt. Mit skalierbarem Cachen und "Bulk Calls"wird ein effizientes Datenmanagement ermöglichen.

Abgrenzung: Die Effizienz der Netzwerkanfragen wird bei der Entwicklung berücksichtigt und fliesst in den berücksichtigten NFA 7 ein.

6. (U) Das Plugin ist mehrsprachig verwendbar, wobei zu Beginn nur Deutsch und Englisch unterstützt werden. Nach dem Umstellen sind alle Texte in der entsprechenden Sprache angezeigt.
   Abgrenzung: Mehrsprachigkeit hat grosse Aufwände zur Folge, gerade mit Hinsicht die Übersetzung der Resultate von Drittparteien. Aus zeitlichen Gründen wird deshalb darauf verzichtet und Englisch als einzige Sprache verwendet.

Kapitel 7

Domänenanalyse

In diesem Kapitel wird die Domäne des zu entwickelnden Systems analysiert, die Hauptkonzepte dokumentiert sowie deren Zusammenhänge aufgezeigt.

7.1 Konzepte

Paketmanager

Um Software-Pakete einheitlich, automatisch und deterministisch zu installieren, werden Paketmanager eingesetzt. Typischerweise ermöglichen sie es, neue Pakete zu installieren, auf neuere Versionen zu aktualisieren oder nicht mehr benötigte Pakete zu entfernen.

Die zwei beliebtesten Paketmanager für Node.js-Pakete sind npm[1] und Yarn[2].

npm Der Node Package Manager (npm), ursprünglich 2010 veröffentlicht, wird automatisch mit Node.js mitinstalliert und ist demenstprechend weit verbreitet.

Yarn Yarn, ursprünglich von Facebook im Jahr 2016 veröffentlicht, wurde mit der Absicht erstellt, die Leistungs- und Sicherheitsmängel von npm (zu dieser Zeit) zu beheben.

[1] https://www.npmjs.com
[2] https://yarnpkg.com

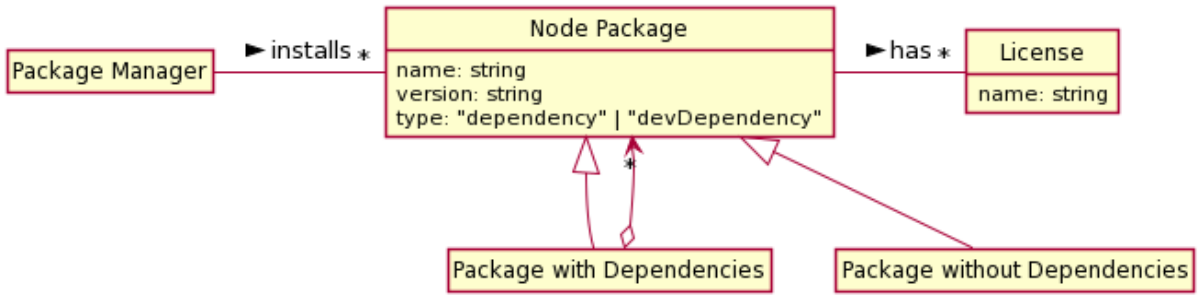

Abbildung 7.1: Aufbau von Node.js Paketen (hier als Komposition)

package.json

Jedes Paket, welches in der npm-Registry publiziert wird, muss eine «package.json»-Datei beinhalten. Diese Datei beinhaltet unter anderem die aktuelle Versionen des Paketes auf und listet alle seine Abhängigkeiten auf.[8]

Abhängigkeiten

Damit Abhängigkeiten eindeutig identifizierbar sind, wird im «package.json» ihre Version mitangegeben. Dabei kann entweder eine fixe Version (z. B. «1.3.4») referenziert oder aber ein Versionsselektor verwendet werden. Mittels Versionsselektoren lässt sich ein Bereich spezifizieren, welcher vom Paketmanager bei der Installation zu einer konkreten Version auflöst wird.
Beispiele für Versionsselektor können sein: die neuste Version eines Paketes («latest»), eines Major-Releases («1.x.x»), eines Minor-Releases (« 1.1.x») usw.

Bei den Abhängigkeiten werden zusätzlich zwischen zwei Typen unterschieden: «devDependencies», welche nur während der Entwicklung, und «dependencies», welche während der Laufzeit des Paketes benötigt werden.[9]

Da die Abhängigkeiten rekursiv sind, kann ein beliebig tiefer Abhängigkeitsbaum entstehen.

Registry

Ein Package kann, muss aber nicht in einer Registry registriert sein, um es öffentlich verfügbar zu machen. Eine Registry kann Metadaten – etwa ein Versionshistorie oder Download-Statistiken – zur Verfügung stellen. Ein Node.js-Paket kann seine Abhängigkeiten von verschiedenen Quellen beziehen, also z. B. auch von einem (Git-)Repository.

Lizenzen

Damit klar ist, wie ein Paket verwendet werden darf, sollte im «package.json» eine Lizenz angegeben werden. Dies kann durch die Angabe des SPDX-Bezeichner[3] oder durch den Verweis auf eine Lizenzdatei – z. B. für eine eigene Lizenzen – erfolgen. Die Angabe von mehreren Lizenzen ist möglich.[9]

Metadaten

Neben der Metadaten der Registries gibt es weitere Instanzen, welche zusätzliche Informationen wie Security-Probleme und Metriken zu Packages anbieten. Diese Metrics dieser Metrics-Provider können sich auf ein Paket generell, oder auch auf eine spezifische Version beziehen.

[3] https://spdx.org/licenses

Kapitel 8

Softwarearchitektur und -design

Um die Systemkomponenten sinnvoll strukturieren und somit die vorgängig erhobenen Anforderungen an das System sauber und effizient umsetzen zu können, ist eine adäquate Softwarearchitektur notwendig. Diese wird im folgenden Kapitel beschrieben, beginnend mit einer Übersicht der fundamentalen Entscheidungen und Lösungsstrategien, die die Architektur des Systems prägen. Danach folgt eine Übersicht über die externen Schnittstellen des Systemes. Schlussendlich wird die statische Zerlegung des Systemes in Building Block und die daraus resultierende Dateistruktur dargestellt und das Deployment der Applikation beschrieben.[27][28][29]

8.1 Lösungsstrategie

1. **TypeScript**: Um sauberen, lesbaren und weniger fehleranfälligen Code zu schreiben, wird die typensichere Programmiersprache TypeScript verwendet.
2. **Schichtenarchitektur**: Um die Komplexität zu reduzieren und eine geringe Koppelung bei gleichzeitig hoher Kohäsion zu erreichen, wird die Software in drei logische Layer strukturiert: *Presentation*, *Business Logic* und *Data Access*.
3. **Code-Aufteilung**: Um die Software nicht nur als Erweiterung in Visual Studio Code, sondern auch in anderen Applikationen verwenden zu können, wird das System in zwei Komponenten aufgeteilt: *Core* und *Visual Studio Code Extension*.
4. **Monorepo**: Um die Entwicklung trotz aufgeteilter Codebasis angenehm zu gestalten, werden die zwei Komponenten im selben (Git-)Repository (ein sogenanntes ≪Monorepo≫) verwaltet. Als Hilfsmittel wird Lerna, ein Tool zur Verwaltung von JavaScript-Projekten mit mehreren Paketen, verwendet.[1]
5. **Fassade**: Um die interne Struktur des Cores zu verstecken und gegen aussen eine vereinfachte Schnittstelle anzubieten, werden die verwendbaren Services via Fassade offengelegt.
6. **DAOs und Business Models**: Für den Austausch von Daten zwischen Business Logic und Data Access werden Data Access Objects (DAOs) verwendet. Für den

[1] https://github.com/lerna/lerna

Austausch zwischen Presentation und Business Logic werden Business Models verwendet. So können die Datenmodelle im entsprechenden Layer einfach angepasst werden, bspw. falls ein externes System seine Datenstruktur verändert, die Business Models aber gleich bleiben sollen.

7. **Respository**: Um die Informationen, welche für den Zugriff auf eine Datenquelle notwendig sind, vom restlichen Code zu enkapsulieren, wird das Repository-Pattern angewendet. Pro Datenquelle (z. B. API, Dateisystem etc.) wird ein eigenes Repository erstellt. Dieses enthält Angaben zu den URLs und definiert die DAOs. Repositories sind im Data Access Layer angesiedelt.[13]

8.2 Systemkontext

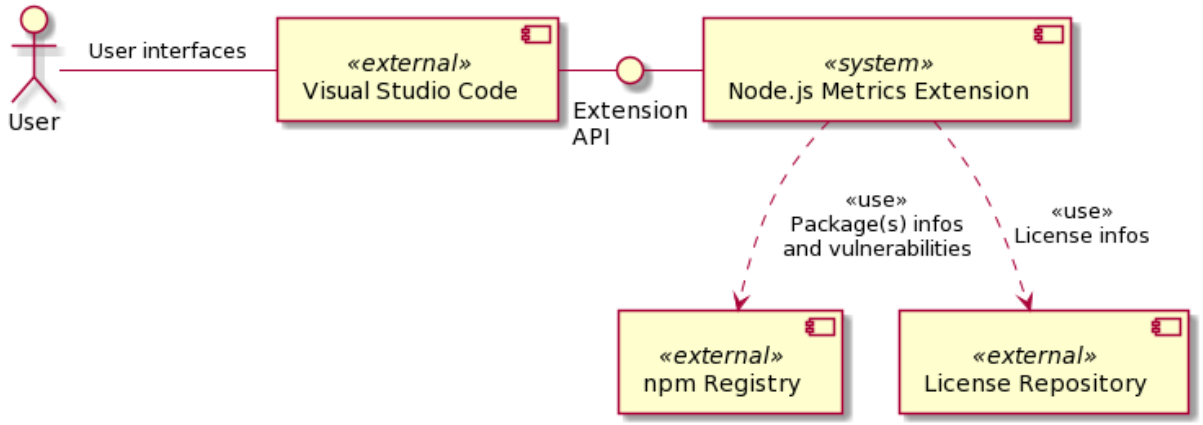

Abbildung 8.1: Systemkontext

System	Beschreibung
Visual Studio Code	Laufzeitumgebung, in welcher die Node.js Metrics Extension ausgeführt wird. Die Kommunikation erfolgt über die Visual Studio Code Extension API.
npm Registry	Beinhaltet Daten und Metadaten zu Paketen und stellt Informationen zu Sicherheitsschwachstellen zur Verfügung.
License Repository	Enthält die gängigsten Lizenzen und ermöglicht die strukturierte Abfrage von dessen Informationen.

8.3 Building Block View

8.3.1 Level 1

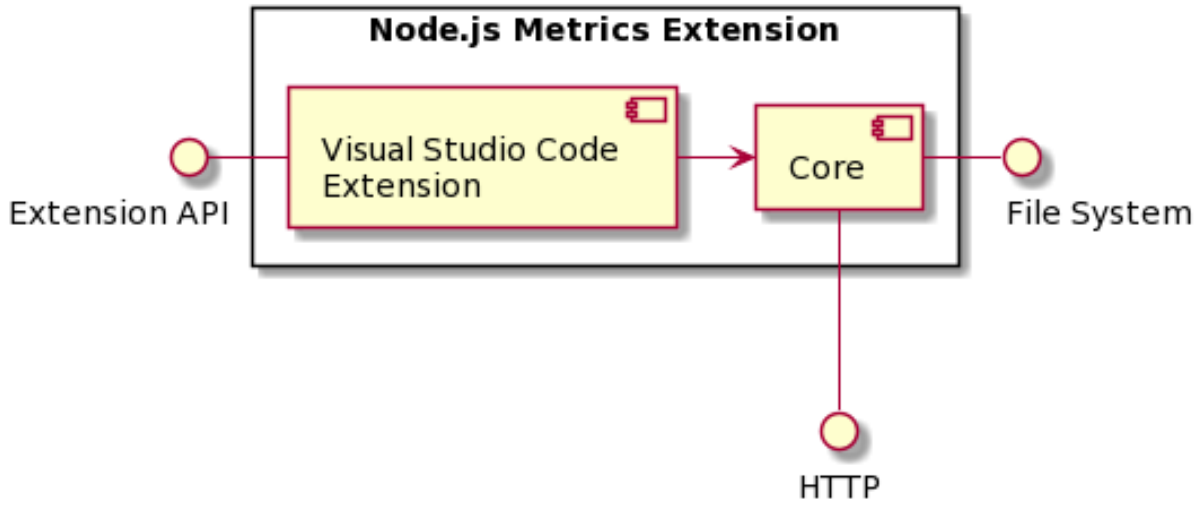

Abbildung 8.2: Building Block View – Level 1

Komponente	Zuständigkeit
Core	Listet die installierten Pakete auf, sammelt Daten darüber, berechnet Metriken und stellt sie den anderen Komponenten zur Verfügung. Beinhaltet die Layer *Business Logic* und *Data Access*.
Visual Studio Code Extension	Visualisiert die durch den Core bereitgestellten Daten und abstrahiert die Extension API von Visual Studio Code. Beinhaltet den *Presentation* Layer.

8.3.2 Level 2

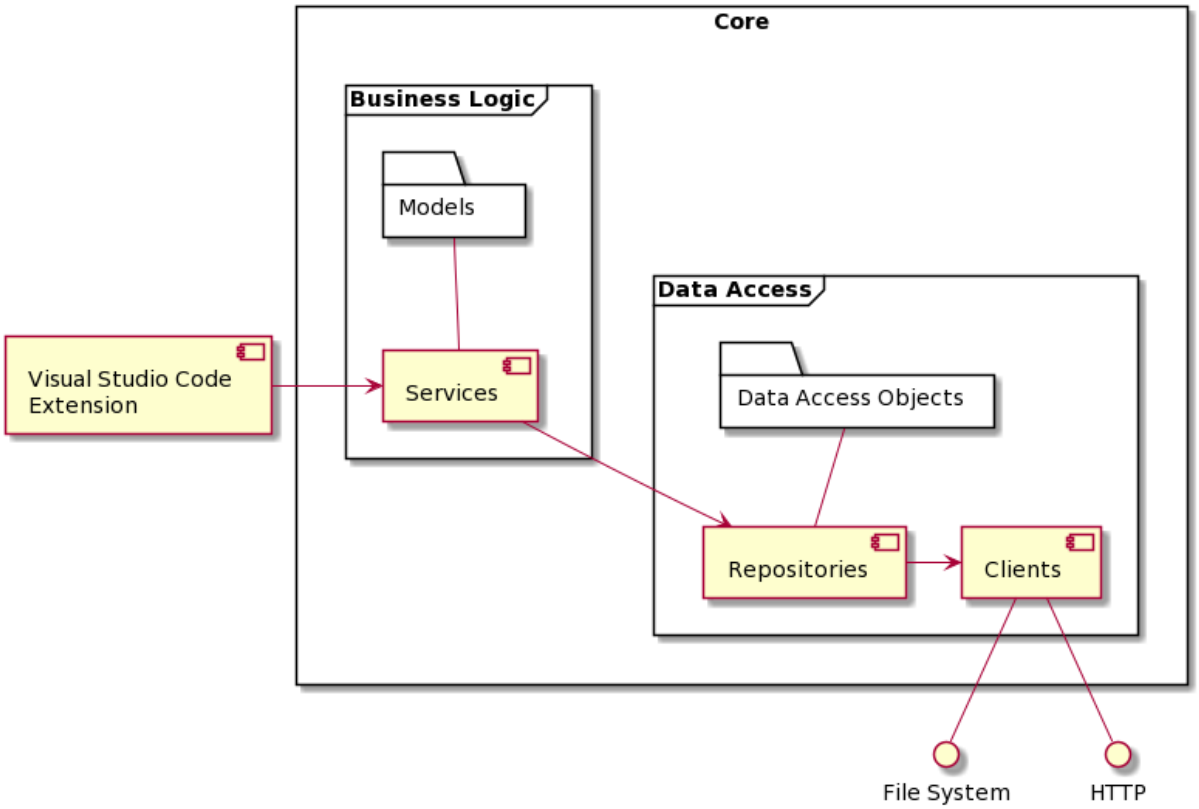

Abbildung 8.3: Building Block View – Level 2

Komponente	Zuständigkeit
Services	Rufen Daten aus den Repositories ab und stellen diese in Form von Models der Presentation zur Verfügung.
Repositories	Greifen via Clients auf eine Datenquelle zu und geben die Daten in Form von Data Access Objects zurück.
Clients	Führen HTTP-Requests oder Zugriffe auf das Dateisystem aus.

8.4 Dateistruktur

Aus den oben getroffenen Architekturentscheidungen ergibt sich die folgende Dateistruktur für das Monorepo:

```
node-package-metrics/
  packages/
      package-metrics/
      vscode-package-metrics/
  tsconfig.build.json
  tsconfig.json
```

Der Core wird im Paket `package-metrics`, die Visual Studio Code Extension im Pa-

ket `vscode-package-metrics` umgesetzt. Die beiden Dateien `tsconfig.build.json` und `tsconfig.json` enthalten Kompilieroptionen, wobei erstere für den Compiler und letztere vom Editor verwendet werden.

`package-metrics` und `vscode-package-metrics`

Die beiden Pakete `package-metrics` und `vscode-package-metrics` haben eine ähnliche interne Struktur. Der Quellcode ist im Verzeichnis `src/` untergebracht, wo auch die Tests abgelegt sind. `assets/` enthält statische Dateien wie die Lizenzen (im JSON-Format) oder Bilder.

```
src/
  assets/
  test/
  ...
tsconfig.build.json
```

`tsconfig.build.json` enthält paketspezifische Kompilieroptionen. Das Verzeichnis `lib/` ist nur im Paket `package-metrics` vorhanden.

8.4.1 Deployment

Extension

Die Extension wird auf dem Visual Studio Marketplace[2] unter dem Namen Node.js Package Metrics[3] publiziert. Durch die Publikation ist die Extension sowohl auf der Marketplace-Website, wie auch in der Extension-Suche innerhalb von Visual Studio Code auffind- und installierbar.

Core

Da es sich bei dem `package-metrics` Paket um ein eigenständiges Modul handelt, kann dieses unabhängig von der Extension verwendet werden. Weiter erlaubt der Publikationsprozess von Extensions zurzeit keine lokal verlinkte Dependencies. Der Core wird deshalb auch auf der npm-Registry unter dem Namen package-metrics[4] publiziert.

Publizierungsablauf

Die beiden Packages können unabhängig voneinander publiziert werden. Bei Änderungen am Core müssen diese jedoch zuerst auf npm publiziert werden, damit diese anschliessend mit einer neuen Version der Extension publiziert werden können. Während der Prozess der Publikation automatisierbar wäre, wird dieser zur Zeit manuell ausgeführt.

Core Neue Versionen des `package-metrics`-Pakets können direkt über die Command-Line publiziert werden. Voraussetzung dafür ist, dass sich der publizierende Entwickler als «node-package-metrics»-Benutzer angemeldet hat.

[2] https://marketplace.visualstudio.com/vscode

[3] https://marketplace.visualstudio.com/items?itemName=NodePackageMetrics.vscode-package-metrics

[4] https://www.npmjs.com/package/package-metrics

```
# Auszufuehren im Root-Ordner des Pakets (package-metrics)
# Anmeldung bei npm
npm login
# Publikation des Pakets
npm publish
```

Extension Zur Publikation der Extension wird der von Visual Studio Code zur Verfügung gestellte Visual Studio Code Extension Manager[5] verwendet. Dieses muss als globales npm-Paket installiert werden. Um neue Versionen auf dem Marketplace publizieren zu können, muss sich der publizierende Entwickler über einen Personal Access Token als Publisher authentifizieren. Da der Publikationsprozess nicht mit lokal verlinkten Dependencies funktioniert, muss das Core-Paket vor der Publikation durch die sich auf der npm-Registry befindenden Version ersetzt werden.

```
# Auszufuehren im Root-Ordner der Extension (vscode-package-metrics)
# Installation des Extension Managers
npm install -g vsce
# Anmeldung als Publisher
vsce login node-package-metrics
# Installation der Dependencies (ersetzt lokale Version
# von package-metrics durch die auf npm publizierte)
npm install
# Publikation einer neuen Version der Extension
vsce publish
```

Accounts

Komponente	Plattform	Benutzername	Profil
Extension	Visual Studio Marketplace	«Node Package Metrics»	Link[6]
Core	npm-Registry	«node-package-metrics»	Link[7]

[5] https://www.npmjs.com/package/vsce
[6] https://marketplace.visualstudio.com/publishers/NodePackageMetrics
[7] https://www.npmjs.com/~node-package-metrics

Glossar

API Application Programming Interface. 9, 13, 17, 40, 50, 56, 70, 72–75, *siehe* Application Programming Interface

Application Programming Interface Ein Programmierschnittstelle (von englisch *Application Programming Interface*) ist eine digitale Schnittstelle zwischen zwei Softwaresystemen und erlaubt deren Anbindung zueinander. 9, 61

Backlog «[Der Produkt-]Backlog ist eine dynamische Liste von durchzuführenden Arbeitsaufträgen. Die Arbeitsaufträge können z.B. in Form von Anforderungen, User Storys oder Funktionsbeschreibungen vorliegen. "Backlog" bedeutet wörtlich übersetzt "Arbeitsrückstand" bzw. "Auftragsbestand"»[3]. 31

BAG Bundesamt für Gesundheit. 38

Building Block Architekturbaustein (Module, Komponenten, Subsysteme, Klassen, Schnittstellen, Pakete, Bibliotheken, Frameworks, Schichten, Partitionen, Tiers, Funktionen, Makros, Operationen, Datenstrukturen, etc.[27]. 55

CLI Command-Line Interface. 8, 12, 73

ClickUp Arbeitsverwaltungstool, siehe `https://clickup.com`. 31

Cloud-Dienst «Internetdienst, der Rechen- oder Speicherkapazität anbietet, auf die über ein Netzwerk (z. B. das Internet) zugegriffen wird»[11]. 39

CPE Common Platform Enumeration. 72

CVE Common Vulnerability and Exposure. 72, 74

CWE Common Weakness Enumeration. 72

IFS Institut für Software. 29

NIST National Institute of Standards und Technology. 64

Notion Notizapplikation, siehe `https://www.notion.so`. 38

npm Node Package Manager. 8, 53, 73

NVD National Vulnerability Database. 72

Open Web Application Security Project Eine Non-Profit-Organisation mit dem Ziel, die Sicherheit von Anwendungen und Diensten im World Wide Web zu verbessern.

Durch Schaffung von Transparenz sollen Endanwender und Organisationen fundierte Entscheidungen über wirkliche Sicherheitsrisiken in Software treffen können. 62, 64

Overleaf Web-basierter LaTeX-Editor, siehe `https://www.overleaf.com`. 38

OWASP Open Web Application Security Project. 64, 72, *siehe* Open Web Application Security Project

REST Representational State Transfer. 73

Sprint Sich wiederholende Arbeitsabschnitte von fixer Dauer (typischerweise eine oder zwei Wochen), während deren die Software weiterentwickelt wird. Der Umfang dieser Weiterentwicklung, auch Inkrement genannt, wird in der jeweiligen Sprint-Planung festgelegt. 30

Story Splitting Eine Technik zum Aufteilen von (zu) grossen User Storys. 44

UP Unified Process. 10, 29, 30

URI Uniform Resource Identifiers. 72

Usability Test . 9

User Story «User Storys sind Entwicklungsaufgaben, die oft als "Kundentyp + Anforderung + Zweck" formuliert werden.»[**user-story**] plural. 31, 44, 62

Webservice Eine Dienstleistung, welche über Rechennetzte konsumiert werden kann.. 10, 13

Literaturverzeichnis

[1] Peter Eeles (IBM). *Capturing Architectural Requirements*. URL: https://www.ibm.com/developerworks/rational/library/4706.html. (besucht am 6.10.2020).

[2] André Duarte André Cruzavatar. *About npms*. URL: https://npms.io/about. (besucht am 15.12.2020).

[3] Dr. Georg Angermeier. *Backlog*. URL: https://www.projektmagazin.de/glossarterm/backlog. (besucht am 18.12.2020).

[4] The MITRE Corporation. *Common Vulnerabilities and Exposures*. URL: https://cve.mitre.org. (besucht am 9.11.2020).

[5] The MITRE Corporation. *Common Weakness Enumeration*. URL: https://cwe.mitre.org. (besucht am 9.11.2020).

[6] Module Counts. *Module Counts*. URL: http://www.modulecounts.com/. (besucht am 12.12.2020).

[7] GitHub Docs. *GitHub flow*. URL: https://docs.github.com/en/free-pro-team@latest/github/collaborating-with-issues-and-pull-requests/github-flow. (besucht am 28.09.2020).

[8] npm Docs. *About packages and modules*. URL: https://docs.npmjs.com/about-packages-and-modules. (besucht am 10.12.2020).

[9] npm Docs. *package.json*. URL: https://docs.npmjs.com/cli/v6/configuring-npm/package-json. (besucht am 10.12.2020).

[10] npm Docs. *The JavaScript Package Registry*. URL: https://docs.npmjs.com/cli/v6/using-npm/registry. (besucht am 18.12.2020).

[11] Duden. *Cloud-Dienst*. URL: https://www.duden.de/rechtschreibung/Cloud_Dienst. (besucht am 9.11.2020).

[12] OpenJS Foundation. *Node.js*. URL: https://nodejs.org. (besucht am 14.12.2020).

[13] Martin Fowler. *Repository*. URL: https://martinfowler.com/eaaCatalog/repository.html. (besucht am 15.12.2020).

[14] Silvan Gehrig. *Aufgabenstellung Semesterarbeit zum Thema «Node.js Package Metrics»*.

[15] Stefan Hagen. *Projektstrukturplan (PSP) – Plan der Pläne (2)*. URL: https://pm-blog.com/2010/07/19/projektstrukturplan-psp-plan-der-plane-2. (besucht am 22.09.2020).

[16] "IEEE Standard for a Software Quality Metrics Methodology". In: *IEEE Std 1061-1998* (Dez. 1998). DOI: 10.1109/IEEESTD.1998.243394.

[17] Jeff Sutherland Ken Schwaber. *The 2020 Scrum Guide™*. URL: https://www.scrumguides.org/scrum-guide.html. (besucht am 15.12.2020).

[18] Microsoft. *Extending Workbench*. URL: https://code.visualstudio.com/api/extension-capabilities/extending-workbench. (besucht am 18.12.2020).

[19] National Institute of Standards und Technology (NIST). *National Vulnerability Database*. URL: https://nvd.nist.gov. (besucht am 9.11.2020).
[20] National Institute of Standards und Technology (NIST). *Official Common Platform Enumeration (CPE) Dictionary*. URL: https://nvd.nist.gov/products/cpe. (besucht am 9.11.2020).
[21] Inc. npm. *About npm*. URL: https://www.npmjs.com/about. (besucht am 14.12.2020).
[22] Renate Oettinger. *Was ist eine Softwarelizenz?* URL: https://www.computerwoche.de/a/was-ist-eine-softwarelizenz,1913465. (besucht am 18.12.2020).
[23] Opensource.org. *Announcing the Open Source License API*. URL: https://opensource.org/node/822. (besucht am 18.12.2020).
[24] Opensource.org. *Licenses Standards*. URL: https://opensource.org/licenses. (besucht am 18.12.2020).
[25] Open Web Application Security Project (OWASP). *OWASP Top Ten*. URL: https://owasp.org/www-project-top-ten. (besucht am 8.11.2020).
[26] PYPL. *PYPL PopularitY of Programming Languages*. URL: https://pypl.github.io/PYPL.html. (besucht am 12.12.2020).
[27] Gernot Starke. *Building Block View*. URL: https://docs.arc42.org/section-5. (besucht am 15.12.2020).
[28] Gernot Starke. *Context and scope*. URL: https://docs.arc42.org/section-3. (besucht am 15.12.2020).
[29] Gernot Starke. *Solution Strategy*. URL: https://docs.arc42.org/section-4. (besucht am 15.12.2020).

Abbildungsverzeichnis

Anhang A

Anhang

A.1 Personas

Marina Tüftulus

«In meiner Freizeit bastle ich passioniert an meiner Heimautomation.»

Alter 21 Jahre

Beruf Lehre als Betriebsinformatikerin

Persönlichkeit einfallsreich, neugierig, technisch affin

Interessen
- Technik
- Open Source
- Smart Home / Heimautomatisierung
- Reisen

Ziele
- Vollautomatisches Zuhause
- Vertieftes Softwarewissen aneignen
- Einen Job bei Google, Facebook oder Co.

Frustrationen
- Angst, gehackt zu werden: in den Medien liest Sie immer wieder von Sicherheitsschwachstellen bei Smart Home Produkten
- Ihre Programme funktionieren nach Updates der externen SW-Pakete oft nicht mehr, da
  - deren Schnittstellen geändert haben
  - diese nicht mehr unterstützt werden

Biografie Nachdem Marina das Gymnasium abgeschlossen hatte, begann Sie ihre Lehre als Betriebsinformatikerin in einem kleinen KMU. Schon seit ihrer Jugend interessiert Sie sich für Technik und hat sich das Programmieren selbst beigebracht. Auf ihrem Raspberry Pi hat Marina schon verschiedene Projekte umgesetzt und dabei auch ab und zu zu Open Source Projekten beigetragen.
Marina liebt Smart Home Gadgets und hat zuhause auch bereits einen beachtliche Anzahl installiert: Türklingel, Audio, Thermostat, Licht… Grosse Teile ihres Zuhauses kann Sie via entsprechende Apps oder mit ihrem Sprachassistenten steuern. Seit neuem ist Marina sehr angetan von Web-Technologien. So kam ihr auch die Idee für ihr nächstes grosses Projekt: Ein Web-Dashboard, welches alle bereits vorhanden Smart Home Geräte einbinden kann, sodass Sie nicht für jedes davon eine einzelne App braucht.

Kurt Hässig

≪Softwarearchitektur im Grossunternehmen ist mit zu viel administrativem Aufwand verbunden.≫

Alter 46 Jahre

Beruf Softwarearchitekt in einem Grossunternehmen

Persönlichkeit gewissenhaft, empfänglich, vernünftig

Interessen
- Technik
- Biken
- Familie

Ziele
- Entwicklung von Webtechnologien voranbringen
- Junge Entwickelnde ins Team holen

Frustrationen
- Überstunden vor Meilensteinende: (zu) viel administrativer Aufwand.
- Viele Rückfragen von Kollegen zu externen Software-Paketen (≪Dürfen wir dieses Paket brauchen?≫).
- Überprüfung der externen Software-Pakete ist mühsam und langwierig:
  - Lizenz
  - Qualität
  - Subpakete
- Pakete, welche die Sicherheitsinspektion nicht bestehen, müssen in letzter Minute ausgetauscht werden.

Biografie Nach seiner Ausbildung als Elektroniker studierte Kurt in den 90er-Jahren Informatik an einem Abendtechnikum. Danach begann er bei dem Grossunternehmen zu arbeiten und ist dort mittlerweile als Softwarearchitekt für ein grosses Team tätig.
Das Unternehmen wird noch sehr traditionell geführt, agile Entwicklung ist zwar in aller Munde, wird aber de facto nicht umgesetzt. Der Unternehmensprozess bei der Softwareentwicklung sieht vor, bei grösseren Meilensteine die externen Softwarekomponenten zu fixieren und einem Qualitätscheck zu unterziehen. Kurt muss dabei als technischer Verantwortlicher die Verantwortung und den administrativen Part übernehmen: Komponenten mit Version auflisten, Qualitätsreport erstellen und beglaubigen.
Kurt liest regelmässig Berichte von technischen Portalen und sieht den Mehrwert und die Zukunft von neuen Technologien. Im Unternehmen sind diese jedoch noch nicht sehr verbreitet: viele Mitarbeiter der Generation X sind eher konservativ eingestellt und misstrauen neuen Technologien. Sie – und auch das Management – sehen darin nur einen Mehraufwand. Als Folge davon ist das Unternehmen für junge Entwicklerinnen nicht sehr attraktiv.

A.2 Konkurrenzanalyse

Es gibt bereits verschiedene Tools und Metriken, um Pakete zu analysieren. In einem ersten Schritt sollen diese gesammelt werden. Die Struktur soll dabei einfach gehalten werden und mit wichtigen Kenndaten festgehalten werden. Um die Anforderungen an unsere Anwendung zu definieren, haben wir zuerst vorhandene Tools auf Basis ihrer Funktionalität analysiert.

npms.io Die Webseite npms.io liefert Repositores für Node Packages mit zusätzlichen Metriken zu Qualität, Wartung und Beliebtheit sowie einem Endergebnis über alle Metriken. Eine erwähnenswerte Komponente ist der npms Analyzer[1], welcher diese Informationen sammelt und das Ergebnis zu einem Node Package speichert. Detailierte Informationen können in der Architekturbeschreibung[2] nachgelesen werden.

npmjs.com Die Webseite der npm Registry [3] liefert Daten für jedes registrierte Package mit zusätzlichen Metriken zu Qualität, Wartung und Bekanntheit. Die Kriterien für die Metriken werden anhand des bereits erwähnten npms-Analyzers[4] erstellt. Des weiteren sind folgende Daten über die Webseite abrufbar:

- Anzahl Dependencies (Dependencies / Dev-Dependencies)
- Anzahl Versionen (aktuell und historisch)
- Anzahl Downloads
- Verwendete Lizenz
- Grösse / Total Dateien
- Bestehende Issues
- Bestehende Pull Requests
- Aktive Entwicklung (Veröffentlicht vor x Jahren)
- Mitwirkende Personen
- Öffentliches / privates Repository

Npm Audit Der Node Package Manager (npm) ist bei vielen JavaScript Entwicklern beliebt. Mit der Version 6 wurde npm um das Konsolen-Tool npm audit[5] ergänzt. Es ermöglicht eine Überprüfung aller Package Dependencies auf Sicherheitsschwachstellen, welche anhand der package.json Datei installiert wurden.

Das Tool kann in der Konsole mit dem Befehl `npm audit` direkt ausgeführt werden. Zusätzlich wird es bei jedem `npm install`-Befehl automatisch ausgeführt und das Ergebnis zusammengefasst auf der Konsole ausgegeben. Gefundene Sicherheitsschwachstellen werden zusätzlich zu Sicherheitshinweisen mit dem Schweregrad und dem Vorgehen zur Behebung der Schwachstelle auf der Konsole ausgegeben. Durch den Befehl `npm audit`

[1] https://npms.io/about
[2] https://github.com/npms-io/npms-analyzer/blob/master/docs/architecture.md
[3] https://npmsjs.com
[4] https://docs.npmjs.com/searching-for-and-choosing-packages-to-download
[5] https://docs.npmjs.com/cli/v6/commands/npm-audit

`fix` ist es möglich direkt auf eine aktuelle Version zu wechseln, bei der die gefundene Schwachstelle behoben wurde.

Dependency Analytics Die Dependency Analytics Visual Studio Code Erweiterung[6] basiert auf der Snyk Intel Vulnerability DB[7] und kann dadurch auf eine sehr umfangreiche Sicherheitsschwachstellen-Datenbank zurückgreifen. Snyk zeichnet sich durch eine gute Integration in Git-basierte Quellcodeverwaltung aus.

Die Erweiterung umfasst folgende Funktionalitäten:

- Generierung eines Vulnerability-Reports über Snyk anhand des `package.json`. Es werden weitere Formate anderer Programmiersprachen unterstützt.
- Der Report kann über die Statusmeldung, ein Icon oberhalb des Editors, oder via Rechtsklick im Kontextmenu der Datei erstellt werden.
- Der Report wird in einem neuen Tab dargestellt.
- Es wird beim Öffnen der `package.json`-Datei eine Statusmeldung der gefunden Schwachstellen eingeblendet.
- Fehler der Analyse werden über eine Statusmeldung und eine Ausgabe in der Konsole ersichtlich.

Sonatype Nexus IQ Die Sonatype Nexus IQ Visual Studio Code Erweiterung [8] hat Zugriff auf die OSS Index Datenbank[9]. Die Datenbank hilft Entwickelnden Schwachstellen zu identifizieren, die Risiken zu verstehen und dadurch die eigene Software möglichst sicher zu halten. Gefundene Schwachstellen werden in einer Liste mit der Paketbezeichnung sowie deren Version und der Farbe, gewichtet nach dem Schweregrad dargestellt. Detaillierte Informationen werden durch die Selektion eines Eintrags direkt in einem neuen Tab dargestellt. In diesem Tab sind auch Lizenzdetails ersichtlich.

Npm Package Quality Über die Website packagequality.com[10] kann die Qualität eines Pakets anhand einer Sucheingabe ermittelt werden. Die Suche ist nur über die Website aufrufbar, es wird keine API zur Verfügung gestellt. Es wird eine Gesamtbewertung mittels 5 Sternen dargestellt. Zusätzlich sind Bewertungen in Prozent zu folgenden Metriken aufgelistet:

- Package Qualität
- Offne Issues
- Langzeitig offene Issues
- Versions-Qualität
- Downloads-Qualität

[6] https://marketplace.visualstudio.com/items?itemName=redhat.fabric8-analytics
[7] https://snyk.io/product/vulnerability-database
[8] https://marketplace.visualstudio.com/items?itemName=SonatypeCommunity.vscode-iq-plugin
[9] https://ossindex.sonatype.org/
[10] https://packagequality.com

Der Algorithmus für die Gewichtung ist im zugehörigen Repository[11] beschrieben.

npq Das Node Paket npq[12] integriert sich in den Installationsprozess von Paketen indem es diese überprüft und falls ein gewisse Schwellenwerte verletzt werden eine Warnung anzeigt. Es werden die folgenden Metriken abgefragt:

- Vorhandene Schwachstellen in der Snyk Datenbank.
- Alter des Packages.
- Anzahl Downloads.
- Enthalten einer Readme-Datei.
- Enthalten einer Lizenz-Datei.
- Enthalten von pre-, oder post-Installations-Skripts.

Die Snyk-Datenbank wird nur abgefragt, wenn das Snyk-eigene Node-Paket installiert ist und der Benutzer damit authentifiziert ist. Ohne eine Snyk-Lizenz ist diese Funktion nicht verwendbar.

[11] https://github.com/alexfernandez/package-quality
[12] https://github.com/lirantal/npq

A.3 Evaluation Vulnerability Provider

Komponenten mit bekannten Schwachstellen

Die OWASP publiziert regelmässig einen Bericht mit den zehn aktuell kritischsten Sicherheitsrisiken in Web-Applikationen, die sogenannte «OWASP Top Ten». Auf Platz neun befindet sich das Risiko «Using Components with Known Vulnerabilities» (*dt. = Nutzung von Komponenten mit bekannten Schwachstellen*).[25]

Offizielle Datenbanken und Standards

Es gibt verschiedene offiziellen Datenbanken, welche bekannte Schwachstellen sammeln und katalogisieren. Nachfolgend soll aufgezeigt werden, welche dieser Datenbanken existieren und über eine API abgefragt werden können.

NVD Sehr bekannt ist die National Vulnerability Database (NVD). Die Datenbank wird von der US-Regierung unterhalten und von vielen Applikationen zur Analyse verwendet und referenziert. Die NVD enthält u.a. sicherheitsrelevante Softwarefehler, Fehlkonfigurationen und Referenzen zu Sicherheits-Checklisten.[19]

CVE Common Vulnerability and Exposure (CVE) (*dt. = Gemeinsame Schwachstellen und Enthüllungen*) ist ein Industriestandard, welcher öffentlich bekannte Cyber-Sicherheitslücken auflistet und jeden Eintrag mit einer eindeutigen Kennnummer versieht. Die CVE-Einträge werden in verschiedensten Cybersicherheitsprodukten und -Diensten verwendet. Mit CVE können Schwachstellen-Datenbanken, -Dienste und -Tools miteinander 'kommunizieren'. Bis zur Erschaffung der CVE im Jahr 1999 war es schwierig, effektiv zu entscheiden, welches Werkzeug für die Bedürfnisse einer Organisation am besten geeignet ist. Jeder Anbieter verwendete eine andere Nomenklatur für Schwachstellen oder Gefährdungen und verwendet unterschiedliche Metriken, um anzugeben, wie viele Schwachstellen oder Gefährdungen er prüft, oder testet. CVE stellt Anbietern eine Standardliste zur Verfügung, mit der sie vergleichen können, so dass sie 'Äpfel mit Äpfeln' vergleichen können. Software-Entwickler greifen regelmässig auf CVE-Datenbanken und deren Bewertungen zu, um das Risiko der Verwendung einer Komponente (Pakete und Binärdateien) in der eigenen Anwendung zu minimieren. Mögliche Schwachstellen müssen laufend überprüft werden. Die Verwendung von CVE ist kostenlos. Es kann die gesamte CVE-Liste durchsucht, referenziert und heruntergeladen werden.[4]

CPE Common Platform Enumeration (CPE) ist ein Industriestandard für die strukturierte Benennung von informationstechnischen Systemen, Software und Paketen. Basierend auf der generischen Syntax für Uniform Resource Identifiers (URI) enthält CPE ein formales Namensformat, eine Methode zur Überprüfung von Namen gegen ein System und ein Beschreibungsformat zur Bindung von Text und Tests an einen Namen. In Kombination mit CVE sollen Schwachstellen in Systemen eindeutig und vergleichbar bezeichnet werden können.[20]

CWE Common Weakness Enumeration (CWE) ist eine von der Community entwickelte Liste von Software- und Hardware-Schwächentypen. Sie dient als gemeinsame Sprache, Messlatte für Sicherheitswerkzeuge und Grundlage für die Identifizierung von Schwachstellen, deren Eindämmung und Prävention.[5]

npm-Registry API

npm Unter dem Namen npm-Registry bzw. npm Open Source wird ein Repository betrieben, über das Pakete unter einer freien Lizenz bereitgestellt werden. Für private Pakete wird eine kommerzielle Version angeboten.

registry Um Pakete nach Namen und Version aufzulösen, gibt es die öffentliche npm-Registry[13], welche die CommonJS [14] Paket Spezifikation zum Lesen von Paketinformationen implementiert. Zusätzlich zu diesen Metadaten beinhaltet die Registry zu jedem Paket eine Auflistung von bekannten Schwachstellen. Diese Daten werden aber in keiner offiziellen API Dokumentation beschrieben. Durch das Analysieren von öffentlich zugänglichen GitHub-Repositories und bekannter Sicherheits-Scanner ist es aber möglich, die Schnittstellen für Abfragen auszulesen.

npm-Audit npm-Audit ist der offizielle npm Scanner, welcher die bekannten Schwachstellen per Kommandozeilen-Befehl (CLI) aus der npm Registry API abfragen kann. Die Hauptidee hinter 'npm audit' besteht darin, den Entwicklern die potentiellen Probleme mit Abhängigkeiten bewusster zu machen. Das Audit-Modul arbeitet durch Scannen der Dateien package.json und package-lock.json, um die vollständige Liste der Abhängigkeiten zu erstellen, und vergleicht dann die Pakete aus dem Abhängigkeitsbaum mit der Liste der bekannten Schwachstellen. Wenn ein Hinweis gefunden wird, der das vom Projekt verwendete Paket betrifft, wird eine Warnung mit detaillierten Informationen angezeigt. Der Inhalt von lokal installierten Paketen wird nicht wirklich untersucht. Der 'npm audit' Scanner ist kostenlos und verwendet intern das Paket `npm-registry-fetch` um Abfragen an die 'npm registry API' zu stellen.

npm-registry-fetch Das Paket `npm-registry-fetch` [15] ist eine Node.js-Bibliothek, die eine API für den konsistenten Zugriff auf die npm Registry implementiert.

npm Registy API Abfrage Es gibt eine Möglichkeit die REST-API direkt anzusprechen. Diese ist zwar nicht dokumentiert, kann aber ohne Einschränkungen verwendet werden. Ein Modul kann direkt mit der Version abgefragt werden[16]. Es sind auch mehrere Dependencies mit einem Aufruf abgefragbar. Dafür wird eine Abfrage an die Registry gestellt, welche gleich aufgebaut ist, wie der intern vom `npm audit`-Scanner verwendete Aufruf. [17]

Verwendbare Daten einer Abfrage der npm-Registry API nach Paketname mit spezifischer Version

- Schwachstellen-Gewichtung (severity): Info, Low, Moderate, High, Critical
- Auflistung der Anzahl gefundenen Schwachstellen nach Gewichtung
- Common Weakness Enumeration (CWE)

[13] https://registry.npmjs.org
[14] http://wiki.commonjs.org/wiki/Packages/1.0
[15] https://www.npmjs.com/package/npm-registry-fetch
[16] https://registry.npmjs.org/-/npm/v1/security/advisories/search?module=MODULE-NAME&version=VERSION
[17] https://registry.npmjs.org/-/npm/v1/security/audits

- Common Vulnerability and Exposure (CVE)
- Link zur npm-Advisorie [18]
- Versionen der Schwachstellen
- Versionen mit Patches
- Schwachstellenbeschreibung
- Vorschläge / Empfehlungen
- Verwertbarkeit
- betroffene Komponenten

GitHub

Bei GitHub können die Maintainer eines Repositories Sicherheitsschwachstellen anhand eines Security Advisory[19] erfassen. Ein Advisory umfasst eine Beschreibung, den Schweregrad (Severity: Low, Moderate, High, Critical), eine CVE-Identifikation sowie eine Liste von der Schwachstelle betroffenen Versionen. Weiter kann spezifiziert werden, wann und mit welcher Version die Schwachstelle behoben wurde.

GraphQL API GitHub bietet eine GraphQL[20] API, welche anhand eines Web-Explorers [21] erkundet werden kann.

Authentifikation Die Benutzung der API benötigt eine Authentifikation. Ein Benutzer einer Applikation kann sich mit seinem GitHub Account anmelden und sich so selbst für API-Anfragen authentifizieren[22]. Abfragen an die API sind pro Stunde limitiert[23].

Weitere Datenbanken mit Einschränkungen

Snyk Snyk hat eine umfangreiche und aktuelle Vulnerability Datenbank[24], welche aber keinen kostenlosen API-Zugriff anbietet. Ohne eine Lizenz kann die API folglich nicht verwendet werden. Das Angebot ist allenfalls geeignet um weitere Informationen zu npm Paketen via Web-Link mit Namen als Query [25] zu verlinken.

Retire.js Es wird keine API angeboten und es steht keine offizielle Schnittstelle zur Verfügung. [26]

[18] Beispiel: https://npmjs.com/advisories/1343
[19] https://github.com/git/git/security/advisories
[20] https://graphql.org
[21] https://developer.github.com/v4/explorer/
[22] https://docs.github.com/en/free-pro-team@latest/developers/apps/authorizing-oauth-apps#web-application-flow
[23] https://docs.github.com/en/free-pro-team@latest/graphql/overview/resource-limitations
[24] https://snyk.io/vuln
[25] Beispiel einer Abfrage: https://snyk.io/vuln/search?q=angular&type=npm
[26] https://retirejs.github.io/retire.js

OWASP Dependency Check Es wird keine API angeboten und es steht keine offizielle Schnittstelle zur Verfügung. [27]

Sonatype OSS Index Dieser Katalog ist leider nur für den internen Gebrauch erlaubt und kann nicht ohne weiteres kommerziell verwendet werden. [28]

[27] https://owasp.org/www-project-dependency-check
[28] https://ossindex.sonatype.org

A.4 Initiale Arbeitspakete

- Projektplan erstellen
- Risikoanalyse erstellen
- Konkurrenzanalyse erstellen
- Metriken festlegen
- Funktionale Anforderungen erstellen
- Nichtfunktionale Anforderungen erstellen
- Domänenanalyse erstellen
- High-Level-Softwarearchitektur erstellen
- Detaillierte Softwarearchitektur ausarbeiten
- Tooling aufsetzen: Git-Repository, CI/CD, Entwicklungsumgebung
- Prototyp erstellen
- Plugin implementieren
- Usability-Prototyp erstellen
- Usability-Tests durchführen
- Systemtests spezifizieren
- Systemtests durchführen und protokollieren
- Qualitätssicherung ausarbeiten, überprüfen und festhalten
- Deployment-Plan erstellen
- Schlussbericht erstellen
- Schlusspräsentation erstellen

A.5 User Stories

US 1.1: Dependency Tree

Als Anwender der Erweiterung möchte ich eine rekursive Auflistung aller installierter Dependencies und Subdependencies sehen, um einen Überblick über die verwendeten Module und deren Zusammenspiel zu erhalten.

Akzeptanzkriterien

- Beim öffnen der package.json-Datei wird eine Tree View in der Explorer-Ansicht eingeblendet
- Die Tree View unterteilt sich in die zwei Kategorien dependencies und dev-Dependencies (struktuiert als Top-Level-Einträge)
- Zu jeder Kategorie werden alle lokal installierten Dependencies angezeigt, welche auch im package.json deklarierte sind.
- Bei einem Mismatch wird eine Fehlermeldung im Output ausgegeben
- Subdependencies werden rekursiv als Kinder der entsprechenden Dependency dargestellt
- Für jede Dependency wird der Name und die installierte Version angezeigt

Technical Hint

- Verwenden von Filesystem für Abfrage von Package-Daten

US 1.2: Gesamtzustand des Projektes

Als Anwender der Erweiterung möchte ich auf einen Blick den Gesamtzustand des aktuellen Projektes sehen, um mir eine Übersicht zu verschaffen und falls nötig Massnahmen zu treffen.

Akzeptanzkriterien

- Status Bar Item zeigt Extension-Name und Icon für folgende Status an:
  - Lade-Icon während Metriken geladen werden
  - Check-Icon wenn keine Probleme gefunden wurden
  - Fehler-Icon wenn Probleme gefunden wurden
- Bei Hover über Statusbar-Icon wird der aktuelle Status genauer ausgeschrieben
  - 'Loading Metrics' während die Metriken geladen werden
  - 'No problems found' wenn keine Probleme gefunden wurden
  - 'X problems found' wenn Probleme gefunden wurden wobei X der Anzahl der Probleme entspricht berücksichtigt

Technical Hint

- Reaktivität prüfen (wie wird der Status automatisch angepasst wenn sich die Metriken ändern)
- Möglichkeit auch Ladezustand anzuzeigen

US 1.3: Zustand von Dependencies in Tree ansehen

Als Anwender der Extension möchte ich in der Tree View eines Packages sehen in welchem Zustand die darin verwendeten Dependencies sind, um allfällige Qualitätsprobleme zu erkennen.

Akzeptanzkriterien

- Zu jeder Dependency wird in der Tree View ein Indikator angezeigt, welcher ausdrückt ob die Dependency
  - ohne Probleme ist (gut)
  - gewisse Probleme hat (Warnung)
  - schwerwiegende Probleme hat (schlecht) oder nicht überprüfbar ist (unbestimmt)
- Der Top-Level-Eintrag im Dependency Tree ist standardmässig aufgeklappt
- Es werden nur Informationen zu Paketen geladen, welche im Tree ersichtlich sind (aufgeklappt)

US 1.4: Aktualisieren des Zustandes

Als Anwender von der Extension möchte ich bei Hinzufügen, Entfernen oder Ändern der Version einer Dependency sehen wie diese eingestuft wird und sich dies auf den Gesamtzustand des Projektes auswirkt.

Akzeptanzkriterien

- Werden Änderungen im package.json gespeichert, wird der Zustand der geänderten Dependencies und der Gesamtzustand neu berechnet.
- Die Visualisierungen der Metriken werden mit den neuen Werten dargestellt.
- Dem Anwender wird in der Statusbar angezeigt, ob die Berechnungen noch laufen oder alles auf dem neusten Stand ist.

US 2.1: Pakettiefe eingrenzen

Als Anwender der Erweiterung möchte ich die Metriken nur von direkten Dependencies sehen und kein Performanceverlust erleiden durch weitere unnötigen Abfragen.

Akzeptanzkriterien

- Der Anwender kann die gewünschte rekursive Abfragetiefe der Dependencies über seine Benutzerspezifischen Konfiguration beeinflussen

US 3.1: License Tree

Als Anwender der Erweiterung möchte ich eine rekursive Auflistung aller Lizenzen sehen, welche in meinem Projekt vorhanden sind, um keine rechtlichen Probleme zu bekommen.

Akzeptanzkriterien

- Beim Auswählen eines Eintrages im Dependency Tree (UC1) wird eine weitere Tree View eingeblendet
- Der Top-Level-Eintrag der Tree View ist die ausgewählte Dependency, darunter erscheinen rekursiv die Subdependencies
- Für jede Dependency wird der Name und die Lizenz angezeigt
- Ist keine Lizenz vorhanden, wird *no license available* angezeigt

Technical Hint

- Fokus zuerst auf *Presentation Layer*
- Verwendung von Funktionalität aus UC1 für Daten

US 3.2: Lizenzdetails

Als Anwender der Erweiterung möchte informationen zu einer Lizenz erhalten, um so zu sehen, ob ich sie korrekt verwenden kann.

Akzeptanzkriterien

- Beim Auswählen eines Eintrages aus öffnet sich eine WebView
- Die Information werden in rudimentärer Form dargestellt
- Es sind Informationen zu den gängigsten Lizenzen vorhanden (Popular Licenses)[a]
- Die Lizenzdetails werden nach folgende Kriterien gruppiert (Permissions, Limitations, Conditions)[b]

Technical Hint

- Reaktivität prüfen (wie wird der Status automatisch angepasst wenn sich die Metriken ändern)
- Möglichkeit auch Ladezustand anzuzeigen

[a] https://opensource.org/licenses
[b] https://github.com/garris/BackstopJS/blob/master/LICENSE

US 4.1: Zusammenfassung des Sicherheitszustandes

Als Anwender der Extension möchte ich eine Zusammenfassung des Sicherheitszustandes meiner direkten Dependencies im Package sehen, um schnell einen Überblick der Gefahren zu erhalten.

Akzeptanzkriterien

- Es wird die Anzahl gefundener Schwachstellen der direkten Dependencies im Gesamtzustand angezeigt (Anzahl Probleme)
- In einer Problemansicht werden gefundene Schwachstellen mit Schweregrad einer Sicherheitsschwachstelle aufgelistet (Kategorien: Info, Low, Moderate, High, Critical).
- Version einer direkten Dependency im Package.json wird markiert, falls eine gefundenen Schwachstelle gefunden wurde.

US 4.2: Verlinkung zu einer Detailbeschreibung

Als Anwender von der Extension möchte ich eine Detailbeschreibung der gefundenen Sicherheitslücke einer Dependency sehen, um nachzulesen wie schwerwiegend die Schwachstelle ist.

Akzeptanzkriterien

- Beim Auswählen eines Eintrages im Dependency Tree der Sicherheitsschwachstellen wird eine Webview eingeblendet
- Es wird detailliert Beschrieben welcher Fehler gefunden wurde und es wird mit Hilfe eines Links auf die Beschreibung verlinkt.

A.6 Risikoanalyse: Revisionen

Nachfolgend sind die Revisionen der Risikoanalyse dokumentiert.

MS3

Erste definitive Version dieses Dokuments.

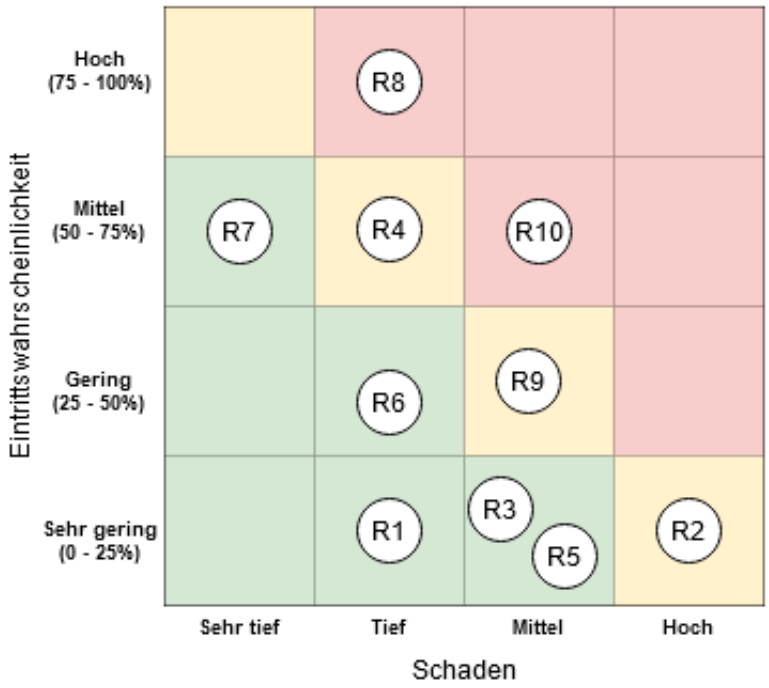

Abbildung A.1: Risikomatrix, Stand MS3

MS4: 16.11.2020

R1 Die Verzögerungen im Abarbeiten der Funktionalität und Zeitdruck führen zu einem erhöhten Potenzial für Konflikte.

Eintritts-Ws. Erhöhung von *sehr gering* zu *mittel*.

Schaden Erhöhung von *tief* zu *mittel*.

R3 Das späte erreichen eines alle Schichten durchstechenden Funktionsumfangs haben dazu geführt, dass der technische Bericht zu wenig weitergeführt und die Test-Coverage noch nicht genügend ist.

Eintritts-Ws. Erhöhung von *sehr gering* zu *gering*

Vorbeugung Neue Massnahme: Nicht User-Story bezogene Tasks

R8 Tasks konnten nur knapp auf Sprint-Ende abgeschlossen werden, da verschiedene Arbeitszeiten zu Wartezeiten geführt haben. Planungssitzungen sind zeitlich knapp und führen zu einer suboptimalen Planung.

Vorbeugung Neue Massnahme: Verschieben des Sprint-Start-Tages

R10 Trotz Bewusstsein und Vorbeugungsmassnahmen ist es zu Verzögerungen aufgrund noch nicht fertiger Features gekommen. Gründe dafür waren jedoch auch gewisse Wissensmängel.

Vorbeugung Neue Massnahme: Pair Programming wenn Probleme oder Unsicherheiten bestehen

A.7 Systemtests

A.7.1 Einführung

Zusätzlich zu den Unit Tests machten wir Anwendertest um die Funktionalität des VS Code Plugins zu testen. Die Systemtests wurden in der finalen Version durchgeführt und können noch auf bestehende Probleme hinweisen, welche bei einer zukünftigen Weiterentwicklung beachtet werden sollten.

Vorbedingung Dependencies aus der *package.json* Datei im Projekt müssen über einen Node Package Manager wie yarn oder npm installiert werden.

A.7.2 Testfall 1: Gesamtzustand des Projektes anzeigen

Testbeschreibung Es soll der Gesamtzustand anhand den Dependencies in der *package.json* berechnet und in der Status Bar von VS Code angezeigt werden.

Testziele

- Starten der Metrikberechnung installierter Dependencies nach dem öffnen der *package.json*Datei
- Anzeige eines Warnungs-Icons in der Status Bar, wenn die Score unter 0.5 ist, ansonsten ein Check-Icon bzw. Lade-Icon währende der Berechnung
- Anzeige einer Score durch Hover in der Status Bar beim Package Metrics Icon

A.7.3 Testfall 2: Zustand von Dependencies anzeigen

Testbeschreibung Es sollen alle Dependencies in der Side Bar im Node Package Dependency Tree mit berechneten Metriken vorhanden sein. Ein Zustands-Icon und eine Score pro Node Package zeigt den Zustand direkt an.

- Starten der Metrikberechnung installierter Dependencies nach dem öffnen der *package.json* Datei
- Aufklappbarer Node Package Dependency Tree in der Side Bar mit Zustandsanzeige via korrekten Icon (Ok ,Warnung, Nicht analysiert)
- Hinzufügen einer Abhängigkeit in der Package.json (inkl. speichern und erneuter Installation) und anschliessender erneuter Ausführung der Metrikberechnung
- Entfernung einer Abhängigkeit (inkl. speichern und erneuter Installation) und schnliessender erneuter Ausführung der Metrikberechnung

- Darstellen einer Score durch das schweben der Maus über einer Abhängigkeit im Node Package Baum

A.7.4 Testfall 3: Metrikberechnug mit angepasster Tiefe

Testbeschreibung Durch das Setzen einer Tiefe von 2 sollen nur die Dependencies bis zur Stufe 2 berechnet und im Node Package Dependency Tree mit Metriken dargestellt werden.

Testziel

- Setzen der Tiefe 2 in der Konfiguration von VS Code
- Starten der Metrikberechnung installierter Dependencies nach dem öffnen der *package.json* Datei
- Aufklappbarer Node Package Dependency Tree in der Side Bar mit Zustandsanzeige via korrekten Icon (Ok ,Warnung, Nicht analysiert) nur bei den ersten 2 Dependency Stufen. Die restlichen Dependencies werden dargestellt, wurden aber nicht analysiert.

A.7.5 Testfall 4: Lizenzdetails anzeigen

Testbeschreibung Zu jedem Eintrag im Node Package Dependency Tree wird die jeweilige Lizenz angezeigt und detaillierte Informationen zu einer Lizenz sind pro Eintrag abrufbar.

Testziel

- Aufklappen der Abhängigkeiten im Node Package Dependency Tree zeigt die Lizenz direkt bei jedem Eintrag an
- Darstellung von Lizenzdetails durch anklicken einer Abhängigkeit im Node Package Dependency Tree

A.7.6 Testfall 5: Vulnerabilities im Editor anzeigen

Testbeschreibung Nachdem alle Node Packages analyisert wurden, werden gefundene Vulnerabilites der direkten Dependencies in der *package.json* Datei und im Problem Panel dargestellt.

Testziel

- Starten der Metrikberechnung installierter Dependencies nach dem öffnen der *package.json* Datei
- Darstellen der Problemliste und Selektion einer gefundenen Vulnerability
- Öffnen der *package.json* Datei und anzeigen der Vulnerability durch Hover über der blau markierten Version einer Dependency (über die Taste F8 wird im Editor das Problem eingeblendet)

A.7.7 Testfall 6: Detailansicht zu Node Packages darstellen

Testbeschreibung Eine Detailansicht zeigt Anzahl analysierte Dependencies, ausgewertete Metriken, Lizenzdetails und gefundene Vulnerabilites für analysierte Node Packages an.

Testziel

- Starten der Metrikberechnung installierter Dependencies nach dem öffnen der package.json
- Das Anklicken von verschiedenen Einträgen im Node Package Dependency Tree öffnet jeweils ein neuer Tab mit einer Detailansicht des Node Packages

A.8 Systemtest Protokolle

Test Case	Tester	Datum	Erfolg	Kommentar
Testfall 1	Flavio Böni	15.12.2020	Ja	Anzeige ok
Testfall 2	Flavio Böni	15.12.2020	Ja	Korrektes Verhalten
Testfall 3	Flavio Böni	15.12.2020	Ja	Wird berücksichtigt
Testfall 4	Flavio Böni	15.12.2020	Ja	Anzeige ok
Testfall 5	Flavio Böni	15.12.2020	Ja	Korrekte Position ist markiert
Testfall 6	Flavio Böni	15.12.2020	Ja	Funktioniert einwandfrei

A.9 Kommentare zu den Tests

- *package.lock* Datei muss nach der Installation der Dependencies vorhanden sein, damit die Node Package Struktur ausgelesen werden kann
- Die Tiefe der Dependencies sollte bei grossen Projekten mit vielen rekursiven Dependencies eingeschränkt werden, um schneller ein Ergebnis zu erhalten
- Das Status Bar Icon wird bei einem Fehler in der Metrikberechnung nicht aktualisiert, so dass es aussieht als ob die Berechnung noch läuft
- Bei vielen Dependencies in grösseren Projekten sollte bei der Metrikberechnung ein Progress dargestellt werden um den Fortschritt sichtbar zu machen
- Das Abbrechen einer laufenden Berechnung ist aktuell nicht möglich
- Nach Änderungen in der *package.json* Datei muss jeweils npm install aufgerufen werden, damit die aktuellen Node Packages verwendet werden können

A.10 Aufgabenstellung

OST
Ostschweizer
Fachhochschule

Semesterarbeit

Thema: Node.js Package Metrics

Autor:	Silvan Gehrig
Version:	1.0
Erstellt am:	13.09.2020
Letzte Änderung am:	22.09.2020

OST
Ostschweizer
Fachhochschule

Änderungsnachweis

Version	Änderungsgrund	Kurz-Z.	Datum
1.0	Initial	sgeh	13.09.20
1.1	Aufgabenstellung	sgeh	19.09.20
1.2	Termine konkretisieren	sgeh	22.09.20

Inhaltsverzeichnis

1. Einführung

Die npm Library enthält über 1 Million Packages. Die frei verfügbaren Packages werden häufig nur sporadisch gepflegt und Bugs in den Packages nur langsam behoben.

Wir wollen mittels Visual Studio Code Plugin den Zustand des aktuellen, sowie der referenzierten Packages direkt im package.json File visualisieren.

2. Aufgabe

Das Ziel der Arbeit besteht darin, von aussagekräftigen Metriken über Packages anhand der package.json Informationen zu erarbeiten und diese zu visualisieren. Packages verlinken häufig wiederum weitere Packages, wodurch eine Rekursive Bewertung von Dependencies notwendig wird.

Die Aufgabenstellung kann bei Bedarf mutiert werden. Die Arbeit besteht aus den folgenden Hauptaufgaben:

2.1 Requirements Analysis

- Ausarbeiten und definieren der Anforderungen anhand einer Konkurrenzanalyse.
- Sicherstellen der Benutzbarkeit durch Usability Tests.

2.2 Ausarbeiten von Metriken

- Evaluieren von diversen Metriken, welche aufgrund der package.json Informationen eruiert werden können.
- Einbeziehen von Dependencies sowie deren Dependencies in die Metriken.
- Bereitstellen eines Metriken-APIs, welcher die Metriken unabhängig von der Darstellung (UI) berechnen.
- Einbeziehen der node.js Package Vulnerabilities in die Metriken, siehe geekflare.com/nodejs-security-scanner.
- Errechnen von Historien, d.h. darstellen von Metriken über mehrere Versionen desselben Projektes.

2.3 Integration in Visual Studio Code

- Direkte Integration der Metriken als Plugin in Visual Studio Code, sobald package.json Files geöffnet werden.
- Live-View der Änderungen am package.json File in die Metriken.
- Darstellen der Metrikendetails in einem zusätzlichen Activity Tool.
- Visualisierung der Metriken auf eine möglichst intuitive Art und Weise unter Einbezug von bestehender Literatur (Lanza & Marinescu, 2006, Object-Oriented Metrics in Practice).
- Möglichst gute Adaptierbarkeit in andere Entwicklungsumgebungen wie IntelliJ.

3. Hinweise

3.1 Themen und Technologien

- Visual Studio Code / Electron
- Node.js
- Diverse WebAPI's (z.B. GitHub API)
- TypeScript

4. Erwartete Resultate

Als Resultat soll ein dokumentiertes und in Visual Studio Code lauffähiges Plugin abgegeben werden. Eine gute Wartbarkeit der Software soll die Weiterentwicklung sicherstellen. Weiteres gilt zu beachten:

- Vorgehen nach den Regeln des Software-Engineering (Scrum und UP), Arbeit nach Projektplan. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten.
- Erfassen des tatsächlichen Arbeitsaufwands. Es muss ersichtlich sein, wer für welchen Teil der Arbeit und des Berichts verantwortlich ist.
- Alle Dokumente sind nachzuführen, d.h. sie sollen den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren.

5. Termine

Die verbindlichen Termine für die Semesterarbeit finden sich in den Studiengangrichtlinien unter: \\hsr.ch\root\alg\skripte\Informatik\Fachbereich\Studienarbeit_Informatik\SAI14\Termine .

Die zeitliche Planung der weiteren Meetings wird von den Studenten durchgeführt:

- Ende der Elaboration (ca. in 1/3 des Projekts) findet ein Meeting statt, bei welchem von den Studenten der «Durchstich» /demonstriert wird.
- Es findet jede Woche ein (oder alternativ alle 2 Wochen ein längeres) Statusmeeting mit dem organisatorischen Betreuer statt. Zusätzliche Besprechungen sind nach Bedarf zu veranlassen.

  Während des Status-Meeting werden anhand des Projektplanes folgende Punkte erläutert:
  - Was wurde erreicht?
  - Wie viele Stunden wurden geleistet?
  - Wo gab es Probleme, werden Hilfestellungen benötigt?
  - Wie ist das weitere Vorgehen?
- Alle Besprechungen sind von den Studenten mit einer Traktandenliste vorzubereiten und Beschlüsse in einem Protokoll zu dokumentieren, das den Betreuern per E-Mail zugestellt wird.

6. Betreuung und Ansprechpartner

- Stake Holder / Product Owner: IFS, Institut für Software, Silvan Gehrig
- Product Owner: IFS, Institut für Software, Mirko Stocker

7. Beurteilung

Eine erfolgreiche Studienarbeit zählt 8 ECTS-Punkte pro Studierenden. Für 1 ECTS Punkt ist eine Arbeitsleistung von ca. 25 bis 30 Stunden budgetiert (vgl. M_SAI14 für die Modulbeschreibung der Studienarbeiten). Für die Beurteilung ist der organisatorische Betreuer verantwortlich.

Die Beurteilung der Arbeiten erfolgt nach einem einheitlichen Bewertungsschema:

Gesichtspunkt	Gewicht
1. Organisation, Durchführung (Projektplanung u. Nachführung Arbeit gemäss Projektplan, Selbständigkeit, Einsatz, Zusammenarbeit mit Auftraggeber, Betreuerin oder Betreuer)	1/5
2. Bericht (Inhalt des Projektschlussberichts, Gliederung, Darstellung, Sprache der gesamten Dokumentation)	1/5
3. Resultat der Arbeit	
3.1 Problemanalyse (Vorstudie, Literaturstudium, Anforderungsspezifikation, Anforderungsanalyse, Domainanalyse)	1/5
3.2 Lösungsentwurf (Lösungsvarianten und deren Beurteilung, Variantenentscheid, Konzept, Entwurf)	1/5
3.3 Realisierung und Test	1/5

Diese Aufstellung basiert auf den Regelungen Studien- und Bachelorarbeiten im Studiengang Informatik \\hsr.ch\root\alg\skripte\Informatik\Fachbereich\Studienarbeit_Informatik\SAI14\ . Es gelten die Bestimmungen der Abteilung Informatik zur Durchführung von Studienarbeiten.

Rapperswil-Jona, 22. September 2020

Silvan Gehrig

Institut für Software

OST – Ostschweizer Fachhochschule

A.11 Zeiterfassung

Im folgenden Abschnitt befinden sich die Resultate der Zeiterfassung. Arbeitspakete wurden in vier Kategorien aufgeteilt:

Admin Administrative Tätigkeiten wie Projektplanung und Meetings.

Coding Programmierung und Aufsetzen von Tooling.

Research Recherche und Analyse.

Documentation Schreiben von Dokumentation.

Bei der Betrachtung der Grafiken sollte berücksichtigt werden, dass die Sprints 3 und 8 statt zwei nur eine Woche lang waren.

Übersicht

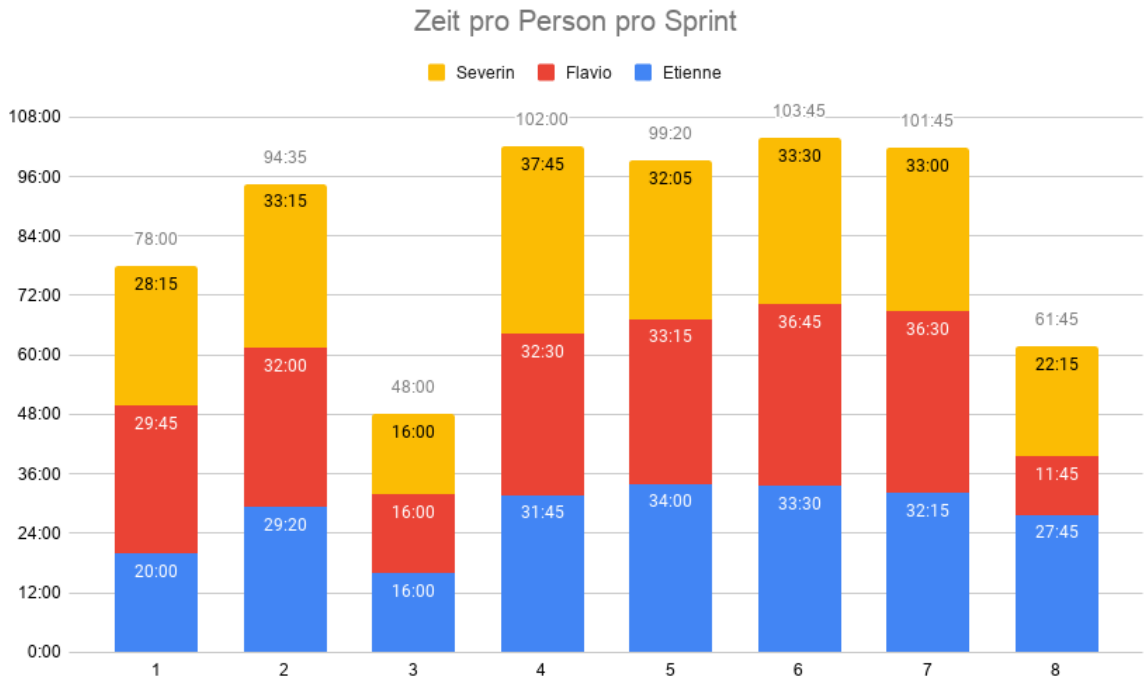

Abbildung A.2: Zeit in Stunden, aufgebrochen nach Teammitgliedern und Sprints

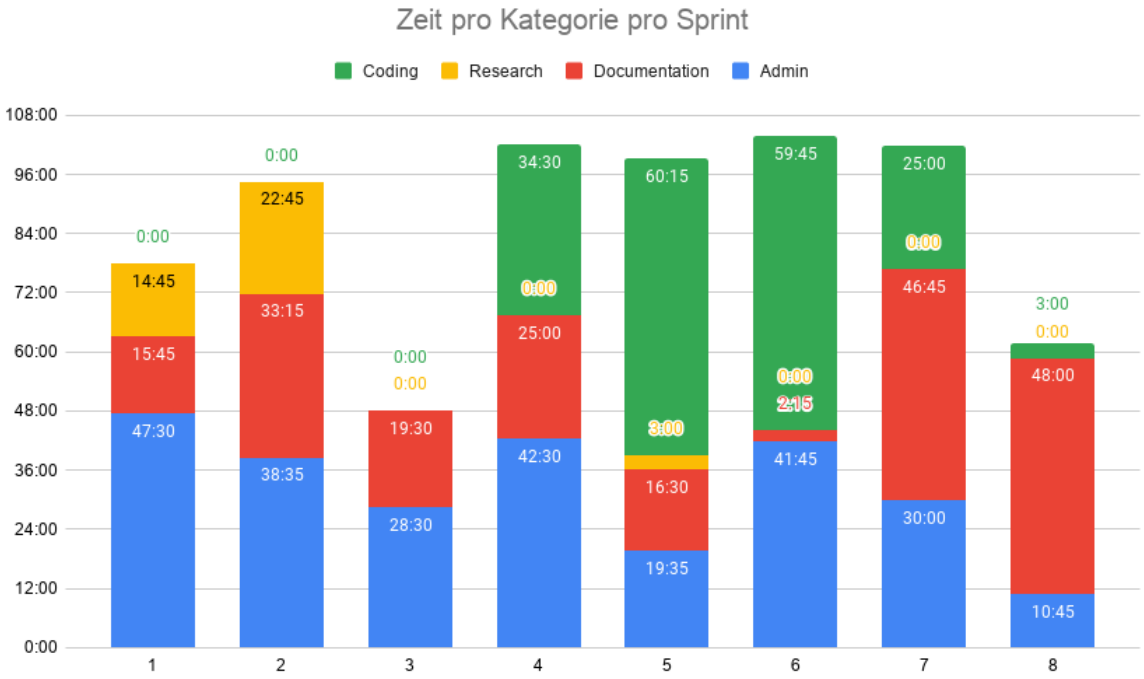

Abbildung A.3: Zeit in Stunden, aufgebrochen nach Kategorie und Sprints

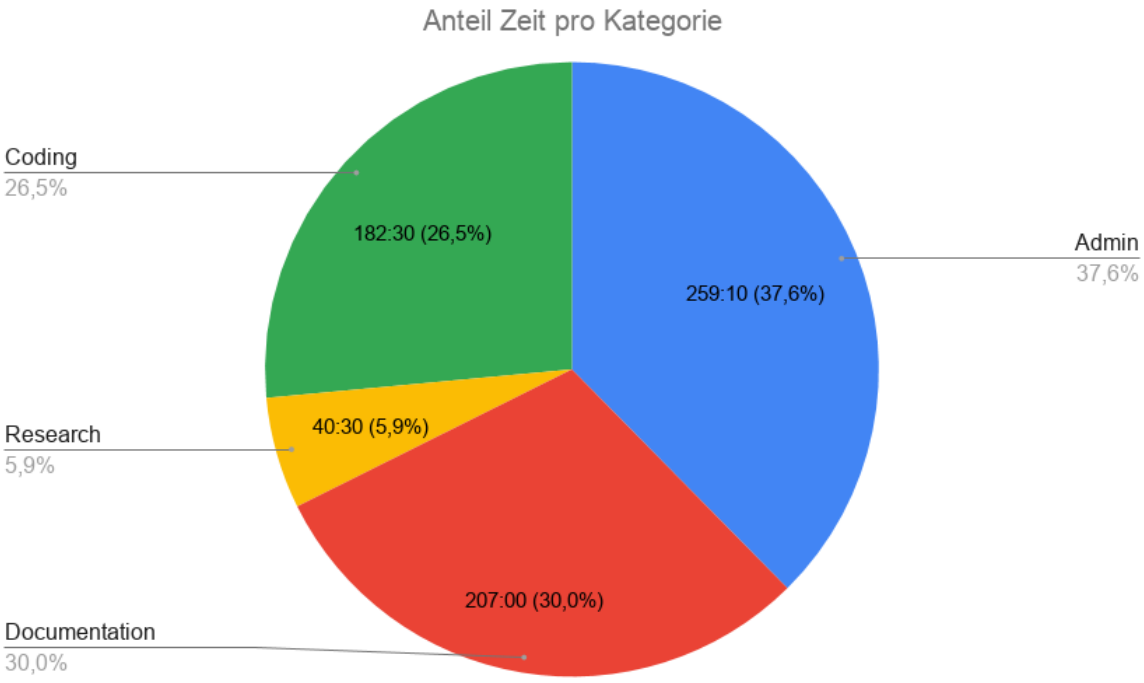

Abbildung A.4: Gesamtaufwand, aufgebrochen pro Kategorie

Zeiten

Gesamtübersicht

Sprint	Etienne	Flavio	Severin	Soll	Ist	Soll Akkumuliert	Ist Akkumuliert
1	20:00	29:45	28:15	96:00	78:00	96:00	78:00
2	29:20	32:00	33:15	96:00	94:35	192:00	172:35
3	16:00	16:00	16:00	48:00	48:00	240:00	220:35
4	31:45	32:30	37:45	96:00	102:00	336:00	322:35
5	34:00	33:15	32:05	96:00	99:20	432:00	421:55
6	33:30	36:45	33:30	96:00	103:45	528:00	525:40
7	32:15	36:30	33:00	96:00	101:45	624:00	627:25
8	27:45	11:45	22:15	48:00	61:45	672:00	689:10
Total	**224:35**	**228:30**	**236:05**	**672:00**	**689:10**		

Sprint	Admin	Documentation	Research	Coding	Total
1	47:30	15:45	14:45	0:00	**78:00**
2	38:35	33:15	22:45	0:00	**94:35**
3	28:30	19:30	0:00	0:00	**48:00**
4	42:30	25:00	0:00	34:30	**102:00**
5	19:35	16:30	3:00	60:15	**99:20**
6	41:45	2:15	0:00	59:45	**103:45**
7	30:00	46:45	0:00	25:00	**101:45**
8	10:45	48:00	0:00	3:00	**61:45**
Total	**259:10**	**207:00**	**40:30**	**182:30**	**689:10**

Sprint 1 (15.09.2020 - 28.09.2020)					**Etienne**		**Flavio**		**Severin**	
Ticket	**CU ID**		**Tags**	**SOLL** **IST**	W1&W2	~~W2~~	W1&W2	~~W2~~	W1&W2	~~W2~~
Aufsetzen Tools	#8et631	Link	admin	00:00 2:30					02:30	
Overleaf einrichten	#8ent1m	Link	admin	00:00 2:00			02:00			
Sprints und Meilensteine in Clickup erstellen	#8gn7mm	Link	admin	00:00 1:00			01:00			
Risikoanalyse erstellen	#8epbvr	Link	documentation	00:00 1:30					01:30	
Research Metriken	#8amx5r	Link	research	00:00 7:45			07:45			
Konkurrenzanalyse erstellen	#8amx5v	Link	research	00:00 3:00			03:00			
Projektplan erstellen	#8amx5k	Link	documentation	00:00 8:00	08:00					
Research VS Code Extension API	#8ethv6	Link	research	00:00 4:00					04:00	
Test Repository erstellen	#8ete83	Link	admin	00:00 5:30					05:30	
Bericht verwendete Tools			documentation	01:00 2:00			02:00			
Clickup aufsetzen			admin	02:00 3:00			03:00			
Qualitätsmassnahmen			documentation	03:00 4:15					04:15	

Meetings Sprint 1	#8et76c	Link	admin	30:00 32:00	11:00		11:00		10:00	
Varia Sprint 1	#8et7ce	Link	admin	06:00 1:30	01:00				00:30	
Total				**42:00** **78:00**	**20:00**		**29:45**		**28:15**	
					20:00	~~0:00~~	29:45	~~0:00~~	28:15	~~0:00~~
Restaufwand				54:00 18:00	4:00	~~-16:00~~	13:45	~~-16:00~~	12:15	~~-16:00~~

Soll pro Woche
16:00

Tag	**Aufwand**
admin	47:30
document	15:45
research	14:45
coding	0:00
Total	**78:00**

Sprint 2 (29.09.2020 - 12.10.2020)					**Etienne**		**Flavio**		**Severin**	
Ticket	**CU ID**	**Tags**	**SOLL**	**IST**	W1	W2	W1	W2	W1	W2
Personas erstellen	#8mqu5f Link	documentation	04:00	03:45	03:45					
Use Case erstellen	#8rup2p Link	documentation	08:00	08:00	04:00		04:00			
Metriken sammeln und strukturieren	#8mqvr6 Link	documentation	04:00	03:30			03:30			
Erste nicht-funktionale Anforderungen erfassen	#8mr6qf Link	documentation	01:00	01:30					01:30	
Nicht-funktionale Anforderungen erweitern	#8pu17x Link	documentation	03:00	04:45	00:45		03:00		01:00	
Hauptdomäne analysieren	#8mr1v9 Link	documentation	02:00	02:15					02:15	
Erweiterung Prototyp	#8ptz2n Link	research	06:00	07:45					07:45	
Projektplan nachführen	#8wvcqw Link	documentation	02:00	01:00	00:45	00:15				
Domainanalyse überarbeiten	#8jumm1 Link	documentation	04:00	02:00				02:00		
Nichtfunktionale Anforderungen überarbeiten	#8wxaav Link	documentation	04:00	04:30		00:30				04:00
Risikoanalyse aktualisieren	#8mr0ew Link	documentation	02:00	02:00						02:00
User Stories für MUST-UCs erstellen	#8wxegh Link	admin	03:00	02:45		00:45		01:00		01:00
Designentwurf erstellen	#8wxqg2 Link	research	06:00	06:00		01:30		02:00		02:30
Prototyp anschauen	#8wxrk1 Link	research	06:00	09:00		04:15		04:45		

Meetings Sprint 2	#8mqpdx Link	admin	30:00	05:15	05:30	04:15	05:30	04:15	05:30	04:15
Varia Sprint 2	#8mqpk1 Link	admin	06:00	06:35	01:20	01:45		02:00		01:30
Total			**91:00**	**94:35**	**29:20**		**32:00**		**33:15**	
					16:05	13:15	16:00	16:00	18:00	15:15
Restaufwand			5:00	1:25	0:05	-2:45	0:00	0:00	2:00	-0:45

Soll pro Woche
16:00

Tag	**Aufwand**
admin	38:35
document	33:15
research	22:45
coding	0:00
Total	**94:35**

Sprint 3 (13.10.2020 - 19.10.2020)					Etienne		Flavio		Severin	
Ticket	**CU ID**	**Tags**	**SOLL**	**IST**	W1	~~W2~~	W1	~~W2~~	W1	~~W2~~
Feedback von Review einarbeiten	#94uhnp Link	documentation	01:00	1:00					01:00	
Tooling aufsetzen	#8jumw2 Link	admin	05:00	7:30			06:00		01:30	
Projektplan nachführen	#8wvcqw Link	documentation	02:00	2:15	02:15					
Domainanalyse überarbeiten	#8jumm1 Link	documentation	04:00	1:00			01:00			
User Stories schreiben	#94uha7 Link	admin	06:00	4:30	01:30		01:00		02:00	
Bestehende Dokumente ins Overleaf übertragen	#8wvb2y Link	documentation	04:00	8:00	05:15				02:45	
Design erstellen	#8jumzt Link	documentation	05:00	3:45					03:45	
Backlog für Sprint 4 vorbereiten	#96pfbt Link	admin	01:00	1:00	00:30		00:30			
High-Level-Softwarearchitektur erstellen	#8jumq3 Link	documentation	02:00	3:30	01:30		02:00			

Meetings Sprint 3	#96pbzp Link	admin	15:00	15:00	05:00		05:00		05:00	
Varia Sprint 3	#96pby4 Link	admin	03:00	0:30			00:30			
Total			**48:00**	**48:00**	**16:00**		**16:00**		**16:00**	
					16:00	~~0:00~~	16:00	~~0:00~~	16:00	~~0:00~~
Restaufwand			48:00	48:00	0:00	~~-16:00~~	0:00	~~-16:00~~	0:00	~~-16:00~~

Soll pro Woche
16:00

Tag	Aufwand
admin	28:30
document	19:30
research	0:00
coding	0:00
Total	**48:00**

Sprint 4 (20.10.2020 - 02.11.2020)					**Etienne**		**Flavio**		**Severin**	
Ticket	**CU ID**	**Kategorie**	**SOLL**	**IST**	W1	W2	W1	W2	W1	W2
Design erstellen	#8jumzt Link	documentation	05:00	0:30	00:30					
Security-API-Provider evaluieren	#9cxyq9 Link	documentation	06:00	7:00			07:00			
Context and Scope	#9cqzpj Link	documentation	04:00	4:15	04:15					
Building Block View – Level 1	#9cr0dq Link	documentation	03:00	4:30	04:30					
[UC1] Dependency Tree	#9aq7b0 Link	coding	16:00	26:30		00:30			11:00	15:00
[UC3] License Tree	#9ay7vq Link	coding	06:00	7:15			01:30	05:45		
CI-Setup abschliessen	#9cr381 Link	admin	02:00	8:45		08:45				
Domainanalyse überarbeiten		documentation	04:00	1:30						01:30
Architektur überarbeiten		documentation	02:00	0:45		00:45				
Projektplan anpassen		documentation	02:00	0:00						
Security-API-Provider dokumentieren	#9myb2k Link	documentation	02:00	2:30				02:30		
Feedback in Dokumentation einpflegen	#9cyuxg Link	documentation	02:00	4:00				04:00		
[UC1] Gesamtzustand des Projektes	#9ay34q Link	coding	08:00	0:45	00:45					
[UC3] Lizenzdetails		coding	16:00	0:00						
Systemsequenzdiagramme erstellen		documentation	04:00	0:00						

Meetings Sprint 4	#9cyum0 Link	admin	30:00	27:00	03:45	05:00	04:30	04:15	04:30	05:00
Varia Sprint 4	#9cyum5 Link	admin	06:00	6:45	01:45	01:15	02:30	00:30		00:45
Total			**118:00**	**102:00**	**31:45**		**32:30**		**37:45**	
					15:30	16:15	15:30	17:00	15:30	22:15
Restaufwand					-0:30	0:15	-0:30	1:00	-0:30	6:15

Soll pro Woche
16:00

Tag	**Aufwand**
admin	42:30
document	25:00
research	0:00
coding	34:30
Total	102:00

Sprint 5 (03.11.2020 - 17.11.2020)					Etienne		Flavio		Severin	
Ticket	**CU ID**		**Kategorie**	**SOLL** **IST**	W1	W2	W1	W2	W1	W2
[UC3] License Tree Pair Programming	#9wt48y	Link	coding	08:00 13:15			08:00		05:15	
[UC1] Gesamtzustand des Projektes	#9ay34q	Link	coding	13:00 26:00			03:00	10:00	07:00	06:00
Dokumentation überarbeiten	#9wrun4	Link	documentation	13:00 16:30	05:00	04:00	02:30		00:30	04:30
[UC3] Lizenzdetails	#8ywu5q	Link	coding	21:00 20:00	09:00	10:00				01:00
[UC1] Dependency Tree (Fixes)	#9aq7b0	Link	coding	1:00					01:00	
Ersatzprovider für npms	#aaum1r	Link	research	3:00				03:00		

Meetings Sprint 5	#9wrvn3	Link	admin	30:00 19:20	02:15	03:30	03:00	03:45	03:20	03:30
Varia Sprint 5	#9wrvn3	Link	admin	06:00 0:15	00:15					
Total				**91:00** **99:20**	**34:00**		**33:15**		**32:05**	
					16:30	17:30	16:30	16:45	17:05	15:00
Restaufwand					0:30	1:30	0:30	0:45	1:05	-1:00

Soll pro Woche
16:00

Tag	Aufwand
admin	19:35
document	16:30
research	3:00
coding	60:15
Total	**99:20**

Sprint 6 (18.11.2020 - 01.12.2020)						Etienne		Flavio		Severin	
Ticket	**CU ID**		**Kategorie**	**SOLL**	**IST**	W1	W2	W1	W2	W1	W2
[UC1] Gesamtzustand des Projektes (follow up)	#acyufc	Link	coding	02:00	2:00			01:30		00:30	
[UC3] Lizenzdetails (follow up)	#acyug7	Link	coding	05:00	5:45	05:15		00:30			
[UC4] Zusammenfassung des Sicherheitszustandes	#94wrnj	Link	coding	08:00	10:30			06:30	02:00		02:00
Interfaces definieren	#aewku9	Link	coding	05:00	4:00	01:15				02:45	
Definitive Struktur von Core Entrypoint (PackageMetrics)	#aeycha	Link	coding	03:00	3:30	00:30		00:15		02:45	
Risikoanalyse aktualisieren	#aeyd3h	Link	documentation	01:00	0:45					00:45	
Collection / Measurement definitiv implementieren	#aeycrr	Link	coding	05:00	9:00				02:00	05:00	02:00
Empty State in Tree View darstellen	#aptnbn	Link	coding	02:00	2:00				02:00		
Scoring implementieren	#aptmcu	Link	coding	03:00	2:15						02:15
Dokumentation überarbeiten	#aey3eq	Link	documentation	13:00	0:30	00:30					
[NFR5] Logging implementieren	#aeym80	Link	coding	05:00	3:00				03:00		
[UC2] Konfiguration	#aeymhe	Link	coding	03:00	2:30		02:30				
[UC1] Zustand von Dependencies in Tree ansehen	#9az0u0	Link	coding	08:00	7:15		07:00				00:15
Ausarbeiten von Outline für Technischer Bericht	#aptkha	Link	documentation	01:00	1:00				01:00		
Diagnostic Prototype	#auz8jc	Link	coding	03:00	3:00				03:00		
Refactorings	#awxe5t	Link	coding		5:00		05:00				

Meetings Sprint 6	#acxrv6	Link	admin	30:00	38:30	08:30	02:00	11:00	03:00	11:00	03:00
Varia Sprint 6	#aczpz1	Link	admin	06:00	3:15	00:15	00:45	00:30	00:30	00:30	00:45
Total				**103:00**	**103:45**	**33:30**		**36:45**		**33:30**	
						16:15	17:15	20:15	16:30	23:15	10:15
Restaufwand						0:15	1:15	4:15	0:30	7:15	-5:45

Soll pro Woche
16:00

Tag	Aufwand
admin	41:45
document	2:15
research	0:00
coding	59:45
Total	**103:45**

Sprint 7 (02.12.2020 - 14.12.2020)						Etienne		Flavio		Severin	
Ticket	**CU ID**		**Kategorie**	**SOLL**	**IST**	W1	W2	W1	W2	W1	W2
Refactoring: Models extrahieren	#ayxvh9	Link	coding	02:00	3:15	02:45				00:30	
Root Dependency-Vulnerabilities darstellen	#ayuta2	Link	coding	03:00	3:30			03:00		00:30	
Inhalt und Struktur des abzugebenden Berichts erstellen	#b0rhju	Link	documentation	03:00	1:30			01:30			
Refactoring von npm-registry Service	#ax0r32	Link	coding	02:00	2:45	00:30				02:15	
Readme in Repositories ergänzen	#ax0zcv	Link	documentation	02:00	1:15					01:15	
Projekt- und Risikomanagement restrukturieren	#b0yea8	Link	documentation	02:00	1:45	01:45					
Konkurrenzanalyse dokumentieren	#b0xkk0	Link	documentation	03:00	5:30		02:00	03:30			
Provider refactorn zu Repositories	#b2ybmk	Link	coding	03:00	5:30	05:30					
Use Case und User Stories überabeiten	#8jun0k	Link	documentation	05:00	4:00			04:00			
Resultate: Zielerreichung ausarbeiten	#b2xg6b	Link	documentation	08:00	8:00				08:00		
Systemtest erstellen	#8jun27	Link	documentation	02:00	2:00			01:00	01:00		
Übersichts-Webview implementieren	#b2xdjr	Link	coding	08:00	8:45					07:45	01:00
Technischer Bericht: Einführung ausarbeiten	#b2xfy6	Link	documentation	05:00	8:30		02:00				06:30
Technischer Bericht: Lizenzdetails			documentation	02:00	1:15	01:15					
Refactoring: Long Method	#b91bmk	Link	coding	01:00	1:00				01:00		
Refactoring: Switch-Cases	#b8xpf1	Link	coding	00:30	0:15						00:15
Kapitel "Domänenanalyse" schreiben			documentation		2:45		02:45				
Resultate: Schlussfolgerungen ausarbeiten	#b2xg78	Link	documentation	03:00	4:00				04:00		
Technischer Bericht: Ausgangslage ausarbeiten	#b2xg8x	Link	documentation	5:00	5:00		02:45				02:15
Abstract erfassen	#ax0wa1	Link	documentation		1:15						01:15
Meetings Sprint 7	#acxrv6	Link	admin	30:00	26:15	03:15	05:30	03:15	05:30	03:15	05:30
Varia Sprint 7	#aczpz1	Link	admin	06:00	3:45	00:30	01:45	00:30	00:15	00:30	00:15
Total				**95:30**	**101:45**	**32:15**		**36:30**		**33:00**	
						15:30	16:45	16:45	19:45	16:00	17:00
Restaufwand						-0:30	0:45	0:45	3:45	0:00	1:00

Soll pro Woche
16:00

Tag	Aufwand
admin	30:00
document	46:45
research	0:00
coding	25:00
Total	**101:45**

Sprint 8 (15.12.2020 - 18.12.2020)						Etienne		Flavio		Severin	
Ticket	**CU ID**		**Kategorie**	**SOLL**	**IST**	W1	~~W2~~	W1	~~W2~~	W1	~~W2~~
Systemtests durchführen und protokollieren	#8jun39	Link	documentation	02:00	2:00			02:00			
Persönlicher Erfahrungsbericht schreiben	#b8xmqn	Link	documentation	03:00	3:45	01:00		01:30		01:15	
Umsetzung, Metriken und Sicherheitsschwachstellen auss	#bezr15	Link	documentation		4:45					04:45	
Extension deployen und in Dokumentation festhalten	#9aqfpp	Link	coding	02:00	3:00					03:00	
Lay-Summary oder Management Summary erstellen	#b0v6bm	Link	documentation	02:00	2:00					02:00	
Deployment beschreiben	#8jun68	Link	documentation	01:00	1:00					01:00	
Dokumentation der Zeitauswertung	#b4t5z8	Link	documentation		3:00					03:00	
Besprechungsprotokolle in den Anhang	#b8uwve	Link	documentation		0:45					00:45	
Doku finalisieren	#b8uv0g	Link	documentation		29:45	23:00		03:00		03:45	
Poster A0 erstellen	#b0xzth	Link	documentation		1:00			01:00			
Meetings Sprint 8	#bezet6	Link	admin	15:00	8:15	02:45		02:45		02:45	
Varia Sprint 8	#bezxzc	Link	admin	03:00	2:30	01:00		01:30			
Total				**28:00**	**61:45**	**27:45**		**11:45**		**22:15**	
						27:45	~~00:00~~	11:45	~~00:00~~	22:15	~~00:00~~
Restaufwand						11:45	~~-16:00~~	-4:15	~~-16:00~~	6:15	~~-16:00~~

Soll pro Woche
16:00

Tag	Aufwand
admin	10:45
document	48:00
research	0:00
coding	3:00
Total	**61:45**