**Gottfried Wilhelm**
**Leibniz Universität Hannover**
**Fakultät für Elektrotechnik und Informatik**
**Institut für Praktische Informatik**
**Fachgebiet Software Engineering**

# Evaluation von bestehenden Konzepten zur Unterstützung von Code-Verständnis mit Graphvisualisierungen

**Evaluation of existing concepts to support code comprehension with graph visualation**

## Bachelorarbeit

im Studiengang Informatik

von

**Darian Heinz Nicola Prusac**

**Prüfer: Prof. Dr. Kurt Schneider**
**Zweitprüfer: Dr. Jil Ann-Christin Klünder**
**Betreuer: Lukas Nagel, Dr. Jil Ann-Christin Klünder**

**Hannover, 29.05.2021**

# Abstract


## German

Source Code zu verstehen ist essenziell für Entwicklung, Wartung und Erweiterung von Software. Das ist besonders kritisch, weil das Verständnis für den Source Code einen Großteil der Wartungskosten ausmacht. Es gibt bereits viele Ansätze für Tools, welche beim Verständnis des Codes helfen sollen.
Softwareprojekte bestehen jedoch nicht nur aus Source Code Dokumenten, sondern auch aus einem Netz von Verbindungen dieser Dokumente. Diese Verbindungen sind sind Imports von Klassen und Packages und Nutzungen von Interfaces.
Es ist also nicht nur der Code selbst, sondern gerade die Verbindungen der einzelnen Klassen und Interfaces untereinander sind dafür entscheidend, ob der Programmierer den Code des Softwareprojekts und vor allem dessen Sinn versteht.
Um diese Verbindungen leichter erkennbar zu machen, wurden eine Reihe von Tools entwickelt. Genau so ein Tool ist auch der Code Explorer, welcher Klassen und Interfaces nach vorher festgelegten Eigenschaften gruppiert und dessen Verbindungen untereinander als Graphen darstellt.

Im Rahmen dieser Arbeit wird eine Studie durchgeführt, welche ermitteln soll, ob der Code Explorer Entwicklern dabei hilft, sich in ein Softwareprojekt einzulesen. Dazu wurde ein Experiment mit 20 Probanden durchgeführt, welche Fragen zu einer festgelegten Klasse innerhalb eines Softwareprojekts beantworten sollten. Dabei wurden die Probanden in zwei Gruppen aufgeteilt. Eine, welche den Code Explorer zu Verfügung gestellt bekommen hat und eine, die lediglich Notepad++ verwenden durfte.

Mit dieser Studie stellt sich heraus, dass eine starke Tendenz zur Nützlichkeit gegeben ist.

## English

Understanding source code is essential for developing, maintaining and improving software. This is especially critical, because the majority of maintenance cost originates from programmers working to understand source code.
There already are many approaches for tools, which are meant for helping developers understand code.
But a software project does not only consist of source code documents, but also of a net of connections between these documents. Connections can be imports of classes and packages or the use of an interface.
In conclusion developers must not only understand the source code document of one class, but the task of that class to achieve this. For that to happen, the developer has to understand the connections between classes and interfaces.
Many tools were developed to make these connections more accessible. The Code Explorer is such a tool, which groups classes and interfaces by given attributes and

visualizes their connections as a graph.

Within the context of this thesis a study was conducted to investigate whether this tool helps developers to understand the structure of a software project. An experiment with 20 test subjects was conducted.
These subjects had the task to answer questions about a given class within a software project. The paricipants were devided into two test groups. One, which has the Code Explorer to their disposal and one which can only use Notepad++.

# Contents

# 1 Einleitung

## 1.1 Motivation

Für neue Mitglieder eines Softwareentwicklerteams ist es in den allermeisten Fällen sehr schwierig sich in neuen Code einzuarbeiten [7]. Dabei ist dies essenziell, um Software warten, erweitern oder modifizieren zu können. Das resultiert darin, dass das Verständnis von Source Code einen Großteil der Wartungskosten von Software ausmacht [19].
Weiterhin ist es für neue Mitarbeiter beim Bearbeiten und Erstellen von Klassen und Methoden nicht ersichtlich, wie sich Veränderungen auf das Softwareprojekt auswirken, ohne sich die Dokumentation oder alle verwandten Klassen durchzulesen. Zweiteres gestaltet sich deswegen als schwierig, da ohne externe Hilfe nicht erkennbar ist, welche Klassen verwandt zu der betrachteten Klasse sind. Je nach Dokumentationsqualität ist auch das Durchlesen der Dokumentation ein sehr aufwändiger, aber in jedem Fall langwieriger Prozess.

Dabei ist Lesbarkeit und Verständlichkeit eine der wichtigsten Eigenschaften damit ein Programm leicht getestet, modifiziert und gewartet werden kann [4].
Jedoch ist es ohne externe Tools oder einer Dokumentation nicht möglich große Softwareprojekte zu durchdringen.
Wenn beispielsweise in Java Code fremde Klassen für ihre Nutzung meistens importiert werden, können Methoden von Klassen aus dem selben Package oder einem importiertem Package genutzt werden, ohne, dass dies im Dateikopf erkennbar ist. Um also herauszufinden welche andere Klassen in der zu bearbeitenden Klasse genutzt werden, muss sich der Entwickler den ganzen Code der zu bearbeitenden Klasse angeschaut haben. Zusätzlich ist es ohne weitere Hilfsmittel nicht zu erkennen, welche anderen Klassen die aktuelle nutzen. Dabei ist es als Entwickler wichtig den Zusammenhang zwischen den zu bearbeitenden Klassen innerhalb eines Softwareprojekts zu verstehen, um den Sinn des Codes zu verstehen [8].

Um Zusammenhänge zu erkennen und Konzepte zu verstehen, neigen Menschen stark dazu Bilder und Metaphern zu verwenden, da diese einfacher zu verarbeiten sind [18]. Gerade in der Wissenschaft werden Modelle dafür genutzt abstrakte Konzepte, wie beispielsweise Atome, greifbarer zu machen [5]. Selbiges gilt auch in der Informatik, was die große Anzahl an Metaphern in der Terminologie der Informatik zeigt. Begriffe wie "Automaten" werden für Konzepte mathematischer Berechnungen und solche wie "Bäume" für Datenstrukturen verwendet [5].

Damit die Zusammenhänge verschiedener Programmteile in einem Softwareprojekt

übersichtlich dargestellt werden können, wurde der Code Explorer entwickelt. Dieses Tool kann die Abhängigkeiten der Klassen innerhalb eines Softwareprojekts als Graphen darstellen. Da graphische Darstellungen leichter für einen Menschen zu verarbeiten sind, bietet sich ein graphischer Ansatz an [15]. Dabei werden Klassen als Knoten und Dependencies als Kanten dargestellt. Dieser Graph soll einen Überblick über die Zusammenhänge innerhalb des Projektes geben und damit neuen Entwicklern im Team dabei helfen sich einzuarbeiten und Zusammenhänge zu verstehen.

## 1.2 Forschungsziel

Der Code Explorer ist bereits entwickelt worden, dennoch gibt es keine Überprüfung der Nützlichkeit des Tools. Deswegen soll im Rahmen dieser Arbeit in einer Studie bewertet werden, ob und wie sehr der Code Explorer dabei hilft fremden Code zu verstehen, indem er die Struktur der Verbindungen der Klassen innerhalb des Projektes darstellt.
Diese Studie besteht aus zwei Testgruppen: Eine Gruppe arbeitet mit Unterstützung durch den Code Explorer, die zweite als Kontrollgruppe hat keinen Zugriff auf das Tool. Beide Gruppen sollen jeweils Fragen zum Verständnis einer Javaklasse beantworten. Gemessen werden die Zeiten, die für die Beantwortung der jeweiligen Fragen benötigt werden, sowie die Qualität der Antworten.

## 1.3 Aufbau der Arbeit

Das zweite Kapitel beinhaltet die Grundlagen dieser Arbeit, bestehend aus einer Vorstellung der verwendeten Materialien sowie aus den Grundlagen des Studiendesigns. Im darauf folgenden Kapitel 3 werden verwandte Arbeiten vorgestellt.
Kapitel 4 beschäftigt sich mit dem Studiendesign. Dabei werden der Ablauf, der Aufbau und die Durchführung erläutert. Weiterführend werden im fünften Kapitel die Ergebnisse präsentiert, um diese in Kapitel 6 zu interpretieren.
Das Fazit folgt im siebten Kapitel. Zum Schluss werden im achten Kapitel noch Ausblicke auf zukünftige Arbeiten gegeben.
Kapitel 9 beinhaltet den Anhang mit Materialien.

# 2 Grundlagen

## 2.1 Verwendete Materialien

### 2.1.1 Code Explorer

Der Code Explorer ist eine Browseranwendung zur Visualisierung der Topologie eines Softwareprojektes. Die Topologie meint dabei für die restliche Arbeit die Verbindungen der einzelnen Klassen des Softwareprojekts untereinander, welche Klassen also von jeweils anderen Klassen abhängen. Das Projekt wird dafür als Graph dargestellt. Die Knoten des Graphen stellen die Klassen und Interfaces dar, die Kanten stellen dessen Abhängigkeiten dar. In Figur 2.1 ist die Benutzeroberfläche des Code Explorers zu sehen.

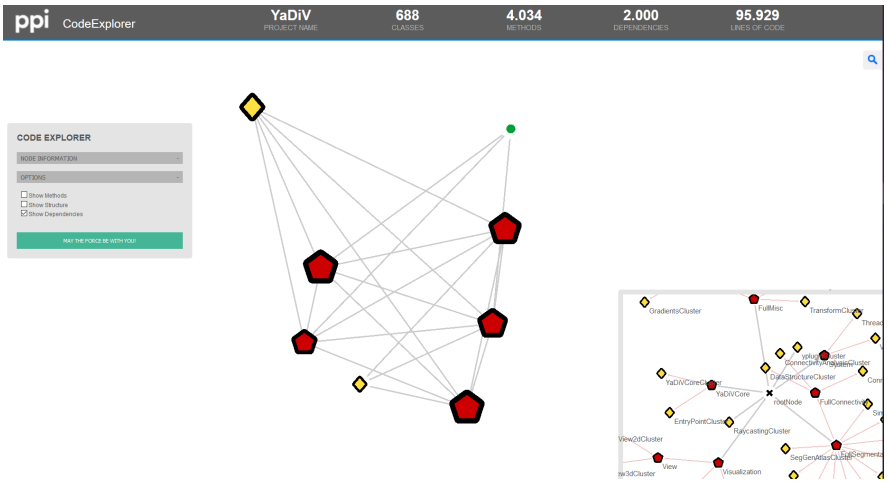



Figure 2.1: UI des Code Explorers

Die Knoten können drei verschiedene Formen haben. Die drei Möglichkeiten sind rote Pentagone, gelbe Rauten und grüne Kreise.
Die grünen Punkte sind dabei die Blätter des Graphen und repräsentieren einzelne Klassen und Interfaces. Die gelben Rauten sind Cluster, welche ausschließlich grüne Punkte zusammenfassen. Die roten Pentagone sind Cluster von Clustern. Sie beinhalten weitere rote Pentagone und/oder gelbe Rauten. Damit der Inhalt eines Clusters sichtbar wird, muss dieser aufgeklappt werden. Dies kann per Doppelklick geschehen oder indem nach Auswahl eines Knotens per Mausklick die Taste "e" betätigt wird. Das sorgt dafür, dass die Knoten, die das Cluster beinhaltet wieder in der Visualisierung auftauchen. Ist ein Knoten bereits ausgeklappt, so kann er mit derselben Methode

wieder eingeklappt werden. Dadurch wird dessen Inhalt wieder unsichtbar und die Knoten, die er beinhaltet, verschwinden wieder aus der Visualisierung. Figur 2.2 zeigt denselben Graphen wie Figur 2.1, jedoch wurden hier ein rotes Cluster und eine dadurch erschienene gelbe Raute ausgeklappt.

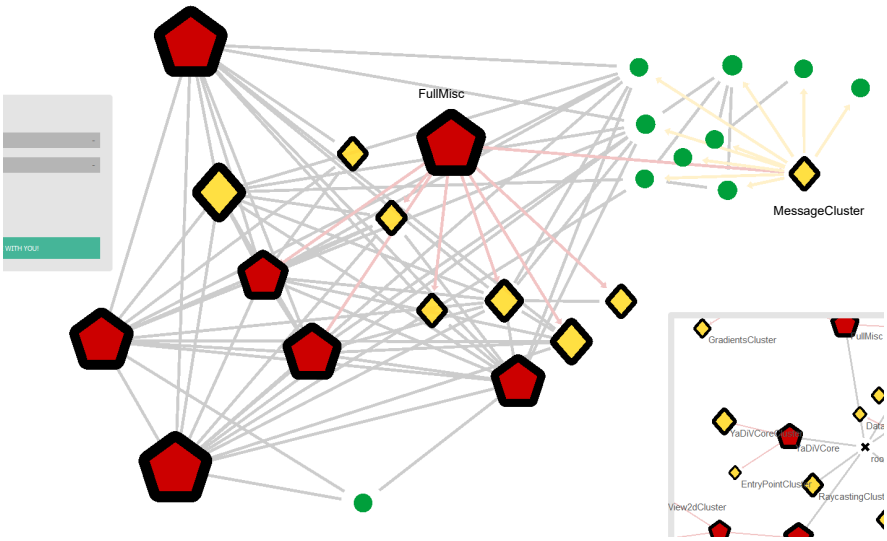



Figure 2.2: UI des Code Explorers nachdem "FullMisc" (Pentagon) und "MessageCluster" (Raute) ausgeklappt wurden.

Welche Klassen und Interfaces jeweils zu welchem Cluster zusammengefasst werden wird von einer Ontologie bestimmt. Dort sind bei jeder Klasse und jedem Cluster Eigenschaften, so genannte Tags, aufgelistet. Knoten mit denselben Tags werden zusammengefasst, wenn das in der Ontologie so notiert ist. Hat ein Knoten mehrere Tags, so ist er Teil mehrerer Cluster.
Für die Darstellung bedeutet das, dass dieser Knoten solange sichtbar ist, bis alle Elternknoten eingeklappt wurden.

Wählt man einen Knoten aus, so färben sich die Verbindungen rot oder blau. Rote Verbindungen zeigen eine Abhängigkeit zu der anderen Klasse an und blaue Verbindungen eine Abhängigkeit der anderen Klasse zu der ausgewählten Klasse. So ist also eine Kante, die, wenn sie bei ausgewähltem Knoten "A" rot und mit "B" verbunden ist, bei ausgewähltem Knoten "B" blau. In Figur 2.3 wurde die Klasse "MessageManager" ausgewählt. Zu sehen sind zwei rote und zwei blaue Verbindungen. Diese zeigen hier also, dass Programmcode aus "YObserver" und "WaitItem" importiert werden und Programmcode nach "YObservable" und "MsgInfo" exportiert wird.

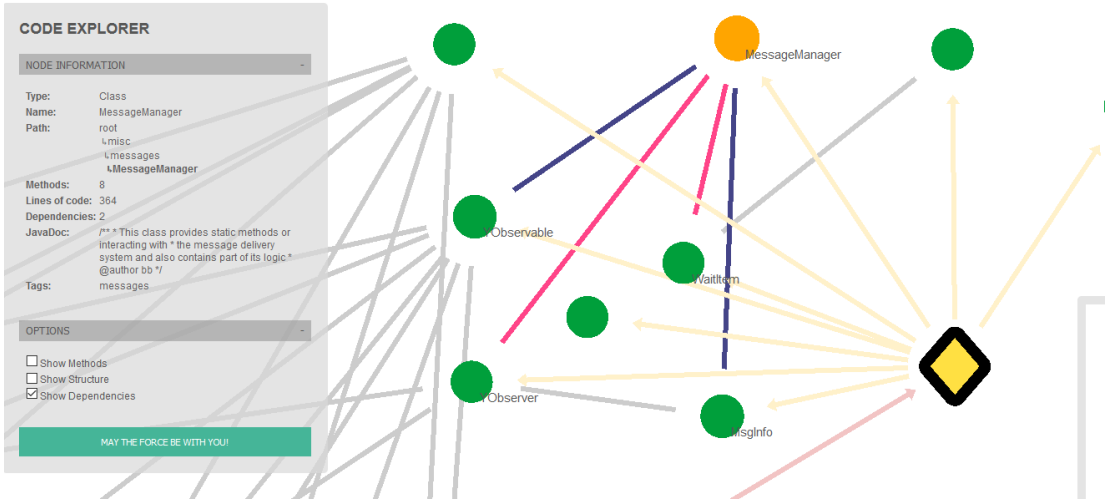



Figure 2.3: Die Klasse "MessageManager" wurde ausgewählt und ihre Verbindungen zu anderen Klassen wurden eingefärbt

Links befindet sich ein Info-Fenster, welches verschiedene Informationen zum ausgewählten Knoten beinhaltet. Egal welcher Typ von Knoten ausgewählt wird, es wird im Info-Fenster immer der Typ des Knotens, der Name des Knotens und der Pfad im Graphen zum Knoten angezeigt. Ist der Knoten grün, also eine Klasse oder ein Interface, dann sind zusätzlich die Anzahl der Methoden, der Wert der Lines of Code-Metrik, die Anzahl der Dependencies, der Javadoc-Kommentar und die Tags dort aufgeführt. Die Anzahl der Dependencies ist die Anzahl der roten Verbindungen, also Klassen, von denen Code importiert wird.

Nach den Node-Informationen kommt ein Options-Bereich, welcher für die Studie nicht relevante Funktionen beinhaltet. Der grüne Button stoppt oder setzt einen Layout-Algorithmus fort, welcher iterativ versucht die Knoten so neu anzuordnen, sodass Anhäufungen vieler Knoten auf engem Raum aufgelöst werden. Dieses Informationsfenster ist in Figur 2.4 dargestellt. Dabei wurde eine Klasse namens "TypeRegistry" ausgewählt.

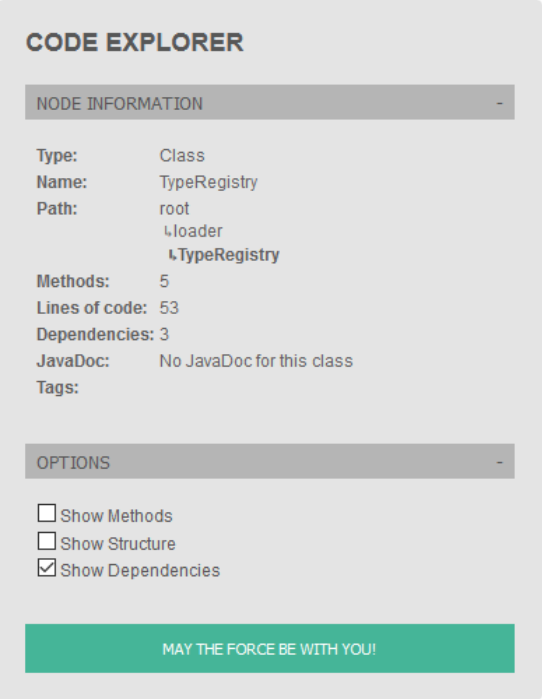



Figure 2.4: Info-Fenster am linken Rand der Code Explorer UI

### 2.1.2 YaDiV

Das Programm, welches in dieser Arbeit und für die Studie verwendet wurde, um vom Code Explorer visualisiert zu werden, ist YaDiV. Auf der Webseite der Entwickler wird es so beschrieben:

"YaDiV ("Yet Another DIcom Viewer") ist ein Programm zur interaktiven Visualisierung von 3D Volumendaten aus dem Bereich der Medizin.
Das Programm kann Daten im DICOM Format (genaugenommen DICOM File Sets) einlesen und enthält u.a. Module zur

- 2D Visualisierung,
- 3D Volumen-Visualisierung (2- und 3D-texturbasiert),
- 3D Segmentierung,
- 3D Segment-Visualisierung,

Unterstützung für stereographische Visualisierung und haptische Eingabegeräte.
Das Programm wird seit 2005 von Dr. Karl-Ingo Friese zusammen mit Studenten des Welfenlab entwickelt und dient derzeit als Plattform für unsere eigene Forschung im Bereich der medizinischen 3D-Datenverarbeitung. Durch die Doktorarbeit von Roman Vlasov wurde das Programm um eine flexibele haptische Schnittstelle erweitert." [6]

Ein Beispiel für diese Visualisierung ist in Figur 2.5 zu finden.

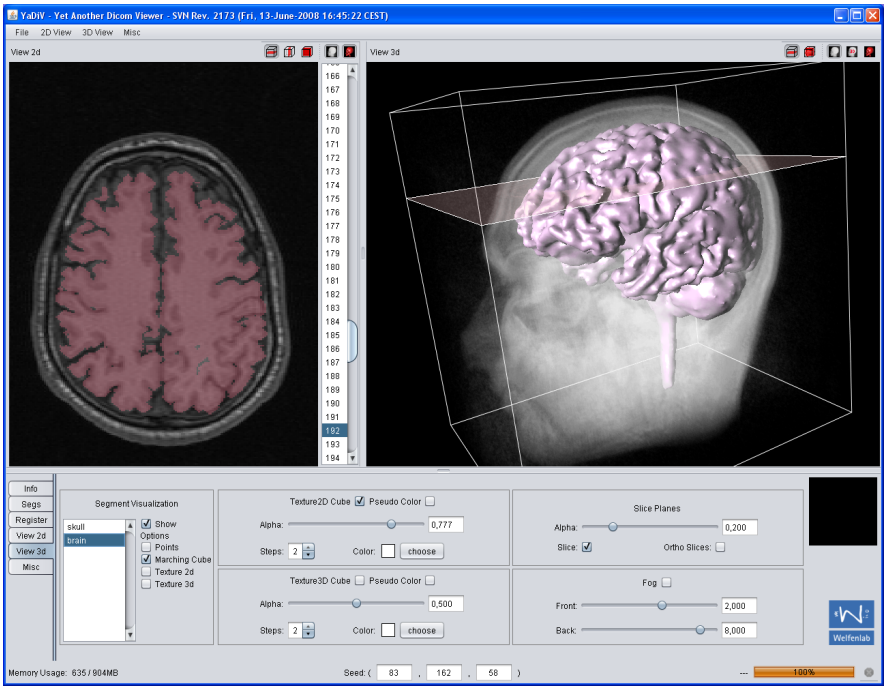



Figure 2.5: YaDiV UI [6]

Die Klasse, welche von den Probanden untersucht werden soll, ist die Klasse MessageManager aus eben diesem Programm.

MessageManager beinhaltet statische Methoden und soll die Nachrichtenüberlieferung zwischen verschiedenen Klassen managen. Auch Teile der Logik des Systems sind dort enthalten. Sie hat eine interne Klasse "MsgInfo" und zusätzlich acht Methoden. Dabei gibt es keinen explizit definierten Konstruktor.

Die interne Klasse "MsgInfo" enthält alle nötigen Informationen, um eine Nachricht zu versenden, sowie Debuginformationen. Sie besteht aus einem Konstruktor und sieben weiteren Methoden.
Im Anhang befinden sich Tabellen zur Übersicht über die beiden Klassen.

## 2.2 Grundlagen Studiendesign

Es gibt verschiedene Arten eine Studie aufzubauen. Grundlegend für das Design einer Studie im Bereich des Software Engineerings ist das Buch "Experimentation in Software Engineering" von Wohlin et al. [23]. Das Buch beschreibt die folgenden vier

verschiedenen Arten von empirischen Studien.

### 2.2.1 Umfrage

Eine Umfrage wird genutzt, um Daten wie Wissen, Verhalten oder Haltungen von, oder über, Menschen zu sammeln. Im Software Engineering wird eine Umfrage zumeist verwendet, wenn die Erfahrung von Personen mit Tools oder Praktiken, welche bereits länger im Einsatz sind, gesammelt werden sollen. Mit den gesammelten Daten soll dann ermittelt werden, ob die Änderungen einen Effekt haben und ob dieser positiv ist. Die gesammelten Informationen werden in eine Form arrangiert, in der sie quantitativ oder qualitativ ausgewertet werden können. Aus den Ergebnissen werden beschreibende und erklärende Schlussfolgerungen gezogen.

### 2.2.2 Fallstudie

Fallstudien werden zur Untersuchung von Projekten, Aktivitäten oder Aufgaben verwendet. Die Daten werden über die gesamte Studiendauer gesammelt. Der Zweck der Datenerhebung ist hier die Beobachtung eines Attributs oder des Zusammenhangs zwischen mehreren Attributen in einem Projekt, einer Aktivität oder Aufgabe. Dabei stützt sich die empirische Untersuchung auf mehrere Beweisquellen, um eine Instanz zu untersuchen. Es wird beispielsweise nicht nur ein einzelnes Softwareprojekt beobachtet, um ein Model der Fehlerrate beim Testvorgang zu erstellen, sondern es werden mehrere untersucht. Der große Unterschied zwischen einer Fallstudie und einem Experiment ist, dass die Fallstudie lediglich observierend durchgeführt wird während sich das Subjekt in seiner realen Umgebung befindet. Experimente hingegen bauen eine kontrollierte Umgebung auf, in welcher das Experiment vollzogen werden soll.

### 2.2.3 Experiment

Bei einem Experiment wird ein Faktor oder eine Variable manipuliert, um zu ermitteln wie sich diese Veränderung auf die zu beobachtende Eigenschaft, oder andere Variable, auswirkt. Auf Zufall basierend werden an mehreren jeweils gleich großen Gruppen von Subjekten jeweils die Variation der einen Variable getestet, während alle anderen Faktoren konstant bleiben. Zum Beispiel kann es zwei Methoden geben eine Aufgabe zu lösen. Hier wird also die Methode je Proband variiert, die Aufgaben und Rahmenbedingungen bleiben aber gleich. Um das gewährleisten zu können, müssen also die Rahmenbedingungen vom Durchführenden kontrolliert werden können, weswegen Experimente in der Regel in einer Laborumgebung durchgeführt werden.
Eine zu stark kontrollierte Umgebung sorgt jedoch für weniger generalisierbare Ergebnisse, da die Umgebung dadurch sehr von der realen Umgebung abweicht. Geben die Durchführenden auf der anderen Seite zu viel Kontrolle über das Experiment auf, um

eine realitätsnahe Umgebung zu schaffen, so riskieren sie, dass die Ergebnisse nicht aus der Änderung der Variable resultieren, sondern aus der unkontrollierten Umgebung.
Die Abwägung des Designs der Umgebung des Experiments ist also wichtig für das Design des Gesamtexperiments.

### 2.2.4 Quasi-Experiment

Die vierte Art der empirischen Studie ist ein Quasi-Experiment, welches im Gegensatz zum Experiment, die Zufälligkeit der Zuweisung des Verfahrens nicht gewährleisten kann, sondern vom Subjekt oder Objekt, also dem Probanden, oder der Methode, selbst abhängt.
Dies geschieht, wenn beispielsweise anzunehmen ist, dass eine komplette Zufälligkeit die Ergebnisse verfälschen könnte, weil die Fähigkeiten der Probanden eine Rolle spielen. Muss der Studiendurchführende also zum Beispiel sicherstellen, dass die Fähigkeitenlevel der beiden Testgruppen ungefähr gleich sind, so handelt es sich um ein Quasi-Experiment.

# 3 Verwandte Arbeiten

## 3.1 Graphische Ansätze

Menschen sind besser dazu in der Lage Daten in visueller, graphisch aufbereiteter Form zu verarbeiten, als mit Tabellen, oder rohen Datensätzen [11]. Diese Daten können natürlich Populationen sein, aber auch Strukturen und dessen Beziehungen untereinander, oder Source Code, fallen darunter.
Texteditoren und IDEs heben Schlüsselwörter im Programmtext hervor, um die Information, dass dieses Schlüsselwort dort verwendet wird und auch, dass es überhaupt eines ist, visuell hervorzuheben. Das ist allerdings bei großen Programmen zur einfachen Verständlichkeit des Codes nicht ausreichend, weswegen Arbeiten wie die von Hendrix et al. sich damit beschäftigen wie Programmierer mit visuellen Hilfestellungen unterstützt werden können [13].
Hendrix et al. [13] haben Source Code an der linken Seite mit einem Control Structure Diagram versehen. Dieses markiert mit Symbolen Variablendeklarationen, Verzweigungen, Schleifen, uvm. Wie in Figur 3.1 zu sehen ist, bezieht sich dies aber nur direkt auf den Source Code.

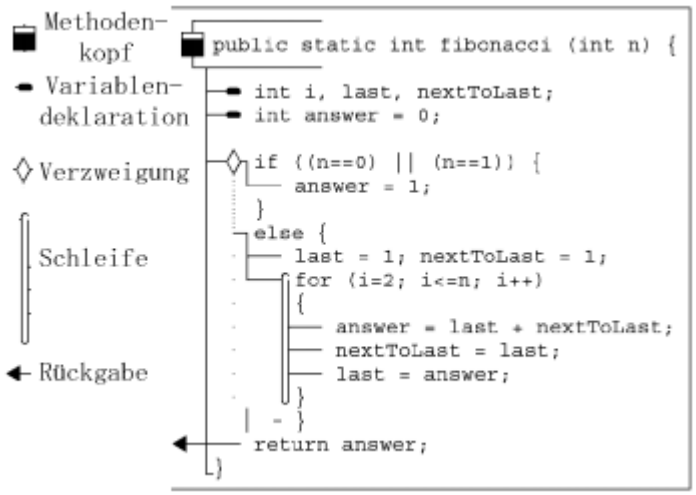



Figure 3.1: Control Structure Diagram im Source Code nach Hendrix et al. [13]

Höhere Programmstrukturen können mit Hendrix' Ansatz allerdings nicht visualisiert werden.
Emden R. Gansner und Stephen C. North schreiben in "Software: practice and experience" [10], dass Graphen geeignete Datenstrukturen sind, um viele Probleme in Computerwissenschaften zu lösen. Genauer gesagt haben Graphen bereits einige

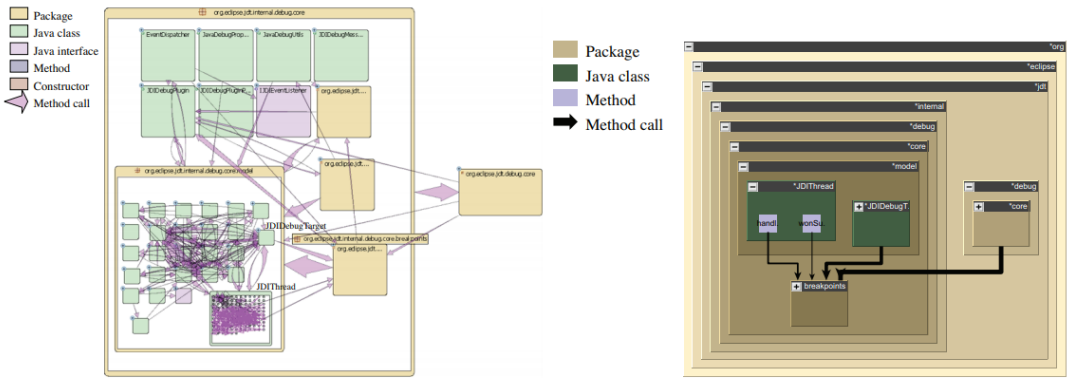
Anwendungen gefunden, darunter im Design und der Analyse von statischen und dynamischen Struktur von Programmen.
Graphen wiederum können also die Topologie eines Softwareprojekts verdeutlichen.
Die Arbeit, auf der die vorliegende aufbaut, ist "An Ontology-Based Approach to Visualize Large Software Graphs" von Lukas Nagel [14] und sie beschäftigt sich mit eben damit, die Topologie von großen Softwareprojekten mittels Graphen leichter verständlich zu machen. Nagels Arbeit fügt die Clusterung von Klassen eines Softwareprojekts nach Tags, also eine Gruppierung nach zugeteilten Eigenschaften, zum Code Explorer hinzu. Der Code Explorer ist ein Tool, welches die Struktur eines Softwareprojekts als Graphen darstellt.
Knoten aus diesem Graphen werden dann möglichst sinnvoll geclustert, sodass die mentale Last des Nutzers reduziert wird. Was die Arbeit nicht abdeckt, ist eine Evaluierung der Effektivität des Gesamtbildes des Code Explorers gegenüber einer Basislinie. Die vorliegende Arbeit soll dies also nachholen.

## 3.2 Andere Tools zu Förderung des Codeverständnisses

Bei Google Scholar gibt es bereits mehr als hier realistisch aufzählbar viele, auf Visualisierung basierende, Arbeiten über Ansätze und Tools zum Codeverständnis zu finden. Dies zeigt, dass visuelle Ansätze ein effektives Mittel zur Unterstützung von Codeverständnis sind.
So sagen auch Alfredo R. Teyseyre and Marcelo R. Campo [22] in ihrer Arbeit, dass der Trend zu stärkerer und vermehrter Visualisierung zur Förderung von Verständnis von Daten, wie Source Code, deutlich zu erkennen ist. Sie beschreiben "Augmented2D views", "Adapted2D views" und "Inherent 3D application domain views", vergleichen Visualisierungen dieser Typen miteinander und analysieren diese auf Schwachstellen in verschiedenen Kategorien, wie zum Beispiel Usability und Integration.
Augmented2D views ist eine Kategorie von zweidimensionalen Visualisierungen, welche aus ästhetischen Gründen in das Dreidimensionale übertragen wurde. Ein Beispiel ist ein dreidimensionales Balkendiagramm.
Adapted 2D views ist eine Kategorie von zweidimensionalen Visualisierungen, welche in das Dreidimensionale übertragen wurde, um zusätzliche Information zu enkodieren. Ein Beispiel ist eine Aufstellung mehrerer Diagramme hintereinander, um die Historie des Diagramms darzustellen.
Inherent 3D application domain views ist eine Kategorie von Anwendungen, welche als dreidimensionale Anwendung konzipiert wurde, wie beispielsweise eine Meatball-Metapher [22].

Dadurch, dass es so viele Ansätze zur Visualisierung von Softwareprojekten gibt, können an dieser Stelle auch noch andere Arbeiten mit verschiedenen Ansätzen genannt werden, die das Selbe erreichen wollen.
Martin Pinzger et al. [17] stellen das Tool DA4Java vor, welches die Topologie eines Softwareprojekts als Graphen darstellt mit Klassen als Knoten und Dependencies als Kanten. Der Graph wird stufenweise erweitert und reduziert. Dadurch wird die men-

tale Last minimiert. Das hat zur Folge, dass der Graph selbst, welcher in Figur 3.2 zu sehen ist, optisch kein Graph im traditionellen Sinne mehr ist, sondern eher an eine Form eines UML-Diagramms erinnert.




(a) Creole (b) DA4Java

Figure 3.2: DA4Java(b) gegenüber der Darstellung desselben Packages mit Creole (a)

Das Ziel dieser Darstellungsart ist nur das aktuell betrachtete Package und Verbindungen zu verwandten Packages auf dem Bildschirm angezeigt zu bekommen. Die Wirksamkeit dessen wurde jedoch in einem separaten von Paper Martin Pinzger et al. [16] bestätigt. Der Nachteil dabei, wenn er auch gewollt ist, besteht darin, dass es keinen Überblick über alle Dependencies gibt.
Dennoch zeigt dies, dass es auch andere Möglichkeiten gibt Dependencies darzustellen als Graphen mit Knoten und Kanten in der UI.

Ein, dem Code Explorer sehr ähnelnder, Ansatz ist ViSE3D von Alexander Fronk et al [9]. ViSE3D stellt die Code-Topologie ebenfalls als Graphen dar - jedoch dreidimensional. Auch hier gibt es, wie in Figur 3.3 zu sehen ist, einen Graphen, der den Code Top-Down darstellt. Dessen Knoten enthalten weitere Teilgraphen, wie in Figur 3.4, in welcher in einen Knoten hineingezoomt wurde, zu erkennen ist.

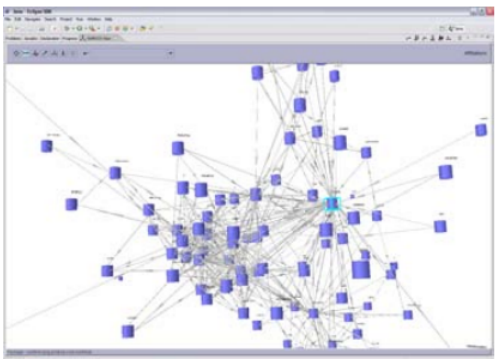


Figure 3.3: ViSE3D UI mit geladenen Graphen

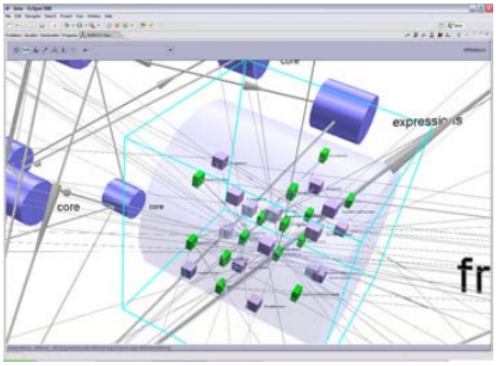



Figure 3.4: ViSE3D Zoom-In in einen Knoten

Während der Benutzer also im Code Explorer Knoten auf- und zuklappt, zoomt er bei ViSE3D in einen Knoten. Der Unterschied in den Knoten ist der, dass der Code Explorer Klassen nach vorher festgelegten Eigenschaften gruppiert, während ViSE3D lediglich die Packages, also die hierarchische Struktur des Programms, anzeigt.

Eine andere Arbeit, welche Dependencies als Graphen darstellt, ist "Comprehending Model Dependencies and Sharing" von Wu et al. [24]. Der Unterschied zum Code Explorer ist, dass das Tool von Wu et al. [24] nicht auf Source Code zugeschnitten ist, sondern auf Beziehungen von Anwendungen in einem Software-Ecosystem, wie Windows.
Figur 3.5 zeigt einen Graphen, den sie auf Basis des Windows Kernels und binärer Instrumentierung generieren, der die Zusammenhänge zwischen einzelnen Programmen darstellt.

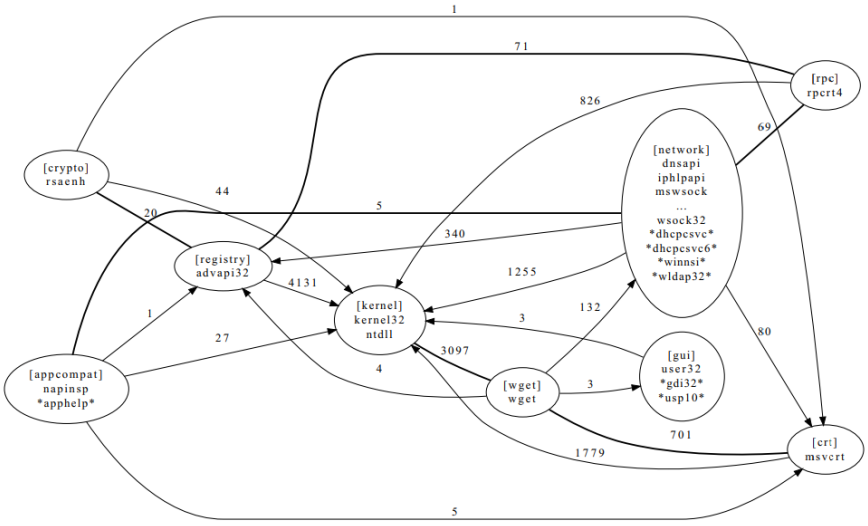



Figure 3.5: Beispiel: DLL dependency graph mit Gruppierungen

Auch hier gibt es eine Gruppierungsfunktion, welche in Figur 3.6 dargestellt wird. In diesem Fall werden Dependencies danach gruppiert, welche Softwares von ihr abhängig sind.

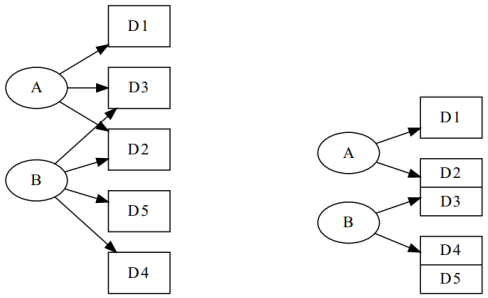



Figure 3.6: Links keine Gruppierung der Dependencies von A und B. Rechts mit Gruppierung der Dependencies von A und B, bestehend aus D1, D2/D3, D4/D5

Die Anwendung von Graphen zur Verdeutlichung von Dependencies ist also bereits in Verwendung. Trotzdem ist das Ziel bei beiden Beziehungen zwischen verschiedenen Modulen darzustellen. Bei Wu et al. [24] sind diese Module verschiedene Programme, beim Code Explorer Klassen.

Geht es jedoch um das Verständnis einzelner Klassen, also des Source Codes selbst und stehen die Dependencies nicht im Hauptfokus, dann werden hauptsächlich andere

Visualisierungsmöglichkeiten verwendet. Bacher et al. [3] benutzen beispielsweise in ihrer Designstudie kompakt dargestellte Bäume. Figur 3.7 und Figur 3.8 zeigen zwei Arten dieser Bäume. Sie sollen innerhalb des Source Code Editors zu sehen sein, um die Struktur des Dokuments darzustellen.

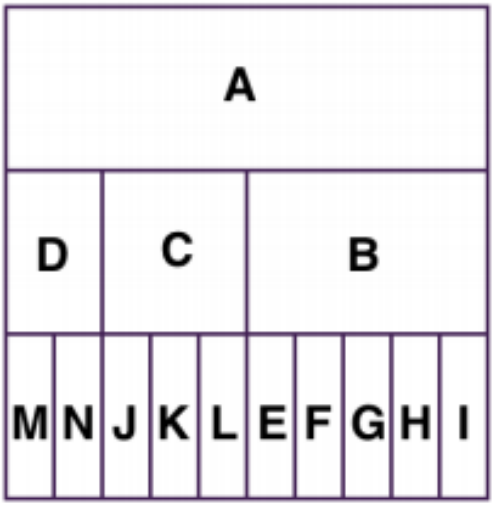



Figure 3.7: Icicle tree, kompakte Darstellung eines Baumes

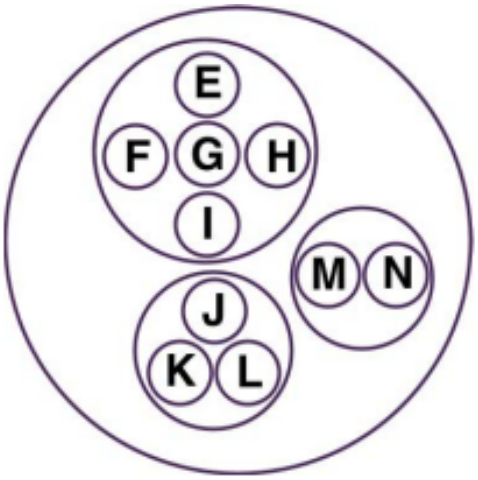



Figure 3.8: Zirkulare Tree-map für die Darstellung von Hierarchien

Diese wandeln dann das Dokument und dessen Funktionen, sowie Verzweigung innerhalb dieser, in einen kompakten Baum um und stellen diese links neben den Code dar, wie in Figur 3.9 gezeigt. Auch die Hierarchie wird mittels einer zirkularen Baumstruktur dargestellt.

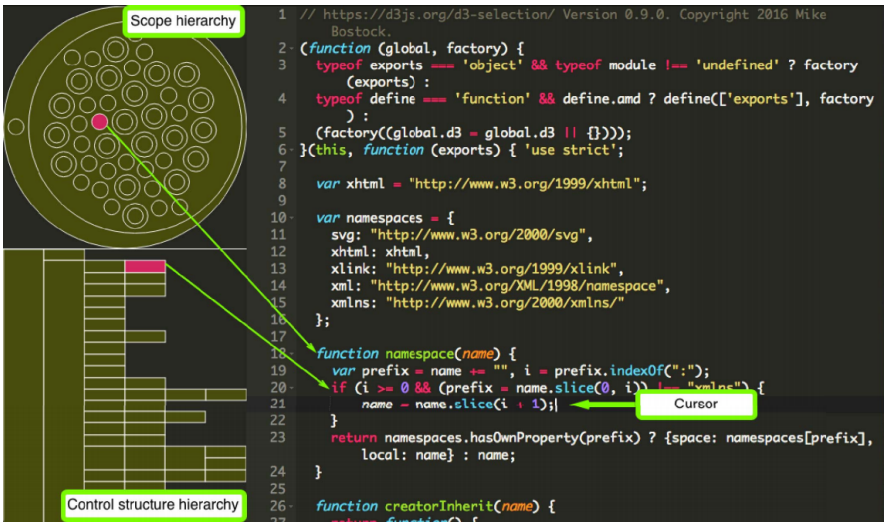



Figure 3.9: Graphen neben dem Code zur Unterstützung des Code Verständnisses

Letzteres bedeutet, dass Graphen nicht die einzige Möglichkeit sind die Beziehungen darzustellen.
Dennoch sind Ansätze, welche nicht mit Graphen arbeiten, hauptsächlich in Arbeiten zu finden, die auf Verständnis eines Code-Dokuments abzielen, wie der von Bacher et al. [3].

## 3.3 Abgrenzung dieser Arbeit

Die meisten hier referenzierten Werke sind verschiedene Ansätze Source Code verständlicher zu machen. Hendrix et al. [13] und Bacher et al. [3] konzentrieren sich dabei darauf den Inhalt des Source Code Dokuments besser verständlich zu gestalten.
Der Code Explorer hingegen konzentriert sich, so wie die Arbeit von Martin Pinzger et al. [17], Fronk et al. [9] und Wu et al. [24], darauf, Dependencies deutlich sichtbar zu machen.
Pinzger et al.s [17] DA4Java ist dabei von der UI her kein Graph mehr, auch wenn die Datenstruktur, die hinter der Visualisierung steckt, ein Graph ist. Dadurch unterscheidet sich diese Arbeit wesentlich von anderen auf Graphen basierenden Tools, wie dem Code Explorer, dem Tool von Wu et al. [24] oder Fronk et al.s [9] ViSE3D.
Dabei ist zu erwähnen, dass Wu et al.s [24] Tool sich nicht auf Softwareprojekte selbst bezieht, sondern auf die Interaktion verschiedener Programme in einem Ecosystem. Damit ist es von der Intention und Funktionsweise dem Code Explorer und ViSE3D sehr ähnlich, erfüllt dennoch einen anderen Zweck.
Der Unterschied zwischen dem Code Explorer und ViSE3D bezieht sich nicht nur darauf, dass der Code Explorer eine zweidimensionale UI hat, während das ViSE3D mit dreidimensionalen Grafiken arbeitet, sondern auch das Clustern einzelner Klassen

nach Tags ist eine Eigenheit des Code Explorers.

Das Paper von Fronk et al. [9] zu ViSE3D beinhaltet keine Evaluation der Konzepte. Wu et al. [24] haben ihr Tool an verschiedenen Programmen getestet und für diese den Dependencygraphen generiert und ausgewertet, jedoch haben sie nicht überprüft wie gut die Probanden diese lesen können.
Die Arbeit von Bacher et al. [3] ist eine Designstudie und daher nicht an Probanden testbar. Deshalb gibt es auch hier keine Evaluation über die Einsetzbarkeit in der Praxis. Pinzger et al. [16] haben in ihrer Arbeit eine von ihnen selbst bereits im Vorfeld durchgeführte Studie zu diesem Tool referenziert.
Die vorliegende Arbeit konzentriert sich jedoch voll und ganz auf die Evaluierung des Code Explorers als Gesamtes gegenüber einer Basislinie. Dafür wird eine Studie in Form eines Experiments durchgeführt, wodurch sie sich nicht nur mit dem evaluierten Tool, sondern auch mit dem Studientyp selbst von der Fallstudie Pinzger et al.s [16] unterscheidet.

Eine weitere zuvor genannte Arbeit ist die von Alfredo R. Teyseyre and Marcelo R. Campo [22]. Teyseyre und Campo geben eine Übersicht über verschiedene Ansätze von dreidimensionaler Softwarevisualisierung und bewerten diese.
Diese Arbeit konzentriert sich hingegen nur auf ein Tool, dem Code Explorer. Zusätzlich besitzt der Code Explorer eine zweidimensionale UI, wodurch sich die vorliegende Arbeit stark von der von Teyseyre und Campo abgrenzt.

# 4 Studiendesign

## 4.1 Forschungsfragen

Wie Nancy J Stone [21] in einem Artikel schreibt, hängt das Design der Studie und dessen Erfolg davon ab, wie gut sie auf den Zweck der Studie zugeschnitten ist.
Der Zweck dieser Studie ist es zu ermitteln, ob und wie hilfreich der Code Explorer ist, um Entwicklern dabei zu helfen sich in ihnen unbekannten Softwareprojekten einzuarbeiten.
Damit stellen sich folgende Fragen über den Code Explorer:

- Wie viel Zeit wird beim Einlesen in das Softwareprojekt durch die Benutzung des Code Explorers, gegenüber Einlesen mit dem Source Code Dokument, gespart?
- Wie sehr hilft der Code Explorer dabei Fehler beim Einlesen zu vermeiden, gegenüber manuellen Lesen des Codes?
- Welche Möglichkeiten zur Förderung des Verständnisses des Softwareprojekts bringt der Code Explorer, welche ohne ein externes Tool nicht erhalten werden können?
- Gibt es Aspekte des Code Explorers, welche beim Einlesen bzw. Verständnis des Codes hinderlich sind?

## 4.2 Auswahl der Studienart

Wie im zweiten Kapitel beschrieben gibt es mehrere Möglichkeiten eine Studie zu designen. Hierbei kann eine Umfrage bereits ausgeschlossen werden, weil dafür die Software bereits länger im Einsatz gewesen sein müsste und sich bei einem noch nicht veröffentlichtem Tool keinerlei Erfahrung mit eben diesem bei Probanden finden lassen kann.

Auch eine Fallstudie beruht darauf lediglich Beobachtungen zu machen und daraus Schlüsse zu ziehen, also beispielsweise Arbeitsabläufe einer Firma vor und nach Einsatz des Code Explorers miteinander zu vergleichen. Eine Fallstudie ist dabei sehr auf das Subjekt selbst in seiner realen Umgebung bezogen. Eine reale Umgebung wäre, wenn ein Entwicklerteam das Programm nutzen würde, ohne als Studiendurchführende einzugreifen. Das bedeutet einen starken Verlust bezüglich der Kontrollierbarkeit der Studie und eine schlechtere Generalisierung der Ergebnisse. Zusätzlich erschwert dies die Probandensuche im Rahmen einer Bachelorarbeit so sehr, dass eine Fallstudie hier nicht durchführbar ist.

Das Experiment hingegen trifft auf den Zweck dieser Studie zu. Bei einem Experiment wird eine einzelne Variable im Ablauf variiert und an verschiedenen Testpersonen unter Erhaltung der anderen Faktoren durchgeführt. Da hier an verschiedenen Probanden der gleiche Ablauf, nur mit zwei verschiedenen Hilfsmitteln getestet werden soll, also genau das, was ein Experiment ausmacht, ist hier das Experiment als Studienschablone zu wählen. Dies wird auch noch dadurch deutlicher, dass das Experiment eine kontrollierte Umgebung erlaubt, um äußere Faktoren möglichst auszuschließen, was dafür sorgt, dass die Ergebnisse besser generalisierbar sind.
Da jedoch bei der Einteilung der Probanden in die Testgruppen darauf geachtet werden muss, dass beide Gruppen ein ähnliches Fähigkeitenlevel haben, ist die Zuweisung der Probanden zu den Gruppen nicht vollständig zufällig. Damit handelt es sich hier um ein Quasi-Experiment.

## 4.3 Material

Da nun erläutert wurde, dass diese Studie ein Quasi-Experiment ist, muss nun geklärt werden, wie dieses aufgebaut ist. Ziel der Arbeit ist es zu überprüfen wie effektiv der Code Explorer ist, um Programmierern das Einsteigen in ein Softwareprojekt zu erleichtern.
Damit sollten die beiden Gruppen von Probanden sich darin unterscheiden, dass sich eine Gruppe mit dem Code Explorer in ein Softwareprojekt einarbeitet und die andere Gruppe sich in selbiges Softwareprojekt ohne das Tool einarbeitet.

Als Softwareprojekt wurde hier das im Grundlagenteil vorgestellte Programm "YaDiV" gewählt, da dieses genügend groß ist, um viele Dependencies zu haben. Gleichzeitig gibt es Teilbereiche, welche isoliert genug sind, sodass sie sich dafür eignen für die Studie verwendet zu werden, ohne, dass dies den zeitlichen Rahmen der Durchführung innerhalb einer viermonatigen Bachelorarbeit übersteigen würde.
Würden Softwareprojekte oder Teilbereiche gewählt werden, welche komplexer sind, würde sich die Suche nach Probanden erschweren und die Durchführung selbst, aber auch Terminfindung, den zeitlichen Rahmen so weit strecken, dass sie den zeitlichen Rahmen der Arbeit übersteigen würde.

Um die Umgebung möglichst neutral und die Ergebnisse unabhängig von der Erfahrung eines Probanden mit einer IDE zu halten, wurde der minimalistische Texteditor Notepad++ zur Darstellung des Source Codes gewählt. Die Wahl der Software, welche dazu genutzt wird den Code selbst zu lesen, wird im Teil "Gefahren für die Legitimität" weiter diskutiert.

## 4.4 Testgruppen

Die Klasse, um die es hier geht, ist die im Grundlagenteil beschriebene Klasse MessageManager.
Um zu ermitteln, ob der Code Explorer beim Verständnis des Codes eines Softwareprojekts hilft, sollen Fragen zu dieser Klasse im Code beantwortet werden.
Die eine Testgruppe benutzt also den Code Explorer, um die Fragen zu beantworten. Diese Gruppe ist Gruppe Eins. Es braucht allerdings noch eine Kontrollgruppe, welche dieses Tool nicht zur Verfügung hat, sondern nur einen Texteditor oder eine IDE benutzt. Diese ist Gruppe Zwei und benutzt Notepad++.

## 4.5 Variablen

### 4.5.1 Abhängige Variablen

Es ergeben sich die folgenden zwei Metriken, die von der Testgruppe abhängen:

- Wie lange der jeweilige Proband benötigt, um die Fragen zu beantworten.
- Wie gut die Qualität / Richtigkeit der Antworten ist.

Diese werden dazu benötigt die Forschungsfragen zu beantworten. Zur Antwortqualität ist zu sagen, dass verschiedene Antworten gleichwertig richtig sein können. Darauf wird im Verlauf dieser Arbeit nochmal eingegangen.

### 4.5.2 Unabhängige Variablen

Es gibt auch einige Variablen, welche nicht von der Testgruppe abhängen sondern vom Probanden.
Diese sind:

- Die Erfahrung des Probanden in der Informatik: Bachelor, Master, Beruf in der IT
- Erfahrung mit Programmiersprachen, besonders Java.
- Vertrautheit mit dem Code Explorer
- Vertrautheit mit YaDiV

Die Erfahrung in der IT ist wichtig, um den Probanden und dessen Ergebnisse einzuordnen. Einen Studienanfänger mit einem Programmierer mit jahrelanger Erfahrung in der Softwareentwicklung zu vergleichen wäre eine Verfälschung der Ergebnisse.
Auch die Erfahrung mit Programmiersprachen hat einen großen Einfluss auf das Ergebnis. Wenn der Proband kaum Erfahrung mit Programmierung und vor allem mit Java oder ähnlichen Sprachen wie C# hat, ist es für diesen wesentlich schwerer, wenn nicht unmöglich, die Fragen des Fragebogens zu beantworten.
Die Vertrautheit mit dem Code Explorer ist dann für das Ergebnis verfälschend, wenn der Proband vertraut mit dem Tool ist und dann dieses bedienen soll. In einem realen Umfeld würde ein neuer Entwickler im Team den Code Explorer noch nicht kennen.
Zuletzt ist noch die Vertrautheit mit YaDiV zu erläutern. Jedoch der einzige Fall, in dem diese von Relevanz wäre, ist, wenn der Proband Entwickler von YaDiV ist und gleichzeitig auch speziell am Package, in welchem MessageManager enthalten ist, gearbeitet hat. Es gab einen einzigen Probanden, welcher an YaDiV entwickelt hat, aber seiner Aussage nach das Messages-Package nicht kannte.

## 4.6 Zusammenfassung des Studienaufbaus

Das Experiment lässt sich mit folgenden Sätzen definieren:
Untersucht werden soll die Wirksamkeit des Code Explorers zur Hilfe des Codeverständnisses neuer Mitglieder eines Softwareentwicklerteams, indem der Code und vor allem dessen Topologie als Graph dargestellt wird. Um dies zu ermitteln wird das Verständnis des Probanden für einen vorgegeben Codeabschnitt, sowie die Zeit, die der Teilnehmer benötigt, um die Fragen zu beantworten, vom Studiendurchführenden erhoben. Die Studie wurde an keinem bestimmten Ort durchgeführt, sondern digital über Teamviewer veranstaltet.

## 4.7 Durchführung

Durch die Coronapandemie bedingt war eine Durchführung in Präsenz nicht möglich, weswegen die Studie digital durchgeführt wurde. Konkret bedeutet dies, dass der Proband den Code Explorer sowie das Notepad++ per Team Viewer [2] bedienen musste. Zur verbalen Kommunikation wurde bis auf bei zwei Teilnehmern Discord genutzt. Ein Proband wollte über Skype kommunizieren, ein anderer über Big Blue Button. Für die Studie macht dies keinen Unterschied.

Bevor die eigentliche Studie begann, wurde jeder Teilnehmer zuvor gebeten eine Einverständniserklärung zu unterschreiben, welche die Verwendung der anonym erhobenen Daten erlaubt.
Dann sollten noch die ersten vier Fragen des Fragebogens beantwortet werden, welche sich ausschließlich auf die Fähigkeiten des Probanden bezogen. Diese Fragen sollten die Erfahrung im Bereich der Informatik, wie Studiendauer oder ob die Person beruflich in diesem Bereich tätig ist, die Erfahrung mit YaDiV und dem Code Explorer, sowie den Kenntnisstand in Java und ähnlichen Programmiersprachen ermitteln. Diese Daten wurden dazu genutzt die Probanden einzuordnen und in der Auswertung dies zu beachten.

Darauf folgte für Testgruppe Eins eine Einführung in den Code Explorer, bei Testgruppe Zwei wurde dieser Schritt jedoch übersprungen.
Jede Testgruppe bekam zusätzlich jeweils einen Hinweis: Gruppe Eins wurde gesagt, dass falls eine Frage nicht mit dem Code Explorer beantwortbar ist, der Proband das bereits offene Notpad++ nutzen darf. Gruppe 2 bekam den Hinweis, dass, wenn eine Frage unbeantwortbar scheint, genau dies eingetragen werden soll.
Daraufhin begann dann die eigentliche Studie und die Beantwortung der Fragen des Fragebogens, die sich auf das Verständnis des Softwareprojektes beziehen. Auf diese Fragen wird in der Arbeit ab jetzt mit "inhaltliche Fragen" referenziert. Während der Beantwortung der inhaltlichen Fragen wurde die Zeit gestoppt. Es wurde je Frage jeweils eine Runde auf der Stoppuhr markiert, sodass die Zeit pro Frage ablesbar ist.
Der Fragebogen selbst ist im Anhang zu finden.

### 4.7.1 Beantwortung des Fragebogens

Die zu beantwortenden, inhaltlichen Fragen beziehen sich auf die Softwareprojekttopologie und sollen auf ein grobes Verständnis der Klasse MessageManager abzielen. Der Fragebogen, mit den im Folgenden erläuterten Fragen, befindet sich im Anhang. Die Fragen wurden so gewählt, dass sie die Informationen erfragen, welche nötig sind, um eine volle Übersicht über MessageManager zu haben. Es soll möglich sein nach Beantwortung der Fragen zu wissen, zu welchen anderen Klassen MessageManager Beziehungen hat und grob welche Aufgabe MessageManager erfüllt. Die vier Fragen dafür sind:

- **F1:** Wie viele Methoden hat die Klasse MessageManager?
- **F2:** Welche andere Klassen musst du dir eventuell anschauen, um den Code von MessageManager vollständig zu verstehen?
- **F3:** Wenn du MessageManager änderst, auf welche Klassen musst du achten, dass sie immer noch gleich funktionieren?
- **F4:** Welche Methode von MessageManager sendet eine Info-Nachricht vom Nachrichten-Stack?

Das Set-Up wurde so gewählt wie man es einem neuen Entwickler geben würde. So war für beide Gruppen die Klasse MessageManager bereits im Notepad++ geöffnet. Für Gruppe Zwei bestand also die erste Frage daraus alle Methoden zu zählen, während Gruppe Eins den Pfad im Code Explorer zur Klasse MessageManager bekommen hat. Gruppe Eins musste die Klasse somit selbst erst finden bevor sie das Ergebnis auf dem Infofenster ablesen konnte. Die meisten Entwickler nutzen eine IDE, die meistens auch übersichtlich darstellt wie viele oder zumindest welche Methoden eine Klasse hat. Das damit verbundene Problem und der Grund warum keine solche genutzt wurde, ist, dass nicht jeder die gleiche IDE nutzt, sowie dass eine IDE programmiersprachenabhängig ist. Somit ist es nicht möglich zu kontrollieren, ob keine oder alle Probanden wissen, wo diese Information in der IDE zu finden ist.
Dennoch sollte ermittelt werden, wie gut die Klasse im Code Explorer zu finden ist und es sollte eine Frage geben, welche das Infofenster nutzt, da bei praktischer Anwendung des Tools, dies regelmäßige Vorgänge sind, die der Entwickler durchführt. Da die Source Code-Datei ohnehin geöffnet werden muss, ganz unabhängig davon, ob der Code Explorer genutzt wird oder nicht, wurde dieser Schritt übersprungen und die Quelldatei war bereits offen für beide Gruppen.

Daher muss die erste Gruppe die Klasse im Code Explorer suchen, während die zweite Gruppe sich nur das Dokument durchlesen muss. Die Fehleranfälligkeit und der Zeitaufwand den Code zu analysieren sollte also damit verglichen werden, wie fehleranfällig und zeitaufwändig es ist, wenn man die Fragen ohne große mentale Anstrengung mit dem Code Explorer beantworten kann, aber die Klasse, unter Angabe des Pfades zur eben genannten Klasse, erst finden muss.
Das basiert darauf, dass in der Praxis der Entwickler das Dokument öffnet und dann

selbiges Dokument im Code Explorer sucht, um die Informationen über die Klasse, welche nur umständlich zu erhalten sind, ablesen zu können.

Die nächsten beiden Fragen beruhen auf den Kernfunktionen des Code Explorers. Sie beziehen sich darauf zu erkennen, wie die Topologie des Softwareprojekts gestaltet ist. Frage Zwei zielt darauf ab zu erkennen, welche anderen Klassen des Softwareprojekts von MessageManager importiert wurden bzw. Methoden welcher anderen Klassen in MessageManager verwendet wurden, *ohne* diese direkt zu importieren. Im Code Explorer ist dies über die Farbe der Kanten zu sehen, während Testgruppe Zwei diese Frage nur lösen kann, wenn sie sich den Code durchlesen.

Die dritte Frage soll die umgekehrten Abhängigkeiten erfragen. Genauer gesagt soll sie erfragen welche Klassen von MessageManager beeinflusst werden, also Methoden von MessageManager nutzen. Dies ist mit Notepad++ nicht beantwortbar, ohne, dass man jeglichen Code des Projekts liest und soll den größten Einsatzzweck vom Code Explorer zeigen. Testgruppe Eins muss lediglich alle Klassen aufführen, welche eine blaue Verbindung vom MessageManager zur jeweiligen Klasse haben. Ohne dieses Tool wäre eine externe Dokumentation oder eine Funktion einer IDE dafür nötig, falls es solch eine gibt, wenn der Entwickler sich nicht alle Klassen des Projekts durchlesen möchte.

Die letzte Frage ist wiederum nicht mit dem Code Explorer zu beantworten. Bei ordentlich geführten Javadoc-Kommentaren gibt es zwar eine Funktion "Show Methods", mit welcher man diese Frage auch sehr schnell beantworten könnte, aber diese Funktion sollte im Rahmen der Studie nicht genutzt werden, weil sie nicht fehlerfrei ist. Deswegen muss Testgruppe Eins an dieser Stelle auch in den Code schauen.
Somit hat hier Testgruppe Zwei einen Vorteil, da sie sich bereits den Code länger angeschaut hat und deswegen die entsprechende Funktion potentiell schneller finden kann. Es soll ermittelt werden wie groß, oder überhaupt signifikant, die zusätzliche Einarbeitungszeit in den Code, durch Nutzung des Code Explorers, ist.

### Fragen nach der Durchführung

Es gab drei weitere Fragen, welche nach der Durchführung der eigentlichen Studie gefragt wurden. Der Zweck dieser Fragen war die Selbsteinschätzung der Probanden bezüglich der Schwierigkeit der Beantwortung der Fragen und wie herausgefordert sie sich gefühlt haben.
Dies kann eine Tendenz anzeigen, warum die Ergebnisse so sind wie sie sind.
Die drei Fragen sind:

- **F1:** Wie aufwändig war es die Fragen zu beantworten?
- **F2:** Wie (nicht) überladen war der Code Explorer?
- **F3:** Mentale Anstrengung die Fragen zu beantworten:

F1 und F2 haben eine Antwortenskala von Eins bis Fünf, wobei Eins für "gar nicht aufwändig" bzw. "gar nicht überladen" und Fünf für "sehr aufwändig" bzw. "sehr überladen" steht. Eine geringe Skala von Eins bis Fünf reicht für eine grobe Einschätzung aus, ohne dass der Proband mit zu vielen Auswahlmöglichkeiten belastet wird.
Die Skala für F3 besitzt eine Antwortenskala von Eins bis Neun, um ganz genau herausfinden zu können, wie sehr die Eigeneinschätzung zur mentalen Last ist. Hier bedeutet Eins "Fahrradfahren" und Neun "Klausur schreiben".

# 5 Ergebnisse

Im Folgenden sollen die Ergebnisse der Erhebung der beiden Metriken vorgestellt werden. Es wurde die Zeit gestoppt, welche die Probanden jeweils brauchen, um die Fragen zu beantworten. Außerdem sind die Informationen, die die Antworten selbst enthalten, eine sehr wichtige, erhobene Metrik, dessen Ergebnisse hier auch präsentiert werden.

Ein einzelner Proband der Code Only Gruppe ist nicht in die Auswertung mit eingeflossen. Dieser Teilnehmer hat sich nicht auf die Beantwortung der Fragen konzentriert und hat sich beispielsweise für die Beantwortung der Frage nach der Anzahl an Methoden, das Dokument nicht einmal bis zur Hälfte durchgelesen.
Das Verhalten der Teilnehmer kann leider nicht kontrolliert werden und daher hat dieser Proband als einziges keine wertvollen Daten geliefert. Die kurzen Zeiten würden das Ergebnis nur verzerren, da diese daraus resultieren, dass der Fragebogen nicht ernsthaft bearbeitet wurde.

Bevor die Präsentation der Ergebnisse jedoch beginnt, muss gesagt werden, dass es, da es hier um das Verständnis der Topologie rund um MessageManager und MessageManager selbst geht, möglich ist, dass trotz Abweichung von der Musterlösung die Fragen nicht falsch beantwortet wurden. Deswegen ist eine Evaluation, welche lediglich über Abgleich von Musterlösung und gegebener Antwort geschieht, nicht zielführend.
Außerdem soll nun "Klasse X importiert Klasse Y" als "Klasse X verwendet Code wie Methoden oder Variablen aus Klasse Y" definiert werden. "Klasse X exportiert nach Klasse Y" soll als "Klasse Y verwendet Code wie Methoden oder Variablen aus Klasse X" definiert werden.

## 5.1 Benötigte Zeit

In dieser Sektion werden die Zeiten präsentiert, welche die Probanden zur Beantwortung der Fragen benötigt haben. Die genauen einzelnen Zeiten sind tabellarisch in Tabelle 1 für die Code Explorer Gruppe und in Tabelle 2 für die Code Only Gruppe aufgeführt.

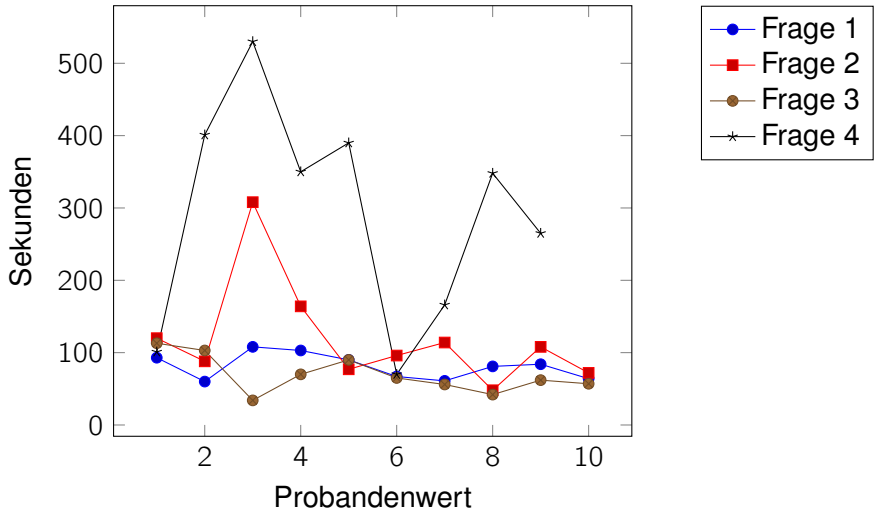
| Wert | Frage 1 | Frage 2 | Frage 3 | Frage 4 |
|---|---|---|---|---|
| W 1 | 93s | 120s | 113s | 101s |
| W 2 | 60s | 88s | 103s | 809s |
| W 3 | 108s | 308s | 34s | 401s |
| W 4 | 103s | 164s | 70s | 530s |
| W 5 | 90s | 77s | 90s | 350s |
| W 6 | 67s | 96s | 65s | 390s |
| W 7 | 61s | 114s | 56s | 70s |
| W 8 | 81s | 48s | 42s | 166s |
| W 9 | 84s | 108s | 62s | 348s |
| W 10 | 64s | 72s | 57s | 265s |
| Mittelwert | 81.1s | 119,5s | 69,2s | 343s |
| Medianwert | 82.5s | 102s | 63,5s | 349s |

Tabelle 1: Tabellarisch aufgelistete zeitliche Ergebnisse der Code Explorer-Gruppe

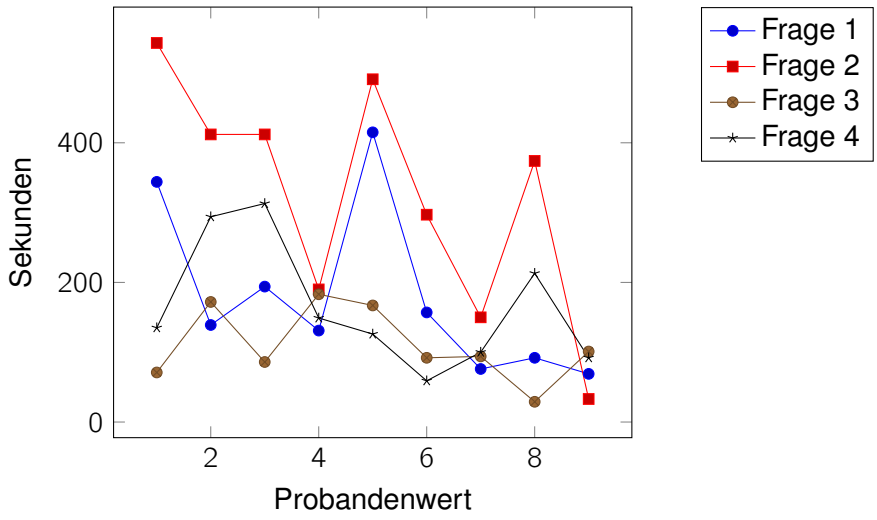
| Wert | Frage 1 | Frage 2 | Frage 3 | Frage 4 |
|---|---|---|---|---|
| W 1 | 344s | 543s | 71s | 135s |
| W 2 | 139s | 412s | 172s | 294s |
| W 3 | 194s | 412s | 86s | 313s |
| W 4 | 131s | 190s | 183s | 149s |
| W 5 | 415s | 491s | 167s | 126s |
| W 6 | 157s | 297s | 92s | 59s |
| W 7 | 76s | 150s | 94s | 100s |
| W 8 | 92s | 374s | 29s | 213s |
| W 9 | 69s | 33s | 101s | 92s |
| Mittelwert | 179,67s | 322,44s | 110.56s | 164.56s |
| Medianwert | 139s | 374s | 94s | 135s |

Tabelle 2: Tabellarisch aufgelistete zeitliche Ergebnisse der Code Only-Gruppe

In jeweils Figur 5.1 und Figur 5.2 sind diese Daten nochmals qualitativ nachvollziehbar.




Figur 5.1: Graphisch dargestellte zeitliche Ergebnisse der Code Explorer-Gruppe




Figur 5.2: Graphisch dargestellte zeitliche Ergebnisse der Code Only-Gruppe

Die Werte, mit denen die beiden Gruppen jedoch miteinander verglichen werden, sind hauptsächlich die jeweiligen Durchschnittswerte. Die Medianwerte sind als Überprüfung der Durchschnittswerte nach Ausreißern zu verwenden. Je weiter die beiden Werte voneinander entfernt sind, desto mehr wurde der Mittelwert von einzelnen Werten beeinflusst.

### 5.1.1 Statistische Signifikanz der Zeiten

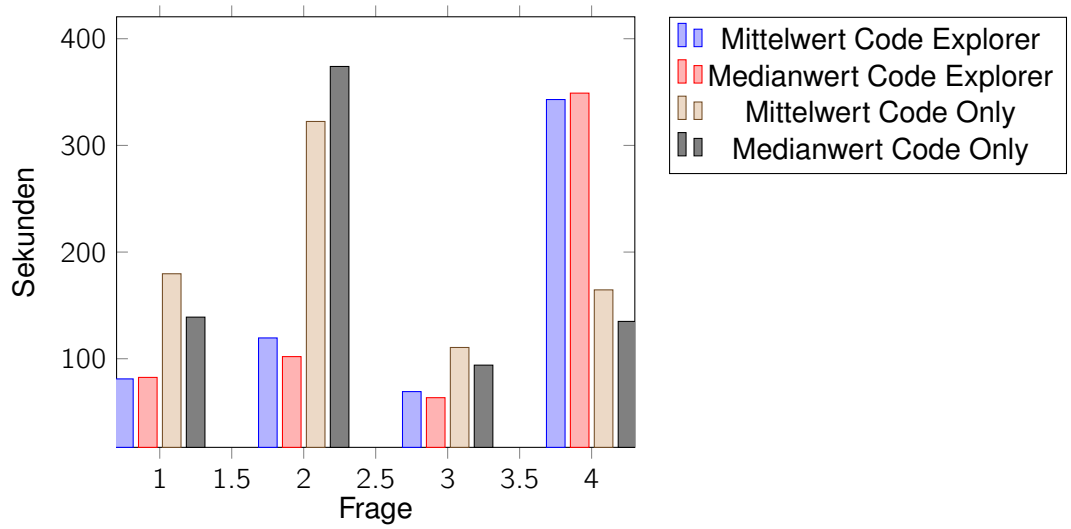
Bevor eine Analyse und Präsentation der Daten geschieht, ist es sinnvoll überhaupt die statistische Signifikanz der Daten zu ermitteln. Zunächst muss dafür eine Überprüfung der Datensets auf eine Normalverteilung durchgeführt werden. Dafür wurde die Webseite statistikguru.de [12] genutzt. Nach Einspeisung jedes Datensetpaares war immer jeweils ein Datenset nicht normalverteilt.

Dabei gilt auch zu beachten, dass nicht jedes Datensetpaar jeweils dieselbe Anzahl an Daten enthält, da die Daten des einen Probanden, wie zuvor erläutert, leider nicht zu gebrauchen sind.
Deswegen wurde der Mann-Whitney-Test gewählt, um die Daten auf statistische Signifikanz zu überprüfen. Dieser Test kann mit Populationen verschiedener Größe umgehen und ist nicht darauf angewiesen, dass die Daten gepaart sind. Um nicht zu riskieren, dass die Implementierung des Tests eine Auswirkung auf das Ergebnis des Tests hat, wurden zwei verschiedene Webseiten genutzt, um den Test durchzuführen. Die erste Webseite ist statskingdom.com [1], die zweite socscistatistics.com [20].
Beide Tests lieferten für jeden der Datenpaare einen sehr ähnlichen p-Wert und kamen immer auf dasselbe Ergebnis bezüglich der Signifikanz.

| Frage | p-Wert | Statistische Signifikanz |
|---|---|---|
| Frage 1 | 0,01596 | Ja |
| | 0,01327 | Ja |
| Frage 2 | 0,01278 | Ja |
| | 0,01272 | Ja |
| Frage 3 | 0,06576 | Nein |
| | 0,06525 | Nein |
| Frage 4 | 0,03752 | Ja |
| | 0,03499 | Ja |

### 5.1.2 Vergleich der Daten

Um die Ergebnisse der Zeiten miteinander zu vergleichen, wurden der Mittel- und Medianwert ermittelt und diese sollen jetzt verglichen werden. Figur 5.3 bietet eine Visualisierung und Übersicht der Median und Durchschnittswerte nach Frage je Gruppe.




Figur 5.3: Mittel- und Medianwerte beider Testgruppen im Vergleich

Zunächst ist anzumerken, dass die jeweiligen Durchschnitts- und Medianwerte bei der Code Explorer Gruppe immer sehr nah bei einander liegen.
Die Code Only Gruppe weist hingegen größere Differenzen zwischen Median- und Durchschnittswert auf.
Während die Code Explorer Gruppe bei Frage zwei diesbezüglich eine Differenz von 17,5 Sekunden aufweist und sich diese sonst nur auf maximal sechs Sekunden beläuft, ist die geringste Differenz zwischen Median- und Durchschnittswert bei der Code Only Gruppe 13,44 Sekunden groß und geht bis zu 51,56 Sekunden bei Frage zwei hoch.

An Figur 5.1 ist zusätzlich qualitativ zu sehen, dass neun von zehn Probanden der Code Explorer Gruppe recht konstant um 80 Sekunden herum benötigt haben, um die ersten drei Fragen zu beantworten. Es gibt jeweils ein Wert für Frage zwei und drei, welche aus dem Raster fallen, für alle anderen Werte, welche zu Frage eins, zwei oder drei gehören, gilt jedoch das eben genannte.
Schaut man sich aber in Figur 5.2 die Ergebnisse der Code Only Gruppe an, ist kein eindeutiger Trend zu beobachten, sondern sehr große Schwankungen in den Zeiten.
Das ändert sich nur bei der letzten Frage. Hier gibt es auch bei der Code Explorer Gruppe große Schwankungen, wobei diese bei der Coder Only Gruppe diesmal geringer ausfallen.

Vergleicht man die Mittelwerte miteinander, ist schnell zu sehen, dass die Code Explorer Gruppe bei Frage Eins bis Drei schneller ist. Wie im vorherigen Kapitel gezeigt, ist dieser Unterschied bei Frage Eins und Zwei auch statistisch signifikant, bei Frage drei jedoch nicht.
Bei der ersten Frage waren Probanden der Code Explorer Gruppe circa doppelt so schnell bei der Beantwortung der Fragen wie Probanden der Code Only Gruppe. Bei Frage zwei waren sie sogar circa drei Mal so schnell.

Dies dreht sich bei Frage Vier allerdings um. Auch hier wurde eine statistische Signifikanz festgestellt und die Code Only Gruppe ist diesmal circa doppelt so schnell gewesen, wie die Code Explorer Gruppe.

## 5.2 Antwortqualität

### 5.2.1 Anzahl der Methoden

Die Antwortenqualität ist bei Gruppe Eins wesentlich besser.
Bei der ersten Frage hat die Code Explorer-Gruppe die Frage "Wie viele Methoden hat MessageManager" ausnahmslos mit Acht und somit richtig beantwortet.
Die Antworten der Probanden der Code-Only-Gruppe variiert bei der ersten Frage bereits stark. Von den Studierenden antworteten drei von sechs mit Acht. Eine Person schrieb Sieben und zwei trugen 17 ein. Bei den Probanden mit einer Tätigkeit in der IT antworteten zwei mit Acht und ein Proband mit Sieben. Die Code Only-Gruppe hatte also zwei Leute, welche sich um eine Methode verzählt oder diese nicht gesehen haben und zwei Probanden, welche die Methoden der internen Klasse mitgezählt haben. Dabei ist zu sagen, dass dies bei einem Teilnehmer der Fall ist, da die interne Klasse nicht als solche erkannt wurde. Ob der andere Proband diese Methoden mitgezählt hat, da er die Klasse nicht erkannt hat, oder weil er MsgInfo als Teil von MessageManager betrachtet hat, ist unbekannt.

Antworten, welche die Probanden auf die erste Frage gegeben haben:

| Gruppe | Antwort | Häufigkeit der gegebenen Antwort |
|---|---|---|
| **Code Explorer** | 8 | 10/10 |
| **Code Only** | 8 | 3/6 Studierende, 2/3 mit IT-Tätigkeit |
| | 17 | 2/6 Studierende, 0/3 mit IT-Tätigkeit |
| | 7 | 1/6 Studierende, 1/3 mit IT-Tätigkeit |

### 5.2.2 Importierte Klassen

Hier ging es darum wie viele Klassen MessageManager importiert. Bei der Code Explorer Gruppe wurde eine Aufzählung der importierten Klassen "YObserver, WaitItem" erwartet. Da diese Frage aber so formuliert wurde, dass es Probanden in der Code Explorer-Gruppe gab, die gesagt haben, dass es auch sinnvoll ist sich die Klassen anzuschauen, in die MessageManger exportiert, wäre es nicht richtig die Antwort "YObserver, WaitItem, YObservable, MsgInfo" als falsch zu markieren, solange dem Teilnehmer der Unterschied zwischen exportierten und importierten Klassen bewusst war. Dies war bei jedem Probanden der Fall, weswegen jede Antwort aller Probanden

der Code Explorer-Gruppe als richtig zu bewerten ist.

Die Bewertung der Code Only Gruppe ist etwas schwieriger. Erwartet wurde mindestens die Antwort: "YObserver, WaitItem, MsgInfo". Der Grund warum MsgInfo hier zusätzlich erwartet wurde, ist der, dass diese Klasse zwar nicht im Code Explorer als importierte Klasse markiert ist, aber in der Quelldatei, welche der Code Explorer einliest, als solche aufgeführt ist. Das Problem, welches das Tool hier hat, ist, dass MsgInfo sowohl eine importierte, als auch eine Klasse ist, in die exportiert wird. Die Kante kann aber nur eine Farbe haben, weswegen die erste Farbe von der zweiten überschrieben wird. Dies betrifft jedoch nicht die Code Only Gruppe, weshalb diese "MsgInfo" mit aufführen sollte.

Da MsgInfo eine interne Klasse ist, wurde diese auch von sechs der neun Probanden aufgeführt. Dennoch variieren die Antworten so stark, dass es bei neun Teilnehmern sechs verschiedene Antworten gibt, mit unter anderem mit Klassen, die nur weitergereicht werden. Es fehlen bei jeder Antwort aber immer mindestens eine Klasse, die wirklich verwendet wird. Beispielsweise wurde "WaitItem" nur von zwei Probanden aufgezählt.
Der Code Explorer zählt Klassen, die zwar in Methodenköpfen sind, dann aber nicht verwendet werden, außer um den Konstruktor einer anderen Klasse wie "MsgInfo" zu füllen; nicht als Import. Dies kann daher sinnvoll sein, da diese Klassen den Message Manger selbst nicht beeinflussen. Solange sie existiert, kann MesageManager das Objekt der Klasse weiterreichen, egal was hinter der Klasse steckt.

Dass diese durchgeschleusten Klassen trotzdem genannt wurden, sollte jedoch nicht als absolut falsch betrachtet werden, denn wenn ein Programmierer komplett verstehen will, was MessageManager tut, dann wäre es auch sinnvoll zu wissen, was MessageManager eigentlich genau weiterreicht.
Die Fragestellung kann also auch so interpretiert werden, dass diese Klassen auch zur Antwort gehören. Dennoch sollten die wichtigsten Klassen genannt werden, welche nie alle gleichzeitig aufgeführt wurden.

Antworten, welche die Probanden auf die zweite Frage gegeben haben:

| Gruppe | Antwort | Häufigkeit der gegebenen Antwort |
|---|---|---|
| **Code Explorer** | YObserver, WaitItem, YObservable, MsgInfo | 3/7 Studierende, 0/3 mit IT-Tätigkeit |
| | YObserver, WaitItem | 4/7 Studierende, 3/3 mit IT-Tätigkeit |
| **Code Only** | Observable; Message; YObserver | 2/6 Studierende, 1/3 mit IT-Tätigkeit |
| | MsgInfo | 1/6 Studierende, 0/3 mit IT-Tätigkeit |
| | MsgInfo, YObservable, Message, YObserve | 1/6 Studierende, 0/3 mit IT-Tätigkeit |
| | MsgInfo, Message | 1/6 Studierende, 0/3 mit IT-Tätigkeit |
| | MsgInfo, YObservable | 0/6 Studierende, 1/3 mit IT-Tätigkeit |
| | WaitItem, MsgInfo | 1/6 Studierende, 0/3 mit IT-Tätigkeit |
| | WaitItem, Message | 0/6 Studierende, 1/3 mit IT-Tätigkeit |

### 5.2.3 Exportierte Klassen

Mit der dritten Frage, in welche Klassen MessageManager exportiert, sollte sichergegangen werden, dass die Probanden der Code Explorer Gruppe den Unterschied zwischen importierten Klassen und Klassen, in die exportiert wird, erkennen. Das Ergebnis dessen ist, dass alle Probanden dies erkannt und mit den beiden Klassen "YObservable, MsgInfo" geantwortet haben, in die exportiert wird.
Für die Code Only Gruppe war die dritte Frage nicht beantwortbar.
Die Teilnehmer hätten jede andere Klasse des Softwareprojekts öffnen und durchlesen müssen, um zu schauen ob MessageManager in einer dieser Klassen verwendet wird.
Stattdessen sollten die Probanden erkennen, dass dies nicht geht und "nicht beantwortbar" antworten. Sechs von neun Probanden haben jedoch mit "MsgInfo" geantwortet, was nicht falsch, sondern richtig ist, da dies eine interne Klasse ist, die direkt von MessageManager abhängt und in die somit exportiert wird.
"YObservable" wurde jedoch von keinem genannt, da diese nicht in MessageManager selbst aufgeführt ist.

Antworten, welche die Probanden auf die dritte Frage gegeben haben:

| Gruppe | Antwort | Häufigkeit der gegebenen Antwort |
|---|---|---|
| **Code Explorer** | YObservable, MsgInfo | 10/10 |
| | | |
| **Code Only** | MsgInfo | 4/6 Studierende, 2/3 mit IT-Tätigkeit |
| | Nicht beantwortbar / alle genutzten Klassen | 2/6 Studierende, 1/3 mit IT-Tätigkeit |

### 5.2.4 Methoden mit gegebenen Zweck finden

Zuletzt sollte die Methode "deliver()" als Methode, welche eine Nachricht vom Nachrichtenstack sendet, erkannt werden.
Bezüglich der Antwortqualität ist zu sagen, dass hier kaum ein Unterschied zu merken ist. Während drei von zehn Teilnehmern "SimplyDeliver()" geantwortet haben, war es bei der Code Only Gruppe nur einer von neun. SimplyDeviler ist tatsächlich die Funktion, die das Gefragte tut, jedoch ist sie Teil von der internen Klasse "MsgInfo" und nicht "MessageManager" selbst. Diese Methode wird zwar von "deliver()" aufgerufen, welche die richtige Antwort ist, aber die Antwort "SimplyDeliver()" lässt darauf schließen, dass nicht erkannt wurde, dass diese zu einer internen Klasse gehört, also die Struktur der Klasse nicht vollständig durchdrungen wurde.

Der Code Explorer betrachtet interne Klassen als eigene Klassen, weswegen es möglich ist, dass diese drei Probanden der Code Explorer Gruppe die interne Klasse deswegen nicht als solche erkannt haben, da sie MsgInfo in einem anderen Dokument erwartet haben. Dazu kann gesagt werden, dass alle drei Probanden Studierende sind. Einer der Probanden mit Beruf im Bereich der IT hat die interne Klasse auch nicht erkannt. Die Rate der Studierenden, welche die Klasse nicht erkannt haben, ist damit jedoch höher, als bei in der IT arbeitenden Teilnehmern.
Die restlichen Studierenden haben alle "deliver()" geantwortet. Die berufstätige Person, welche die interne Klasse nicht erkannte, hat mit der Methode "waitFor()" der internen Klasse geantwortet. Die anderen beiden Probanden mit Beruf in der IT haben mit "deliver()" und "schedule()" geantwortet. "schedule" ist zwar eine Methode von MessageManager, aber wie der Name suggeriert, wird dort nur die Queue an Nachrichten verwaltet, aber keine Nachricht versendet.
Insgesamt lässt sich zur Code Explorer Gruppe sagen, dass neun von zehn Probanden eine Methode mit der Funktion eine Nachricht zu versenden auswählten, also eine Methode mit richtigem Inhalt. Dafür haben vier von zehn die interne Klasse nicht als solche erkannt.

Nur eine Person der Code Only Gruppe konnte die interne Klasse nicht erkennen und antwortete "SimplyDeliver()". Eine Person hatte nicht die Java-Fähigkeiten, die Frage richtig zu beantworten und wurde deswegen aus der Bewertung genommen.
Alle anderen Probanden antworteten entweder "deliver()" oder "schedule()". Interessanterweise schrieben alle Studierenden "deliver()" und alle Probanden mit Beruf in der IT "schedule()". Nur ein Studierender antwortete "async: schedule, sync: deliver". Da "schedule()" nur die Nachrichtenqueue verwaltet, dass nur vier von acht Probanden eine Methode mit der richtigen Funktion gewählt haben und drei von acht Personen eine mit einer ganz anderen Funktion. Der letzte Teilnehmer ist der, der "async: schedule, sync: deliver" schrieb. Auch wenn die Antwort nicht ganz richtig ist, wies die Person während der Durchführung ein gutes Verständnis von MessageManager als Ganzes auf.
Insgesamt lässt sich zur Code Only Gruppe sagen, dass nur vier von acht Probanden eine Methode mit der Funktion eine Nachricht zu versenden auswählten, also eine

Methode mit richtigem Inhalt. Dafür hat auch nur eine Person von acht die interne Klasse nicht als solche erkannt und hat somit die Struktur der Klasse nicht durchdrungen.

Antworten, welche die Probanden auf die vierte Frage gegeben haben:

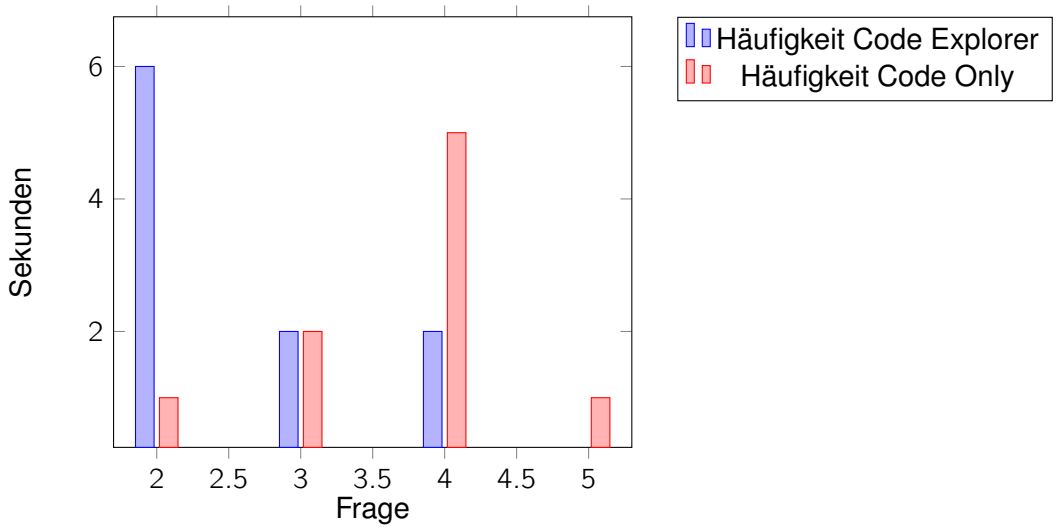
| Gruppe | Antwort | Häufigkeit der gegebenen Antwort |
|---|---|---|
| **Code Explorer** | SimplyDeliver() | 3/7 Studierende, 0/3 mit IT-Tatigkeit |
| | deliver() | 4/7 Studierende, 1/3 mit IT-Tätigkeit |
| | schedule() | 0/7 Studierende, 1/3 mit IT-Tätigkeit |
| | waitfor() | 0/7 Studierende, 1/3 mit IT-Tätigkeit |
| **Code Only** | SimplyDeliver() | 1/6 Studierende, 0/3 mit IT-Tatigkeit |
| | deliver() | 3/6 Studierende, 0/3 mit IT-Tätigkeit |
| | schedule() | 0/6 Studierende, 3/3 mit IT-Tätigkeit |
| | async: schedule, sync: deliver | 1/6 Studierende, 0/3 mit IT-Tätigkeit |
| | Konnte nicht antworten | 1/6 Studierende, 0/3 mit IT-Tätigkeit |

## 5.3 Fragen zur Selbsteinschätzung

### 5.3.1 Schwierigkeit der Fragen und mentale Last

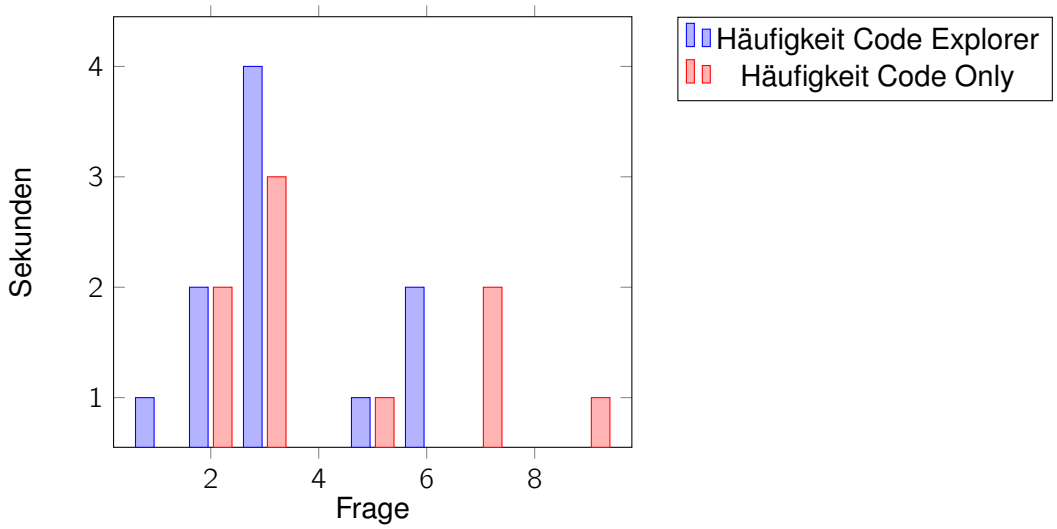
Zunächst sollen die Frage nach der Schwierigkeit der Beantwortung der Fragen und die Frage der mentalen Last des Probanden nach dessen Eigeneinschätzung ausgewertet werden, da diese beiden Fragen von jeder Gruppe beantwortet werden konnten.

Bezüglich der Einschätzung der Schwierigkeit die Fragen zu beantworten, kann gesagt werden, dass die Code Explorer Gruppe diese als wesentlich leichter einschätzten als die Code Only Gruppe.
Während die Code Explorer Gruppe dazu tendierte die Zwei anzukreuzen, war es in der Code Only Gruppe die Vier, welche am öftesten angekreuzt wurde, wie es Figur 5.4 zeigt.




Figur 5.4: Häufigkeitsverteilung der Antworten bei der Frage nach der Schwierigkeit, die Fragen zu beantworten

Bei der Frage nach der gefühlten mentalen Last, sieht das Ergebnis anders aus.
Figur 5.5 stellt die Antwortenhäufigkeiten dar und dort lässt sich kein eindeutiger Trend sehen.
Dadurch dass die maximale, empfundene Last bei der Code Explorer Gruppe bei maximal Sechs liegt, während die Code Only Gruppe drei Mal eine Antwort von Sieben oder höher angegeben hat und gleichzeitig die Antworten der Code Explorer Gruppe sich bei Zwei bis Drei häufen, lässt sich eine leichte Tendenz erkennen. Jedoch ist der Unterschied nicht groß genug, um eindeutige Aussagen zu treffen.




Figur 5.5: Häufigkeitsverteilung der Antworten bei der Frage nach der gefühlten mentalen Last

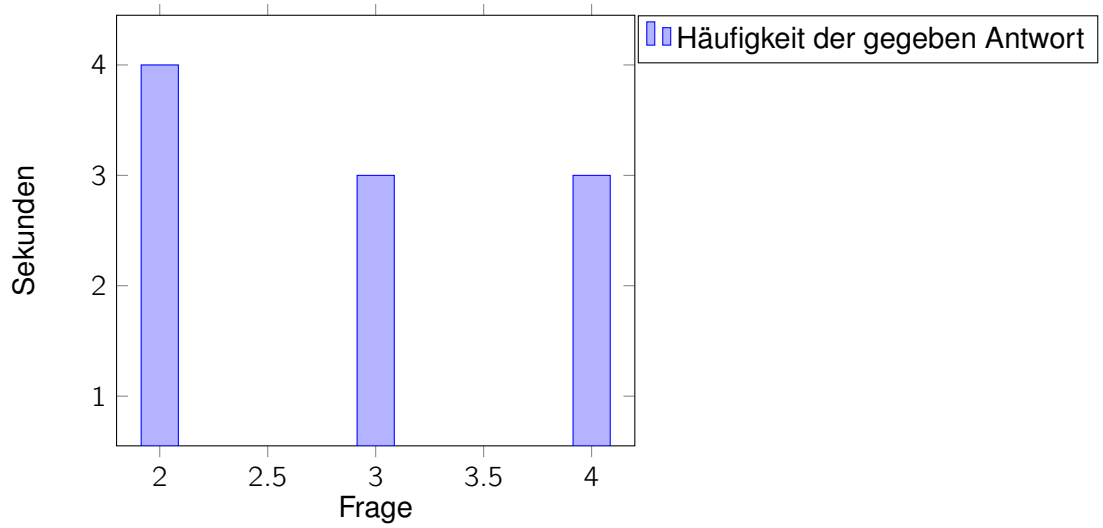
### Statistische Signifikanz

Das eben genannte spiegelt sich auch in der statistischen Signifikanz wieder.
Es wurden wieder dieselben Tests verwendet wie in Kapitel 5.1.1.
F1 steht hier für die Frage nach der Schwierigkeit der Beantwortung der Fragen und F3 für die Selbsteinschätzung der mentalen Last.

| Frage | p-Wert | Statistische Signifikanz |
|---|---|---|
| F1 | 0,02204 | Ja |
| | 0,03078 | Ja |
| | | |
| F3 | 0,3541 | Nein |
| | 0,36812 | Nein |

### Überladenheit des Code Explorers

Es gibt noch eine weitere Frage, dessen Ergebnisse hier aufgeführt werden müssen. Bei der Frage, ob und wie überladen die Probanden der Code Explorer Gruppe den Code Explorer finden, gab es niemanden, der ihn komplett überladen fand, aber auch niemanden, der ihn absolut übersichtlich fand.
Wie die Antwortenhäufigkeitsverteilung in Figur 5.6 zeigt, wurde vier Mal mit Zwei, drei Mal mit Drei und drei Mal mit Vier geantwortet. Es gab auch Wünsche von Probanden, wie zum Beispiel eine Legende, um die Navigation zu vereinfachen.



# 6 Interpretation

## 6.1 Schwankungen in den Zeiten

Im letzten Kapitel wurden die Schwankungen der Zeiten zur Beantwortung der Fragen angesprochen. Es wurde festgestellt, dass diese bei der Beantwortung der ersten drei Fragen bei der Code Explorer Gruppe sehr gering ausfielen und qualitativ um circa 80 Sekunden pendelten. Dann hingegen gab es bei der Kontrollgruppe sehr große Schwankungen.
Dies weist darauf hin, dass der Code Explorer insgesamt dafür sorgt, dass die Zeit, die ein Programmierer für die Ermittlung der Daten, welche erfragt wurden, benötigt nicht mehr von dessen Fähigkeiten abhängt.
Während also die Probanden der Kontrollgruppe je nach Erfahrungs- und Kenntnisstand sehr unterschiedliche Zeiten aufwiesen, scheint dies dank des Tools bei der anderen Gruppe keine Rolle zu spielen.
Zu beobachten war auch, dass dieser Trend nicht mehr zu sehen war, als die Code Explorer Gruppe auch in den Code schauen sollte. Bei der letzten Frage hatte die Zeit, die der jeweilige Proband benötigt hatte, auch wieder eine enge Bindung zum individuellen Teilnehmer.

## 6.2 Anzahl der Methoden

Bei den ersten Frage war die Code Explorer-Gruppe bei der Beantwortung der Fragen signifikant schneller als die Gruppe, welche nur den Code zur Verfügung hatte. Allerdings sind nicht nur die Zeiten besser, sondern auch die Antwortqualität.
Daraus, dass beide Metriken bei der Code Explorer Gruppe klar besser ausgefallen sind, lässt sich schließen, dass das Infofenster ein sinnvoller Ort ist Daten über die Klasse, die sonst bei manueller Ermittlung fehleranfällig sind oder auch nur öfter abgerufen werden müssen, aufzuzählen. Dadurch wird die Komponente "Mensch" herausgenommen, was den Prozess beschleunigt und die Anzahl Fehlerquellen, wie zum Beispiel Verzählen, reduziert.

## 6.3 Importierte Klassen

Die zweite Frage, welche Klassen zum vollständigen Verständnis von MessageManager hilfreich sind, zeigt die größte Stärke des Code Explorers. Nicht nur, dass die Gruppe, welche das Tool nutzen durfte, signifikant schneller war, sondern auch die Anzahl der verschiedenen Antworten beläuft sich lediglich auf Zwei, wobei beide

Antworten richtig sind. Die andere Gruppe hatte sieben verschiedene Antworten, wobei keine einzige die drei Klassen gleichzeitig beinhalten, welche mindestens erwartet wurden.

Dies lässt darauf schließen, dass der Code Explorer nicht nur dafür sorgt, dass der Programmierer Beziehungen zu anderen Klassen schneller erkennt, sondern auch keine übersieht bzw. die Fehleranfälligkeit durch Übersehen von Codezeilen nicht mehr gegeben ist.
Zu beachten gilt dennoch, dass der Fehler, dass die Kante nur eine Farbe haben kann, dafür sorgt, dass trotzdem nicht alle importierten Klassen von der Code Explorer Gruppe erkannt werden konnten. Sobald dieser Fehler behoben wird, ist der Code Explorer aber ein Tool, welches dem Programmierer erlaubt alle Zusammenhänge sehr schnell zu sehen.

## 6.4 Exportierte Klassen

Bevor beide Gruppen miteinander verglichen werden, gilt es auch zu beachten,
dass die Teilnehmer der Code Explorer Gruppe hier statt durchschnittlichen 119,5 Sekunden nun nur noch 69,2 Sekunden gebraucht haben, um die Frage zu beantworten. Dieser starke Unterschied lässt darauf schließen, dass sobald der Benutzer verstanden hat wie der Code Explorer zu bedienen und zu lesen ist, dieser die Softwaretopologie wesentlich schneller lesen kann. Dadurch, dass die Teilnehmer bei Frage Zwei noch nicht so vertraut waren mit dem Code Explorer, wird der Unterschied in den Zeiten bei Frage Zwei noch signifikanter.

Wenn man nun aber beide Gruppe bei der dritten Frage vergleicht, stellt man fest, dass die Code Only Gruppe ähnlich lange für die Beantwortung der dritten Frage gebraucht hat.
Jedoch sind die Zeiten nicht zu vergleichen, da diese Frage für diese Gruppe im Rahmen der Studie nicht beantwortbar war.
Der Sinn dieser Frage war also zu überprüfen, ob die Probanden der Code Explorer Gruppe den Unterschied zwischen importierten Klassen und Klassen, in die exportiert wird, erkennen.
Wie im Ergebnisteil erläutert, ist dies der Fall gewesen.

Insgesamt lässt sich zu dieser Frage also sagen, dass der Code Explorer die Funktion bietet, die Klassen zu sehen, in welche die aktuell betrachtete Klasse exportiert. Diese existiert bei einem Texteditor realistisch nicht. Der Programmierer könnte sich alle anderen Klassen anschauen, um die gewünschte Information zu erhalten, dies ist aber außerhalb der praktikablen Anwendung. Der Code Explorer macht diese Information jedoch innerhalb kurzer Zeit einsehbar.

## 6.5 Methoden mit gegebenen Zweck finden

Bei der letzten Frage sollte auch die Code Explorer Gruppe in den Code selbst hineinschauen. Zunächst fällt auf, dass Gruppe Zwei diesmal signifikant schneller war als Gruppe Eins. Die Code Explorer Gruppe hat zur Beantwortung dieser Frage circa doppelt so lange gebraucht, wie die Code Only Gruppe.

Der zeitliche Unterschied zwischen den beiden Gruppen kann sich damit erklären lassen, dass die Code Only Gruppe bereits länger den Code vor Augen hatte und somit besser darüber Bescheid wusste wie Klasse intern aufgebaut ist. Auch, dass die interne Klasse als solche besser erkannt wurde, kann damit erklärt werden.
Was dieser Vorteil nicht verbessert hat, ist die Antwortqualität. Geht man danach, hat sich diese sogar durch das längere Arbeiten mit dem Code verschlechtert. Zwar wurde die interne Klasse öfter erkannt, aber es wurde mit Methoden geantwortet, welche eine ganz andere Funktion haben, als die gefragte.
Die Code Explorer Gruppe hat zwar länger gebraucht und die interne Klasse weniger oft erkannt und damit die Struktur des Source Code Dokuments nicht so gut durchdrungen, aber dafür mit Methoden geantwortet, welche die gefragte Funktion erfüllen.

Eindeutige Schlüsse darüber zu ziehen, warum das so ist, kann an dieser Stelle nicht gemacht werden. Zu vermuten ist jedoch, dass es anstrengender war die vorherigen Fragen mit dem rohen Code, statt dem Code Explorer, zu beantworten, weswegen die Code Explorer Gruppe geduldiger war und einen frischeren Kopf hatte, um die Fragen zu beantworten.

## 6.6 Fragen zur Selbsteinschätzung

Hier ist es interessant zu beobachten, dass obwohl die Code Explorer Gruppe die Fragen als wesentlich einfacher eingeschätzt haben, sich das in der Selbsteinschätzung zu mentalen Last nicht widerspiegelt.
Das kann mehrere Gründe haben. Der erste Grund wäre, dass der Code Explorer bezüglich der Überladenheit noch verbessert werden könnte. Mehr als die Hälfte der Probanden haben mit einer Drei oder Vier geantwortet und haben noch Raum für Verbesserungen gesehen.
Ein anderer Grund kann sein, dass die Probanden der Code Explorer für die letzte Frage selbst in den Code schauen sollten und somit bei der Selbsteinschätzung der mentalen Last, diese nicht nur auf den Code Explorer bezogen haben.
Der dritte Grund kann eine zu geringe Probandenmenge sein, sodass das Ergebnis eindeutiger ausfallen würde, wenn die Studie mit mehr Teilnehmern durchgeführt worden wäre.

In jedem Fall lässt sich zum aktuellen Zeitpunkt keine Aussage über die mentale Last der Probanden der Code Explorer Gruppe im Vergleich zur Code Only Gruppe treffen.

## 6.7 Gefahren für die Legitimität

### 6.7.1 Legitimität der Schlussfolgerung

Auch wenn die Ergebnisse recht eindeutig waren, bleibt immer noch die Gefahr, dass diese nur durch die geringe Anzahl der 19 Probanden entstanden ist. So ist auch die jeweilige Größe der Populationen zur Bestimmung der statistischen Signifikanz mittels des Mann-Whitney U-Tests sehr gering und könnte sich bei größeren Populationen ändern.

### 6.7.2 Interne Gefahren für die Legitimität

Es besteht die Möglichkeit, dass sich das Ergebnis der Studie mit Auswahl der Software, welche analysiert wird, ändert, wenn diese nicht YaDiV ist. YaDiV ist zwar für die Studie durch die Größe und durch die vielen Packages geeignet gewesen, um im Rahmen dieser Studie als Grundlage zu dienen, aber es gibt noch komplexere Softwareprojekte. Selbst innerhalb YaDiVs gibt es Cluster, welche wesentlich komplexer zu durchblicken sind, welche aber nicht verhältnismäßig für eine Studie im Rahmen einer Bachelorarbeit gewesen wären.
Diese Cluster würden aber in der Realität einem Programmierer begegnen, weswegen das Ergebnis eventuell nicht auf diesen anwendbar ist.

Eine andere Gefahr ist, dass weitere Probanden, wie der im Ergebnisteil ausgeschlossene Teilnehmer, bei der Teilnahme nicht motiviert waren und deshalb Fehler gemacht haben, weil sie die Durchführung schnell hinter sich bringen wollten.

### 6.7.3 Legitimität des Studienkonstruktes

Im Rahmen der Studie wurde der Texteditor Notepad++ verwendet, um den Code von MessageManager darzustellen. Da in der Praxis Programmierer jedoch IDEs benutzen, wäre es besser gewesen den Code Explorer mit einer IDE zu vergleichen.
Demnach wäre es optimal gewesen eine IDE zu haben, welche kein Teilnehmer kennt, um die gleichen Bedingungen für beide Gruppen zu haben, da der Code Explorer ebenfalls jedem Probanden unbekannt ist. Damit hätten beide Gruppen eine kurze Einführung darüber erhalten können welche für die Studie relevanten Funktionen es gibt und dann eine ähnliche fremde Umgebung gehabt. Da dies jedoch nicht möglich ist, wurde der Notepad++ gewählt, da diese Anwendung keine relevanten Funktionen besitzt, die für eine spezifische Programmiersprache gedacht sind, oder IDE-spezifischen Funktionen besitzt.
Es gibt jedoch noch einen weiteren Grund, warum es auch besser sein könnte im ersten Schritt nur einen Texteditor zu nehmen. Dieser ist, dass die Code Explorer Gruppe dann zwei neue Umgebungen hätte, in die sie sich einfinden müsste, die Code Only Gruppe nur eine.

### 6.7.4 Externe Gefahren für die Legitimität

#### Schwierigkeiten bei der Durchführung

Die Durchführung lief allerdings nicht ganz reibungslos. Neben technischen Herausforderungen, wie einer variablen Verzögerungszeit bei der Fremdsteuerung meines Desktops, gab es vereinzelt kurze Verbindungsabbrüche. Auch wenn die Stoppuhr während den Abbrüchen pausiert wurde, könnten die Zeiten leicht von der wahren Zeit abweichen.

#### Bugs

Der Code Explorer selbst ist nicht ganz optimiert.
Ein Bug oder unbekanntes Feature, welches aufgetreten ist, ist, dass unter unbekannten Bedingungen zusätzlich ein weiterer roter, pentagonförmiger, randloser Knoten, aufgetaucht ist. Dieses Pentagon war lediglich mit einer dicken, pfeilförmige Verbindung vom zuletzt betrachteten Knoten ausgehend verbunden. Das Auftreten führte zu kurzer Verwirrung des Probanden.
Es gab Fälle, in denen Probanden mit dem Fenster rumprobiert haben, obwohl im Vorhinein gesagt wurde, diese Optionen nicht zu verwenden. Dadurch, dass die Umschaltung zwischen "Show Methods", "Show Structure" und "Show Dependencies" im linken Fenster sehr klobig ist, führte es in diesen Einzelfällen dazu, dass die angezeigte Ontologie nicht mehr wiederherzustellen war, außer indem die Seite neu geladen wurde.
Deswegen könnte die gestoppte Zeit leicht von der wahren Zeit abweichen.

# 7 Fazit

Am Ende muss jeder Entwickler mit dem Code selbst arbeiten und deswegen bringt ein Tool wenig, wenn es zwar im ersten Moment den Prozess des Verstehens beschleunigt, sich der Entwickler dann aber trotzdem erst einlesen muss und die gewonnene Zeit wieder nichtig wird, weil dies länger dauert als sonst.
Dazu muss angemerkt werden, dass es die zusätzlich Zeit gibt. Die Ergebnisse zur vierten Frage zeigen eindeutig, dass die Code Explorer Gruppe länger gebraucht hat sich die Methoden anzuschauen als die Code Only Gruppe.
Hierzu kann jedoch gesagt werden, dass mit Hilfe des Code Explorers zusammenhängende Klassen wesentlich schneller und akkurater gesehen werden können, dass die zusätzliche Arbeitszeit, die benötigt wird, um sich in den Code einzulesen, wieder wett macht.
Zusätzlich war die Code Only Gruppe zwar schneller bei der vierten Frage, aber inhaltlich hat die Code Explorer Gruppe besser abgeschnitten. Dies lässt sich damit erklären, dass die Probanden der Code Explorer Gruppe einen frischeren Kopf hatte nach Beantwortung der ersten drei Fragen als die Code Only Gruppe.

Insgesamt kann also gesagt werden, dass es eine eindeutige Tendenz dazu gibt, dass der Code Explorer ein Tool ist, welches den Einarbeitungsprozess in ein Softwareprojekt stark vereinfacht und beschleunigt.

# 8 Zukünftige Arbeiten

In dieser Studie wurde nur ein Teilbereich innerhalb des Programms "YaDiV" von den Probanden analysiert. Zudem war die Anzahl der Probanden beschränkt. Es wäre empfehlenswert in zukünftigen Arbeiten dieses Experiment mit mehr Probanden und komplexeren Teilprogrammen zu wiederholen, um die hier festgestellte Tendenz zu stärken oder gegebenenfalls zu widerlegen. Dies war im Rahmen einer Bachelorarbeit nicht möglich.

Des Weiteren wäre eine erneute Evaluation nötig bzw. sinnvoll, sobald der Code Explorer überarbeitet und funktional erweitert wurde.
Es gab während der Durchführung einige Vorschläge der Probanden zur Verbesserung des Code Explorers. Einige Probanden haben sich eine Legende gewünscht, die die Knoten und Farben der Kanten kurz erklärt, andere vermissten eine eindeutige Markierung von bereits ausgeklappten Knoten.
Der Code Explorer hat sehr viel Potential, welches jedoch noch ausgeschöpft werden muss.
Wurden die neuen Funktionen hinzugefügt, sollte diese Studie in einem größeren Rahmen wiederholt werden, um den Effekt der Änderungen zu messen.

Eine Studie, dessen Durchführung außerdem sehr empfehlenswert wäre, wäre eine, die den Unterstützungsfaktor des Code Explorers bei Benutzung einer IDE untersucht. Wie im Teil der Gefahren für die Legitimität erläutert, benutzen Entwickler in der Praxis IDEs, weswegen es sinnvoll wäre zu untersuchen, ob der Code Explorer beim Verständnis der Softwaretopologie hilft, wenn er zusätzlich zur IDE eingesetzt wird.
In dieser Studie sollten dann alle Probanden bereits mit der IDE vertraut sein, um gleiche Bedingungen für alle Teilnehmer zu haben und die Gruppen sollten sich dem entsprechend darin unterscheiden, dass eine der beiden Gruppen den Code Explorer zusätzlich benutzt und die andere nicht.
Dieses Szenario würde nah an eine reale Anwendung des Tools herankommen, weswegen sich die gewonnenen Daten als wertvoll herausstellen dürften.

# 9 Anhang

Übersicht über MessageManager:

| Methodenname | Beschreibung | Argumente | Rückgabewert |
|---|---|---|---|
| run | Initiiert einen Worker-Thread und lässt diesen bis zum Abschluss der Aufgabe laufen | | void |
| schedule | Rekursive Methode zur asynchronen Verarbeitung / zum asynchronen Senden einer Nachricht | YObservable sender, Message message, YObserver recipient, Object recComment, Object msgComment | WaitItem |
| processInSwing | Verbindet die Klasse mit der UI (ruft schedule oder deliver auf) | YObservable sender, Message message, YObserver recipient, Object recComment, Object msgComment | void |
| deliver | Verarbeitet / sendet synchron eine Nachricht | YObservable sender, Message message, YObserver recipient, Object recComment, Object msgComment | void |
| printStackTrace | Gibt die Verarbeitungskette der Nachricht des derzeitigen Threads in der Konsole aus | PrintStream to | void |
| printStackTrace | Gibt Fehle der Verarbeitungskette der Nachricht des derzeitigen Threads in der Konsole aus | | void |
| schedule | Plant die asynchrone Verarbeitung der Nachricht | MsgInfo msgInfo | WaitItem |
| scheduleWorker | Versichert, dass der Worker Thread später im AWT Thread ausgeführt wird | | void |

Übersicht über MsgInfo:

| Methodenname | Beschreibung | Argumente | Rückgabewert |
|---|---|---|---|
| MsgInfo | Konstruktor | YObservable sender2, Message message2, YObserver recipient2, Object recComment2, Object msgComment2, MsgInfo next2 | |
| waitFor | Wartet darauf, dass eine Nachricht verarbeitet wurde und verschickt diese, wenn dies bereits geschah. Kann nur für "urgent deleveries" verwendet werden. Will der Empfänger die Nachricht in der EDT verarbeiten, kann keine "urgent delivery" durchgeführt werden. | boolean urgent | void |
| simplyDeliver | Verschickt die Nachricht an einen Empfänger. Nur für Unbedingte synchrone Aufrufe verwendbar. | | void |
| isEnqueued | Überprüft ob dieses Objekt (this) in der MessageManager-Queue ist | | boolean |
| isQueueHead | Überprüft, ob dieses Objekt (this) ganz vorne in der Verarbeitungsschlange ist | | boolean |
| dequeue | Entfernt dieses Objekt (this) aus der MessageManager-Queue. Diese Queue muss mit diesem Thread synchronisiert sein | | void |
| enqueue | Fügt dieses Objekt (this) der MessageManager-Queue hinzu. Diese Queue muss mit diesem Thread synchronisiert sein | | void |
| printStackTrace | Gibt alle Nachrichten des MsgInfo stacks auf der Konsole aus | PrintStream to | void |

# Fragebogen

## Fragen zur Person:

1. Arbeits-/Studienumfeld:
   - ☐ Beruf in IT
   - ☐ Masterabschluss in Informatik
   - ☐ Bachelorabschluss Informatik
   - ☐ Semester im Informatikstudium: ____
2. Programmiererfahrung
   1. Java:
      - ☐ sehr gut
      - ☐ gut
      - ☐ grundlegend
      - ☐ keine
   2. sonstige ____
      1. ☐ sehr gut
      2. ☐ gut
      3. ☐ grundlegend
      4. ☐ keine
3. Haben Sie Erfahrung mit YaDiV?
   - ☐ Bereits daran entwickelt
   - ☐ Bereits genutzt
   - ☐ Nein
4. Haben Sie Erfahrung mit dem Code Explorer?
   - 1 = nein, keine – 5 = ja, schon oft genutzt
     ☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5

**Fragebogen bei der Durchführung:**

1. Wie viele Methoden hat die Klasse MessageManager?
   - ____
2. Welche andere Klassen musst du dir eventuell anschauen, um den Code von MessageManager vollständig zu verstehen?
   - ____
3. Wenn du MessageManager änderst, auf welche Klassen musst du achten, dass sie immer noch gleich funktionieren?
   - ____
4. Welche Methode von MessageManager sendet eine Info-Nachricht vom Nachrichten-Stack?
   - ____

**Fragen nach der Durchführung:**

1. Wie aufwändig war es die Fragen zu beantworten
   - 1 = gar nicht aufwändig – 5 = sehr aufwändig
     ☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5

2. Wie (nicht) überladen war der Code Explorer?
   - 1 = gar nicht überladen – 5 sehr überladen
     ☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5
3. Mentale Anstrengung die Fragen zu beantworten:
   - 1 Fahrradfahren – 9 Klausur schreiben
     ☐ 1, ☐ 2, ☐ 3, ☐ 4, ☐ 5, ☐ 6, ☐ 7, ☐ 8, ☐ 9

# Bibliography


[1] Mann whitney u test calculator auf statskingdom.com, Zuletzt besucht: 13.05.2021.

[2] TeamViewer AG. Teamviewer, https://www.teamviewer.com/de/download/windows/, Zuletzt besucht: 27.05.2021.

[3] Ivan Bacher, Brian Mac Namee, and John D. Kelleher. On using tree visualisation techniques to support source code comprehension. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 91–95, 2016.

[4] Mitchell H Clifton. A technique for making structured programs more readable. *ACM Sigplan Notices*, 13(4):58–63, 1978.

[5] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[6] Welfenlab Dr Karl-Ingo Friese. Yadiv, http://www.welfenlab.de/yadiv.html, Zuletzt besucht: 06.05.2021.

[7] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 286–28610, 2018.

[8] A.R. Fasolino and G. Visaggio. Improving software comprehension through an automated dependency tracer. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 58–65, 1999.

[9] Alexander Fronk, Armin Bruckhoff, and Michael Kern. 3d visualisation of code structures in java software systems. In *Proceedings of the 2006 ACM symposium on Software visualization*, pages 145–146, 2006.

[10] Emden R Gansner and Stephen C North. An open graph visualization system and its applications to software engineering. *Software: practice and experience*, 30(11):1203–1233, 2000.

[11] Michael J Haass, Andrew T Wilson, Laura E Matzen, and Kristin M Divis. Modeling human comprehension of data visualizations. In *International Conference on Virtual, Augmented and Mixed Reality*, pages 125–134. Springer, 2016.

[12] W.A. Hemmerich. statistikguru.de, https://statistikguru.de/rechner/normalverteilung-rechner.html, Zuletzt besucht: 13.05.2021.

[13] D. Hendrix, J.H. Cross, and S. Maghsoodloo. The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Transactions on Software Engineering*, 28(5):463–477, 2002.

[14] Lukas Nagel. An ontology-based approach to visualize large software graphs, 2020.

[15] Steven Pinker. A theory of graph comprehension. *Artificial intelligence and the future of testing*, pages 73–126, 1990.

[16] M Pinzger, K Gräfenhain, P Knab, and HC Gall. Incremental visual understanding of java source code. *Technical Report*, 2008.

[17] Martin Pinzger, Katja Grafenhain, Patrick Knab, and Harald C. Gall. A tool for visual understanding of source code dependencies. In *2008 16th IEEE International Conference on Program Comprehension*, pages 254–259, 2008.

[18] Glenda C Rakes, Thomas A Rakes, and Lana J Smith. Using visuals to enhance secondary students' reading comprehension of expository texts. *Journal of adolescent & adult literacy*, 39(1):46–54, 1995.

[19] Thomas A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, 1984.

[20] Jeremy Stangroom. Socsistatistics, https://www.socscistatistics.com/tests/mannwhitney/default2.a Zuletzt besucht: 13.05.2021.

[21] Nancy J Stone. Designing effective study environments. *Journal of environmental psychology*, 21(2):179–190, 2001.

[22] Alfredo R. Teyseyre and Marcelo R. Campo. An overview of 3d software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, 2009.

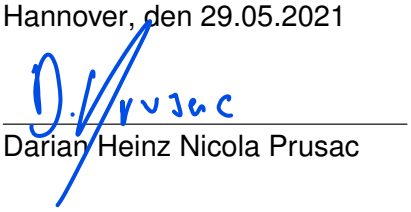
[23] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[24] Yongzheng Wu, Roland H. C. Yap, and Rajiv Ramnath. Comprehending module dependencies and sharing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, page 89–98, New York, NY, USA, 2010. Association for Computing Machinery.

# Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 29.05.2021



Darian Heinz Nicola Prusac