TECHNISCHE UNIVERSITÄT
CHEMNITZ

Einfluss des Dependency Degree auf die Verständlichkeit von Code

Bachelorarbeit

Fakultät für Informatik
Professur Softwaretechnik

4. August 2022

Betreuerinnen: Prof. Janet Siegmund
Elisa Madeleine Hartmann

Zusammenfassung

In meiner Bachelorarbeit befasse ich mich mit dem Thema der Verständlichkeit von Code und wie sich der Datenfluss, operationalisiert durch den Dependency Degree (DepDegree), auf eben diese auswirkt. Hierzu habe ich eine empirische Studie unter Informatik-Studierenden durchgeführt. Die Auswertung stützt sich dabei vor allem auf die beiden Größen Korrektheit der Antworten und Antwortzeit, bezieht zusätzlich allerdings auch Klickmuster und Heatmaps mit ein. Trotz der relativ kleinen Datenbasis lassen sich bereits Tendenzen zu einem Zusammenhang erkennen. In zukünftigen Arbeiten kann dieser Zusammenhang noch einmal genauer untersucht werden.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CSV	comma-seperated values
DepDegree	Dependency Degree
fMRI	functional magentic resonance imaging
IDE	integrated development environment
LOC	lines of code
SLOC	source lines of code
CLOC	comment lines of code
BLOC	blank lines of code
PLOC	physical lines of code
LLOC	logical lines of code
HC	Halstead's complexity

1. Einleitung

Wartung von Code ist ein Thema, welches oft nicht besonders beliebt ist. Es macht oft Aufwand und ist kompliziert. Besonders schwer ist es meistens, wenn man fremden Code bearbeiten und dafür zuerst verstehen muss. Genau dort stellt sich die Frage: was macht Codeverstehen so schwer? Zwar kann eine saubere Dokumentation hilfreich sein, jedoch wird man immer Zeit benötigen, den Fremdcode zu lesen und die Strukturen zu verstehen.
Die gleiche Frage stellt sich bei der Produktion von Code. Gerade für Anfänger:innen stellt es sich oft als eine schwere Tätigkeit heraus. Dabei spielen Sprachkenntnisse zwar oft auch eine wichtige Rolle, es gibt jedoch noch zahlreiche Einflüsse mehr. [Che20, Rob19]

Seit vielen Jahren versucht die Wissenschaft herauszufinden, was Programmieren so schwer macht und warum dies so ist. Über die Jahre wurden so verschiedene Faktoren entdeckt und Metriken entwickelt, die Code anhand bestimmter Eigenschaften beschreiben und bewerten sollen. Zu den Faktoren zählen unter anderem die Linearität von Code, der Grad der Komprimiertheit von Code oder der Entwicklungsansatz bezüglich Rekursion (Top-Down oder Bottom-Up). Zu den gängigsten Metriken zählen *Halstead's Complexity* (Vokabulargröße), McCabe's *cyclomatic complexity* (Kontrollfluss), *Lines of Code* und der *Dependency Degree* (Datenfluss). In neuesten Forschungen [PAP+21] hat sich jedoch gezeigt, dass nicht alle Metriken die konzipierte Aussagekraft besitzen. Eine vielversprechende Metrik ist dabei der DepDegree, welcher den Datenfluss beschreibt, weshalb ich den Einfluss des DepDegree auf das Codeverständnis näher betrachten möchte.

Im folgenden Kapitel 2 betrachte ich daher die theoretischen Grundlagen sowie den vorhandenen Forschungsstand zu diesem Thema. Anschließend erläutere ich in Kapitel 3 meine Vorgehensweisen bei der Umsetzung der Forschung. In Kapitel 4 analysiere ich die Ergebnisse meiner Studie und diskutiere sie in Kapitel 5. In Kapitel 6 fasse ich die wichtigsten Punkte noch einmal zusammen und gehe kurz auf mögliche weitere Forschungen ein.

2. Theoretische Grundlagen

2.1. Metriken

In bisherigen Forschungen wurden verschiedene Komplexitätsmetriken entwickelt, die alle das Ziel verfolgen, Code-Komplexität messbar zu machen, um so eine Aussage über die Schwierigkeiten beim Verstehen von konkretem Code treffen zu können. Damit haben sich mit der Zeit verschiedene Gruppen von Metriken ergeben [PAP+21]:

- Codeumfang
- Umfang des Vokabulars
- kontrollflussbasierte Metriken
- datenflussbasierte Metriken

Codeumfang

Der Codeumfang lässt sich im Allgemeinen am leichtesten bestimmen, indem die Codezeilen gezählt werden. Folgt man Yahya Tashtoush et. al [TAA14], gibt es hierzu jedoch unterschiedliche Varianten. Die einfachste Möglichkeit ist das Zählen aller Codezeilen (LOC). Diese umfasst jedoch nicht nur die „source lines of code“ (SLOC), welche nur ausführbare Codezeilen umfasst, sondern auch die „comment lines of code“ (CLOC), sowie die Leerzeilen („blank lines of code“, BLOC). Die SLOC kann man zudem auch in „physical lines of code“ (PLOC) und „logical lines of code“ (LLOC) unterteilen. Der Unterschied zwischen diesen beiden Zählweisen wird im folgenden Beispiel sichtbar:

```
1 for (int i = 0; i <= 100; i++) System.out.println(i);
```

Physisch betrachtet ist dies eine Codezeile. Logisch betrachtet sind dies zwei Codezeilen, da jedes Statement als einzelne Zeile gezählt wird.
Ein großer Nachteil dieser Metrik ist, dass sie keinerlei inhaltliche Aspekte betrachtet, auf die es beim Verstehen von Code jedoch ankommt.

Vokabularumfang

Neben dem Codeumfang existiert auch der Umfang des Vokabulars als Metrik. Zu den bekanntesten Metriken dieser Gruppe zählt Halstead's complexity (HC) [Hal77, Hal72]. Diese Metrik definiert sechs Grundgrößen:

η_1**:** Zahl der einzelnen Operatoren,

η_2**:** Zahl der einzelnen Operanden,

$f_{1,j}$**:** Vorkommenshäufigkeit der j-häufigsten Operatoren,

$f_{2,j}$**:** Vorkommenshäufigkeit der j-häufigsten Operanden,

N_1**:** Vorkommenshäufigkeit aller Operatoren, wobei gilt: $N_1 = \sum_{j=1}^{j=\eta_1} f_{1,j}$,

N_2**:** Vorkommenshäufigkeit aller Operanden, wobei gilt: $N_2 = \sum_{j=1}^{j=\eta_2} f_{2,j}$,

Zu den Operatoren sei zu sagen, dass diese nach Funktionseinheiten definiert sind. Klammernpaare werden also aufgrund ihrer funktionalen Einheit als einzelner Operator gewertet.
Aus diesen Größen werden weitere Größen wie die Vokabulargröße $\eta = \eta_1 + \eta_2$, also die Zahl aller unterschiedlichen Operatoren und Operanden, sowie die Programmlänge $N = N_1 + N_2$, also die Gesamtzahl aller Verwendungen von Operatoren und Operanden, definiert. Nachfolgend werden daraus weitere Größen berechnet:

Volume: $V = N * log_2\eta$
beschreibt die Zahl der mentalen Entscheidungen, um aus einem Vokabular die nötigen Operatoren und Operanden zu wählen

Difficulty: $D = (\frac{\eta_1}{2}) * (\frac{N_2}{\eta_2})$
ist die Schwierigkeit bezogen auf das Verstehen bzw. die Produktion von Code

Level: $L = \frac{\eta_1^* * \eta_2}{\eta_1 * N_2}$
ist das Verhältnis zwischen der tatsächlichen Programmgröße und der minimal möglichen Programmgröße

Intelligent content: $I = L * V$
beschreibt die inhaltliche Komplexität (unabhängig von der Programmiersprache)

Effort: $E = D * V$
beschreibt abhängig von Volumen und Level den mentalen Aufwand, ein Programm zu verstehen oder zu produzieren, und quantifiziert diesen in Entscheidungseinheiten,

Time: $T = \frac{E}{18}$
definiert die Zeit, welche für das Verstehen bzw. die Produktion benötigt wird abhängig vom Effort und einer Konstante, bspw. Stroud's Number [Hal77, S. 48, S. 52]

Für die Bewertung von Code sind letztendlich die Größen *Difficulty*, *Effort* und *Time* von Bedeutung. Anhand des nachfolgenden Code Snippets möchte ich das einmal kurz demonstrieren.

Listing 2.1: Beispielfunktion

```
1 public String reverseString(String word) {
2
3     String result = "";
4
5     for ( int j = word.length() - 1; j >= 0; j-- )
6         result += word.charAt(j);
7
8     return result;
9 }
```

Bezüglich der Zählweise bei modernen Sprachen existieren unterschiedliche Auffassungen. In meinem Beispiel möchte ich mich an Halstead's Einschätzung halten, dass all jene Symbole als Operatoren gezählt werden, welche nicht als Variablen oder Konstanten angesehen werden und somit keine Operanden sind. [Hal77, S. 7]

Die unterschiedlichen Operatoren aus Listing 2.1 sind demzufolge (sortiert nach ihrem ersten Auftreten): `public`, `String`, `reverseString()`, `( )`, `{ }`, `=`, `" "`, `;`, `for`, `int`, `length()`, `-`, `>=`, `--`, `+=`, `charAt()`, `return`
Daraus ergeben sich $\eta_1 = 17$ und $N_1 = 24$.
Weiterhin besteht das Code Snippet aus den folgenden Operanden: `word`, `result`, `j`, `1`, `0`
Daraus ergeben sich $\eta_2 = 5$ und $N_2 = 9$.
$f_{1,j}$ und $f_{2,j}$ habe ich hier implizit gezählt, da diese Werte im weiteren Verlauf keine weitere Relevanz haben. Weiterhin wird für die Berechnung des Levels der Wert η_1^* benötigt, welcher die minimal nötigen Operatoren beziffert. Laut Halstead ist dieser Wert mindestens 2 – ein Funktionsbezeichner und ein Operator für eine Zuweisung oder Gruppierung. In meinem Beispiel folge ich also dieser Einschätzung und nehme $\eta_1^* = 2$ an.
Die übrigen Werte berechnen sich nun wie folgt:

- $\eta = \eta_1 + \eta_2 = 22$ Elemente
- $N = N_1 + N_2 = 33$ Auswahlen
- $V = N * log_2\eta \approx 147.2$ mentale Vergleiche
- $D = (\frac{\eta_1}{2}) * (\frac{N_2}{\eta_2}) = 15.3$ mentale Unterscheidungen
- $L = \frac{2*\eta_2}{\eta_1 * N_2} \approx 0.1$
- $I = L * V \approx 9.6$

- $E = D * V \approx 2251.6$ grundlegende mentale Unterscheidungen
- $T = \frac{E}{18} \approx 125.1$ s

Laut Halstead benötigt es also rund 15 mentale Unterscheidungen, bestehend aus rund 2252 grundlegenden mentalen Unterscheidungen, um in rund 125 Sekunden das Code Snippet 2.1 zu generieren.

Damit wird ein Vorteil dieser Metrik deutlich: da sich die Metrik allein auf äußere Merkmale stützt, ist sie einfach zu berechnen. Gleichzeitig ist dies jedoch auch ein Nachteil, denn wie ich in Kapitel 2.3 näher beschreiben werde, hat der Kontrollfluss ebenso eine Auswirkung auf die Komplexität von Code.

Kontrollflussbasierte Metriken

Die bisher genannten Gruppen bezogen sich ausschließlich auf äußerliche Elemente. Im Gegensatz dazu beziehen sich kontrollflussbasierte Metriken auch auf inhaltliche Informationen. Eine dieser Metriken ist McCabe's cyclomatic complexity[McC76] (CC). Die CC gibt an, wie viele linear unabhängige Pfade durch ein Programm hindurch existieren. Hierzu wird der Kontrollfluss in einem Graphen dargestellt und anschließend die Komplexität mithilfe $V(G) = E - N + 2$ berechnet, wobei G der Graph, E die Zahl der Kanten (edges) und N die Zahl der Knoten (nodes) ist.
Diese Metrik hat zwar den Vorteil, dass der Einfluss von Verzweigungen und ähnlichen Konstrukten beachtet wird, jedoch hat die Programmlänge keinerlei Einfluss auf das Ergebnis.

Datenflussbasierte Metriken

Um dieses Problem zu lösen, wurden datenflussbasierte Metriken entwickelt. Eine dieser Metriken ist der Dependency Degree (DepDegree). Da dieser Grundlage meiner Arbeit ist, werde ich im folgenden Kapitel näher darauf eingehen.

2.2. Dependency Degree

Der DepDegree beschreibt nach Beyer und Fararooy [BF10] die Zahl der Abhängigkeiten zu anderen Operationen eines Programmes. Mit anderen Worten: der DepDegree gibt an, wie viele Operationen notwendig sind, um die betrachtete Operation auszuwerten. „Operationen“ schließen dabei auch Variableninitialisierungen ein. Die in Listing 2.1 (Seite 13) beispielhaft genannte Funktion besitzt einen DepDegree von 12. Die Grafik 2.1 veranschaulicht hierfür den Datenfluss. Die einzelnen Operationen werden durch die Knoten dargestellt, während die Kanten den Datenfluss selbst darstellen. Für jeden Variablenaufruf erhält man eine Kante. Um nun den DepDegree

zu erhalten, ermittelt man die Zahl der Kanten. Hierbei sei anzumerken, dass Kurzformen wie `j--` in ihrer Langform `j = j - 1` betrachtet werden. Damit ist auch ersichtlich, wie die beiden Schleifen zustande kommen.

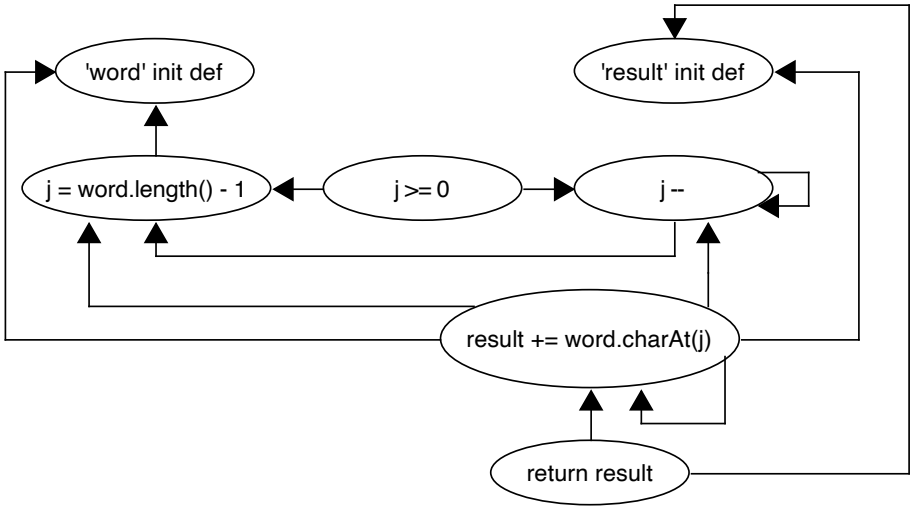

Abb. 2.1.: der Graph visualisiert die Abhängigkeiten für den DepDegree

Wie bereits im vorigen Kapitel erwähnt, gehört der DepDegree zu den datenflussbasierten Metriken. Das bedeutet, dass nicht allein äußere Elemente (wie bei LOC oder HC) ausschlaggebend sind, sondern in erster Linie logische Strukturen. Vergleicht man den DepDegree bspw. mit den LOC, fällt eine komplexere Berechnung auf. Dies liegt daran, dass nicht allein äußere Merkmale gezählt werden, sondern logische Zusammenhänge. Damit geht gleichzeitig auch der Vorteil einher, dass wichtige Aspekte wie Verschachtelungen und Methodenrufe nicht außer Acht gelassen werden und die Metrik somit eine höhere Aussagekraft hat. Mehr dazu jedoch in Kapitel 2.3.

2.3. Verwandte Arbeit

Grundlage aller Metriken ist die Tatsache, dass Codeverstehen eine Tätigkeit ist, welche Hirnaktivität – auch *cognitive load* genannt – erfordert. Je komplexer ein Stück Code dabei ist, desto schwerer ist es, dieses zu verstehen. Jain und Singh [JS20] haben diesen Umstand näher untersucht und einige Faktoren bestimmt, die sich auf das Codeverstehen auswirken. Um ihre Annahmen zu erklären, haben sie dazu auch Code Snippets betrachtet, die sich in dem jeweils betrachteten Aspekt unterschieden und somit auch unterschiedlich auf die Hirnaktivität auswirken sollen. Unter anderem beinhaltet dies die Anzahl vorhandener Konstanten und Variablen, sowie die Zahl der Veränderungen. Eine Validierung mittels bildgebender Verfahren

(bspw. functional magentic resonance imaging (fMRI) oder Eye-Tracking) wurde in dieser Arbeit jedoch nicht vorgenommen.

Eine solche fMRI-Studie haben Peitek et al. [PAP+21] durchgeführt. Sie haben untersucht, wie sehr diese verschiedenen Metriken mit der beobachtbaren Hirnaktivität korrelieren. Während dieser Studie bekamen die Teilnehmenden Code Snippets gezeigt und sollten daraufhin jeweils bestimmen, welche Ausgabe bei der Ausführung der Snippets erzeugt wird. Gemessen wurden die Korrektheit und die Bearbeitungsdauer, sowie die subjektive Komplexität. Weiterhin wurde die Hirnaktivität gemessen. Grundlage für diese Untersuchung ist, das Denkprozesse den Sauerstoffgehalt im Gehirn verändern, was sich mittels bildgebender Verfahren visualisieren lässt. Je höher die Hirnaktivität ist, umso höher ist der Sauerstoffgehalt. Dabei fanden sie heraus, dass gerade datenflussbasierte Metriken, wozu der DepDegree zählt, eine hohe Korrelation aufweisen. Weiterhin haben jedoch auch Text- und Vokabulargröße einen Einfluss, auch wenn dieser weniger stark mit der Hirnaktivität korreliert.
Durch den fMRI-Ansatz ist es möglich, Kausalzusammenhänge zwischen einzelnen Code-Elementen und der zugeschriebenen Komplexität herzustellen. Daher ist dieser Ansatz auch sehr vielversprechend.

Ein anderes Verfahren, welches für meine Arbeit von Bedeutung ist, ist Eye-Tracking. Dabei geht es darum, die Augenbewegungen von Studienteilnehmenden mit Hilfe einer Kamera oder speziellen Brille zu verfolgen und so zu bestimmen, wohin die Personen geschaut haben. Aus diesen Ergebnissen lassen sich Schlussfolgerungen ziehen, welche Bereiche beispielsweise in Code Snippets für die Teilnehmenden besonders interessant bzw. kompliziert waren. Konkret habe ich mich hierbei auf die Arbeit von Mucke et al. [MSS21] gestützt, da diese ein Verfahren implementiert haben, welches ein remote Eye-Tracking erlaubt.

3. Methodik

3.1. Forschungsfrage und Hypothesen

Basierend auf den bisherigen Erkenntnissen aus den beiden vorangegangenen Kapiteln sowie meiner allgemeinen Zielsetzung aus Kapitel 1 habe ich die folgende Forschungsfrage formuliert:

> *Wie wirkt sich die Schwierigkeit des Datenflusses gemessen an der Höhe des DepDegree bei Informatik-Studierenden auf die Korrektheit und Antwortzeit beim Programmverständnis von Code Snippets aus?*

Daraus ergeben sich zwei Hypothesen:

> ***H1:*** *Je höher der DepDegree, desto länger benötigen die Studierenden für das Verständnis.*
>
> ***H2:*** *Je höher der DepDegree, desto weniger häufig ist die Antwort der Studierenden korrekt.*

Zugehörig zu den beiden Hypothesen lassen sich zwei Nullhypothesen formulieren:

> ***$H0_1$:*** *Wenn der DepDegree höher ist, hat das keine beobachtbaren Auswirkungen auf die Antwortzeit der Studierenden oder die Antwortzeit ist kürzer.*
>
> ***$H0_2$:*** *Wenn der DepDegree höher ist, hat das keine beobachtbaren Auswirkungen auf die Zahl der korrekten Antworten der Studierenden oder die Zahl ist geringer.*

3.2. DD-Tool

Um den DepDegree zu bestimmen, verwendete ich ein Tool, welches in der Eclipse IDE die jeweiligen DepDegrees berechnet und anzeigt. Dabei werden die jeweiligen DepDegrees der einzelnen Operationen berechnet und anschließend je Methode zusammengefasst. Um die Berechnung besser nachvollziehen zu können, werden bei Auswahl einer Operation auch die jeweiligen bedingenden Operationen aufgelistet und markiert. Die Grafiken 3.1 und 3.2 verdeutlichen dies. Grafik 3.1 zeigt die Ansicht, während der Cursor sich an einer anderen Stelle im Code befindet. Die unterschiedlichen DepDegrees sind mit unterschiedlichen Rottönen kenntlich gemacht,

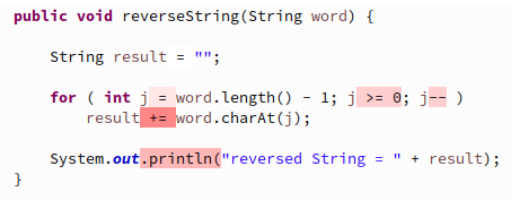

Abb. 3.1.: DepDegrees ohne Markierung

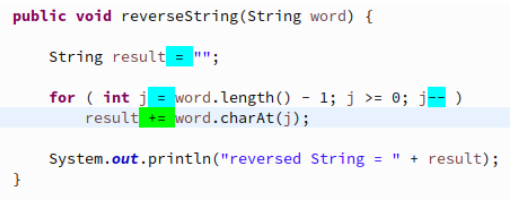

Abb. 3.2.: DepDegree mit Markierung

wobei eine höhere Farbintensität ein höherer DepDegree-Wert bedeutet. Grafik 3.2 zeigt die Ansicht, wenn sich der Cursor auf einer Operation (grün markiert) befindet. Dann werden die Operationen, auf welche sich die ausgewählte operation bezieht, blau markiert.

3.3. Studiendesign

Um meine in Kapitel 3.1 genannten Hypothesen zu untersuchen habe ich eine empirische Studie durchgeführt. Aufgrund der pandemischen Situation konnte diese nicht wie anfangs geplant, als Studie vor Ort stattfinden, sondern musste als Online-Survey umgesetzt werden. Dies hatte auch Auswirkungen auf das Design der Studie. So beinhaltete der ursprüngliche Plan ein kamerabasiertes Eye-Tracking, sowie ein paralleles Think Aloud, bei dem die Teilnehmenden ihre Gedanken während der Durchführung der Studie kommunizieren.

Hauptbestandteil der Studie waren sieben Aufgaben, welche auf Java Code Snippets[1], wie zuvor bereits in Listing 2.1 beispielhaft dargestellt, basierten. Die Teilnehmenden sollten hierbei, ausgehend von einer vorgegeben Eingabe, für ein bestimmtes Code Snippet die Ausgabe der Methode ermitteln. Die Reihenfolge der Aufgaben war willkürlich gewählt, jedoch für alle Teilnehmenden identisch. Die Reihenfolge der DepDegrees (40, 12, 15, 18, 28, 24, 30) war ebenfalls weitestgehend durchmischt, um den Einfluss von Ermüdung zu verringern.

Für die Auswahl der Snippets konnte ich auf eine vorhandene Sammlung der Professur zurückgreifen, wobei die Auswahl einiger Snippets aus dieser Sammlung nach verschiedenen Gesichtspunkten geschah. In erster Linie war mir wichtig, verschiedene DepDegrees abzubilden. Damit erhoffte ich mir eine ausreichende Aussagekraft der Ergebnisse zu erreichen, sollte der angenommene Effekt vorhanden sein. Weiterhin war mir wichtig, Aufgaben aus verschiedenen Bereichen (Arithmetik, Array-Handling, String-Handling) zu nehmen, um den Einfluss möglicher Stärken und Schwächen der Teilnehmenden verringern zu können. Nachdem ich mir einige Snippets aus verschiedenen Bereichen herausgesucht habe, habe ich mit mithilfe des in Kapitel 3.2 beschriebenen DD-Tools die DepDegrees bestimmt und mir einige Methoden mit unterschiedlichen DepDegrees herausgesucht. Zudem wollte ich mit der

[1] eine vollständige Auflistung findet sich im Anhang A

Wahl aussagefreier Variablen- und Methodennamen vermeiden, dass Vorwissen zu bestimmten Algorithmen das Ergebnis verfälscht – vorausgesetzt, die Teilnehmenden erkennen den Algorithmus nicht auch ohne diese Informationen. Gänzlich vermeiden ließ sich dies nicht, denn bei den Snippets handelte es sich um Algorithmen, die bereits in den ersten Semestern des Informatik-Studiums Inhalt verschiedener Veranstaltungen waren. Letztlich habe ich die Zahl auf sieben Snippets beschränkt, um die Bearbeitungszeit der Studie nicht zu sehr über die veranschlagten 20 bis 30 Minuten zu ziehen.

Um das Vorgehen der Teilnehmenden besser nachvollziehen zu können, sollte ein Eye-Tracking-Verfahren zur Anwendung kommen. Durch die Remote-Durchführung konnte ein klassisches Verfahren mit Hilfe einer Kamera nicht verwendet werden, weshalb ich mich dann für REyeker [MSS21] entschieden habe. REyeker bewirkt, dass die Bilder von den Code Snippets verschwommen dargestellt werden und nur ein Bereich in definierter Größe um eine angeklickte Stelle herum nicht verwischt wird. Die Klickposition sowie die Klickzeit werden dabei jeweils aufgenommen und in der entsprechenden Variable des verwendeten SosciSurvey gespeichert.

Da REyeker für die Implementierung der Studie bereits mit SosciSurvey erprobt war, habe ich mich ebenfalls dafür entschieden. Die Implementierung von REyeker geschieht dabei über ein HTML Snippet (s. Grafik 3.3). Dieser bindet die JavaScript-Dateien[2] ein, welche zuvor in SosciSurvey hochgeladen werden. Zudem können durch die Verwendung bestimmter Tags verschiedene Einstellungen wie die Form und Größe des „sichtbaren“ Bereiches vorgenommen werden. Auf ähnliche Weise wurden auch die Variablen sowie Fragentexte implementiert. Die Zusammenstellung des Fragebogens erforderte anschließend nur noch eine Zusammenstellung der einzelnen Blöcke und eine Ergänzung um einen weiteren Block, welcher das Bild lädt.

Um den subjektiven Eindruck der Schwierigkeit der Teilnehmenden zu ermitteln, wollte ich zusätzlich die „Think Aloud“-Methodik verwenden, bei welcher die Teilnehmenden ihre Gedanken während der Aufgabenbearbeitung laut aussprechen. Da dies bei einer Remote-Durchführung jedoch sehr schwierig ist, habe ich mich entschieden, diese Gedanken retrospektiv aufnehmen zu lassen. Daher folgte im Anschluss zu jeder Aufgabe eine Seite, bei der die Gedanken während der Aufgabenbearbeitung abgefragt wurden.
Vorgelagert zum Hauptteil wurden demografische Daten abgefragt. Konkret erfragte ich die Programmiererfahrung in Jahren bezüglich Programmierung im Allgemeinen, Programmierung mit Java und Programmierung im professionellen Umfeld. Weiterhin nahm ich den aktuellen Beschäftigungsstatus (Bachelor/Master/Promotions/-berufstätig/Anderes) sowie einen subjektiven Vergleich zu Personen des gleichen Bildungsstandes auf. Ziel war es, die aufgenommenen Daten möglicherweise noch kategorisiert nach Erfahrungslevel betrachten zu können.

[2] `https://github.com/brains-on-code/REyeker/tree/Directing-Gaze/REyekerTool/scripts`

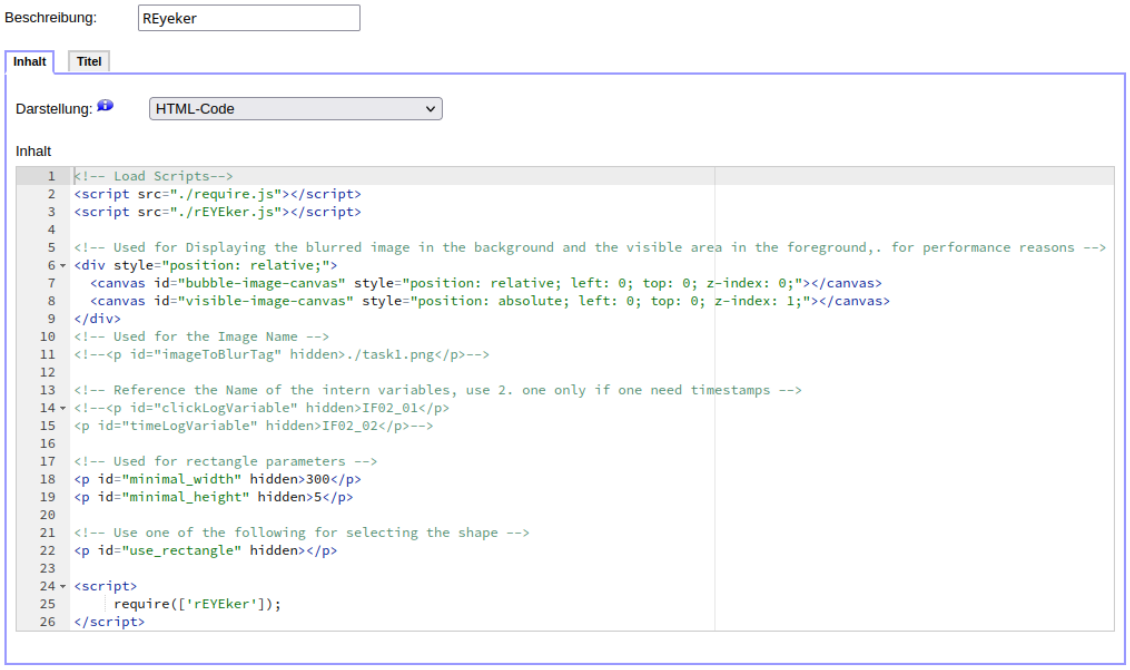

Abb. 3.3.: REyeker in SosciSurvey

3.4. Teilnehmende der Studie

Die Teilnehmenden für diese Studie habe ich aus der Gruppe der Informatik-Studierenden gewonnen. Hintergrund dieser Gruppenwahl war, dass ich Probleme beim Codeverständnis durch fehlende Kenntnis der Syntax ausschließen wollte. Weitere Einschränkungen wurden jedoch nicht gemacht. Demografisch betrachtet habe ich mit dieser Wahl ebenso keine Einschränkungen getroffen, auch wenn sich durch den Status der Studierenden eine geringere Programmiererfahrung erwarten ließ. Diese habe ich zu Beginn der Studie ebenso abgefragt.

Insgesamt haben sich sieben Teilnehmende beteiligt. Die folgende Tabelle 3.1 zeigt die Zusammensetzung hinsichtlich des Beschäftigungsstatus (Bachelor/Master/Promotion/berufstätig/anderes (Nennung)) zum Zeitpunkt der Studie.

Person	Status
1	berufstätig
2	Promotion
3	berufstätig
4	Master-Student:in
5	berufstätig
6	Bachelor-Student:in
7	Bachelor-Student:in, berufstätig

Tab. 3.1.: Status der Teilnehmenden

3.5. Durchführung der Studie

Wie im vorigen Kapitel beschrieben, gliederte sich meine Survey in verschiedene Teile. Nach einem kurzen Einführungstext, welcher die Aufgabe beschrieb und ein paar Rahmeninformationen zu Zweck und Umfang gab, wurden basierend auf der Forschung von Siegmund et. al [SKL+14] ein paar Fragen zum Erfahrungsstand gestellt. Ziel dessen war es, bei ausreichender Gruppengröße auch Betrachtungen zwischen verschiedenen Erfahrungsgruppen treffen zu können. Darauf folgte eine Testaufgabe (s. Grafik 3.4), mit der sich die Teilnehmenden mit dem Vorgehen vertraut machen konnten.

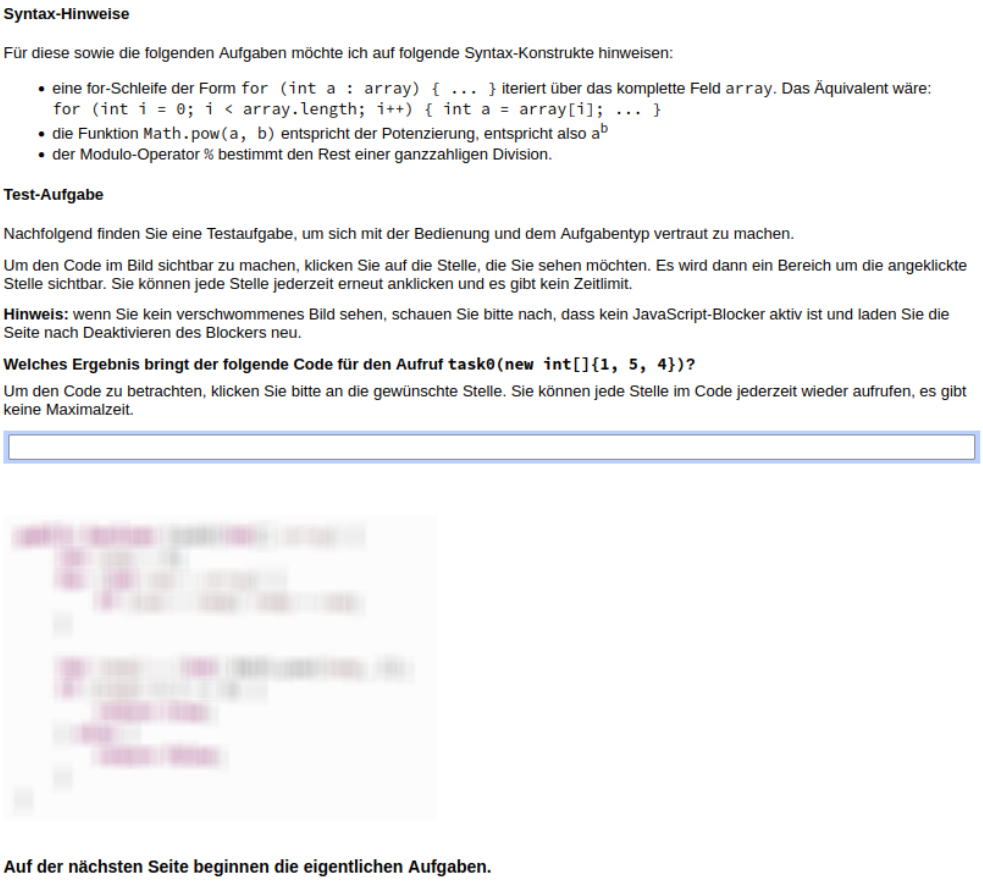

Abb. 3.4.: So stellte sich die Testaufgabe den Teilnehmenden beim Öffnen dar.

Danach folgten die eigentlichen Aufgaben, welche stets aus den beschriebenen zwei Seiten – Aufgabe (Grafik 3.5) und Retrospektive (Grafik 3.6) – bestanden. Die Aufgabenstellung versuchte ich hierbei immer so kurz wie möglich zu halten, damit sich die Teilnehmenden auf die wichtigen Informationen konzertieren konnten. Unterhalb der Frage befand sich für jede Aufgabe der formale Hinweis zur Benutzung von REyeker. Nachfolgend wurde bei Seitenaufruf das gesamte Bild des Code Snippets verschwommen dargestellt. Erst nach dem ersten Klick auf eine Stelle innerhalb des

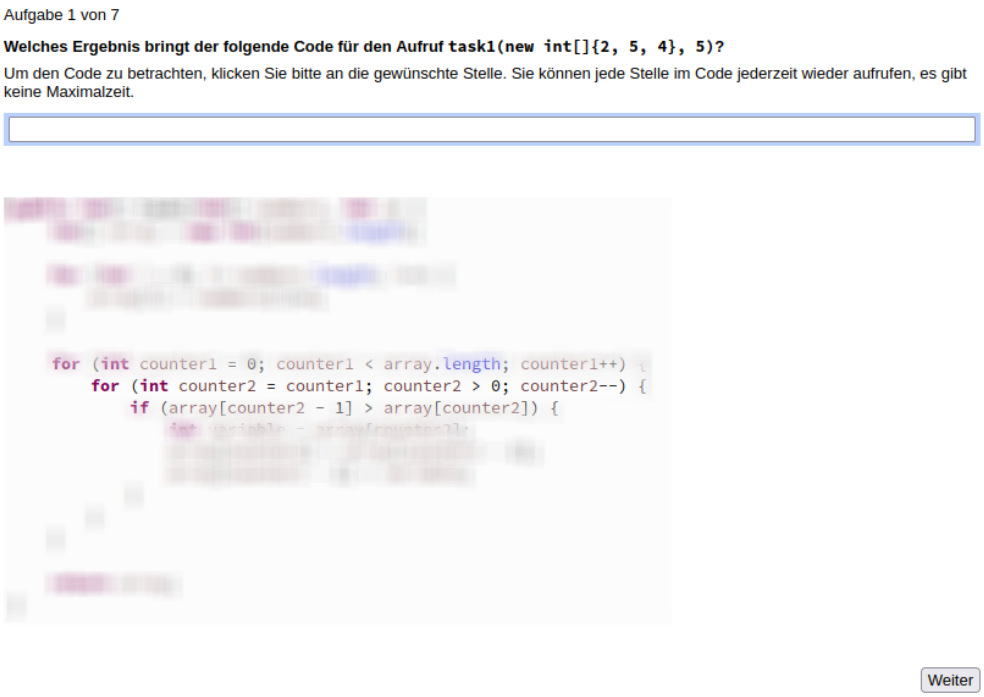

Abb. 3.5.: Die Aufgabe nach dem ersten Klick auf das Snippet ...

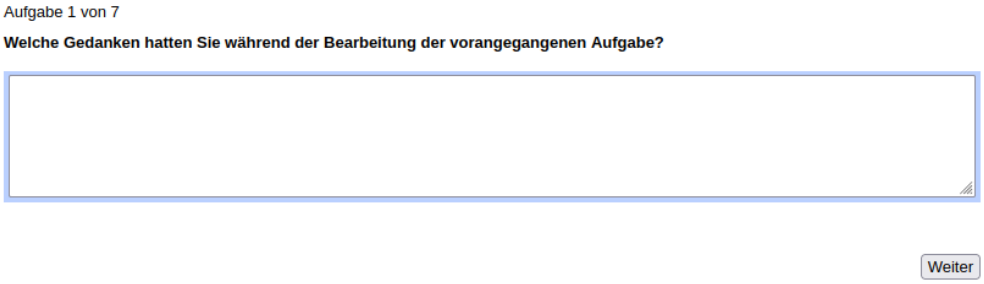

Abb. 3.6.: ... und die Seite für die Retrospektive im Anschluss.

Bildes wurde ein rechteckiger Bereich von 300 Pixeln in der Breite zu 5 Pixeln in der Höhe scharf dargestellt. Bei jedem weiteren Klick wird dieser Bereich auf die neue Stelle verschoben. Die Aufgabe gilt als beendet, wenn die Teilnehmenden ihr Ergebnis in das Eingabefeld eingetragen und auf „Weiter“ geklickt haben. Anschließend wurde auf der Folgeseite retrospektiv nach den Gedanken während der Bearbeitung der vorangegangenen Aufgabe gefragt. Dieses Vorgehen war für alle 7 Aufgaben identisch. Danach war die Survey für die Teilnehmenden beendet.

4. Analyse

Im folgenden möchte ich nun die Ergebnisse der im vorigen Kapitel beschriebenen Studie näher betrachten. Um das Vorgehen zu veranschaulichen, betrachte ich einen konkreten Datensatz näher. Mein Vorgehen ist hierbei quantitativ.

Die Daten aus dem SosciSurvey-Projekt erhielt ich in Form einer CSV-Tabelle. Die folgenden Tabellen 4.2 und 4.3 sind hierbei Auszüge aus der Gesamttabelle. Neben den für meine Betrachtung relevanten Daten, wie z. B. den gespeicherten Klickdaten, enthielt die Tabelle auch nicht relevante Daten, wie beispielsweise den Titel des verwendeten Fragebogens oder den Zeitstempel des Bearbeitungsbeginns. Diese Daten werde ich hier nicht weiter betrachten.

Zur Einordnung betrachte ich als erstes die Antworten auf die Erfahrungsfragen. Insgesamt haben sich 7 Teilnehmende beteiligt. Die Aufteilung auf Bachelor-Studierende, Master-Studierende, berufstätige Personen und anderes (entsprechend genannt) wurde bereits in Kapitel 3.4 betrachtet. Entsprechend dieser Varianz variierten auch die Erfahrungsjahre. In Bezug auf die Erfahrung allgemein bewegten sich die Werte zwischen 4 und 20 Jahren, wobei das arithmetische Mittel bei 10 Jahren lag. Bezüglich der Programmierung mit Java bewegten sich die Angaben zwischen 0 und 10 Jahren, mit einem Mittelwert bei 3 Jahren. Bei der professionellen Programmierung wurden Werte zwischen 1,5 und 14 Jahren angegeben, mit einem Mittelwert bei 4,5 Jahren.

Person	**Erfahrungsjahre in Programmierung ...**			**Vergleich**
	... allgemein	**... mit Java**	**... professionell**	
1	10	1	3	ungefähr gleich
2	5	1	2	höher
3	20	10	14	ungefähr gleich
4	7	1	3	ungefähr gleich
5	13	0	5	höher
6	4	0	1.5	geringer
7	11	8	3	ungefähr gleich

Tab. 4.1.: Erfahrungsdaten

Anschließend widmete ich mich den Antworten auf die eigentlichen Tasks. Outlier konnte ich hierbei keine feststellen, da es zu wenige Datensätze waren, um sinnvoll Outlier bestimmen zu können. Jedoch war ein Datensatz dabei, welcher bei ei-

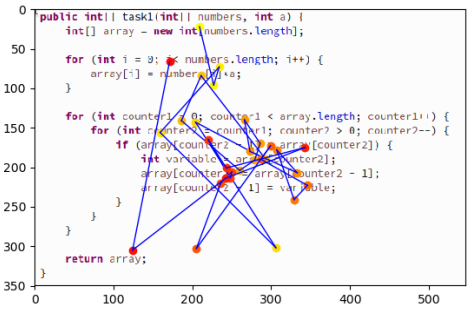

Abb. 4.1.: Vorgehen Person 1 in Aufgabe 1

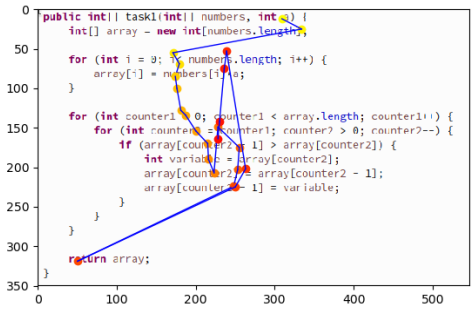

Abb. 4.2.: Vorgehen Person 2 in Aufgabe 1

nem größeren Sample möglicher Outlier-Kandidat hätte sein können. Während die übrigen Teilnehmenden zwischen 5 und 7 richtige Antworten abgegeben hatten, hat diese Person nur zwei korrekte Antworten abgegeben (Aufgabe 3 mit DepDegree 15 und Aufgabe 5 mit DepDegree 28). Da hierbei auch die Aufgaben mit den niedrigsten DepDegrees falsch beantwortet wurden, kann man diesen Datensatz als möglichen Outlier betrachten.

int Var T1: clickLogVariable	**int Var T1: timeLogVariable**
249-11 211-64 227-36 163-70 227-38 185-89 229-43 245-132 246-158 255-179 233-68 273-172 319-200 365-231 355-181 140-290 191-76 298-168 232-136 257-191 274-156	1460 7667 8303 21241 32475 34567 36031 39953 48581 56744 62150 62912 77616 84936 85847 90877 92106 106917 114182 116214 117702

Tab. 4.2.: die Klickdaten von REyeker

Als erstes habe ich mir einen Überblick über die Klickdaten verschafft, indem ich mir mithilfe von Python zu jeder Aufgabe die Klickpunkte der einzelnen Personen über die Snippets zeichnen lassen habe (s. Grafik 4.1). Hierfür wählte ich mir einen gleichmäßigen Farbverlauf, welcher bei gelb beginnt und bei rot endet. Damit wollte ich in erster Linie ein Gefühl für die Daten erhalten, weiterhin jedoch auch sehen, wie die verschiedenen Teilnehmenden die Snippets gelesen haben. Dabei fiel mir bereits auf, dass manche Personen viel hin und her geklickt haben und andere relativ strukturiert nur in eine Richtung weitergeklickt haben. Ein sehr deutliches Beispiel ist Aufgabe 1. Hierbei sind Person 1 (Grafik 4.1) und Person 2 (Grafik 4.2) sehr unterschiedlich vorgegangen. Während Person 1 viel hin und her geklickt hat, hat Person 2 sich zuerst einen Überblick über den Code verschafft und anschließend die Verschachtelung der Schleifen genauer betrachtet. Dieses Vorgehen ist strukturierter.

Anschließend habe ich basierend auf den gleichen Daten mit Hilfe von REyeker für jede Person je Aufgabe Heatmaps generiert, welche die betrachteten Stellen anhand

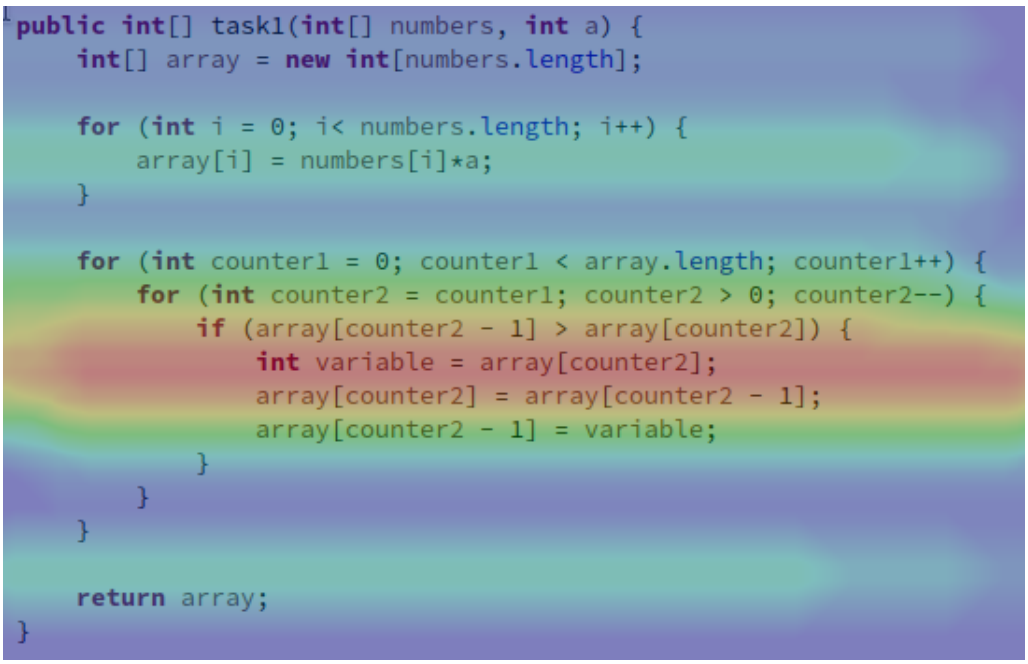

Abb. 4.3.: Die Heatmap zur Grafik 4.1

der gesamten Verweildauer verfärbt haben. Dabei flossen jedoch alle Zeitpunkte ein, sodass eine Reihenfolge nicht mehr erkennbar war. Dennoch vermittelten sie relativ gut, welche Bereiche von den Teilnehmenden länger betrachtet wurden und demnach vermutlich auch schwieriger waren. Grafik 4.3 veranschaulicht dieses am Beispiel von Aufgabe 1 von Person 1. Eine rote („warme“) Stelle signalisiert hierbei, dass insgesamt relativ lange dort verweilt wurde, eine blaue („kalte“) Stelle hingegen weist eine kaum oder gar nicht betrachtete Stelle aus. Dabei ist die Größe der Stelle auch von der in Kapitel 3.5 beschriebenen Größe des „sichtbaren“ Bereiches abhängig.

Da bei meiner Betrachtung die Korrektheit eine bedeutende Rolle spielt, habe ich anschließend zu jeder Aufgabe Gesamt-Heatmaps erstellt. Als erstes erstellte ich Heatmaps, die sämtliche Antworten einbezogen. Anschließend nutzte ich nur die Datensätze, welche ein korrektes Ergebnis zur Folge hatten, um so Heatmaps über alle richtigen Antworten zu der betrachteten Aufgabe zu erstellen. Die Abbildungen 4.4 und 4.5 zeigen die jeweiligen Heatmaps über alle Antworten bzw. nur alle korrekten Antworten zu Aufgabe 1. Zwar ähneln sich beide Abbildungen auf den ersten Blick, bei genauerer Betrachtung jedoch fällt auf, dass der rote Bereich in Abbildung 4.5 etwas kleiner ist. Da dies bei den anderen Aufgaben ähnlich aussah, lässt sich vermuten, dass diejenigen, die richtig geantwortet haben, besser wussten, wo sie hinschauen müssen.

Nachdem ich so einen Eindruck über die verschiedenen Daten gewonnen hatte, habe ich mir konkret noch einmal die Daten zur Korrektheit und Bearbeitungsdauer herausgesucht. Für die Bearbeitungsdauer verwendete ich dabei die Verweildauer auf der entsprechenden Seite (siehe letzte zwei Spalten in Tabelle 4.3). Person 1 verweilte demnach 346 Sekunden auf Seite 4, welche die erste Aufgabe enthielt. Dies schließt

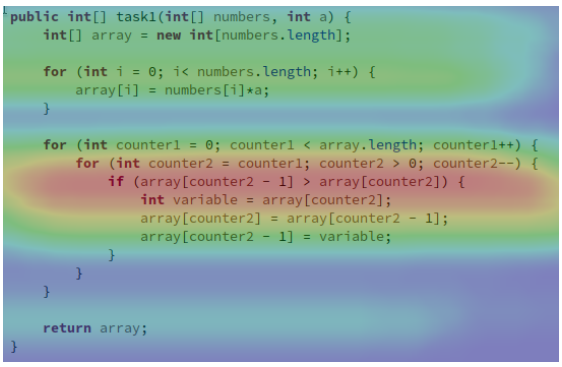

Abb. 4.4.: Heatmap über alle Antworten zu Aufgabe 1

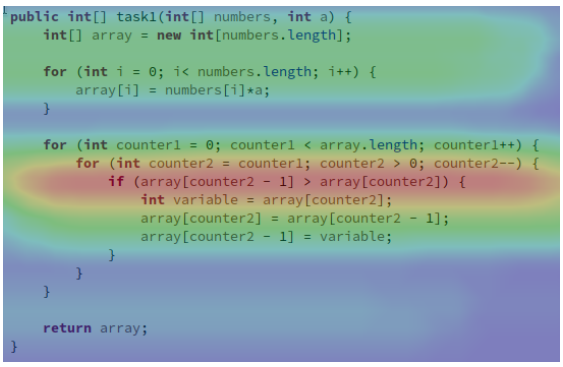

Abb. 4.5.: Heatmap über alle korrekten Antworten zu Aufgabe 1

Task01: [01]	**task02: [01]**	**...**	**Verweildauer Seite 4**	**Verweildauer Seite 6**
[25, 20, 10]	true	...	346	106

Tab. 4.3.: links: die Antworten; rechts: die Verweilzeiten für die betreffenden Seiten

die gesamte Zeit von Aufruf der Seite bis zum Absenden der Antwort ein. Damit lag die Zeit ein wenig über dem Durchschnitt, welcher bei 303 Sekunden lag.

Weiterhin habe ich die Antworten mit den korrekten Antworten verglichen und somit nach korrekt („true“), falsch („false“) und nicht beantwortet („none“) kategorisiert. Beides habe ich für jede Person einzeln jeweils über den DepDegree abgetragen und so Graphen wie in Grafik 4.6 erhalten. Um abschließend eine Gesamtbetrachtung durchführen zu können, habe ich schließlich die gleiche Betrachtung noch einmal unabhängig der einzelnen Personen durchgeführt. Grafik 4.7 beinhaltet sämtliche Daten, auch jene mit falschen Antworten, während Grafik 4.8 lediglich die Daten mit den korrekten Antworten enthält. Auffällig sind hierbei die höheren Durchschnittszeiten in Grafik 4.8. Während der Durchschnitt für einen DepDegree von 30 in Grafik 4.7 bei knapp über 275 Sekunden liegt, liegt dieser in Grafik 4.8 bei ungefähr 325 Sekunden. Dies bedeutet, dass zu den in Grafik 4.8 nicht mehr betrachteten Teilnehmenden auch Personen gehörten, die deutlich kürzer auf der Seite verweilten und sich demnach weniger mit der Aufgabe befasst haben.

Anhand der Graphen war dann auch erkennbar, dass es mit steigendem DepDegree auch eine steigende Tendenz bei der Bearbeitungszeit und eine sinkende Tendenz bei der Korrektheit gibt. Bei näherer Betrachtung fällt zudem ein möglicher Outlier-Wert für einen DepDegree von 30 auf, welcher eine niedrigere Zahl korrekter Antworten nach sich zog, als der DepDegree von 40. Jedoch sei hier erwähnt, dass mir eine Streichung dieses Wertes nicht sinnvoll erschien, da ich mit 7 Datensätzen zu 7 DepDegrees eine verhältnismäßig kleine Datenbasis hatte. Aus diesem Grund kann die Forschungsfrage auch nicht abschließend beantwortet werden, sondern nur ein möglicher Zusammenhang prognostiziert werden.

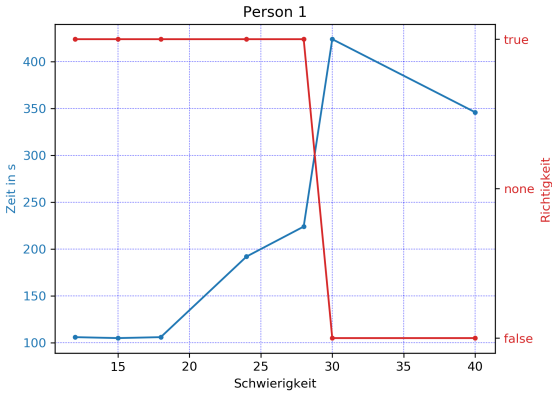

Abb. 4.6.: Visualisierung Korrektheit und Zeit für Person 1

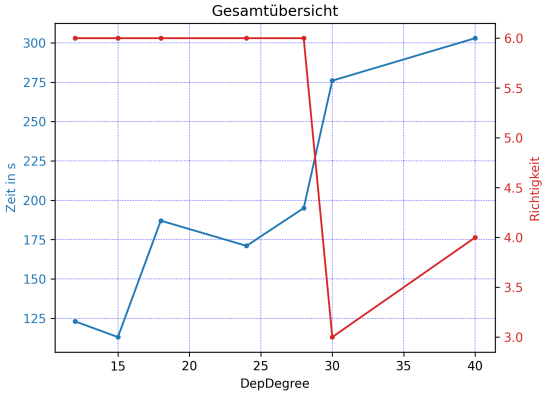

Abb. 4.7.: Visualisierung Korrektheit und Zeit für alle Personen

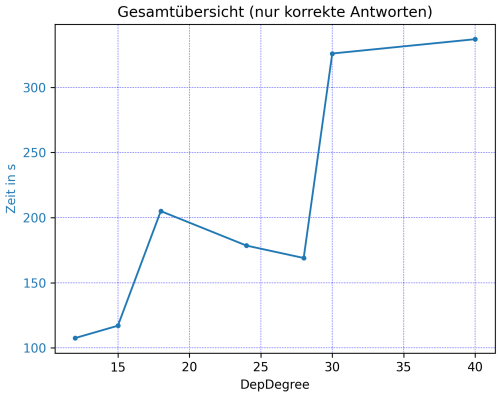

Abb. 4.8.: Visualisierung Zeit für alle Personen mit korrekten Antworten

5. Diskussion

Während ich im vorigen Kapitel lediglich die Ergebnisse betrachtet und analysiert habe, möchte ich mir in diesem Kapitel noch einmal den Kontext und die Rahmenbedingungen ansehen.

5.1. Externe Validität

Als erstes möchte ich mich den Threats der externen Validität widmen. Die externe Validität beschreibt, wie generalisierbar die Ergebnisse in der „realen Welt“ sind. Threats bzgl. dieser beschreiben die Einschränkungen, die bei der Betrachtung der Ergebnisse gemacht werden müssen.

Wie in der Analyse bereits angesprochen, sind die Ergebnisse der Studie nicht ausreichend, um ein abschließendes Ergebnis zu erhalten. Betrachtet man die Ergebnisse genauer, fällt als erstes die sehr kleine Datenbasis von 7 Datensätzen zu je 7 Aufgaben (DepDegrees) auf. Aus Sicht der externen Validität ist dies in erster Linie auf die geringe Teilnehmendenzahl zurückzuführen, welche meiner Ansicht nach zu einem guten Teil auch aus der pandemiebedingten Situation resultiert. Dadurch war eine Präsenzdurchführung nicht möglich und könnte so auch einige Studierende von einer Teilnahme abgehalten haben. Wäre eine größere Datenbasis, bestehend aus mehr Datensätzen, gegeben, wäre einerseits eine Outlierbestimmung machbar gewesen. Andererseits wären auch statistische Tests durchführbar gewesen.

Betrachtet man die Daten aus Tabelle 4.1 (Seite 23) noch einmal, fällt auf, dass die Teilnehmenden zum Teil noch nicht sehr erfahren sind. Während die mittlere Erfahrung bzgl. des Programmierens allgemein bei 10 Jahren liegt, liegen die mittlere professionelle Programmiererfahrung bei 4,5 Jahren und die mittlere Programmiererfahrung mit Java bei lediglich 3 Jahren. Für die externe Validität bedeutet dies, dass die Ergebnisse möglicherweise nicht auf die gesamte Gruppe der Programmierenden generalisierbar sind.

Weiterhin kann man die Art der Tasks bzgl. der externen Validität betrachten. In meiner Studie habe ich vor allem kurze Algorithmen verwendet, wie sie zwar im Feld ebenso vorkommen, jedoch meist vorimplementiert verwendet werden. Eine Möglichkeit, den Einfluss dieses Faktors zu verringern, wäre eine Verwendung von Code Snippets aus dem realen Softwareeinsatz.

5.2. Interne Validität

Nachdem ich nun die Threats der externen Validität betrachtet habe, möchte ich mich im Folgenden auch die Threats der internen Validität betrachten. Die interne Validität beschreibt die Qualität des gesamten Designs. Eine hohe interne Validität ist Grundlage für eine korrekte Auswertung. Threats bzgl. dieser sind Faktoren, die das Ergebnis verfälschen und somit falsche Schlüsse zur Folge haben können.

Bei der Betrachtung der internen Validität möchte ich beim Design der Studie auf Durchführungsebene beginnen. Die Remote-Ausführung stellt hierbei nämlich eines der größten Probleme dar. Während sich bei einer Präsenzdurchführung im Labor die Umgebung kontrollieren und zwischen den Teilnehmenden einheitlich gestalten lässt, ist dies bei einer Remote-Durchführung nicht realisierbar. So lassen sich Störfaktoren, welche zu Ablenkung oder Verbindungsabbrüchen führen können, nicht regulieren. Dies kann die Konzentration der Teilnehmenden negativ beeinflusst haben, wodurch die Ergebnisse verfälscht sein können.

Bezogen auf die Konzentrationsfähigkeit spielen verschiedene Faktoren eine Rolle, welche ich nachfolgend näher betrachten möchte.
Als ersten Faktor betrachte ich die Durchführungsdauer. Diese könnte hier vielleicht ein wenig verkürzt oder entsprechend mit Pausen versehen werden, denn bei Task 7 benannten drei Teilnehmende in der Retrospektion, dass sie mit Ermüdung zu tun hatten („Ich mag nicht mehr“[Person 3, s. Anhang B.g], „Langsam wirds doch öde“[Person 5, s. Anhang B.g]).
Im Zusammenhang mit der Dauer stehen auch die Zahl und Reihenfolge der Tasks. Während die Reihenfolge im Voraus willkürlich von mir festgelegt worden war, ist die Verteilung möglicherweise unglücklich erfolgt. So könnte die Tatsache, dass die Aufgabe mit dem höchsten DepDegree als erstes gestellt worden ist, über mögliche Ermüdung oder Demotivation einen negativen Einfluss gehabt haben. Hier wäre eine gleichmäßige Verteilung oder zwei Gruppen mit gegenläufigen Verteilungen möglicherweise sinnvoller gewesen. Zusätzlich würden mehr Tasks und damit mehr DepDegrees eine genauere Betrachtung möglich machen und die Verteilung ließe sich leichter umsetzen.
Die vorgenannten Faktoren sind jedoch nicht der einzige Einfluss auf eine mögliche Ermüdung, denn bereits in Task 1 benannte eine Person, dass die Variablennamen „furchtbar [sind]“[Person 3, s. Anhang B.a]. Dieser Threat hat meiner Einschätzung nach jedoch einen geringeren Einfluss als der dadurch minimierte Einfluss von vorhandenem Algorithmenwissen (z. B. Bubblesort im zweiten Teil von Task 1). Eine weitere Person äußerte, dass der „Gesamtüberblick schwerer [fiele]“, „wenn man nur einen Teil des Codes sieht“[Person 4, s. Anhang B.a]. Damit zusammenhängend lässt sich auch ein möglicher Einfluss durch die Notwendigkeit des Klickens vermuten. Für die meisten Teilnehmenden war es wahrscheinlich eher kontraintuitiv zu klicken, um einen bestimmten Bereich sehen zu können. Lediglich eine Person erwähnte in der Retrospektion, dass sie kürzlich eine „ähnliche Umfrage mit [...] viel kleinerem Sichtfenster gemacht“ habe, und „die Navigation [...] hier deutlich besser“ [Person 7, s.

Anhang B.a] gewesen wäre. Auch hier stellt die gewählte Einstellung einen in meinen Augen guten Kompromiss zwischen „zu wenig“ und „zu viel“ Sichtbereich dar. Ausgeschlossen werden kann dieser Einfluss nur bei einer kamera- oder brillenbasierten Durchführung, da hierbei keine zusätzlichen Aktionen seitens der Teilnehmenden nötig sind und sie den gesamten Code sehen.

Die aufgezählten Probleme machen eine finale Beantwortung der Forschungsfrage zwar unmöglich, dennoch lässt sich eine Tendenz formulieren. Um die Forschungsfrage final beantworten zu können benötigt es jedoch weiterer Forschungen.

6. Schlussfolgerungen und Ausblick

Mit Hilfe einer empirischen Studie konnte ich die Vermutung von Peitek et al. [PAP+21] unterstützen, dass ein Zusammenhang zwischen datenflussbasierten Metriken – konkret dem Dependency Degree – und dem Verständnis von Code besteht. Konkret bedeutet dies, dass es länger zu dauern scheint und/oder mehr Fehler gemacht werden, wenn dessen DepDegree höher ist. Aufgrund einer zu kleinen Datenbasis lässt sich diese Vermutung jedoch nicht abschließend bestätigen oder widerlegen.

Um die Vermutung abschließend zu bestätigen oder zu widerlegen, sollte dieser Ansatz in künftigen Studien noch einmal betrachtet werden. Hierzu gibt es entsprechend der Diskussion (Kapitel 5) verschiedene Punkte, welche angepasst bzw. verbessert werden können.
Insbesondere die Größe der Datenbasis sollte durch mehr Teilnehmende und/oder mehr Tasks erweitert werden. Parallel kann auch die Gruppe der Teilnehmenden verändert werden, um erfahrungs- oder wissensbasierte Einflüsse zu verringern.
Bezüglich der Durchführung ist auch eine Präsenzdurchführung mit brillen- oder kamerabasiertem Eye-Tracking empfehlenswert, um Störfaktoren zu verringern und das Lesen des Codes intuitiver zu gestalten.

In meiner Arbeit habe ich einen quantitativen Ansatz gewählt. Um ein umfassenderes Bild zu gewinnen und auch die Empfindungen der Teilnehmenden einzubeziehen, halte ich eine weitere qualitative Betrachtung in künftigen Forschungen für sinnvoll.

Abbildungsverzeichnis

Literaturverzeichnis

[BF10] Beyer, Dirk; Fararooy, Ashgan: A Simple and Effective Measure for Complex Low-Level Dependencies. In: *2010 IEEE 18th International Conference on Program Comprehension*, 2010, Seiten 80–83

[Che20] Cheah, Chin S.: Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. In: *Contemporary Educational Technology* 12 (2020), Nr. 2, Seiten ep272

[Hal72] Halstead, M. H.: Natural Laws Controlling Algorithm Structure? In: *SIGPLAN Not.* 7 (1972), feb, Nr. 2, Seiten 19–26. – ISSN 0362–1340

[Hal77] Halstead, Maurice H.: *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., 1977

[JS20] Jain, Leena; Singh, Satinderjit: Underlying Mental Factors Contributing to Software Complexity. In: *International Journal of Engineering and Advanced Technology (IJEAT)* 9 (2020), Februar, Nr. 3, Seiten 4352–4358

[McC76] McCabe, T.J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), Dec, Nr. 4, Seiten 308–320. – ISSN 1939–3520

[MSS21] Mucke, Jonas; Schwarzkopf, Marc; Siegmund, Janet: REyeker: Remote Eye Tracker. In: *ACM Symposium on Eye Tracking Research and Applications.* New York, NY, USA : Association for Computing Machinery, 2021 (ETRA '21 Short Papers). – ISBN 9781450383455

[PAP+21] Peitek, Norman; Apel, Sven; Parnin, Chris; Brechmann, André; Siegmund, Janet: Program Comprehension and Code Complexity Metrics: An fMRI Study. In: *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering. IEEE*, 2021

[Rob19] Robins, Anthony V.: 12 Novice Programmers and Introductory Programming. In: *The Cambridge handbook of computing education research* (2019), Seiten 327

[SKL+14] Siegmund, Janet; Kästner, Christian; Liebig, Jörg; Apel, Sven; Hanenberg, Stefan: Measuring and modeling programming experience. In:

Empirical Software Engineering 19 (2014), Oct, Nr. 5, Seiten 1299–1334. – ISSN 1573–7616

[TAA14] Tashtoush, Yahya M.; Al-Maolegi, Mohammed; Arkok, Bassam: The Correlation among Software Complexity Metrics with Case Study. In: *CoRR* abs/1408.4523 (2014)

A. Tasks

Listing A.1: Task 1

```
public int[] task1(int[] numbers, int a) {
        int[] array = new int[numbers.length];

        for (int i = 0; i< numbers.length; i++) {
            array[i] = numbers[i]*a;
        }

        for (int counter1 = 0; counter1 < array.length; counter1++) {
            for (int counter2 = counter1; counter2 > 0; counter2--) {
                if (array[counter2 - 1] > array[counter2]) {
                    int variable = array[counter2];
                    array[counter2] = array[counter2 - 1];
                    array[counter2 - 1] = variable;
                }
            }
        }

        return array;
    }
```

Listing A.2: Task 2

```
task2(String word) {
        for (int i = 0; i < word.length() / 2; i++) {
            int j = word.length() - 1 - i;
            if (word.charAt(i) != word.charAt(j)) {
                return false;
            }
        }
        return true;
    }
```

Listing A.3: Task 3

```
public boolean task3(int[] numbers) {
        int theOne = numbers[0];
        for (int i : numbers) {
            if (i > theOne) theOne = i;
        }

        for (int j = 2; j < theOne; j++) {
            if (theOne % j == 0) return false;
        }
        return true;
    }
```

Listing A.4: Task 4

```
public int task4(int[] array, int key) {
        int index1 = 0;
        int index2 = array.length - 1;

        while (index1 <= index2) {
            int m = (index1 + index2) / 2;
            if (key < array[m]) index2 = m-1;
            else if (key > array[m]) index1 = m+1;
            else return m;
        }
        return -1;
    }
```

Listing A.5: Task 5

```
public int task5(int num1, int num2) {
        int temp;
        do {
            if (num1 < num2) {
                temp = num1;
                num1 = num2;
                num2 = temp;
            }
            temp = num1 % num2;
            if (temp != 0) {
                num1 = num2;
                num2 = temp;
            }
        } while (temp != 0);
        return num2;
    }
```

Listing A.6: Task 6

```
public int task6(int number1, int number2) {
        int min, max;
        int result = -1;

        if (number1>number2) {
            max = number1;
            min = number2;
        } else {
            max = number2;
            min = number1;
        }

        for(int i=1; i<=min; i++) {
            if( (max*i)%min == 0 ) {
                result = i*max;
                break;
            }
        }

        return result;
    }
```

Listing A.7: Task 7

```
public int task7(int[] numbers) {
        int temp = 0;
        int variable = 0;
        int[] integers = new int[numbers.length];
        for (int j = 0; j < numbers.length; j++) {
            temp = numbers[j];
            for (int i = 0; temp > 0; i++) {
                variable = temp % 10;
                integers[j] = integers[j] + variable * (int)
                    Math.pow(2, i);
                temp = temp / 10;
            }
        }

        int a = 0;
        for (int num : integers) {
            a += num;
        }

        return a;
    }
```

B. Antworten der Retrospektive

B.a. Task 1

Person 1: Verwirrung, bis ich verstanden habe, dass die Zahlen sortiert werden. Habe mich ziemlich dumm gefühlt.

Person 2: Es war verwirrend, da plötzlich einige Variablen beim Zählen durch das Array kamen, über die man schlecht einen Überblick bekommen konnte. Vor allem die Funktion in der Schleife hat verwirrt

Person 3: * Benennungen der Variablen sind furchbar
* Kommentare im Code fehlen
* Verschachtelungen der Schleifen sind eigenartig

Person 4: Wenn man nur einen Teil des Codes sieht fällt der Gesamtüberblick schwerer, vor allem die Counter im Kopf zu behalten ist schwierig.
Aus diesem Grund habe ich nur grob überflogen, was die Funktion "berechnetünd meine Lösung daraus abgeleitet.

Person 5: 1. Multiplizieren
2. aha hmm durchiterieren, wenn größer tauschen, ganz oft → Bublesortartig

Person 6: Erst Skalieren und dann BubbleSort.
Etwas durcheinander gekommen, da counter1 inkrementiert und counter2 dekrementiert wird.

Person 7: Vor kurzem ähnliche Umfrage mit schwarzen Codeschnipseln und viel kleinerem Sichtfenster gemacht, die Navigation geht hier deutlich besser ... Signatur schauen Eingabe aufschreiben ... array gleicher länge wird erstellt ... erstelltes array wird initialisiert ... Werte aufschreiben ... for Schleifenblock anschauen ... Zählerlauf noch mal kompakter aufschreiben ... (c1: 0 → arr.l-1) (c2: c1 → 1) Schrittweite bei beiden eins in jeweilige Richtung ... Was passiert denn in der Schleife ... bedingter Swap elemente be c2, c2-1 ... hier also invariante über Elementreihenfolge in abhängigkeit der größe hergestellt ... Paare (c1, c2) durchspielen (0,1), (2,1), (1,0), ... erkenntnis so was wie Bubblesort ... noch mal schauen wie rum sortiert wird ... mit Zahlen durchgehen Ergebnis aufschreiben. Weiter

B.b. Task 2

Person 1: War einfach, weil kürzer als task1, zumindest gefühlt.

Person 2: es war etwas leichter. Ich merke, dass ich schon länger nicht mehr mit solchen Funktionen programmiert habe, da ich mich schwer tue, die Logik dahinter zu verstehen. Und ich glaube ich habe in der Aufgabe davor etwas falsch berechnet

Person 3: * sieht eigentlich einfach aus, hat aber doch länger gedauert als gedacht

Person 4: Diese Aufgabe war leichter zu lösen. Die Lösung erfolgte analog zu Aufgabe 1.

Person 5: i iteriert durch die hälfte, j fängt von hinten an, wenn i!=j also wenn der string nicht gespiegelt ist gibts falsch zurück

Person 6: Die Funktion überprüft, ob der String symmetrisch ist.

Person 7: Ok wieder komische Schleifen indizierung nur bis zur hälfte der länge ??? ... gegenläufigen Zähler, -1-i Wo geht dass los, oder besser wie wird das dann am Ende treffen die sich, Überlappung? gerade, ungerade? Jetzt keine Lust zu schauen ... erstmal körper prüfen ... Bedingung ... Wieder invariante diesmal über gesamte Funktion so lange wie indizierten Elementpaare gleich sind ist Resultat war sonst falsch ... → Palindrom erkennung Eingabe anschauen ... Ergebniss hinschreiben

B.c. Task 3

Person 1: theOne ist ein seltsamer Name für einen Bezeichner. Evtl 'biggest', wenn es aussagekräftig sein soll oder 'number' wenn nicht.

Person 2: Warum wird false zurück gegeben, wenn der Abgleich stimmt???

Person 3: * recht einfach
* max ist 7; 7 ist prim

Person 4: Im ersten Teil der Funktion wurde scheinbar das Maximum des Arrays gesucht, der zweite Teil erinnerte an die Bestimmung von Primzahlen. Da 7 das Maximum war und eine Primzahl ist gehe ich davon aus, das die Lösung 'true' ist.

Person 5: gehe durch array finde maximum
Maximum = 7
rechne maximum mod 2 → 7%2=1 return false
→ True wenn die Zahl durch alle vorherigen teilbar ist

Im nachhinein: vermutlich hab ich die gleichheit vertauscht und es sollte nach Primzahlen suchen

Person 6: Zunächst wird das Maximum des Arrays bestimmt und danach geprüft, ob es eine Primzahl ist (wobei Zahlen ¡= 1 hier auch als Primzahl erkannt werden würden).

Person 7: theOne ? echt jetzt ? ... ok für die Invarianten und klaren Funktionsblöcke wie Initialisierungen einzelne funktionen zu erstellen und semantisch aussagekräftig zu benennen geht ja vielleicht bei einer Aufgabe über Codestruktur schlecht aber Variablennamen wie der ... den Code möchte ich nicht warten (Jedenfalls interpretiere ich so dass an dieser Stelle aufkommende Gefühl, nur ein sehr schwaches, irgendwo zwischen verärgerung und ekel einzuordnen) ... weiter schauen nächster Abschnitt offenbart besseren Namen: Maximum ... der Rest Teilbarkeit der Zahl durch alle kleineren ab 2 prüfen um so funktionsinvariante, Ergebnis ist false, wenn das Maximum der Elemente im gegebenen Array keine Primzahl ist, herzustellen

B.d. Task 4

Person 1: Schnell erkannt, dass es sich um eine Suche handelt, dann den code nur noch überflogen.

Person 2: Das macht mich fertig. Dieses Mal ging die Logik, aber dass am Schluss trotzdem -1 zurück gegeben wird war ein curveball

Person 3: * bei der Division von integer musste ich nachdenken

Person 4: Es könnte sich bei der Funktion um eine binäre Suche handeln. Gesucht wurde das Element 20, welches sich an Position 3 (bei 1. Element = 0) befindet.

Person 5: index1 ist am anfang
index2 ist am ende
prüfe den mittleren Index
verändere index1 bzw. index2
→ suche in sortiererter Liste.

Person 6: Die Funktion sucht in dem Array nach der Position des Integers key (falls im Array vorhanden). Das Suchintervall wird pro Schritt nur um 1 verkleinert, also lineare Laufzeit. Das Array muss für diese Art von Suche aufsteigend sortiert sein.

Person 7: i1 low, i2 high, ... m ist Durchschnitt aus i1 und i2 ... Element bei m mit key vergleichen grenze Anpassen oder m ausgeben ... bis low nicht mehr über high liegt oder ke gefunden wurde, also binäre Suche, eingabe anschauen, Ergebniss aufschreiben

B.e. Task 5

Person 1: Weis nicht

Person 2: Zwei Schleifendurchläufe sind richtig fies zum berechnen. Am liebsten hätte ich mir ein Blatt Papier genommen, um händisch mitzurechnen. Das Aufrufen der Zeilen ging relativ, inzwischen ist eine Routine drin und ich kann mir relativ leicht merken, wo welcher Codeschnipsel ist

Person 3: * bin leicht angestrengt

Person 4: Im Nachhinein würde ich sagen, task5 war eine ggT-Funktion. Während der Bearbeitung der Aufgabe war es etwas schwierig für mich nachzuvollziehen bzw. mir zu merken welche Werte num1 und num2 jetzt genau besitzen, da viele verschiedene Zuweisungen an diese Variablen getätigt wurden.

Person 5: Tausche input so dass num1 größer ist
num1 modulu num2
solange !=0 tausche rum und entferne die größere Zahl
→ finde den größten gemeinsamen teiler

Person 6: Die Funktion setzt den Euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen um.

Person 7: Um ehrlich zu sein werden die Gedanken die ich habe immer weniger, der je mehr ich mit diesen Aufgaben Zeit verbringe desto weniger Bewusst wird das bearbeiten. Vielleicht sinkt auch nur die Konzentration, so dass das Bearbeiten der Aufgabe mehr geistige Ressourcen verbraucht und dadurch die Geistigen Ressourcen nicht mehr zur Wahrnehmung der Reflektion ausreichen oder es wandert der Fokus mehr und mehr zu Problemlösestrategien und weg von kommunikativen oder sozial reflektierenden strategien. Also ich denke eine kurze Pause wäre angebracht. dieser Gedanke kam mir irgendwann in der mitte der bearbeitung. Ansonsten war nur: Hier wieder Swap so dass num1 immer ¿= num2 ... teiler ermitteln teilbarkeit auf 0 prüfen ... wiederhohlen, sieht nach euklidschen Algorithmus zum finden des ggT aus, will ich aber nicht im kopf prüfen, Eingabe müsste schneller durchzuspielen sein. Stimmt auch dafür ... hinschreiben.

B.f. Task 6

Person 1: Bessere Bezeichnernamen als in den anderen Aufgaben (min, max) haben mir beim Verständnis geholfen

Person 2: Die Aufgabe hat sich zumindest leichter angefühlt als die ersten beiden. Eventuell komm ich aber inzwischen schon wieder mehr in das Denken rein. Habe auch die Eingabeleiste genutzt, um Notizen zu machen, falls sich wieder mal die Reihenfolge von 6, 8 ändern sollte oder so ein Spaß

Person 3: * Konzentration lässt nach

Person 4: Der erste Teil wo min und max festgelegt wurden sind war einfach nachzuvollziehen, um auf das Ergebnis zu kommen bin ich jedoch einmal die for-Schleife komplett im Kopf durchgegangen.
kgV-Funktion?

Person 5: bestimme min und max
tue zeugs
hmm vermutlich größtes gemeinsames vielfaches
ja genau das ist es
8 und 10 $\rightarrow$ 40

Person 6: Die Funktion bestimmt das kleinste gemeinsame Vielfache zweier ganzer Zahlen.

Person 7: Ah hier ist das finden von max und min schön verständlich, gehört jetzt nur noch in eigene Funktion ... da hat man doch gleich wieder lust die Schleifeninvariante genau zu verstehen und nicht nur die Eingabe durchzugehen ... warum wird hier einmal max*i und einmal i*max gerechnet würde eins nicht reichen und dann in einer Variablen mit guten Namen zwischenspeichern wie MinsMultiple ... die Namen sind ja doch irreführend ... hier geht es gar nicht um Minimum und Maximum sondern nur um eine größere und eine kleinere Zahl ... als name der zwischengespeicherten Variable wäre so was wie NextSmallestMultiple nicht schlecht ... und damit auch die Funktion des ganzen einfacher zu erkennen es wird das kgV der Eingabezahlen ermittelt ... kgV Eingabezahlen ermitteln hinschreiben

B.g. Task 7

Person 1: Ich hab immer noch nicht ganz verstanden was der erste Teil macht, also bin ich dass im Kopf Schritt für Schritt durchgegangen

Person 2: Sorry, das war zu viel für mich. Da komm ich im Kopf nicht mehr mit. Zu viele Variablen mit zu vielen Berechnungen, die man sich auf einmal merken muss. Papier will ich nicht benutzen, da nirgends steht, dass es als Hilfsmittel erlaubt ist und die Zeile reicht nicht mehr aus

Person 3: Ich mag nicht mehr

Person 4: Hier bin ich mir nicht ganz sicher, ob ich den Code verstanden habe. Also falls beim Anlegen des integer-Arrays die Feldwerte mit 0 initialisiert werden sollte als Ergebnis 1 herauskommen. Ich habe den Code so verstanden dass die Feldwerte uninitialisiert sind (aber das liegt vlt. auch daran, dass ich bisher nicht viel mit JAVA gearbeitet habe), also würde in der Zeile integer[j] = integer[j] + ... mit unbekannten Werten gerechnet werden.

Person 5: Langsam wirds doch öde
was tut der? hm:
- zeug durchiterieren
- modulu/div zeug machen also die einzelnen dezimalstellen auswerten
- diese dann in integer packen allerdings mit basis 2 statt 10
- am ende aufsummieren
→ input wird als binärzahlen betrachtet und eine summe gebildet

Person 6: Die Funktion interpretiert die übergebenen Integer als Binärzahlen und berechnet jeweils die korrespondierende Dezimalzahl. Danach werden alle Zahlen im Array addiert und das Ergebnis zurückgegeben.

Person 7: Erst mal der Obere Teil sieht interessant aber nicht all zu leicht aus ... ersmal schauen ob man den überhaupt braucht ... a wird zurückgegeben ... a ist die Summe der Elemente von Integers ... ok dafür braucht man jetzt doch den Rest also mal schauen ... gut hier werden immer die Reste bei Division durch 10 abgespalten und diese dienen dann als faktoren für Zweierpotenzen, und das wird auf jeden fall für alle Elemente des Eingabearrays gemacht ... weis nicht wie man das nennt aber es wird sozusagen so getan als währen die Ziffern der Eingabezahlen im Zehnersystem die der Zahlen des integer feldes im dualsystem ... dann noch zusammen rechnen und fertig.