Institut für Visualisierung und Interaktive Systeme
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelorarbeit Nr. 289

Visuelle Eyetracking-Analyse von Quellcodedarstellungen

Aretina Iazzolino

Studiengang:	Informatik
Prüfer/in:	Prof. Daniel Weiskopf
Betreuer/in:	Dipl.-Inf. Kuno Kurzhals, Dipl.-Inf. Dr. rer. nat. Fabian Beck
Beginn am:	20. November 2015
Beendet am:	24. Mai 2016
CR-Nummer:	H.5.1,H.5.2

Kurzfassung

Im Entstehungsprozess von Software gibt es eine Reihe von Vorgänge, um die Qualität und die Sicherheit eines Systems zu testen. Die Codeanalyse ist eines der wichtigsten Vorgänge, die in allen Phasen der Entwicklung vorgenommen wird. Es gilt, den Fehler so schnell wie möglich zu finden, da das Beheben von Fehlern in späteren Phasen der Entwicklung oftmals mit höheren Kosten verbunden ist. Der Wunsch, in Zukunft diesen Vorgang so effizient wie möglich zu gestallten, führt zu zahlreichen Studien, die sich intensiv mit der Analyse von Review Vorgänge beschäftigen. Es ist dennoch schwer, einen allgemeinen Analysevorgang zu bestimmen, da menschliche Faktoren und die individuelle Performance einen prädominanten Einfluss beim Review Vorgang haben. Ziel dieser Arbeit ist es, die Durchführung einer Benutzerstudie, sowie die Erstellung eines Werkzeugs, für die visuelle Analyse der Daten der Benutzerstudie. Mit Hilfe eines Eyetrackers, soll das Blickverhalten der Teilnehmer aufgenommen werden und anschließend mit dem entwickelten Werkzeug analysiert werden.

Abstract

During the development process of software there are various stages to test the quality and safety of a system. Code analysis is one of the most important operations that is performed at all stages of development. It is meant to find and fix errors as quickly as possible, since doing it in later stages of development is often associated with higher costs. It is still difficult to determine a general analysis approach, because human factors and individual performance have a predominant influence in the review process. The aim of this work is to conduct a user study, as well as the creation of a tool for visual analysis of the obtained data from the user study. Using an eyetracking system, gaze behavior of the participants should be captured and finally analyzed with the developed tool.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Code-Review, was unter anderem das Verständnis und die Fehlersuche im Programm beinhaltet, ist einer der wichtigsten Vorgänge bei der Software Entwicklung. Durch die Inspektion des Codes soll fortlaufend die korrekte Funktionsweise des Produktes getestet und die Reduzierung von Fehlern ermöglicht werden. Denn das Beheben von Fehlern in späteren Phasen der Entwicklung ist oft mit höheren Kosten verbunden. Die zentrale Frage, mit der sich Wissenschaftler seit Jahren beschäftigen ist, wie man diesen Prozess so effizient wie möglich gestalten kann. Dabei sind viele Techniken und Prinzipien entstanden, um den Review Vorgang zu unterstützen, wie beispielsweise Style-Guides, die als Ziel haben dem Programmierer eine Richtlinie bei der Software Entwicklung zu geben und auch helfen sollen, eine Struktur im Code zu schaffen, damit Aufgaben wie Wiederverwendbarkeit und Wartbarkeit durch eine Dritte Person mit geringem Aufwand möglich sind. Auch durch Code-Highlighting, also bestimmte Stellen oder Schlüsselwörter farblich hervorzuheben wurde die Verständlichkeit unterstützt. Zahlreiche Studien, wie z.B. die von Feigenspan [FKA+13], haben sich genau mit dieser Frage auseinandergesetzt, und mit farblicher Hintergrund Markierung bestimmter Bereiche festgestellt, dass dadurch das Programmverständnis durchaus gesteigert werden kann. Da beim Programmverständnis kognitive Prozesse involviert sind, die man nicht direkt beobachten kann [KR91], hat man in den letzten 15 Jahren Eyetracking Studien durchgeführt, die den kognitiven Vorgang von Menschen bei der Code-Inspektion analysierten. Dabei wurden besondere Muster entdeckt, wie z.B. das Scan-Muster [UNM+06], die im weiteren Verlauf näher erläutert werden.

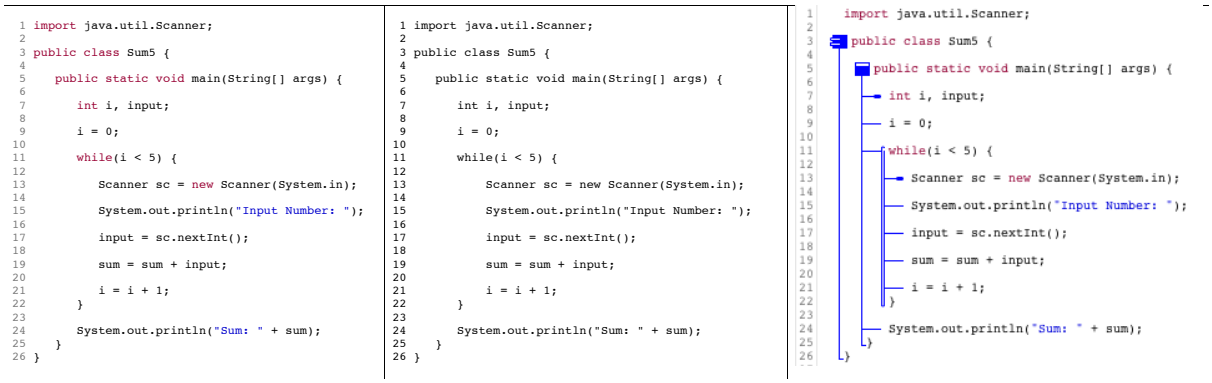
Mit Hilfe der seit über hundert Jahren erforschten und weiterentwickelten Eyetracking Technologie und dank seiner vielfältigen Anwendungsmöglichkeit in vielen Bereichen, ist es nicht nur möglich, die Augenbewegungen von Menschen zu erfassen und zu analysieren es wurde in vielen Studien auch gezeigt, dass man Rückschlüsse auf den Prozess der Wahrnehmung ziehen kann. Die Ergebnisse solcher Studien und die damit verbundene Kenntnis über die Kognition, soll in Zukunft idealerweise den Inspektionsaufwand minimieren und somit den essentiellen Vorgang des Code-Reviews bei der Software-Entwicklung effizienter gestallten.

Durch die Erfassung der Blickbewegungen mittels Eyetracking entsteht eine Menge von Daten. Eine große Herausforderung ist die Repräsentation dieser selbst. Die einfache und vor allem verständliche Darstellung von Daten spielt eine wichtige Rolle in vielen Bereichen. Dabei soll die Information besser dargestellt werden, nicht nur aus ästhetischen Gründen, sondern um sie intuitiver erfassen zu können. Aus graphischen Repräsentationen ist es oft einfacher gewisse Verhaltensmuster zu erkennen, oder schwierige Konzepte verständlicher zu gestallten.

Im Vergleich zur ausgereiften Eyetracking Technologie sind die teilweise nur beschränkt existierenden Visualisierungstechniken, insbesondere in Verbindung mit Code-Review, der Grund für die Entstehung dieser Arbeit.

Der Umfang stellt sich zusammen aus einer durchgeführten Studie und die Erstellung eines Visualisierungswerkzeugs für die aufgenommenen Daten. Die Studie dient dazu, mit Hilfe eines Eyetracking Apparates, die Blickbewegungen der Teilnehmer zu erfassen. Die Aufgabe besteht darin, Java Programme, die jeweils in drei unterschiedlichen Darstellungen den Teilnehmern präsentiert werden, visuell auf semantischen Fehlern zu durchsuchen. Abbildung 1.1 zeigt die drei unterschiedlichen Darstellungen - Plain Text, Java Syntax Highlighting und letzteres noch zusätzlich mit Control Structure Diagrams(CSD) angereichert, die im weiteren Verlauf detaillierter erklärt werden.

Das Visualisierungswerkzeug wird für die retrospektive Analyse der aufgenommenen Daten entwickelt. Mit Hilfe dessen soll insbesondere analysiert werden, wie Menschen mit unterschiedlicher Programmiererfahrung individuell nach Fehlern im Programmcode suchen und in welchem Maße die verwendeten Darstellungen zum Verständnis und bei der Fehlersuche hilfreich waren. Die Werkzeuge, die in den Vergangenen Studien verwendet wurden, waren beispielsweise in der Lage den Review Vorgang in Form einer Timeline darzustellen, in welcher man den Lesefluss des Teilnehmers nachempfinden konnte. Das entwickelte Visualisierungswerkzeug soll zusätzlich noch ermöglichen, Bereiche von besonderem Interesse, sogenannte Areas of Interest (AOI), die auf den Programmen definiert werden, in der Timeline farblich zu unterscheiden. Dadurch kann man den Fokus auf spezielle Komponenten des Codes setzen, beispielsweise auf Schlüsselwörter, CSD- oder Kontrollstrukturen, um somit eine detaillierte Analyse durchführen zu können. Zum Schluss werden anhand der Visualisierung die Ergebnisse der Teilnehmer untereinander verglichen und ausgewertet.



```
import java.util.Scanner;

public class Sum5 {

    public static void main(String[] args) {

        int i, input;

        i = 0;

        while(i < 5) {

            Scanner sc = new Scanner(System.in);

            System.out.println("Input Number: ");

            input = sc.nextInt();

            sum = sum + input;

            i = i + 1;
        }

        System.out.println("Sum: " + sum);
    }
}
```

Abbildung 1.1: Code Darstellungen

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Einführung in Eyetracking: In Kapitel 2 wird zuerst eine Einführung in die Anatomie des menschlichen Auges gegeben, sowie einige Wahrnehmungsstrategien und die Eyetracking Technologie vorgestellt.

Kapitel 3 – Verwandte Arbeiten: Kapitel 3 ist der Arbeit gewidmet, die in den letzten Jahren im Bereich Code-Review in Verbindung mit Eyetracking absolviert wurde.

Kapitel 4 – Visualisierungswerkzeug: Kapitel 4 beschreibt die konkrete Realisierung des Visualisierungswerkzeugs sowie die Funktionsweise.

Kapitel 5 – Eyetracking-Studie: In Kapitel 5 wird der Ablauf der durchgeführten Studie vorgestellt.

Kapitel 6 – Auswertung: Kapitel 6 umschließt die gewonnenen Erkenntnisse aus der Studie.

Kapitel 7 – Diskussion In Kapitel 7 werden die Ergebnisse aus Kapitel 6 diskutiert.

Kapitel 8 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Einführung in Eyetracking

2.1 Motivation zur Forschung der Blickbewegung

Das Beobachten und Analysieren menschlicher Verhaltensweisen liegt uns in der Natur. Das Auge, zusammen mit den restlichen Sinnesorganen, stellt die Schnittstelle zur Umgebung dar. Das ermöglicht uns mit der Außenwelt zu interagieren. Studien haben bewiesen, dass etwa 80 % der Perzeption [Züh11] über das Auge erfolgt, wodurch ständig Reize, auch Stimuli genannt, aus der Umgebung wahrgenommen werden. Deshalb ist es bei Eyetracking Studien interessant die Augenbewegungen zu untersuchen, um Informationen über Wahrnehmungs- und Verständnisprozesse zu erhalten. Das Auge ist somit eines der wichtigsten Sinnesorgane des Menschen. Im Gehirn findet der Prozess der Umwandlung dieser Reize statt, wodurch dann eine individuelle Reaktion auf den eingehenden Stimulus determiniert wird. In diesem Kapitel wird eine kurze Einführung in die Anatomie des menschlichen Auges gegeben. Dies soll dazu dienen, die Funktionsweise der Informationsverarbeitung zu verstehen. Anhand der Videookulographie, die wichtigste Technik die heutzutage verwendet wird um Blickbewegungen aufzunehmen, soll erklärt werden wie Eyetracking funktioniert. Der Unterschied zwischen stationäre und head-mounted Systeme, sowie wichtige Begriffe die in Verbindung mit Code-Review in Kapitel drei nochmal genauer erklärt werden, werden auch vorgestellt. Zum Schluss werden wichtige Datentypen vorgestellt die von Eyetrackern aufgezeichnet werden und für diese Arbeit auch relevant sind.

2.2 Die Anatomie des Auges

Abbildung 2.1 ist eine Illustration des Querschnittes des menschlichen Auges. Der hintere Teil des Glaskörpers, ab den Ziliarmuskeln, hat eine kreisrunde Form, während im vorderen Bereich die Hornhaut eine deutlich hervorgehobene Wölbung aufzeigt. Im Innenraum des Glaskörpers befindet sich die Netzhaut, die mit Rezeptoren ausgestattet ist. Sie erreichen ihre maximale Dichte dort, wo die optische Achse des Auges die Netzhaut schneidet, in der *fovea centralis*. An dieser Stelle ist es uns möglich am schärfsten zu sehen. Je weiter man sich von diesem Bereich entfernt, desto rapide fällt die Rezeptordichte der Netzhaut ab. Die Aufmerksamkeit auf ein bestimmtes Objekt, hängt somit eng mit der Lage der optischen Achse zusammen. Insgesamt beträgt die Größe des visuellen Feldes etwa 130° vertikal und 180° horizontal [Duc07]. Abbildung 2.2 zeigt das Innenleben des menschlichen Auges und

verdeutlicht anhand des Aufbaues der Rezeptoren, die aus Stäben und Zapfen besteht (engl. Rod and Cones), die eben beschriebene Funktionsweise. Die Zapfen sind insbesondere für das Farbsehen verantwortlich, während die Stäbe für das Helligkeitsempfinden zuständig sind. Der Bereich, in dem keine Rezeptoren vorhanden sind, da sie in den Sehnerv übergehen, wird *blinder Felck* genannt.

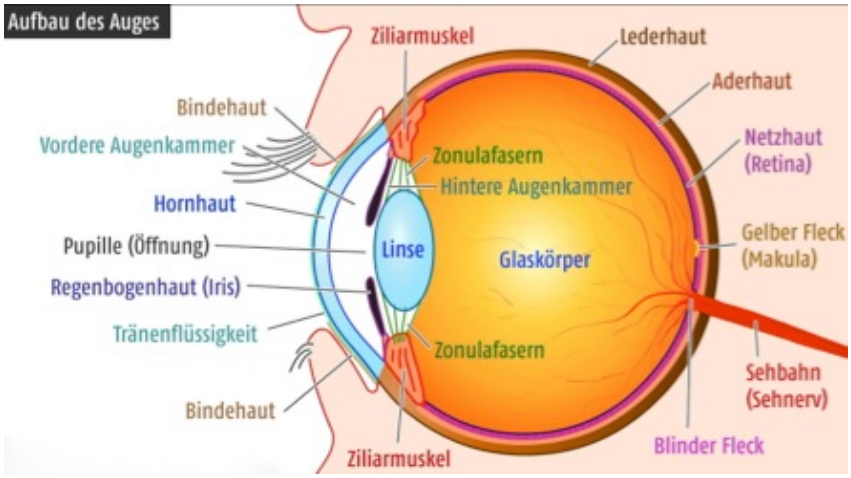


Abbildung 2.1: [http://www.lasikon.de/]. Aufbau des menschlichen Auges. Hier sind die einzelnen Bestandteile des Auges zu erkennen die oben erklärt wurden.

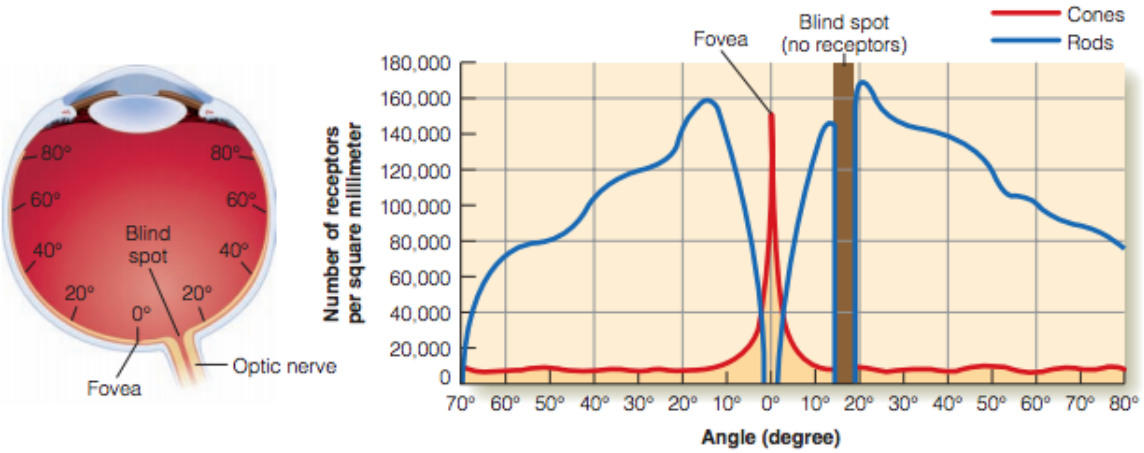


Abbildung 2.2: Dieses Bild [GB09] zeigt die Verteilung der Stäbe und Zapfen in der Netzhaut. Das Linke Auge zeigt insbesondere die Lage auf der Netzhaut, die in Grad, relativ zur *fovea centralis*, angegeben wird. In der rechten Abbildung, entspricht dies der x-Achse. Der braune Balken stellt den *blinden Fleck* dar, der keine Rezeptoren besitzt.

2.2.1 Die Augenbewegungen

Die zwei wichtigsten Augenbewegungen, die unbewusst im Alltag am häufigsten stattfinden werden nun vorgestellt [Duc07]:

Die Fixation

Fixationen sind im engeren Sinne keine Augenbewegung, sondern viel mehr ein zeitlich begrenzter Moment, in welchem die Aufmerksamkeit auf ein Objekt gerichtet ist. Die Dauer einer Fixation beträgt im Durchschnitt 100– 600ms. Nur hier fängt das Gehirn an die visuelle Information zu verarbeiten.

Die Sakkade

Sakkaden sind schnelle sprunghafte Augenbewegungen zwischen Fixationen. Die Durchschnittsdauer einer Sakkade beträgt zwischen 10ms und 100ms.

2.2.2 Wahrnehmung und Informationsverarbeitung

Im Allgemeinen lassen sich zwei Vorgehensweisen beim Wahrnehmungsprozess unterscheiden, die *Bottom-Up* und die *Top-Down* Strategie [GB09]. Beide Strategien werden in Kapitel drei nochmal aufgegriffen und die genauere Verbindung zum Programmverständis wird hergestellt.

Bottom-up

Das Bottom-up Verfahren, auch data-based Verfahren genannt [GB09], beschreibt die Datenverarbeitung der von der Außenwelt eingehenden Information. Durch das einfallende Licht auf die Rezeptoren der Netzhaut, wird die Information visuell aufgenommen. Erst durch das Erkennen von Eigenschaften, das im Gehirn stattfindet, erhält das betrachtete Objekt eine exakte Form und einen Namen.

Top-Down

Die Top-Down Strategie, auch knowledge-based Verfahren genannt [GB09], stellt ein Verarbeitungsmechanismus dar, das sich auf schon vorhandenes Wissen stützt. Das bedeutet, der Betrachter stellt eine allgemeine Hypothese über das betrachtete Objekt auf. Wenn beim Betrachten diese Eigenschaften mit dem übereinstimmen, was angenommen wurde, findet der Prozess der Erkennung und Kategorisierung des Objektes statt.

In der Regel finden beide Prozesse auf natürliche und nebenläufige Weise statt. Das Zusammenspiel beider Strategien wird in Abbildung 2.2 gezeigt.

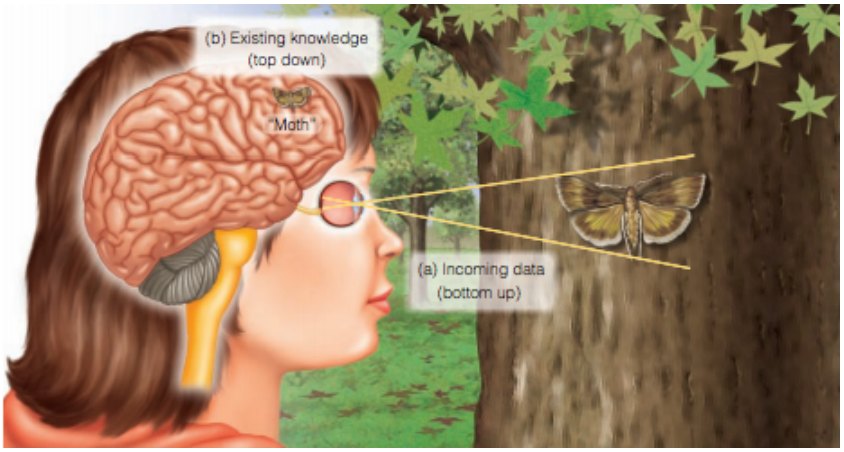


Abbildung 2.3: In diesem Bild[GB09] wird die Wahrnehmung und das Zusammenspiel des *Bottom-Up* und *Top-Down* Verfahren deutlich. Dabei wird das betrachtete Objekt zunächst via *Bottom-Up* Verfahren auf die Rezeptoren abgebildet. Anschließen wird mittels *Top-Down* das benötigte Wissen abgerufen, um das Objekt zu erkennen.

Präattentive Percepeption

Die Präattentive Wahrnehmung ist ein Mechanismus der uns erlaubt unbewusst Informationen aufzunehmen. Diese Art von Information wird innerhalb von 200–250 ms aufgenommen. Eigenschaften wie die Farbe oder Form eines Objekts, gehören zu dieser Wahrnehmungskategorie[GB09].

2.3 Eyetracking

Zur Erfassung und Analyse visueller Reize bedarf es spezieller Werkzeuge wie z.B. der Eyetracker Tobii T60XL, welcher die Augenbewegungen des menschlichen Auges registriert und

in dieser Studie auch zum Einsatz kommt. Anhand der aufgenommen Daten kann dann im einem nächsten Schritt analysiert werden, wie die Probanden die Stimuli betrachtet haben. Der folgende Abschnitt stellt die aktuellen Systeme vor und erklärt die Technik der Videookulographie, die bei modernen Eytrackern zum Einsatz kommt und für die Blickbewegunsaufnahme eingesetzt wird.

2.3.1 Eyetracking Techniken und Systeme

Bei den Eyetrackern die es aktuell auf dem Markt gibt, unterscheidet man zwischen *stationäre* und *head-mounted* Systeme. Die Wahl des Eyetrackers ist stark von der gestellten Aufgabe abhängig. Beim Betrachten von Webseiten oder Benutzeroberflächen wird oft ein stationäres System eingesetzt. Während man für Aufgaben die mehr Bewegungsfreiheit erfordern gerne head-mounted Systeme einsetzt. In diesem Kapitel wird der unterschied zwischen stationären und head-moundet Systemen erklärt, die Technik der Videookulographie vorgestellt, sowie weitere Datentypen erklärt, die bei Eyetrackingstudien aufgenommen werden.

Stationäre Systeme

Stationäre, auch Remote Eyetracking Systeme genannt, sind direkt im Monitor eines Computers integriert oder können frei im Raum aufgestellt werden. Der Eyetracker von der Firma Tobii T60XL, der auch in dieser Studie zum Einsatz kommt, gehört zu dieser Kategorie. Diese Eyetracker basieren auf der Technik der Videookulaographie[Duc07], welche Infrarotstrahlen verwendet, um die Blickbewegungen aufzufassen. Die Infrarotstrahlen werden vom Eyetracker ausgesendet, die dann von der Hornhaut reflektiert werden und anschließend von einer Kamera aufgenommen werden. Dadurch ist es dem Teilnehmer gestattet, bis zu einem gewissen Grad uneingeschränkt mit dem Gerät zu interagieren und sich relativ frei zu bewegen.

Head-Mounted Systeme

Bei head-mounted Geräten wird das System hingegen direkt im Rahmen einer Brille montiert. Dadurch kann sich der Benutzer frei im Raum bewegen, ohne an ein bestimmtes Gerät gebunden zu sein. Sie werden oft bei Usability Testing verwendet, um z.B. das Kaufverhalten in einem Supermarkt zu analysieren.

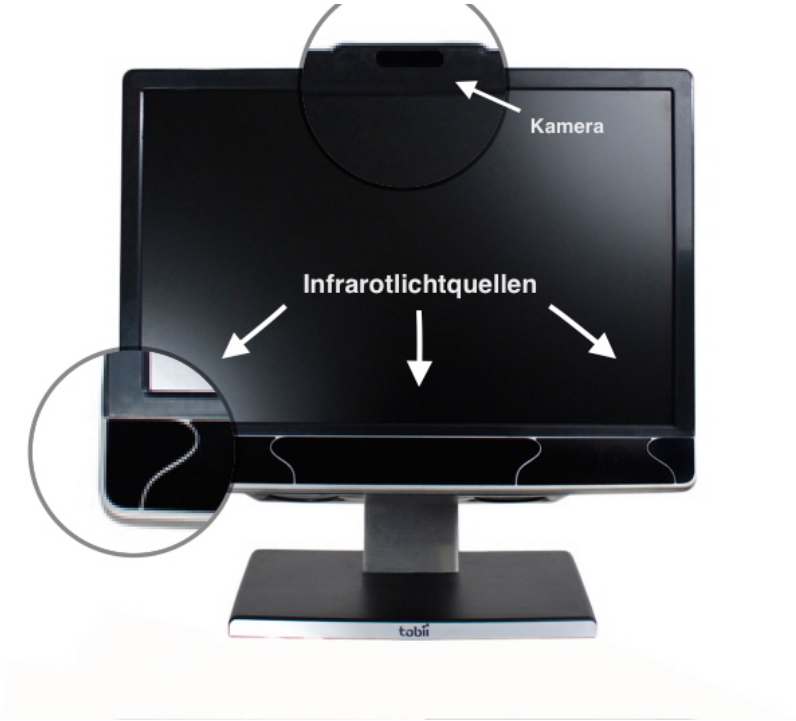


Abbildung 2.4: [www.tobii.com]Remoter Eyetracker Tobii T60XL des Herstellers Tobii.

2.3.2 Visualisierung von Eyetrackingdaten

Zu den wichtigsten Visualisierungstechniken gehören Scanpaths und Heatmaps.

Heatmap

Mit Hilfe von Heatmaps ist es mölich, aggregierte Fixationen zu visualisieren. Dadurch hat man eine generelle Sicht der Daten und man erkennt, welche Bereiche besonders stark fixiert wurden. Diese Bereiche werden auf dem Stimulus farblich hervorgehoben. Dabei wird die Fixationsdichte durch einen Farbverlauf von grün nach rot dargestellt. Je höher die Fixationsdichte in einem Bereich ist, desto rötlicher wird dieser Bereich gefärbt [BKR+14].

Scanpath

Ein Scanpath stellt die Augenbewegungen in Form eines Pfades dar. Der Pfad besteht aus Fixationen, die als Kreise dargestellt werden und aus Sakkaden, die als Linien zwischen diesen Kreisen gezeichnet werden. Dabei wird der Kreis größer, je länger man einen Punkt fixiert. Zusätzlich werden die Punkte im Scanpath durchnummeriert, wodurch nachvollziehbar ist, in welcher Reihenfolge die Punkte angeschaut werden [BKR+14].

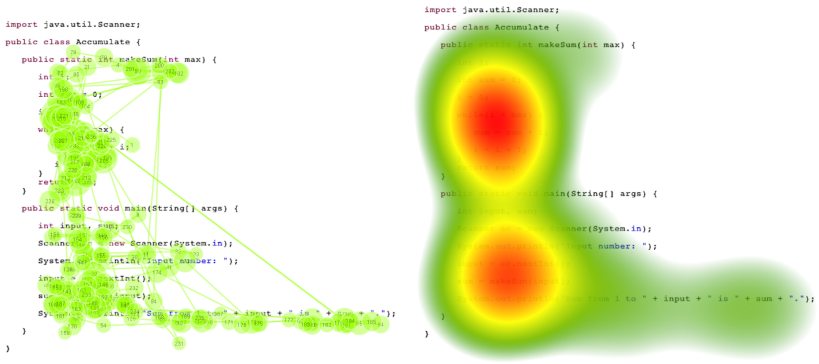


Abbildung 2.5: Bild eines Scanpaths (links) und einer Heatmap (rechts).

2.3.3 Datentypen

Die Datentypen die vom Eyetracker geliefert werden und auch in der Benutzer-Studie eine zentrale Rolle spielen werden nun vorgestellt und erklärt. Die in Abschnitt 2.2.2 eingeführten Bewegungen, Fixation und Sakkade, gehören auch dazu [TobiiStudio2.XUserManual].

Stimulus

Als Stimulus bezeichnet man in einer Benutzerstudie alles was die visuelle Aufmerksamkeit des Teilnehmers fängt. In diesem Kontext ist das ein statisches Bild, also die präsentierten Java Programme. Es kann aber auch ein Video oder eine Webseite als Stimulus verwendet werden.

Gaze

Gaze gilt als Oberbegriff für eine Ansammlung von Fixationen in einem bestimmten Bereich, sogenannte Area of Interest.

Area of Interest (AOI)

Eine Area of Interest schränkt einen Bereich auf dem Stimulus ein, dem man besondere Aufmerksamkeit widmet und über welchen Daten aufgenommen werden. Dadurch kann man

bestimmten AOIs, Fixationen zuordnen und gruppiert analysieren. Der Eyetracker annotiert zu jeder AOI eine AoiID und einen AoiName.

Timestamp

Der Zeitstempel gibt die Zeit in Millisekunden an, über die gesamte Dauer in der die Blickdaten aufgenommen werden.

MappedFixationPointX/Y

MappedFixationPointX/Y gibt jeweils die x und y Koordinate des Auges an. Sie werden relativ zum Ursprung des Koordinatensystems angegeben, wobei der Ursprung der obere linke Punkt ist.

3 Verwandte Arbeiten

Code-Review ist ein Verfahren, das angewendet wird, um Fehler im Code zu entdecken. Es wird hauptsächlich von Menschen durchgeführt, ohne das Programm auszuführen. Bei der Entwicklung von Software nimmt es eine wichtige Rolle ein, denn dadurch wird fortlaufend die korrekte Funktionsweise des Produktes geprüft. Zudem ist es wichtig Fehler im Code frühzeitig zu entdecken. Es gilt, je früher ein Fehler entdeckt wird, desto günstiger ist es ihn zu beheben [Boe+81]. Abbildung 3.1 stellt die wichtigsten Phasen der Software-Entwicklung dar. Dabei ist deutlich zu erkennen, wie die Kosten für das Finden und Beheben von Fehlern exponentiell mit fortschreiten der Entwicklungsphasen zunimmt.

Abbildung 3.1: [http://de.slideshare.net/]. Die Graphik betont das exponentielle Wachstum der Kosten um einen Fehler zu beheben in jeder Entwicklungsphase von Software.

Weigers [Wie96] verdeutlicht die Wichtigkeit von Code-Review durch eine seiner Studien. Er fand heraus, dass dadurch 50-70% der Fehler entdeckt werden können. Wenn man zusätzlich den Aspekt betrachtet, dass die Wartung der Software meist nicht von derselben Person

durchgeführt wird, die das Programm geschrieben hat, wird die Wichtigkeit des Programmverständnisses noch deutlicher. Programmverständnis ist jedoch etwas sehr individuelles, was von vielen menschlichen Faktoren abhängt, wie zum Beispiel die Programmiererfahrung. Deshalb ist es von besonderem Interesse die kognitiven Prozesse bei diesem Vorgang genauer zu untersuchen und auf Erkenntnisse zu stoßen, die in Zukunft den Inspektionsaufwand effizienter und so gering wie möglich halten. Uwano [UNM+06] setzt als langfristiges Ziel, ein effizientes Verfahren zu schaffen, dass dem Reviewer erlaubt so viele Mängel wie möglich im Softwareprodukt zu finden.

3.1 Programmverständnis Modelle

Es gibt unterschiedliche Stratiegien, die jeder Reviewer bewusst oder unbewusst beim analysieren eines Quellcodes verfolgt. Uwano zitiert unter anderem Methodologien wie das *Ad-Hoc Review (AHR)*, bei welchem man den Code ohne bestimmte Kriterien liest. Das *Perspektive Based Review*, wo man den Code aus unterschiedlichen Perspektiven liest (z.B. Programmierer, Tester etc.), *Checklist Based Review*, mit welchem man den Code anhand einer Liste typischer Fehler untersucht und weitere, die an dieser Stelle nicht weiter erläutert werden. Diese Methoden haben als Ziel gewisse allgemeingültige Kriterien beim Lesen zu schaffen, um der Inspektion einen durchdachten Ablauf zu geben. Jedoch haben viele empirische Studien gezeigt, dass man keine genauen Aussagen treffen kann, welche der Methoden mehr Fehler im Code aufdeckt [LV00].

In vergangenen Studien haben sich einige Modelle herauskristallisiert, wie Programmierer den Quellcode perzipieren und lesen [FKA+13]. Allgemein gibt es Programmverständnismodelle wie *Top-Down, Bottom-Up und Integrated Modells*. Die ersten zwei Begriffe wurden Allgemein in Kapitel 2 eingeführt. In Verbindung mit Programmkognition kann man sich das folgendermaßen vorstellen: Ein Entwickler setzt das *Top-Down* Verfahren ein, wenn schon gewisse Vorkenntnisse vorhanden sind, z.B. wenn er die Programmiersprache des Quelltextes beherrscht. Dabei formuliert er eine generelle Hypothese zur Funktionsweise des Programms, die er dann verifiziert oder falsifiziert. Beim *Bottom-Up* Verfahren hingegen wird der Code stückweise analysiert. Diese Stücke werden dann zu größeren kombiniert. Der Programmierer puzzelt sich somit sein Wissen zusammen. *Integrierte Modelle* sind ein Hybrid aus *Bottom-Up* und *Top-Down*. Generell verwendet ein Programmierer *Top-Down* wo es möglich ist und *Bottom-Up* wo es nötig ist [FKA+13]. Diese Modelle sind eine gute Grundlage, um den Programmierer anhand seiner Fähigkeiten zu Kategorisieren. Was jedoch an dieser Stelle fehlt ist eine Metrik, um das Programmverständnis zu messen.

3.1.1 Programmverständnis messen

Eine der zentralen Fragen vieler Studien ist, wie man Programmverständnis am besten messen kann. Durch konkrete Metriken, wie z.B. die durchschnittliche Antwortzeit, oder die Korrektheit

der Antwort lässt sich Programmverständnis bis zu einem gewissen Grad messen. Wenn aber kognitive Prozesse involviert sind, die schwer zu messen sind, muss man nach alternativen Methoden suchen, wie im Folgenden deutlich wird. In vielen Studien wird die Arbeit von Ericson und Simon [1984] zitiert, welche sich ausgiebig mit verbalen Protokollen befasst haben. Sie beschreiben unter anderem die *Think Aloud* Methode, bei welcher man die Gedanken, die im Review Prozess entstehen, protokolliert [BT06]. Um das Ganze abzurunden und um verbale Protokolle zu validieren, hat man zusätzlich Videoprotokolle oder die direkte Beobachtung eingeführt. Die Zuverlässigkeit dieser Verfahren wurde jedoch stark kritisiert, unter anderem weil die kognitiven Vorgänge durch bloße Beobachtung nicht objektiv genug beurteilt werden können [UNM+06]. Es fehlt somit an Abstration und Neutralität.

3.2 Erkenntnisse aus Eyetrackingstudien

Um den kognitiven Vorgang besser zu studieren, setzt man die Eyetracking Technologie ein. Die frühen Anfänge reichen bis zu den Jahren 1990 zurück, als Crosby und Stelovsky [CS90] eine Eyetrackingstudie durchführten, wo die visuelle Aufmerksamkeit der Teilnehmer bei der Inspektion eines in Pascal geschriebenen binären Suchalgorithmus aufgenommen wurde. Durch die Blickbewegungsaufnahme fand man heraus, dass erfahrene Programmierer sich mehr auf relevante Bereiche des Codes fokussierten, während sich die visuelle Aufmerksamkeit der Anfänger eher auf Kommentare und Vergleiche limitierte [BT06].

Uwano führte eine Eyetrackingstudie durch, um die kognitiven Prozesse genauer zu studieren. Die Aufgabe der Reviewer bestand darin Fehler semantischer Art im Quelltext zu entdecken. Bei der Studie wurden unterschiedliche Muster erkannt, wie z.B. das Scan Muster in Abbildung 3.2. Sie fanden heraus, dass einige Teilnehmer dazu neigten, den Code zuerst von oben bis unten kurz durchzuschauen, um sich dann auf bestimmte Teile des Codes zu konzentrieren. Dieser Vorgang wurde als kognitiver Prozess aufgefasst, denn der Teilnehmer versucht zuerst die gesamte Programmstruktur zu verstehen und währenddessen identifiziert er suspekte Stellen im Code, die einen Fehler beinhalten könnten. Ein weiteres Muster, das entdeckt wurde ist das *Retrace Declaration Pattern*, das in Abbildung 3.3 zu sehen ist. Wenn ein Teilnehmer auf eine Variable stößt, die zum ersten Mal benutzt wird, springt er zu der Stelle zurück, wo sie deklariert wird. Die Ergebnisse zeigten vor allem, dass die Teilnehmer, die nicht viel Zeit investierten den Code zu *scannen*, mehr Zeit benötigten um die Fehler aufzuspüren. Sharif [SFM12] wiederholte teilweise die Studie von Uwano, indem Eyetracking Messungen durchgeführt wurden und eine größere Anzahl an Teilnehmer rekrutiert wurde, mit unterschiedlicher Programmiererfahrung. Die Ergebnisse bestätigten die Hypothese von Uwano, und stellten auch fest, dass durch Eyetracking durchaus die individuelle Performance der Teilnehmer vorhergesagt werden kann.

Allgemein kann man die bis dato durchgeführten Studien folgendermaßen differenzieren. Zum einen analysieren sie den Review-Vorgang durch Verwendung von Tools, die speziell

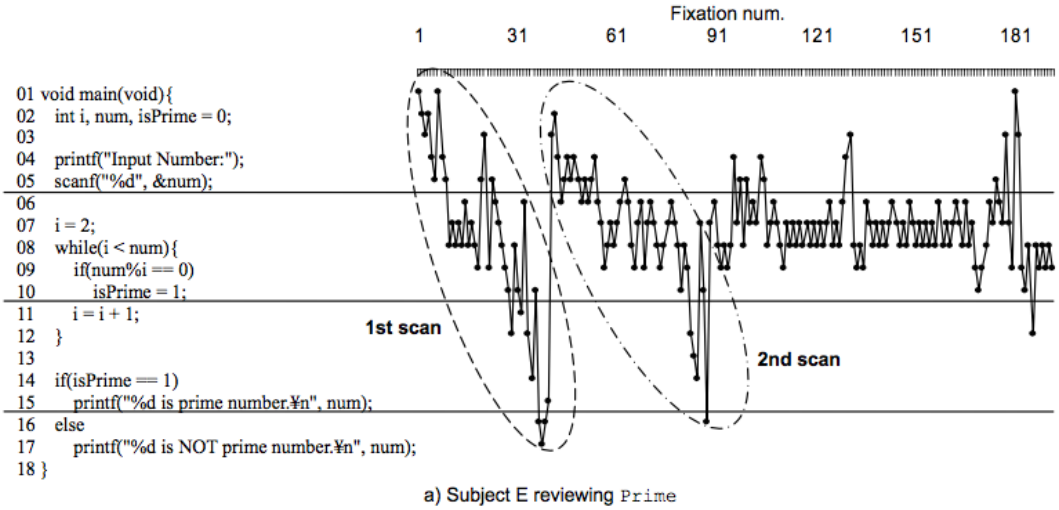


Abbildung 3.2: Diese Abbildung [UNM+06] zeigt, wie ein Teilnehmer ein Stimulus der Studie betrachtete. Dabei wurde ein besonderes Muster entdeckt, das Scan Muster. Der Teilnehmer macht sich ein generelles Bild vom Code bevor er ins Detail geht.

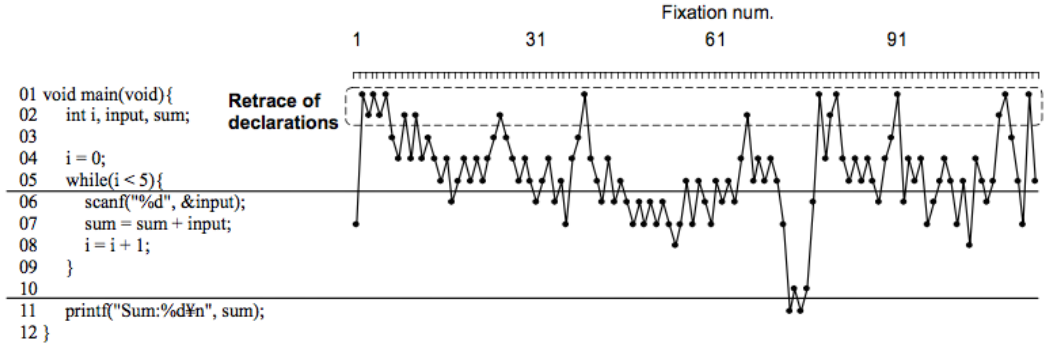


Abbildung 3.3: Graphische Darstellung [UNM+06] des *Retrace Declaration* Musters.

für das Programmverständnis entwickelt werden. Zum anderen wird das Verständnis unterstützt, indem man die Programme mit Farbelemente oder Symbole anreichert. Bednarik [BT06] führte beispielsweise eine Eyetrackingstudie durch, wo Programmierer mit Hilfe eines Visualisierungsprogramms, Jeliot 3 [MMS+04], Programme auf Fehler durchsuchen sollten. Dabei wurde insbesondere beobachtet, wie sie mit dem Tool interagieren und wie sich die visuelle Aufmerksamkeit zwischen der simplen Quelltext Repräsentation und der animierten Version des Codes verhält. Studien wie die von Hendrix [HCM+02] versuchten durch das Einfügen graphischer Symbole im Code, sogenannte Control Structure Diagrams (CSD), den Reviewer visuell zu unterstützen. Die Forscher erklären, wie wichtig graphische Elemente im Review-Vorgang sein können, und untermauern ihre Hypothese mit den Ergebnissen der Studie, die in Punkt Schnelligkeit und Korrektheit der Antworten sehr gut ausgefallen sind.

Da Farben präattentiv wahrgenommen werden [GB09], haben Studien wie die von Feigenspan [2012] den Einfluss erforscht, sie im Programmcode einzubinden. In *Software Product Line Engineering* beispielsweise werden aus einem Basiscode viele Varianten erzeugt, indem man je nach Funktionsalitätswunsch Featurecode einsetzt. Dabei kann die Identifikation von Features bei großen Programmen sehr schwer und mühsam werden, insbesondere beim Wartungsvorgang. Die Studie von Feigenspan [FKA+13] hat versucht diesem Problem entgegenzuwirken, durch die farbliche Markierung von Featurestellen im Code. Das Ergebnis war, dass nicht nur die Idee gut ankam, auch das Programmverständnis, selbst für umfangreiche Programme, konnte gesteigert werden. Es gibt aber auch Studien die Code Highlighting nicht befürworten. Die Studie von Hakala [HNS06] beispielsweise beschäftigt sich mit drei unterschiedlichen Code-Highlighting-Techniken. Die Aufgabe der Teilnehmer bestand darin nach Kontrollstrukturen, Methodenaufrufe, etc. in Java Programmen zu suchen. Dabei hat man Schlüsselwörter, wie in den Standard-Editoren, sowie umfangreiche Syntaxblöcke farblich gekennzeichnet und für Kontrollstrukruren kein Highlighting verwendet. Obwohl die Kommentare der Teilnehmer sehr positiv ausfielen, sprachen die Ergebnisse der Studie eher gegen einer Verbesserung der Performance durch Code-Highlighting.

Alle Studien versuchen das Verständnis auf eigene Weise zu messen und zu unterstützen. Durch Eyetracking ist es in den vergangenen Jahren gelungen Details zu entdecken, die durch das bloße Beobachten möglicherweise nie zum Vorschein gekommen wären. Deshalb ist es von besonderem Interesse weiterhin in diesem Gebiet zu forschen und Studien durchzuführen.

4 Visualisierungswerkzeug

In dieser Arbeit soll der Einfluss unterschiedlicher Codevisualisierungen beim Programmverständnis mit Hilfe von Eyetracking untersucht werden. In den bisherigen Studien war es bereits möglich, den Lesefluss der Teilnehmer in Form einer Timeline darzustellen, weshalb diese Arbeit teilweise eine Wiederholung der Studie von Uwano [UNM+06] ist. Das vorgestellte Visualisierungswerkzeug ist zusätzlich in der Lage, Areas of Interest (AOI) farblich zu unterscheiden. Damit soll eine detailreichere Analyse möglich sein, indem man das Augenmerk beispielsweise auf bestimmte Schlüsselwörter setzt. Die Realisierung, die genaue Funktionsweise, sowie die verwendeten Technologien werden im folgenden Kapitel vorgestellt.

4.1 Technologien

Für die Realisierung der Timeline wurden die Webtechnologien JavaScript, HTML, CSS und die Bibliothek D3.js verwendet. Diese Technologien und weitere wichtige Begriffe werden nun näher erklärt.

4.1.1 HTML

HTML steht für Hypertext Markup Language. Es ist "eine textbasierte Auszeichnungssprache zur Strukturierung von digitalen Inhalten wie Texten, Bildern und Hyperlinks"[Gul14]. Für die Darstellung solcher Dokumente werden Browser eingesetzt, mit deren Hilfe man die Inhalte einsehen kann. HTML wird vom *World Wide Web Consotrium (W3C)* standardisiert und weiterentwickelt, wobei seine aktuellste Version HTML5 ist. Mit diesem Standard werden im Vergleich zur Vorgängerversion diverse Sachen ermöglicht, ohne zusätzliche Plug-ins installieren zu müssen. Es lassen sich beispielsweise Audio- und Video Dateien direkt einbinden [www.w3.org]. Die Struktur des Dokuments wird mit Hilfe von "Tags"modelliert. Der obere Kasten in Abbildung 4.2 ist ein Beispiel dafür. Eine Einschränkung, sowohl bei der Entwicklung, als auch bei der Darstellung der Webseite ist, dass nicht jeder Browser immer die aktuellste Version unterstützt. Die Konsequenz ist, dass dieselbe Webseite oder Web-Anwendung, nicht auf allen Browsern gleich dargestellt wird. Eine Möglichkeit dies zu umgehen ist, gewisse Voreinstellungen von JavaScript zu nützen, durch die man in der Lage ist, den verwendeten Browsertypen zu identifizieren. Somit kann man unterschiedliche Operationen für jeden Browsertypen implementieren [www.w3.org].

DOM

Für die Entwicklung mit HTML und insbesondere mit der Bibliothek D3.js ist es wichtig den Aufbau und die Funktionsweise des *Document Object Model (DOM)* zu kennen. Das DOM kann als abstrakte Version einer HTML Seite interpretiert werden, das ebenfalls vom *World Wide Web Consortium (W3C)* standardisiert und weiterentwickelt wird [www.w3schools.com].

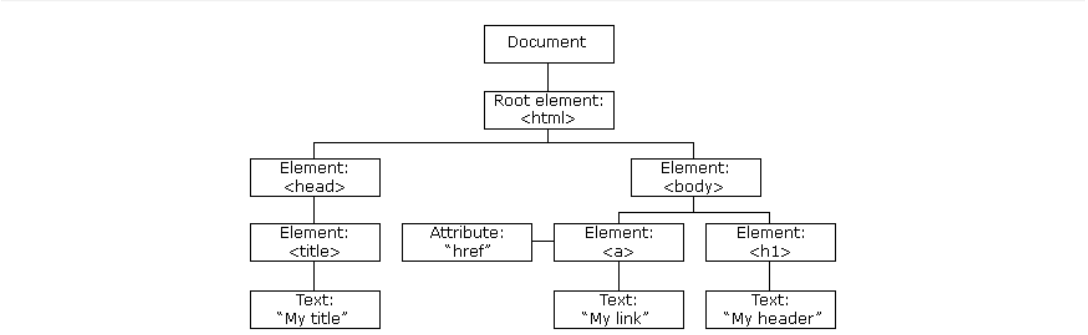


Abbildung 4.1: Hierarchische Struktur einer Webseite. Bildquelle: www.w3schools.com

Abbildung 4.1 zeigt die hierarchische Struktur einer Webseite. Wenn eine Seite vom Browser geladen wird, wird automatisch ein DOM Element davon generiert [www.w3schools.com]. Das DOM hat eine baumartige Struktur und stellt die Schnittstelle für die Kommunikation zwischen HTML und JavaScript dar. Alle ihm angehängten Elemente werden nämlich zu Objekten, die mittels JavaScript dynamisch manipuliert werden können [wiki.selfhtml.org].

4.1.2 JavaScript

JavaScript ist eine vielseitige und flexible Sprache mit objektorientierten Fähigkeiten und kann direkt im HTML Dokument eingebettet, oder in einem externen Dokument verwaltet werden, das dann im HTML Dokument eingebunden wird. Im Unterschied zu vielen anderen Skriptsprachen wie PHP, ASP oder JSP, die serverseitig ausgeführt werden, wird JavaScript hauptsächlich vom Browser lokal interpretiert. Seine Hauptaufgabe ist, statische HTML Dokumente mit Funktionalität zu versehen, wodurch es möglich ist dynamische und interaktive Webseiten zu erstellen. Der Vorteil dieser Sprache ist, dass sie plattformunabhängig ist und somit von nahezu jedem Webbrowser interpretiert werden kann [www.itwissen.info]. JavaScript verwendet außerdem keine Variablentypen und durch die Verwendung unterschiedlicher Frameworks wie jQuery [https://jquery.com/] wird die Webentwicklung erheblich erleichtert. Durch die Fähigkeit Aktualisierungen des DOMs leicht umzusetzen, wurde die Verwaltung der Objekte die im DOM enthalten sind quasi komplett JavaScript übergeben. So können HTML Elemente, Attribute, CSS-Styles entfernt, hinzufügt oder modifiziert werden [www.w3schools.com]. Dank seiner Flexibilität hat sich JavaScript in den letzten Jahren immer mehr zu einem essentiellen Bestandteil von Web-Applikationen entwickelt.

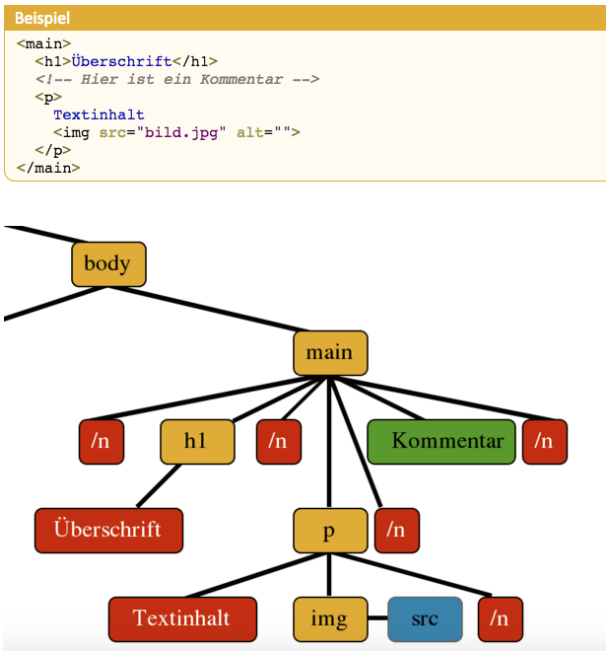


Abbildung 4.2: Beispiel eines HTML Dokuments und das zugehörige DOM. Bildquelle: www.w3schools.com

4.1.3 CSS

CSS ist die Abkürzung für Cascading Style Sheets. Mit diesem Instrument kann man die graphische Darstellung (Layout, Farbe, Abstände, Schriftart, usw.) der Elemente einer Webseite modifizieren. Dadurch hat man, verglichen zum Standard HTML, designtechnisch viel mehr Möglichkeiten die Elemente zu modifizieren und zu verwalten. Der Vorteil liegt darin, eine einzige Datei zu verwenden, die dann mit allen HTML Dokumenten, die darauf referenzieren, kommuniziert. Somit kann man schon verwendete graphische Einstellungen auch für andere Elemente verwenden. Sie sind eine Empfehlung des W3C Konsortiums, das schon mehrere Versionen auf dem Markt gebracht hat. Die aktuellste ist CSS3, die im HTML5 Standard integriert ist. Allgemein kann man sagen, dass HTML für die Struktur der Webseite zuständig ist, während man mit CSS das Design bestimmt [wiki.selfhtml.org].

4.1.4 JSON

Für die Speicherung der Daten wurde unter anderem eine dateibasierte Datenspeicherung verwendet, das JSON Objekt. JSON steht für *JavaScript Object Notation*. Der Vorteil dieser Notation ist, dass es sowohl client- als auch serverseitig eingesetzt werden kann. Der Aufbau eines JSON-Objekts kann man in Abbildung 4.2 sehen. Die Objekte sind jeweils von geschweiften Klammern umgeben. Die Daten werden als *Key-Value-Pairs* gespeichert [wiki.selfhtml.org].

JSON-Objekte stellen ein leichtgewichtiges Datenaustauschformat dar, wodurch die Arbeit mit der Bibliothek D3.js erheblich erleichtert wird.

Beispiel

```
{
  "Name": "Georg",
  "Alter": 47,
  "Verheiratet": false,
  "Beruf": null,
  "Kinder": [
    {
      "Name": "Lukas",
      "Alter": 19,
      "Schulabschluss": "Realschule"
    },
    {
      "Name": "Lisa",
      "Alter": 14,
      "Schulabschluss": null
    }
  ]
}
```

Abbildung 4.3: Das Bild zeigt den Aufbau eines JSON-Objekts. Dabei wird ein Objet von geschweiften Klammern umschlossen. Jeder Wert besitzt einen eindeutigen Schlüssel, über welchen man auf den Wert zugreifen kann. Ein Schlüssel-Werte-Paar wird durch einen Doppelpunkt getrennt. [wiki.selfhtml.org]

4.1.5 SVG

SVG steht für *Scalable Vector Graphic* und ist ein Dateiformat für Vektorgrafiken. Der Vorteil solcher Grafiken ist, dass sich Bilder, verglichen zu Pixelgrafiken, ohne Verluste in jede Größe skalieren lassen [Gul14]. Das ergibt sich dadurch, dass SVG-Grafiken Bilder anhand von Vektoren beschreiben, anstatt der Rastertechnik, die in Pixelgrafiken verwendet wird. SVG Grafiken können mit verschiedenen Javascript Bibliotheken realisiert werden. In dieser Arbeit werden sie mit Hilfe der Bibliothek D3.js erstellt.

4.1.6 D3.js

Das Akronym D3 steht für *Data Driven Documents*. Damit wird auf die Funktionsweise hingewiesen, mit deren Hilfe sich Dokumente (HTML-Seiten) auf Datenbasis dynamisch erzeugen lassen. Um dies zu ermöglichen, werden in D3 mehrere Technologien wie HTML, SVG und CSS integriert. Dabei werden die geladenen Datensätze an das oben erklärte DOM gebunden und anschließend manipuliert und transformiert. Mit Hilfe eines sogenanntes *Selection*-Ansatzes kann die Auswahl der Elemente im DOM erheblich vereinfacht werden [Gul14]. Abbildung

Abbildung 4.4: Unterschied zwischen einer Pixelgrafik und einer Vektorgrafik. Bildquelle: www.wikibooks.org

4.5 und 4.6 zeigen den Unterscheid zwischen einer Selektion von Paragraphen mit der DOM Schnittstelle, verglichen zu dem Ansatz der von D3 verwendet wird. Dadurch wird die Komplexität um ein vielfaches reduziert und somit die Performance gesteigert. Des Weiteren arbeitet diese Bibliothek hervorragend mit JSON Objekten zusammen, da sie, nachdem sie Zugang zum Objekt hat, die Werte direkt über die Keys selektieren kann. Die Bibliothek ist außerdem sehr beliebt, aufgrund der vielen Dokumentierungen und Tutorials die im Internet zu finden sind.

```
1 //Änderung der Farbe aller Paragraphen mittels DOM-API
2 var absatz = document.getElementByTagName("p"); //selektiere alle Paragraphen p im DOM
3 for (var i=0; i<absatz.length; i++) { //für jedes Item i
4     var absatz = absatz.item(i);
5     absatz.style["color"] = "green"; //ändere Farbe
6 }
```

Abbildung 4.5: Paragraphen-Auswahl mittels DOM-API.

```
1 //Änderung der Farbe aller Paragraphen mittels D3
2 d3.selectAll("p").style("color","green");
```

Abbildung 4.6: Paragraphen-Auswahl mittels *Selection* Ansatz.

4.2 Realisierung

Die Aufgabe bestand darin, ein Visualiserungswerkzeug zu erstellen, das die aufgenommenen Eyetracking Daten grafisch darstellt. Dies soll für die retrospektive Analyse dienen. Dafür wurden die oben genannten Webtechnologien verwendet, um eine HTML basierte Webanwendung zu erstellen. Die zu verarbeitenden Daten sind zum einen die vom Eyetracker generierten Daten. Sie können im Tobii Studio im .tsv Format exportiert werden. Um sie in der Timeline anzeigen zu lassen, müssen sie allerdings im .csv Format umgewandelt werden. Zum anderen werden weitere externe Daten geladen, die für die graphische Repräsentation wichtig sind, wie beispielsweise die Koordinaten der annotierten AOIs. Diese werden in einer .csv Datei gespeichert. Die Bibliothek D3.js bietet eine Funktion *queue()* an, mit welcher man externe Daten dynamisch laden kann. Diese Funktion wandelt .csv-Dateien direkt in JSON Objekte um. Dabei muss geachtet werden, dass die erste Zeile der Datei als Schlüssel (Key) von der Funktion interpretiert wird. Im Programm selbst, werden intern die nötigen Daten aus der .csv Datei herausgefiltert (siehe Abschnitt 2.3.2) und in ein JSON Objekt gespeichert. In vielen Literaturbüchern wird angemerkt, dass das Laden externer Dateien, besonders in Verbindung mit der queue()-Methode, zu Problemen führen kann. Dies ist auch hier der Fall, weshalb man zur Umgehung des Problems einen simplen lokalen Server startet. Das Programm an sich bestehst aus drei Dateien: *timeline.html*, *style.css* und *timeline.js*. Diese befinden sich alle im selben Ordner. Um das Programm zu starten, ruft man im Browser, nachdem man den simplen Server gestartet hat, die Datei *timeline.html* auf. Der große Vorteil Browser-basierter Anwendungen ist, dass man sie nicht lokal auf der Festplatte installieren muss. Somit kann man die Datei, sofern sie lokal gespeichert ist und man in diesem Fall eine Internetverbindung hat, direkt im Browser abrufen.

4.2.1 Datei auswählen

Die Daten der Teilnehmer werden vom Tobii Studio exportiert und in zwei Ordner unterteilt. Ein Ordner beinhaltet die Review Vorgänge der einzelnen Teilnehmer, der andere enthält die Review Vorgänge geordnet nach Programmen und Programmdarstellung. Diese befinden sich lokal auf dem Rechner. Die ausgewählten Dateien werden untereinander in Form einer Timeline angezeigt, die aus vielen Rechtecken besteht. Die Dauer einer Fixation erkennt man dabei an der Breite des Rechtecks. Je breiter, desto Länger dauerte die Fixation in diesem Punkt. Somit kann man die unterschiedlichen Leistungen der Teilnehmer singulär oder untereinander vergleichen. Abbildung 4.7 zeigt den eben beschriebenen Vorgang. Für die x-Achse wird aus der .csv Datei der Wert *TimeStamp* verwendet. Dies ist ein Zeitstempel, der die Zeit in Millisekunden aufzeichnet, ab dem Beginn der Blickbewegungsaufzeichnung. Dabei wird eine Fixation aus mehreren aufeinanderfolgenden *GazePoints* zusammengesetzt (siehe Abbildung 4.8), wenn der Abstand zwischen ihnen innerhalb bestimmter Bereiche bleibt [TobiiStudio2.XUserManual]. Die Fixationsdauer wird aus der Differenz zwischen aufeinanderfolgenden Fixationspunkten berechnet.

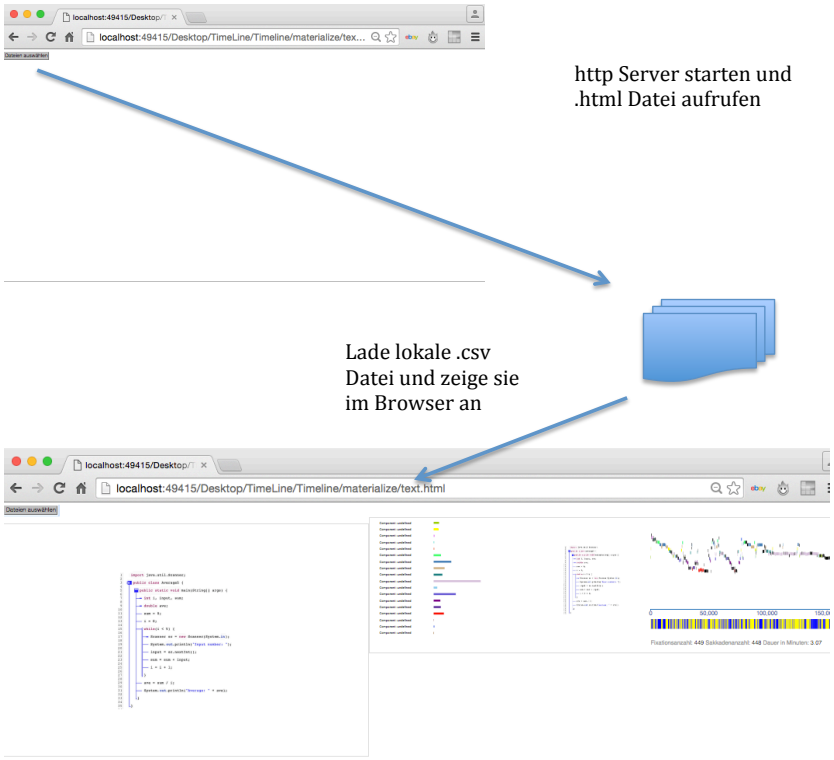


Abbildung 4.7: Visualisierungsprogramm und Darstellung der Timeline im Browser

Zusätzlich zur Visualisierung werden Fixationsanzahl und Sakkadenanzahl angezeigt, sowie die Gesamtdauer der Betrachtung des Stimulus. Die im Vorfeld im Tobii Studio definierten AOIs werden farblich unterschieden. Somit kann man in der Timeline nachvollziehen, welche AOI angeschaut wurde. Links neben der Timeline wird jeweils immer der Stimulus angezeigt, sowie die Programmkomponenten. Die länge der Balken gibt jeweils an, wie lange eine AOI verglichen zu den anderen angeschaut wurde. Links neben den Hits wird das aktuell betrachtete Stimulus im Original Format 1900 x 1200 Pixel angezeigt siehe (Abbildung 4.9).

Die Balken der Koponenten haben dieselbe Farbe der ihnen zugehörigen AOIs in der Timeline. Schwarze Rechtecke in der Timeline signalisieren, dass keine AOI angeschaut wurde. Wenn man mit der Mouse über die Elemente gleitet, erscheint ein Tooltip mit den Namen der AOI. Somit hat man eine kompakte Darstellung der zusammengehörigen AOIs. Unterhalb der Timeline kann man zusätzlich den Wechsel der Leserichtung der betrachteten Zeilen in Form eines Balkens ansehen(siehe Abbildung 4.10). Dabei zeigt gelb die Richtung an, wenn man von rechts nach links schaut und blau im anderen Fall.

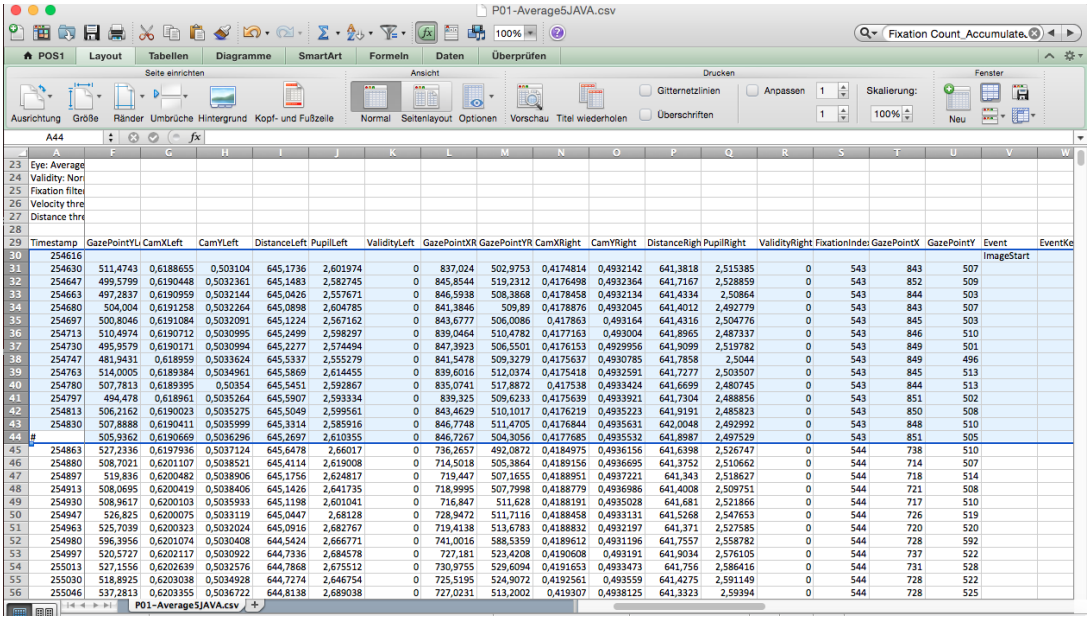


Abbildung 4.8: GazePoints werden zu einer Fixation zusammengefasst.

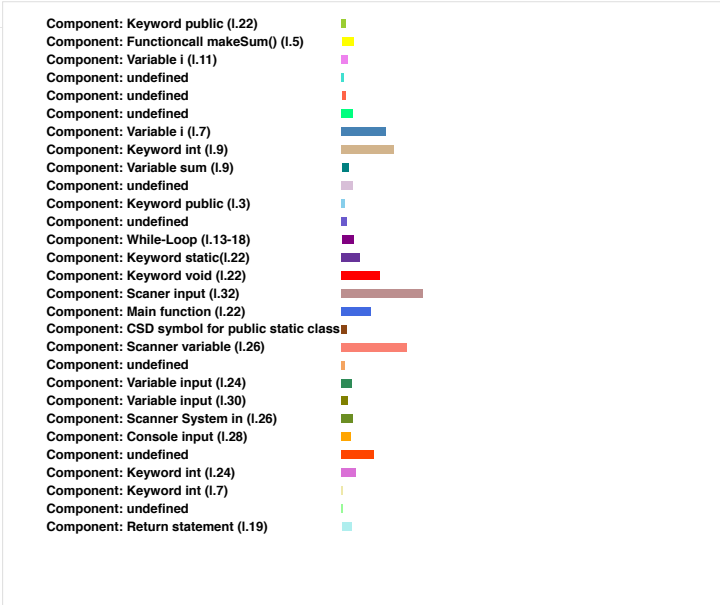


Abbildung 4.9: Hier werden die einzelnen Programmkomponenten angezeigt, und die jeweilige Zeile in welche sie sich befinden. Die Balken signalisieren wie lange sie im vergleich zu den anderen Komponenten angeschaut wurden.

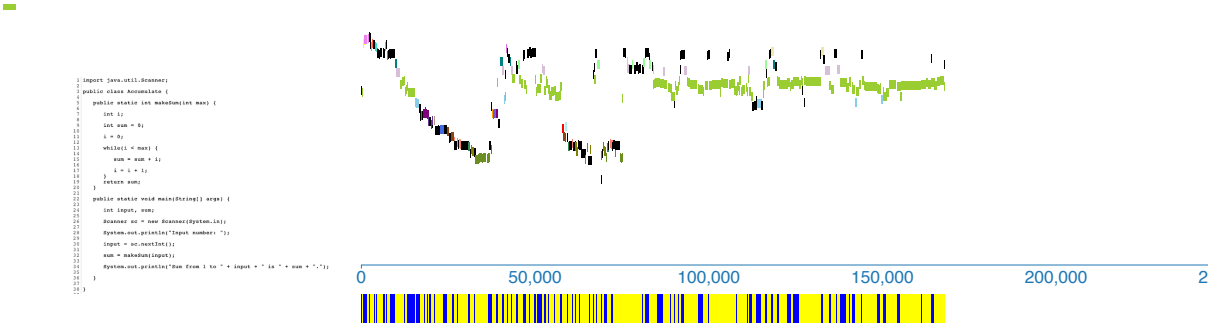


Abbildung 4.10: Hier sieht man den Wechsel der Leserichtung. Gelb signalisiert, dass der Leser von rechts nach links liest und blau im anderen Fall.

5 Eyetracking-Studie

In diesem Kapitel wird die Durchführung der Eyetracking-Studie vorgestellt. Die Aufgabe der Teilnehmer bestand darin, 6 Java Programme in jeweils drei unterschiedlichen Darstellungen, Plaintext, Java-Syntax-Highlighting und letzteres noch inklusive mit *Control Structure Diagrams*, jeweils auf einen Semantikfehler zu untersuchen und die Funktionsweise zu bestimmen. Die Studie wurde zum Teil bereits von Uwano [UNM+06] mit 5 Teilnehmer und von Sharif [SFM12] mit 15 Teilnehmer nochmals durchgeführt (siehe Kapitel 3.2).

5.1 Ablauf der Studie

Vor der eigentlichen Studie wurde eine Pilotstudie mit zwei Probanden durchgeführt, um mögliche Fehlerquellen beim Studienverlauf frühzeitig zu erkennen. Der Versuchsaufbau im Tobii Studio bestand insgesamt aus 12 Folien, die nacheinander eine Introduktionsfolie und das Stimulus aufzeigte. Der Aufbau wurde optimiert, indem eine zusätzliche Folie eingefügt wurde, zwischen der Introduktionsfolie und dem eigentlichen Stimulus, auf der das Programm nochmals angezeigt wurde mit Zeilenangabe. Sie diente dazu, den Teilnehmern die Zeit für Kommentare zu gewähren. Für die eigentlichen Stimuli hatten sie maximal 5 Minuten Zeit. Die Zeit wurde dabei nebenher geprüft und nach Ablauf, oder nachdem sie den Fehler gefunden hatten, wurden die Teilnehmer aufgefordert auf die Spacetaste zu drücken. Es wurden auch einige eher Formale Fehler, die den Fragebogen betrafen angemerkt. Diese wurden vor der durchführung der Studie behoben.

Der formale Ablauf sah für jeden Probanden folgendermaßen aus:

1. Einverständniserklärung:

   Zu Beginn der Studie bekommt jeder Teilnehmer eine Einverständniserklärung. Darin steht, dass er jeder Zeit dazu berechtigt ist, die Studie abzubrechen und die Daten anonym aufgenommen werden.

2. Fragebogen zur Person:

   Nach der Einverständniserklärung füllt der Teilnehmer den ersten Teil des Fragebogens. Er enthält Fragen zu Alter, Geschlecht, Sehhilfe, Hochschulabschluss und angestrebter Abschluss (siehe Anhang A.1).

3. Farb- und Sehtest:

   Um zu testen, ob der Teilnehmer eine Sehschwäche hat, wurde ein Farb- und Sehtest durchgeführt. Somit wurde sichergestellt, dass die Daten nicht beeinträchtigt werden (siehe Kapitel 5.6).

4. Tutorial:

   Vor der Studie war es wichtig die Teilnehmer auf denselben Kenntnisstand zu bringen, weshalb ein Tutorial zu den CSDs erstellt wurde. Dafür wurden Beispielcodes (siehe Beispiel in Abbildung 5.2) verwendet um die eingesetzten Symbole zu erklären. Anschließend wurde der Allgemeine Ablauf der Studie erklärt.

5. Kalibrierung:

   Durch die Kalibrierung wird gewährleistet, dass die Augenbewegungen von der Kamera richtig aufgefasst werden. Dabei müssen 9 Punkte auf dem Bildschirm verfolgt und fixiert werden. Mit einer Kinnstütze wird die Position des Kopfes stabilisiert und ein Abstand zwischen 63 und 66 cm zum Bildschirm sichergestellt.

6. Betrachten der Programme:

   Jeder Teilnehmer besitzt eine Identifikationsnummer, wobei jede Nummer durch die *Graeco Latin Square*-Aufteilung eine unterschiedliche Sequenz der Bilder aufzeigt(siehe Abbildung 5.3). Nach einer Folie wird ein Stimulus aufgezeigt, gefolgt von einer Zwischenfolie. Dort hat jeder Teilnehmer Zeit, die Funktionsweise und den Fehler des betrachteten Programms wiederzugeben. Die maximale Zeit pro Stimulus betrug 6 Minuten.

7. Fragebogen zu den Programmen:

   Nach Ablauf der Studie füllt der Teilnehmer den zweiten Teil des Fragebogens(siehe Anhang A.2) aus, der sich auf die betrachteten Stimuli bezieht. Es wurden Fragen zu Vorkenntnisse, Verständnis, Anstrengung gestellt. Um zu wissen, welche Darstellung er bevorzugt, wird er gebeten, die Codedarstellungen zu bewerten. Die letzten Seite des Fragebogens ist für Kommentare gedacht.

8. Aufwandsentschädigung:

   Nach Beendigung der Studie wird dem Teilnehmer als Aufwandsentschädigung ein überraschungsei gegeben.

5.1.1 Technischer Aufbau

Die Studie fand im Labor des VISUS-Gebäudes der Universität Stuttgart statt. Damit mögliche Störfaktoren keinen Einfluss auf die Studie haben konnten, wurden sowohl die Tür als auch die Fenster geschlossen gehalten. Die Rollläden wurden auch heruntergelassen. Außerdem wurden die Teilnehmer darum gebeten ihre Handys auszuschalten. Für die Blickbewegungsaufnahme

wurde der Eyetracker T60 XL der Firma Tobii verwendet. Für die Datenaufnahme wurde die Software Tobii Studio 2.2.8 verwendet. Die Programmsequenz wurde auf einem 24“Monitor mit einer Bildschirmauflösung von 1920 x 1200 Pixel dargestellt.

5.2 Hypothesen

Folgende Hypothesen wurden in den vergangenen Studien formuliert (Uwano [UNM+06] und Sharif [SFM12]):

Hypothese 1

Die durchschnittliche Dauer, bis der Teilnehmer den Fehler findet, ist gleich unabhängig davon, ob der Code anfangs gescannt wurde oder nicht .

Hypothese 2

Die Genauigkeit, mit der der Fehler gefunden wird, ist gleich unabhängig von der Scan-Zeit.

Hypothese 3

Die visuelle Anstrengung ist gleich, unabhängig, von der Scan-Zeit.

Hypothese 4

Die Erfahrung hat keinen Einfluss auf die Scan-Zeit, und somit keine Auswirkungen auf die Zeit bis zur Fehlerdetektion und Genauigkeit.

Die Hypothesen, die während dieser Studie entstanden sind, werden im Diskussionskapitel vorgestellt.

5.3 Variablen

Man unterscheidet zwei Typen von Variablen, *unabhängigen* und *abhängigen*, die die Hypothesen betreffen. Bei den *unabhängigen* Variablen handelt es sich um Variablen, die während eines Experiments geändern werden können. In dieser Studie sind das die verwendeten Java Programme in den unterschiedlichen Darstellungen(siehe Kapitel 5.4).

Die *abhängigen* Variablen hängen von den unabhängigen ab. Sie sollten sich während einer Studie nicht ändern, denn in ihnen Zeigt sich die Auswirkung der *unabhängigen* und werden in einer Studie gemessen. Die *abhängigen* Variablen dieser Studie sind z.B. die Zeit bis zur Fehlererkennung oder die Fixationsanzahl pro AOI.

5.4 Stimuli

Die verwendeten Stimuli für die Studie sind 6 Programme, die ursprünglich von Uwano in der Programmiersprache C geschrieben wurden. Für diese Studie wurden sie in der Programmiersprache Java umgeschrieben, da die Teilnehmer Studenten waren, die als erste Programmiersprache Java gelernt haben und beinhalten jeweils denselben logischen Fehler. Die Größe der Bilder wurde auf 1900 x 1200 Pixel gesetzt, wobei jede Zeile eine Höhe von 21 Pixel besitzt. In der folgenden Tabelle sind alle wichtigen Details zu den Programmen enthalten.

Zusätzlich wurden drei unterschiedliche Codedarstellungen verwendet, Plaintext, Java Systax Highlighting und letzteres mit Control Strukture Diagrams (CSD) ergänzt. Dabei soll der unterschiedliche Einfluss, den sie während der Coderanalyse haben können, untersucht werden.

Control Strukture Diagrams (CSD) wurden entwickelt, um den Programmierer beim Programmverständnis zu unterstützen. Dabei sollen die Diagramme im Programm integriert werden, indem sie wie eine natürliche Erweiterung des Programms selbst wirken, ohne das Erscheinungsbild drastisch zu ändern [HCM+02]. GRASP Editoren [www.eng.auburn.edu] automatisieren die Generierung von CSDs im Programmcode und wurden auch hier verwendet. Das Ergebnis kann man in Abbildung 5.2 sehen. Um eine ausgeglichene Verteilung der Stimuli zu erreichen und Lerneffekte durch Wiederholung zu minimieren, wurden *Graeco Latin Squares* [Hol03] verwendet. Die Verteilung der Programme wird in Abbildung 5.3 gezeigt. Dabei steht P für Plaintext, J für Java Syntax Highlighting und C für Java Syntax Highlighting inklusive CSD. Das zugehörige Programm zu der Nummer, kann man in der Tabelle 5.1 ablesen.

5.5 Aufgabe

Die Aufgabe der Teilnehmer bestand darin eine Sequenz von 6 Java Programmen die jeweils in drei unterschiedlichen Codedarstellungen präsentiert wurden, auf einen semantischen Fehler

Nummer	Programm Name	Funktionsweise	Semantik Fehler	Zeilen
1	Accumulate	Der User gibt eine nicht negative Integer Zahl n ein. Das Programm bildet die Summe aus allen Zahlen von 1..n.	Die Schleifenbedingung ist falsch. Die Besingung sollte lauten (n<=0) statt (n<0).	38
2	Average-5	Der User gibt 5 Integer Zahlen ein und das Programm berechnet daraus den Durchschnitt.	Eine Typkonvertierung von Integer zu double fehlt. somit entsteht ein Rundungsfehler im Durchschnitt.	35
3	Average-Any	Der User gibt eine beliebige Anzahl an Integer Zahlen ein (max.255) bis 0 eingegeben wird. Das Programm berechnet den Durchschnitt der eingegebenen Zahlen.	Die Anzahl der Schleifendurchläufe ist falsch. Das Programm berechnet immer den Durchschnitt von 255 Zahlen unabhängig von den eingegebenen Zahlen.	46
4	Prime	Der User gibt n Integer Zahlen ein. Das Programm überprüft, ob n eine Primzahl ist oder nicht.	Die Logik in der Bedingungsabfrage ist verkehrt, somit wird das Ergebnis auch umgekehrt.	40
5	Swap	Der User gibt zwei Zahlen ein. Das Programm vertauscht die, durch die swap()-Funktion und gibt dann das Ergebnis aus.	Die Pointer werden falsch verwendet und somit werden die Zahlen nicht ausgetauscht.	46
6	Sum-5	Der User gibt 5 Integer Zahlen ein. Das Programm berechent die Summe dieser Zahlen.	Die Variable sum wurde nicht initialisiert.	26

Tabelle 5.1: Informationen zu den verwendeten Stimuli

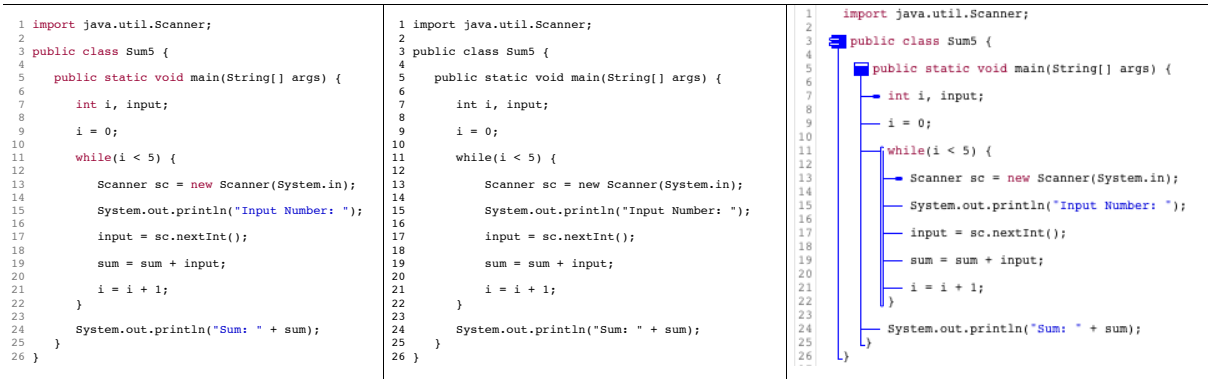


Abbildung 5.1: Die Abbildung zeigt denselben Code in den 3 unterschiedlichen Darstellungen. (von links nach rechts: Java Syntax Highlighting, PlainText, Java Syntax Highlighting inkl. CSD)

Teilnehmer1	P,1	J,2	C,6	P,3	J,5	C,4
Teilnehmer2	J,2	C,3	P,1	J,4	C,6	P,5
Teilnehmer3	C,3	P,4	J,2	C,5	P,1	J,6
Teilnehmer4	J,4	C,5	J,3	J,6	C,2	C,1
Teilnehmer5	J,5	C,6	P,4	J,1	C,3	P,2
Teilnehmer6	J,6	J,1	P,5	C,2	C,4	J,3

Teilnehmer7	C,4	J,5	P,3	C,6	J,2	P,1
Teilnehmer8	P,5	P,6	J,4	C,1	J,3	P,2
Teilnehmer9	P,6	J,1	C,5	C,2	C,4	P,3
Teilnehmer10	C,1	J,2	P,6	C,3	J,5	P,4
Teilnehmer11	P,2	C,3	J,1	P,4	C,6	J,5
Teilnehmer12	J,3	P,4	C,2	J,5	P,1	C,6

Abbildung 5.2: Programmverteilung mittels *Graeco Latin Square*

zu untersuchen. Zusätzlich sollte auch die Funktionsweise des Algorithmus bestimmt werden, um, infolgedessen, den logischen Fehler zu finden. Pro Programm hatten sie maximal 6 Minuten Zeit. Spätestens dann wurden sie aufgefordert, auf einer nächsten Folie im Studienverlauf, mittels *Think Aloud* Methode die Funktionsweise und den Fehler laut zu sagen. Jedes Programm beinhaltet einen semantischen Fehler. In Tabelle 5.1 sind alle Programme aufgelistet, mit der entsprechenden Funktionsweise und den Programmfehler.

5.6 Teilnehmer

Insgesamt haben 12 Personen an der Studie teilgenommen. Davon waren 3 weiblich und 9 männlich. Das Durschnittsalter lag bei 26,28 Jahren mit einer Alterspanne von 22-36 Jahren. 8 Personen waren Brillenträger und 3 davon haben Kontaktlinsen getragen. Vor der Eyetracking Studie, musste jeder Teilnehmer ein Sehtest und ein Farbtest (Ishihara-Test) durchführen. Der Test ergab, dass alle Teilnehmer eine normale Sehkraft hatten. Durch die Ergebnisse des Farbtests wurde festgestellt, dass keine Farbfehlsichtigkeiten der Teilnehmer vorhanden waren.

Unter den Teilnehmern waren 8 Bachelorstudenten, 3 Masterstudenten und ein Diplomstudent. Die Studienänge umfassten: Informatik, Softwaretechnik, Physik, Elektrotechnik, Infomationstechnik, Technische Kybernetik und Luft- und Raumfahrttechnik. Alle hatten Programmiererfahrung und im speziellen mit der Programmiersprache Java.

6 Auswertung

Es folgt nun die Auswertung der Daten, mittels der von Tobii Studio ermittelten deskriptiven Statistik und die visuelle Analyse der Daten, mit Hilfe des entwickelten Visualisierungsprogramms. Im Folgenden wird der Einfachheit halber die Java Syntax Highlighting Darstellung inklusive CSD mit CSD abgekürzt und die Java Syntax Highlighting Darstellung mit Java.

6.1 Vorverarbeitung der Daten

4 Probanden mussten wieder neu aufgenommen werden, um die Ausgeglichenheit der betrachteten Stimuli zu gewährleisten.

Annotation der AOIs

Bevor man die Daten in Tobii Studio auswertet, müssen die AOIs annotiert werden. Dieser Vorgang ist auch für das Entwickelte Tool wichtig. Bei der Definition der AOIs wurde dabei die Lage des semantischen Fehlers berücksichtigt. Beispielsweise wurde im Programm *Accumulate PlainText* die While-Schleife als AOI definiert, wie in der folgenden Auswertung erkennbar ist, weil sich dort der Fehler befand.

6.2 Ergebnisse der unabhängigen Variablen

Es folgt nun die statistische und visuelle Auswertung der Daten. Es wird keine inferentielle Statistik durchgeführt. Die Hypothesen werden aus den Ergebnissen der durchgeführten Studie und des entwickelten Tools formuliert und sollten in einer weiteren Studie vertieft und statistisch ausgewertet werden. Dennoch wird ein Blick auf die deskriptive Statistik geworfen, um eine generelle Sicht auf die Daten zu erhalten.

In den vergangenen Studien wurden die Hypothesen aus Kapitel 5 analysiert. Dabei hat Sharif [SFM12], basierend auf die Ergebnisse der Studie von Uwano [UNM+06], beispielsweise untersucht, wie die Zeit bis zur Fehlerdetektion und die Scan-Zeit in Beziehung zueinander stehen (siehe Hypothese 1). Dabei wurde festgestellt, dass die Scan-Zeit einen relevanten

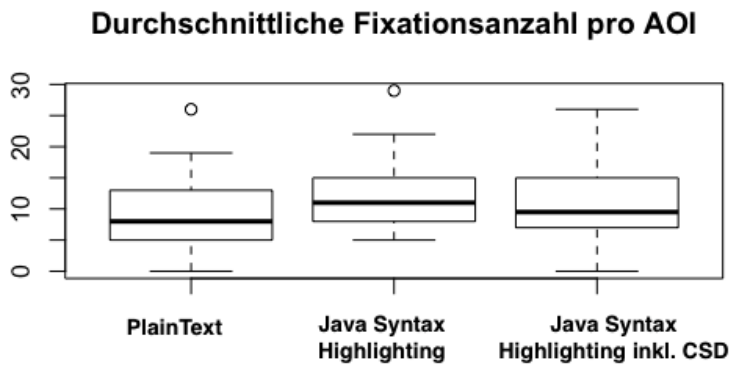


Abbildung 6.1: Durchschnittliche Fixationsanzahl pro AOI in den drei unterschiedlichen Codedarstellungen.

Einfluss auf die Zeit bis zur Fehlerdetektion hat. Je sorgfältiger das Programm gescannt wird, desto schneller kommt man zum Ergebnis. Somit wurde die erste Hypothese verworfen. Da die Scan-Zeit in Verbindung steht mit der Anzahl der Fixationen, wurde in der Studie somit die dritte Hypothese auch verworfen. Die zweite Hypothese konnte nicht verworfen werden, da man es nicht für die Gesamtheit der Daten verallgemeinern konnte. Hypothese 4 wurde auch verworfen, da man herausfand, dass Anfänger im Schnitt ca. 25,8 Sekunden länger brauchten beim Review-Vorgang, als Fortgeschrittene. Uwano fand unterschiedliche Muster in den Review-Vorgängen der Teilnehmer, die neue Einblicke in die Leseweise des Codes verschaffen konnten. Beispielsweise das *Scan Muster*, das auch in dieser Arbeit in vielen Reviews erkannt wird, oder das *Retrace Declaration Pattern* (siehe Kapitel 3 Abb. 3.3). Das Scan Muster bezieht sich dabei auf die Zeit, bis 80% des Codes gelesen wurde. Beide Studien differenzieren sich nicht nur in der Anzahl der Teilnehmer (Sharif: 15 Teilnehmer, Uwano: 5), Sharif fürhrte mehr statistische Tests durch, die die Hypothesen von Uwano teilweise unterstützen (Beispiel: Relevanz der Scan-Zeit bei der Fehlererkennung).

Die subjektive Empfindung der Teilnehmer in dieser Studie, lässt vermuten, dass sie sich durch das fehlende Highlighting mehr anstrengen mussten beim Code Review. Die deskriptive Statistik in Abbildung 6.2 zeigt, dass die durchschnittliche Dauer bis zur Fehlererkennung pro Programmdarstellung, bei der CSD Variante am größten war. Für die PlainText Darstellung, haben die Teilnehmer im Schnitt 181,5 Sekunden, für die Java Darstellung 167 Sekunden und für die CSD Darstellung 225,5 Sekunden gebraucht. Somit hat die Fehlersuche in der CSD Darstellung 58,5 Sekunden länger gedauert als in der Java Darstellung und 44 Sekunden länger als in der PlainText Darstellung.

Die deskriptive Statistik in Abbildung 6.1 zeigt außerdem, dass bei der PlainText Darstellung eine AOI im Schnitt 8 Mal angeschaut wurde, bei der Java Darstellung 11 Mal und bei der CSD Darstellung 9,5 Mal.

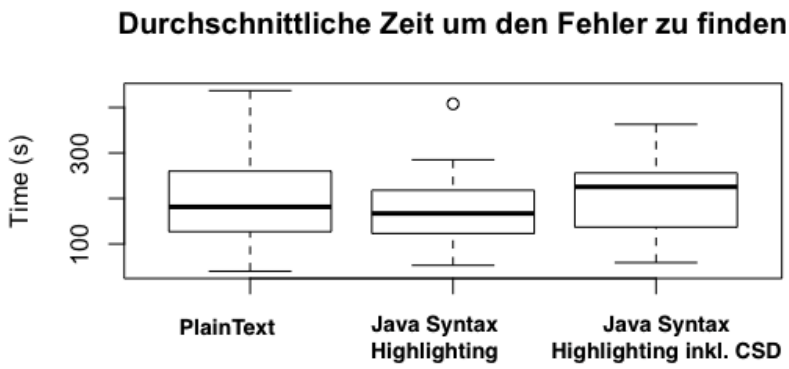


Abbildung 6.2: Dieses Boxplot zeigt die durchschnittliche Zeit in Sekunden an, mit der der Fehler in den unterschiedlichen Codedarstellungen gefunden wurde. Dabei sieht man, dass man in der Java Darstellung im Durchschnitt den Fehler schneller gefunden hat, verglichen zur CSD und PlainText Darstellung. Die PlainText Darstellung war im Schnitt schneller als die CSD Darstellung. Betrachtet man jedoch die individuelle Performance der Teilnehmer (siehe gestrichelte Linien), variieren die Werte zwischen den Codedarstellungen sehr, sodass PlainText insgesamt schlechter ausfällt.

6.3 Visuelle Analyse der Daten

In diesem Abschnitt werden die Daten, mit Hilfe des entwickelten Tools, visuell analysiert.

In Abbildung 6.3 wird ein Vergleich gemacht zwischen allen Teilnehmern die das Stimulus *Accumulate PlainText* angeschaut haben. Im ersten Fenster sieht man den Review-Vorgang von Teilnehmer Nummer 1. Mit 14,6 Minuten hat er verglichen zu den anderen Teilnehmern, für den gesamten Studienversuch am wenigsten gebraucht. Für diese Aufgabe (*Accumulate PlainText*) hat er 2,8 Minuten gebraucht. Es ist deutlich erkennbar, wie der Code zuerst gescannt wurde, um dann den Fokus auf die While-Schleife zu setzten, wo sich der Fehler auch befand. Wenn man seinen gesamten Studienverlauf anschaut kann dieses Verhalten in 83% der Review-Vorgänge beobachtet werden.

Teilnehmer Nummer 2 hat 4,54 Minuten gebraucht für dieselbe Aufgabe. Der Review-Vorgang von Teilnehmer zwei sieht im Vergleich dazu anders aus. Man findet in seiner Visualisierung auch einen Scan Muster, allerdings zieht sich das in die Länge, bis das komplette Programm angeschaut wurde. Sein Blick verweilt in der While-Schleife und man erkennt, dass sich die Leserichtung innerhalb der Zeilen selbst sich oft ändern (siehe Abbildung 6.3 Balken unterhalb der Timeline). Man sieht, dass er den Fehler möglicherweise schon zu Beginn gefunden hat, weshalb er den Fokus hauptsächlich auf die While-Schleife setzt, dann den Rest des Programms beobachtet, um wiederrum zum Fehler zu springen. Die Fixationsanzahl und Dauer sind auch

Abbildung 6.3: Hier werden die Review-Vorgänge von 5 Teilnehmer verglichen, die dasselbe Programm *Accumulate PlainText* angeschaut haben. Man sieht, die unterschiedlichen Performances der Teilnehmer untereinander aufgelistet. Dadurch ist ein direkter Vergleich möglich. Hier erkennt man unter anderem verschiedene Scan Muster, sowie die unterschiedliche Dauer der individuellen Review-Vorgänge. Es wird auch der Richtungswechsel innerhalb der Zeilen gezeigt unterhalb der Timeline (blau, wenn der Blick vom aktuellen Stand nach rechts wandert, gelb im umgekehrten Fall). Zusätzlich wird links die Häufigkeit, mit der eine AOI angeschaut wird angezeigt. Dies varriert auch je nach Performance.

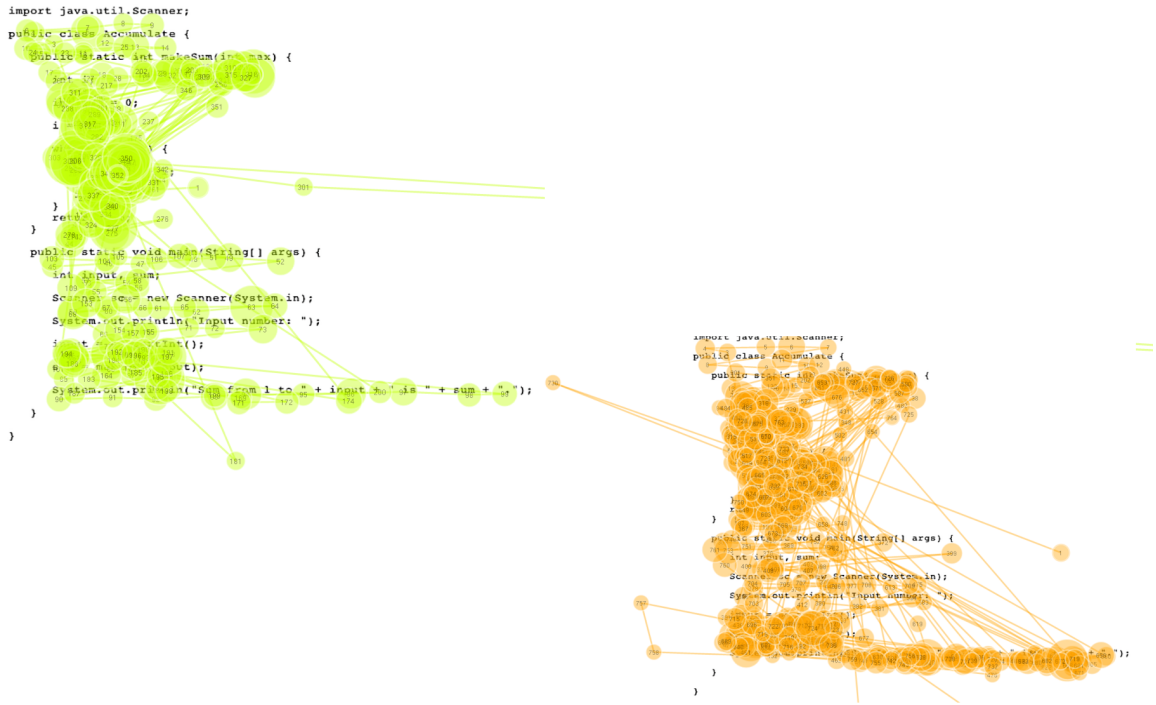


Abbildung 6.4: Scanpath von Teilnehmer 1 und 2 im Vergleich. Man erkennt, dass die Fixationen bei Teilnehmer 2(organge) viel weniger dauern (siehe Größe des Kreises) und die Anzahl größer ist.

unterschiedlich (siehe Scanpath Abb. 6.4). Dieses Verhalten sieht man in all seine Review-Vorgänge. Er hat für den gesamten Versuch 34,4 Minuten gebraucht.

Der dritte Teilnehmer, hat im Vergleich zu den ersten zwei mit 1,4 Minuten am kürzesten gebraucht. Sein Scan Muster ist nicht gleichmäßig, wie das des ersten Teilnehmers. Es scheint sogar, als würde er die While-Schleife überspringen, um zuerst die main()-Methode zu analysieren und anschließend springt er wieder hoch zur While-Schleife. Seine Strategie könnte ihm einen zeitlichen Vorsprung gegeben haben, da er versucht den Kontrollfluss, beginnend von der Hauptmethode des Programms, zu analysieren. Was man bei diesem Teilnehmer feststellen kann ist, dass er diese Strategie nicht in all seinen Review-Vorgängen konsequent durchgeführt hat (siehe Abbildung 6.5).

50% seines gesamten Review-Vorgangs ähnelt dem Review-Vorgang des zweiten Teilnehmer. In den restlichen 50% fällt auf, dass seine Strategie in der PlainText Darstellung mit Scan Muster versehen ist. Dies könnte ein Indiz dafür sein, dass durch das fehlende Highlithing, sein Blick nicht von spezifischen Programmkomponenten gefangen wird, weshalb er gezwungen ist, eine Struktur im Programm künstlich herzustellen und im Schnitt ca. 2 Minuten länger gebraucht hat, um den Fehler zu finden. In der CSD und in der Java Darstellung, sind die Vorgänge

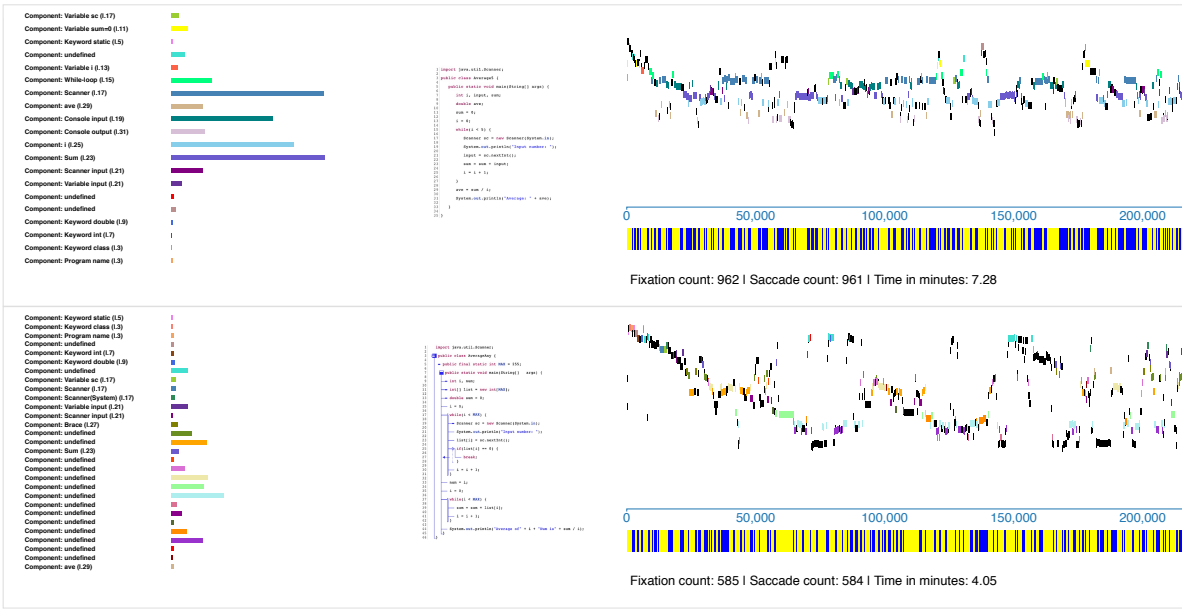


Abbildung 6.5: Hier werden weitere Review-Vorgänge von Teilnehmer Nummer 3 gezeigt. Dabei sieht man deutlich, dass er in anderen Reviews unstrukturierter vorgegangen ist und länger gebraucht hat.

eher unstrukturiert und von langen Fixationen unterschiedlicher Programmkomponenten charakterisiert. Insgesamt hat er 26,3 Minuten gebraucht.

Beim vierten Teilnehmer springt als erstes ins Auge, dass er mit 5,8 Minuten am längsten für dieselbe Aufgabe gebraucht hat. Die Komponenten, die er am längsten betrachtet hat, sind hauptsächlich Schlüsselwörter und die While-Schleife, wo sich der Fehler befindet. Verglichen zu den anderen Vorgängen erkennt man, dass er vom oberen Anfang des Programms zum Ende wandert und umgekehrt, und das mehrmals im Review-Verlauf. Diesen Lesevorgang erkennt man in allen Reviews seines Studien-Vorgangs. Zudem sieht man deutlich, dass er bei der CSD und der PlainText Darstellung um einiges länger gebraucht hat (siehe Abbildung 6.6).

Dies könnte daran liegen, dass er bei der PlainText Darstellung zu wenig Farbreferenzen zu den Komponenten hat, um den Code strukturiert zu analysieren, somit muss er auch, wie Teilnehmer Nummer 3, eine Struktur im Code bilden, um dann den Fehler zu finden. Bei der Java Darstellung hat er hingegen durch das Highlighting eine Struktur im Programm, somit wird er beim Review unterstützt. Man muss auch erwähnen, dass die Programme eine unterschiedliche Länge haben. Dies kann auch ein Grund, für die unterschiedliche Dauer der Reviews sein. Insgesamt benötigte er 37,1 Minuten.

Teilnehmer Nummer 5 hat einen ähnlichen Lesefluss wie Teilnehmer Nummer 1 und 3. Man sieht, dass er im Vergleich zu den anderen Kandidaten weniger AOIs angeschaut hat und sich hauptsächlich auf den Fehler in der While-Schleife konzentriert. Innerhalb seines gesamten Review-Vorgangs erkennt man keine genaue Lesestrategie, da sich die Review-Vorgänge nicht

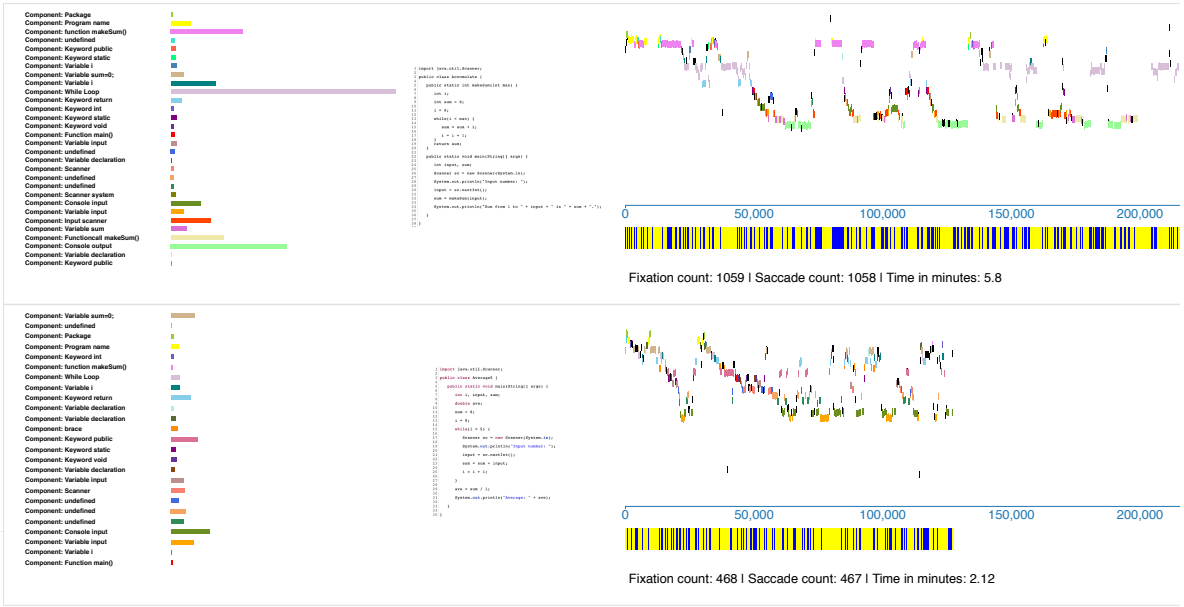


Abbildung 6.6: Reviews von Teilnehmer 4. Hier sieht man den unterschied, wie er die PlainText Aufgabe und die Java Aufgabe gelöst hat und man erkennt, dass er für die PlainText Aufgabe länger gebraucht hat um den Fehler zu erkennen.

ähneln und zusätzlich haben die Codedarstellungen auch keinen Einfluss gehabt, da der Fehler auch in unterschiedlichen Codedarstellungen gefunden wurde. Insgesamt hat er 28,4 Minuten gebraucht.

Die Programmiererfahrung von Teilnehmer 1 und 3 liegen im hohen Niveau (5-6). Teilnehmer Nummer 2 und 5 haben eine fortgeschrittene Programmiererfahrung (3-4) und Teilnehmer Nummer 4 ist ein Programmieranfänger.

Es folgt nun die Anaylse aller Teilnehmer die *Accumulate JAVA* angeschaut haben die in Abbildung 6.7 zu sehen ist.

Teilnehmer Nummer 1, hat in seinem Review-Vorgang die While-Schleife am längsten angeschaut und es ist keine genaue Lesestrategie zu erkennen. Die Fixationspunkte sind kurz und springen häufig von einem Extrem zum andern des Programms. Was man auch in seinen restlichen Review-Vorgängen beobachten kann. Für den gesamten Versuch hat er 28,6 Minuten gebraucht.

Bei Teilnehmer Nummer 2 hingegen, erkennt man das Scan Muster und die Komponente die am meisten angeschaut wurde, ist ebenfalls die While-Schleife. Diese Lesestrategie behält er in jedem seiner Review-Vorgänge bei. In 67% der Review-Vorgänge seiner gesamten Studie, hat er den Fehler durch diese Strategie gefunden. Insgesamt brauchte er 21 Minuten. Verglichen zum ersten Kandidaten erkennt man eine strukturierte Vorgehensweise beim Lesen des Programms. Abbildung 6.8 zeigt die unterschiedlichen Scanpaths von Teilnehmer 1 und 2 .

Abbildung 6.7: Hier werden die Teilnehmer angezeigt, die das Programm *Accumulate Java* angeschaut haben. Der direkte Vergleich zeigt die unterchiedlich lange Dauer der Review-Vorgänge, sowie der Fokus, den die jeweiligen Teilnehmer auf die Komponenten des Programms gesetzt haben.

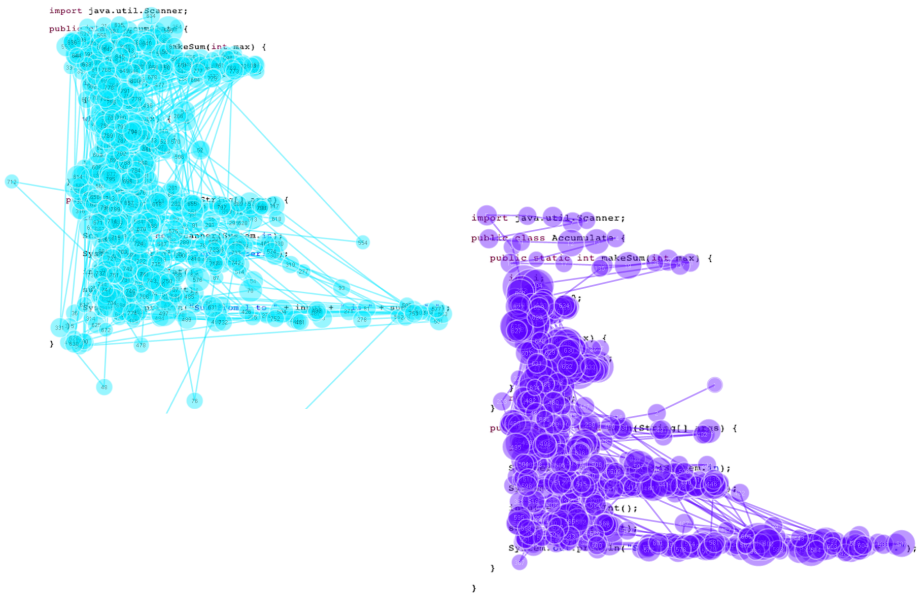


Abbildung 6.8: Scanpath von Teilnehmer 1 und 2 im Vergleich. Teilnehmer 1 (links oben) hat eine hohe Anzahl an Fixationen, verglichen zu Teilnehmer 2.

Der Review-Vorgang von Teilnehmer Nummer 3, sieht kompakter aus und man sieht im Vergleich zu den andern Teilnehmern, keine großen unterschiede. Er brauchte insgesamt 22,5 Minuten für den gesamten Versuch und auch er fand in 67% der Review-Vorgänge einen Fehler. Diese Lesestrategie findet man in seinem gesamten Studienverlauf.

Der letzte Teilnehmer hat den Fehler am schnellsten gefunden. Er wandert zur main()-Methode und simuliert den Kontrollfluss des Programms, wie Teilnehmer Nummer 4 im *Accumulate PT* Stimulus. Er hat 100% der Fehler gefunden und eine Gesamtdauer von 21,35 Minuten gebraucht. Wenn man seinen gesamten Review-Vorgang getrachtet, erkennt man viele kleine Scan Muster und die Performance zwischen den jeweiligen Codedarstellungen unterscheidet sich kaum, sie variiert nur in der Dauer, abhängig von der Länge des Programms.

Teilnehmer 1 ist ein Programmieranfänger(1-2), der Teilnehmer Nummer 2 hat eine mittelmäßige Programmiererfahrung (3-4) und die restlichen zwei haben eine fortgeschrittene Programmiererfahrung(5-6).

Es werden nun alle Teilnehmer verglichen, die das *Accumulate CSD* Programm angeschaut haben. Die Review-Vorgänge sieht man in Abbildung 6.9. Der Review-Vorgang vom ersten Teilnehmer weist gewisse Scan Muster auf, über den gesamten Ablauf. Er hat eine hohe Anzahl an Fixationen benötigt, dennoch hat er den Fokus nicht auf die While-Schleife gesetzt, im Gegensatz zu den anderen Teilnehmen. Seine restlichen Review-Vorgänge sind durch wiederholte Scan Muster charakterisiert. Insgesamt brauchte er 32,3 Minuten und hat in 50% des Review-Vorgangs den Fehler gefunden.

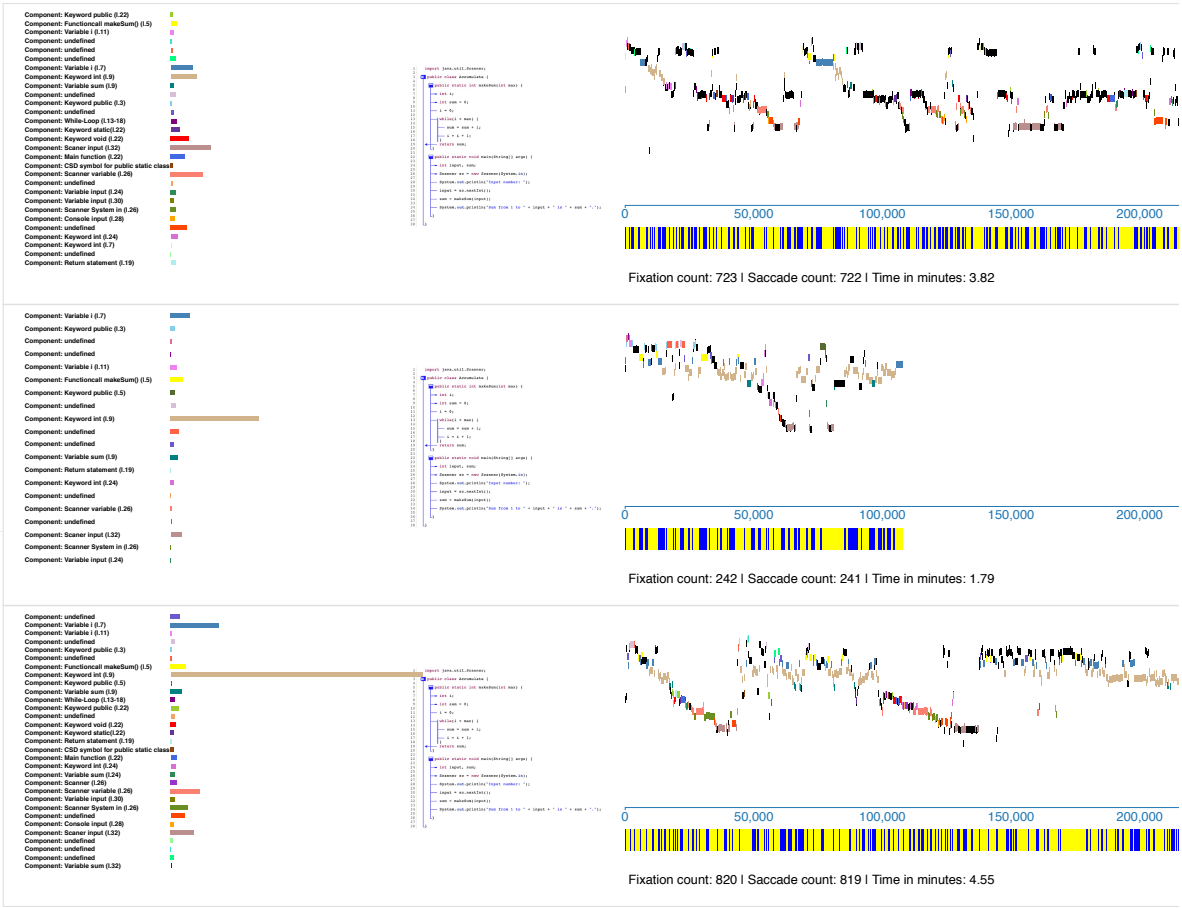


Abbildung 6.9: Hier werden die Review-Vorgänge der Teilnehmer gezeigt, die das Programm *Accumulate CSD* angeschaut haben.

Teilnehmer Nummer 2 hat den Fehler in kürzerer Zeit gefunden. Seine Fixationsanzahl ist geringer und die Leserichtung ändert sich seltener als bei Teilnehmer 1. Die Anzahl der betrachten AOIs ist auch kleiner und die am längsten betrachtete ist die While-Schleife. Er brauchte 20,5 Minuten und fand auch in 50% des Review-Vorgangs den Fehler. Er hat den Fokus hauptsächlich auf die While-Schleife setzte.

Teilnehmer Nummer 3, hat im Vergleich zu den anderen Teilnehmern am längsten gebraucht und zeigt einen strukturierten Vorgang beim Review. Er scannt den Code von oben bis unten, springt dann wieder zur Funktion makeSum() und hält sich für eine gewisse Zeit auf der While-Schleife auf und wiederholt erneut diesen Vorgang. Wenn man seinen gesamten Verlauf anschaut, erkennt man dass er für die CSD Darstellung länger gebraucht hat (im Schnitt 1,2 Minuten), im Vergleich zur Java oder PlainText Darstellung.

Die Programmiererfahrung des ersten Teilnehmers ist im mittelmäßig(3-4), während der letzte Teilnehmer im Bild ein Programmieranfänger ist.

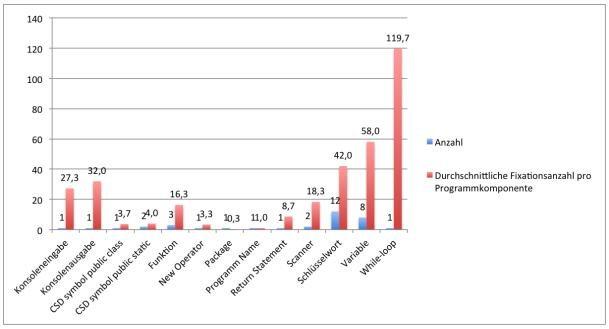


Abbildung 6.10: Diese Abbildung zeigt, die durchschnittliche Anzahl der Betrachteten AOIs im Programm *Accumulate CSD*.

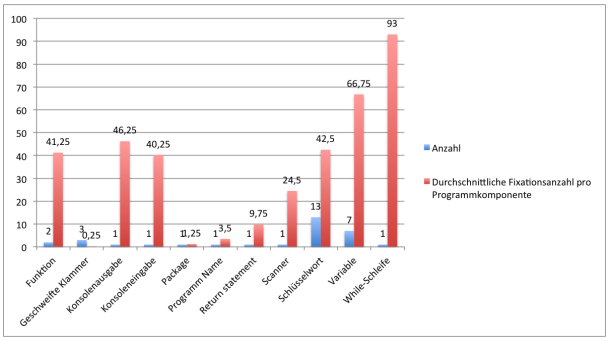


Abbildung 6.11: Diese Abbildung zeigt, die durchschnittliche Anzahl der Betrachteten AOIs im Programm *Accumulate JAVA*.

Abbildung 6.10, 6.11, 6.12 zeigen die durchschnittliche Anzahl der betrachteten AOIs im Programm *Accumulate* in den jeweiligen Codedarstellungen. Dabei erkennt man z.B. dass in der PlainText Darstellung, im Schnitt weniger AOIs angeschaut wurden, verglichen zur Java oder CSD Darstellung. Schlüsselwörter, wie public oder class, werden in der Java Darstellung im Durchschnitt am meisten angeschaut mit 42,5 Mal, gefolgt von der CSD Darstellung mit 42,0 während in der PlainText Darstellung nur 24,2 Mal angeschaut wurden. Die Variablen wurden bei der PlainText Darstellung auch relativ wenig mit 40,4 Mal angeschaut, in der CSD Darstellung 58,0 Mal und in der Java Darstellung 66,75 Mal. Die While-Schleife wurde von allen am meisten angeschaut, dort befand sich auch der Fehler. In der CSD Darstellung wurden die CSD Symbole relativ wenig angeschaut im Vergleich zu anderen Komponenten (im Schnitt 3,9 Mal). Im Schnitt wird jede AOI in der PlainText Darstellung 10.7 Mal angeschaut, in der Java Darstellung 10,25 und in der CSD Darstellung 9,2 Mal angeschaut.

Alle Review-Vorgänge haben schlussendlich dazu geführt, dass die Funktionsweise des Programms richtig erfasst und auch der Fehler gefunden wurde, trotz der zum Teil großen Diskrepanzen zwischen der Zeit, die sie für den Review-Vorgang gebraucht haben und der Lesestrategie. Vergleicht man die Performances untereinander, so kann man, außer der unterschiedlichen Dauer der Betrachtung, in jeder Visualisierung eine andere Lesestrategie erkennen.

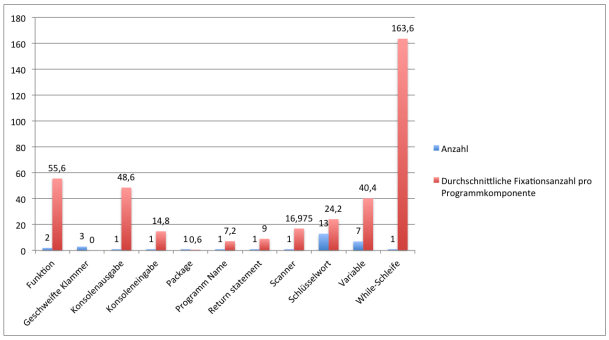


Abbildung 6.12: Diese Abbildung zeigt, die durchschnittliche Anzahl der Betrachteten AOIs im Programm *Accumulate PT*.

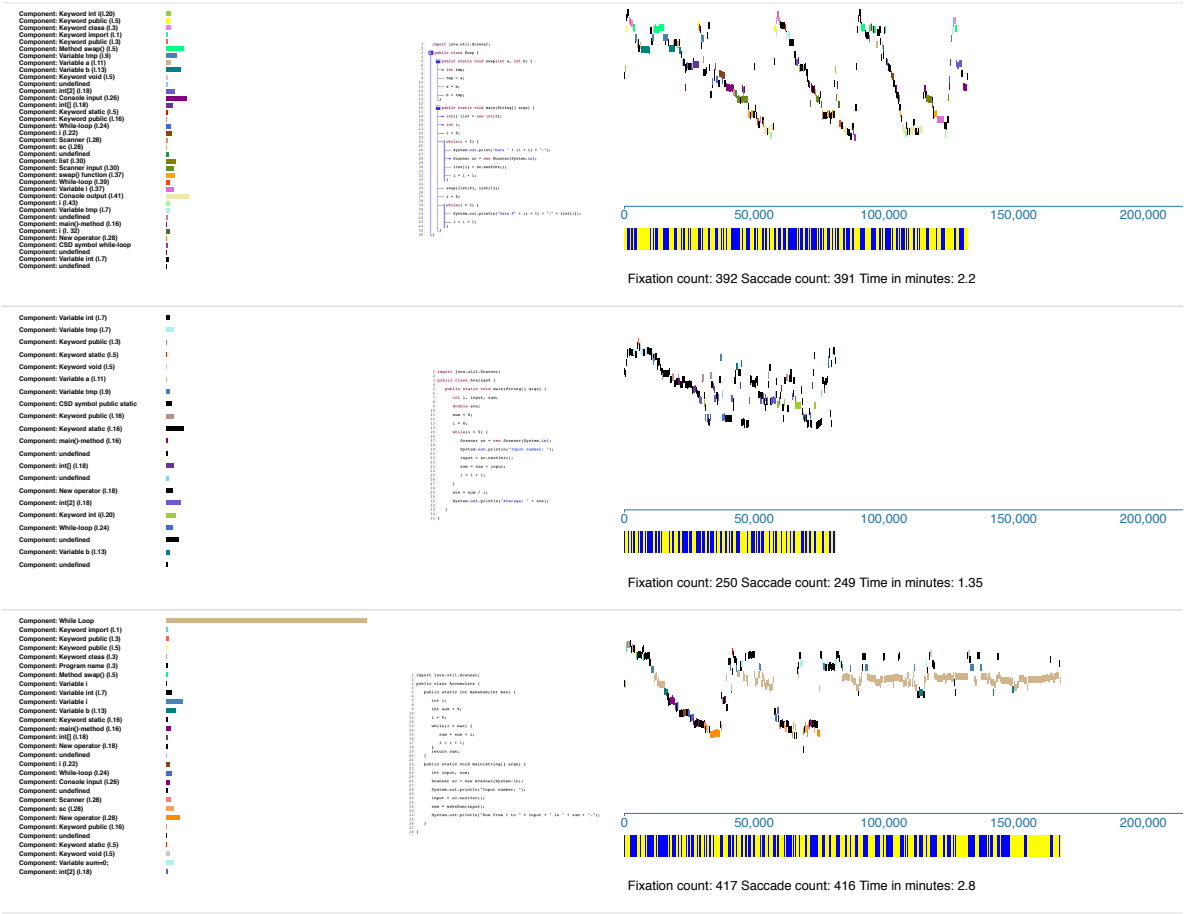


Abbildung 6.13: Diese Abbildung zeigt das Verhalten eines Teilenehmers, beim der Codeanalyse von 3 Programmen in jeweils 3 unterschiedlichen Codedarstellungen. Man erkennt, dass die Fehlersuche in der Java Darstellung am schnellsten war, sowie eine Lesestrategie, charakterisiert durch viele Scan Muster, die der Leser in allen Vorgängen anwendet.

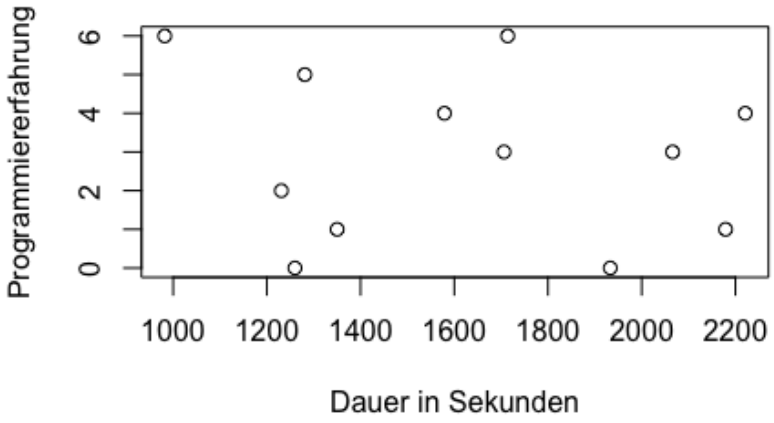


Abbildung 6.14: Diese Grafik zeigt die Relation zwischen der Programmiererfahrung des Teilnehmers und die gesamte Dauer des Review-Vorgangs in Sekunden. Dabei ist zu erkennen, dass Programmierer mit derselben Programmiererfahrung auch unterschiedliche Performances haben können.

In Abbildung 6.13 werden die Review-Vorgänge eines einzelnen Teilnehmers verglichen. Dabei werden für den Vergleich jeweils 3 unterschiedliche Programme in den drei Darstellungen gewählt. Die Programme sind *Accumulate PlainText Average5 Java* und *Swap CSD*. Man erkennt in jedem Analyse Vorgang eine Lesestrategie, die aus Scan Mustern zusammengesetzt ist. Der Teilnehmer fand den Fehler in der Java Darstellung in 1,35 Minuten, in der PlainText Darstellung in 2,8 und in der CSD Darstellung in 2,2 Minuten. Somit war die Fehlersuche in der Java Darstellung am schnellsten, gefolgt von der CSD und der PlainText Darstellung.

Abbildung 6.15 zeigt zusammenfassend die durchschnittliche Dauer in Sekunden, um den Fehler zu finden. Dabei wird die Dauer für die jeweiligen Stimuli in den unterschiedlichen Codedarstellungen angezeigt. Man erkennt, dass die CSD Darstellung und die PlainText Darstellung, im Vergleich zur Java Darstellung schlechter abschneiden.

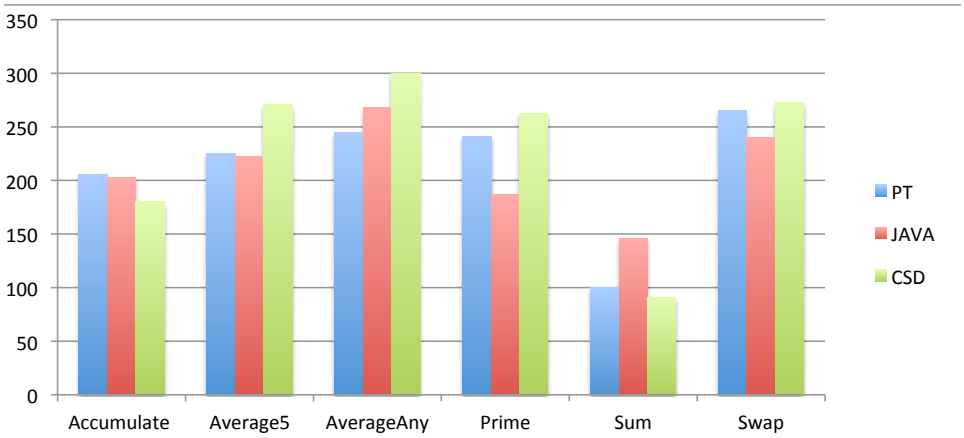


Abbildung 6.15: Durchschnittliche Zeit in Sekunden bis der Fehler gefunden wird, in den unterschiedlichen Codedarstellungen.

6.4 Auswertung des Fragebogens

Im Folgenden werden die Ergebnisse des Fragebogens vorgestellt. Diese dienen zur Messung des subjektiven Empfindens der einzelnen Codedarstellungen. Um eine bessere Aufteilung zu erhalten, wurde die Skala nochmals in 3 Kategorien aufgeteilt, niedrig(1-2), mittelmäßig(3-4) und fortgeschritten(5-6).

6.4.1 Vorkenntnisse

1. Ich habe Programmiererfahrung: Skala: 1 = wenig,6 = viel

Insgesamt lag der Mittelwert bei 3,8. 2 von 12 Teilnehmer haben eine hohe Programmiererfahrung, 7 von 12 eine mittelmäßige und 3 von 12 eine niedrige Programmiererfahrung.

2. Ich habe Programmiererfahrung in Java : Skala = 1 wenig, 6 = viel

Hier lag der Mittelwert bei 2,4. Insgesamt gab es 2 Teilnehmer mit fortgeschrittenen Programmiererkenntnissen in Java, 4 Teilnehmer mit einer mittelmäßigen und 6 Teilnehmer mit einer niedrigen Programmiererfahrung in Java.

6.4.2 Anstrengung

Wie sehr mussten Sie sich anstrengen, um die gestellte Aufgabe zu erfüllen? Skala 1 = gar nicht, 6 = viel

Der Mittelwert lag bei 3,7. Abbildung 6.16 stellt die Anstrengung grafisch dar.

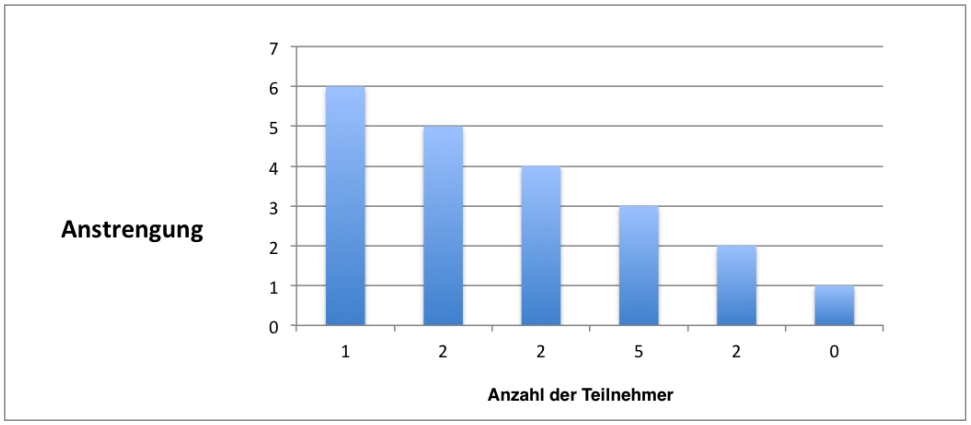


Abbildung 6.16: Diese Abbildung zeigt, wie sehr sich die Teilnehmer anstrengen mussten, von einer Skala von 1 bis 6.

Verständnis

Wie sehr haben die unterschiedlichen Visualisierungen dazu beigetragen, den Code besser schneller zu verstehen?

1. Plaintext : Skala: 1 = gar nicht, 6 = viel

Der Mittelwert lag bei 1,5. 6 Teilnehmer haben es auf einer Skala von 1 bis 6 mit 1 bewertet, 2 mit 3 und 3 mit 2.

2. Java Syntax Highlighting : Skala: 1 = gar nicht, 6 = viel

Der Mittelwert lag bei 5,3. 6 Teilnehmer haben es mit 6 bewertet, 3 mit 5 und 3 mit 4.

3. Java Code inkl. COntrol Strukture Diagrams : Skala: 1 = gar nicht, 6 = viel

Der Mittelwert lag bei 3,0. Es gab ein Teilnehmer der des mit 3 bewertet hat, d mit 3, einer mit 4, einer mit 5 und zwei mit 6.

Ranking

Welche der Visualisierungen würden Sie bevorzugen, wenn Sie den Java-Code auf Fehlerexistenz analysieren müssten? (Ranking von 1(hoch) bis 3(niedrig))

1. Plaintext

2. Java-Syntax-Hyghlighting

3. Java Code mit Control Structure Diagrams

Die Mittelwerte waren, für PlainText 2,9 , für CSD 2,1 und für Java 1,0. (siehe Abbildung 6.17)

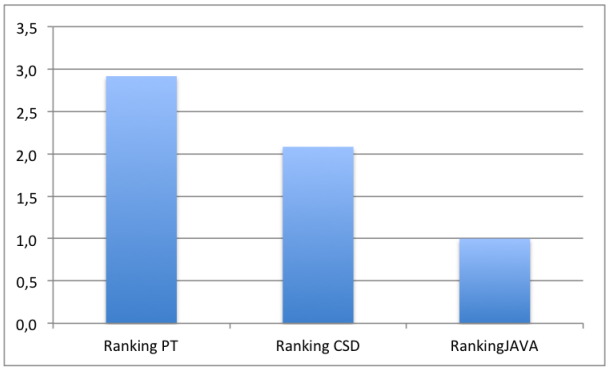


Abbildung 6.17: Durchschnittliche Scores des Rankings. Man sieht, dass Java bevorzugt wird, gefolgt von CSD und PlainText(PT)

7 Diskussion

In diesem Kapitel werden die Ergebnisse der Studie anhand der entstandenen Hypothesen diskutiert.

Hypothese 1

Control Structure Diagrams beschleunigen die Fehlersuche im Programm, verglichen zu den anderen Darstellungen.

Sowohl die subjektive Empfindung der Teilnehmer als auch die deskriptive Statistik haben gezeigt, dass die CSD Diagramme, keinen unterstützenden Effekt bei der Fehlersuche im Programm hatten. Die Statistik in Abbildung 6.16 zeigt, dass man in den CSD Darstellungen der Stimuli, im Schnitt länger gebraucht hat, um den Fehler zu finden, verglichen zu den anderen Darstellungen. Während Programmieranfänger die CSDs als gute Ergänzung zum Code empfanden, wurden sie von fortgeschrittenen Programmierern eher als überflüssig oder ablenkend empfunden. Dies könnte dazu geführt haben, dass sie beim Review Vorgang eher ignoriert wurden, oder den Reviewer durch die zusätzlichen Informationen eher verwirrt haben.

Auf visueller Ebene konnte man durch das Visualisierungswerkzeug in einzelnen Review Vorgängen jedoch beobachten, dass die CSD Darstellung (siehe visuelle Analyse in Kapitel 6), oft genauso lange Zeit in Anspruch genommen hat, wie die anderen zwei Darstellungen. Somit könnte der Teilnehmer in diesen Fällen die CSDs ignoriert haben. In anderen Beispielen, wurde sogar mehr Zeit für die Fehlersuche gebraucht im Vergleich zu den anderen Darstellungen. Das zeigt, dass die Reviewperformance und das Programmverständnis von vielen menschlichen Faktoren abhängt, wie auch Uwano [UNM+06] erkannte und man, zumindest in dieser Studie, keine konkreten Aussagen über den Effekt von CSDs im Programmcode treffen kann.

Hypothese 2

Die durchschnittliche Fixationsanzahl pro AOI in der PlainText Darstellung ist kleiner, verglichen zu den zwei anderen Darstellungen.

Die deskriptive Statistik in 6.1 unterstützt diesen Gedanken. Der Grund für den niedrigeren Wert bei der PlainText Darstellung könnte sein, dass durch das fehlende Highlighting die Teilnehmer nicht dazu animiert werden, bestimmte Komponenten zu fixieren, die für das

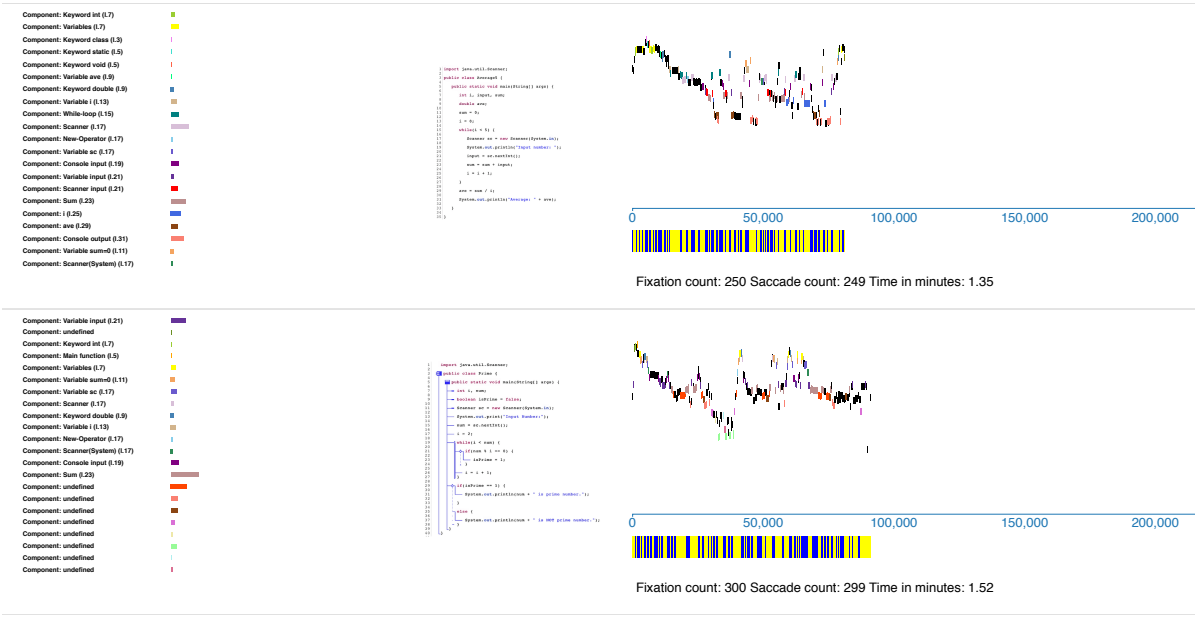


Abbildung 7.1: Review Unterschied zwischen CSD und Java Darstellung

Programmverständnis relevant sein könnten. Dies könnte unter anderem dazu geführt haben, dass die Fehlerdetektion bei dieser Darstellung länger gedauert hat, gefolgt von der CSD Darstellung, wo der Zusatz an Symbolen den Review Vorgang gestört haben könnte.

Bei der Auswertung der durchschnittlichen Fixationsanzahl pro AOI, für die Darstellungen des Programmes *Accumulate* (Abbildung 6.10 bis 6.12) wurde bemerkt, dass die Schlüsselwörter (public, class, static etc.) in der Java Darstellung im Durchschnitt am meisten angeschaut wurden mit 42,5 Hits, gefolgt von der CSD Darstellung mit 42,0 Hits während in der PlainText Darstellung sie nur 24,2 Mal angeschaut wurden. Dies weist darauf hin, dass durch das Highlighing bestimmte Programmkomponenten hervorgehoben werden, und die Aufmerksamkeit dadurch leichter gefangen wird. Außerdem fällt auf, dass auch die Variablen in der PlainText Darstellung weniger angeschaut wurden. Das könnte an ihrer Position im Programm liegen. Wenn man bei der Java Darstellung die Schlüsselwörter der Variablendeklaration anschaut, wandert der Blick automatisch zur zugehörigen deklarierten Variable. Da diese Farbinformation in der PlainText Darstellung entfällt, werden sie auch automatisch weniger betrachtet.

Durch die visuelle Analyse konnte man aber auch oft beobachten, dass der Review Vorgang in den PlainText Darstellungen sogar strukturierter war, als in den Java Reviews. Das kann man als Lesestrategie interpretieren. Die PlainText Darstellung, zwingt den Leser selbst eine Struktur im Code zu schaffen, da sie ohne Hilfsmittel auskommen müssen, während die Java Darstellung, durch das Highlighting, dem Teilnehmer die nötige Struktur vorgibt, um darauf die Fehlersuche zu starten.

Hypothese 3

Die durchschnittliche Dauer, bis der Teilnehmer den Fehler findet, ist in der PlainText Darstellung am größten, verglichen zu den anderen zwei Darstellungen.

Anders als erwartet, war die Fehlersuche im Schnitt in der CSD Darstellung am langsamsten. Was man aber auch noch erkennt ist, dass die individuelle Performance der Teilnehmer in den unterschiedlichen Darstellungen variiert, sodass PlainText insgesamt schlechter ausfällt.

Was man außerdem feststellen konnte ist, dass, selbst wenn der Fehler nicht immer gefunden wurde, die Funktionsweise des Programms von 96% der Teilnehmer erfasst wurde. In diesen Vorgängen wurde jeweils auch immer der Programmname mit einer durchschnittlichen Anzahl von 5,5 Fixationen durchgeführt. Der Name hat einen hilfreichen Hinweis auf die Funktionsweise gegeben, wie beispielsweise Sum5(bildet die Summe von 5 Zahlen) oder Prime(entscheidet, ob eingegebene Zahl eine Primzahl ist oder nicht). So konnte man aus einer rein logischen Schlussfolgerung die Funktionsweise herleiten. Daraus kann man schließen, dass ein gut gewählter Funktions- oder Programmname, den Review Vorgang durchaus unterstützen kann und sollte bei der Programmentwicklung berücksichtigt werden.

Die Programmiererfahrung der Teilnehmer, stimmt nicht in allen Fällen mit der Performance überein. Dies bestätigt auch Abbildung 6.14, in welcher die Relation zwischen der Programmiererfahrung und der gesamten Dauer in Sekunden gezeigt wird. Teilnehmer mit derselben Programmiererfahrung können trotzdem unterschiedlich lange für dieselbe Aufgabe gebraucht haben. Es ist außerdem schwer, die Performance anhand der Programmiererfahrung zu klassifizieren, da sie vom Teilnehmer selbst angegeben wird und somit eine subjektive Einschätzung ist.

Das Scan Muster, das erstmals von Uwano [UNM+06] entdeckt wurde, ist auch in dieser Studie in vielen Review Vorgängen zu finden. Er definiert die Scan-Zeit als die Zeit, bis die ersten 80% des Codes gelesen wurden. Sharif [SFM12] bestätigt durch statistischen Tests, dass die Länge der Scan-Zeit einen eindeutigen Einfluss auf die Zeit bis zur Fehlererkennung hat. Je akkurater der Scan Vorgang am Anfang, desto schneller oder genauer wurde der Fehler in ihrer Studie gefunden. In dieser Studie konnte man beobachten, dass Teilnehmer mit mehr Programmiererfahrung (5 oder 6), den Scanvorgang sogar mehrmals durchgeführt haben, bis sie zu einem Ergebnis gekommen sind. Tatsächlich, wenn man Abbildung 6.3 anschaut, und beispielsweise die ersten zwei Teilnehmer vergleicht, erkennt man, dass der erste Teilnehmer den Code in kürzerer Zeit scannt, und auch den Fehler schneller findet. Der zweite Teilnehmer hingegen, braucht für das Scanning länger und findet den Fehler auch später. Das Scan Muster kann somit ein Indiz für eine strukturierte und kognitive Vorgehensweise beim Review interpretiert werden. In Hendrix [HCM+02] Studie wurde der Effekt von CSDs im Programmverständnisprozess analysiert. Dabei wurde festgestellt, dass sie einen positiven Effekt bei diesen Vorgang haben. Die Teilnehmer dieser Studie wurden hauptsächlich in Anfänger und Fortgeschrittene Programmierer unterteilt. Besonders für Programmieranfänger wurden positive Effekte erzielt, in Bezug auf das Verständnis des Programms. Was sich in

dieser Studie auch zeigt, jedoch ist die Anzahl der Teilnehmer in Hendrix Studie um einiges größer.

Durch die Analyse mit Hilfe des Visualisierungswerkzeugs, konnte man vor allem bei Programmiernfänger beobachten, dass die Review Vorgänge länger und unstrukturierter waren. Die Sakkadenrichtungen änderten sich häufig wie man am Balken unterhalb der Timeline sehen konnte. Bei fortgeschrittenen Programmieren konnte man hingegen eine genaue Struktur im Leseverhalten sehen und die Sakkadensprünge waren insgesamt ausgeglichener im Balken verteilt. Einen Vergleich zwischen einem Anfänger und einem Fortgeschrittenen Programmieranfänger kann man in den Abbildungen 7.2 und 7.3 sehen.

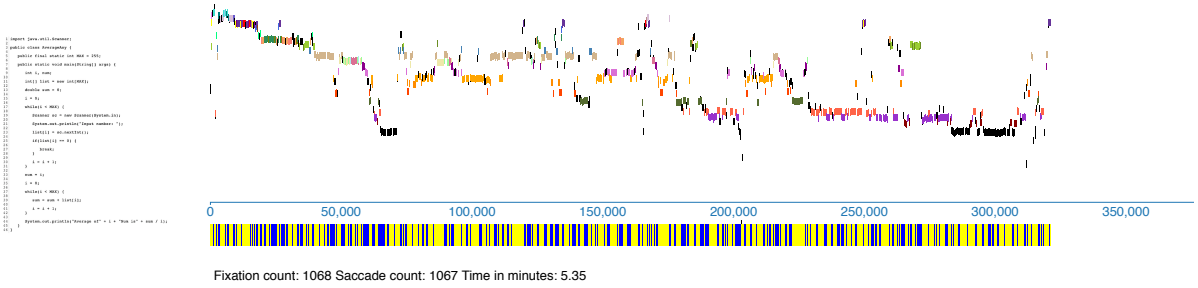


Abbildung 7.2: Review Vorgang eines Programmieranfängers. Der Vorgang ist von vielen Fixationen charakterisiert, der Sakkadenbalken unter der Timeline signalisiert den häufigen sprünghaften Wechsel zwischen den Zeilen im Programm.

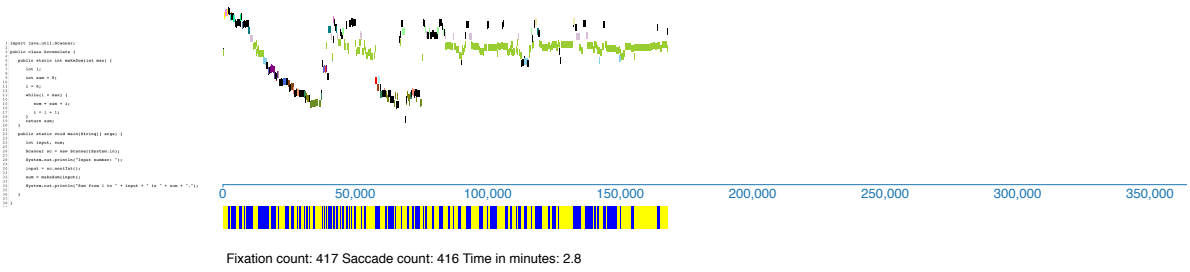


Abbildung 7.3: Review Vorgang eines fortgeschrittenen Programmierer. Der Review Vorgang ist gleichmäßig, die Fixationsanzahl ist gering.

8 Zusammenfassung und Ausblick

Zusammenfassend kann man sagen, dass die unterschiedlichen Darstellungsarten evtl. in einigen Fällen geholfen haben, den Code besser und schneller aufzufassen, oder den Review Vorgang sogar verlangsamt haben. Die strikte Unterteilung folgt, da die individuelle Performance einen stärkeren Einfluss auf das Ergebnis hat. Die Ergebnisse der Studie, vor allem die visuelle Analyse, haben gezeigt, dass man die Performance der einzelnen Teilnehmer durchaus durch Eyetracking bestimmen kann. Die CSD Diagramme wurden vor allem von Programmieranfängern sehr hoch gerankt und von fortgeschrittenen Programmierer als überflüssig empfunden. Was aber nicht bedeutet, dass sie komplett ignorieret wurden, die deskriptive Statistik zeigt sogar, dass sie zu einer schlechteren durchschnittlichen Zeit, bis zur Fehlerdetektion geführt haben. Gründe könnten sein, dadurch dass keiner davor mit CSDs gearbeitet hat, die Teilnehmer durch die zusätzliche Information überfordert waren. Die PlainText Darstellung wurde nicht positiv bewertet, aufgrund mangelnder Informationen in Form von Highlighting. Das Highlighting erzwingt nämlich im Code eine gewisse Strukturierung, die wiederrum den Teilnehmer dazu bringt, das Programm kritischer und zu analysieren, weshalb das Java Syntax Highlighling bevorzugt wurde.

Insgesamt schneidet die CSD Visualisierung schlechter ab, in Bezug auf die durchschnittliche Zeit bis der Fehler gefunden wird, gefolgt von der PlainText Darstellung. Java Syntax Highlighting hat am besten abgeschnitten und wurde auch von den Teilnehmern bevorzugt. Ein möglicher Grund, weshalb die CSD Darstellung nicht so gut abgeschnitten hat wie gedacht ist, dass es sich bei den Aufgaben hauptsächlich um Wartungsaufgaben handelt. Weshalb sich die meisten nicht darauf konzentriert haben, das Programm anhand der Symbole zu verstehen, sondern eher darauf, unabhängig von der möglichen Hilfestellung, den Fehler so schnell wie möglich zu finden. Würde die Aufgabe lauten, den syntaktischen Fehler zu finden, könnten die CSDs durchaus hilfreich sein, da man zusammenhängende Blöcke schneller auf einen Blick erkennt.

Durch die Analyse mit Hilfe des entwickelten Visualisierungswerzeugs, kann man die Performance der einzelnen Teilnehmer untersuchen und vor allem miteinander vergleichen. Man sieht, wie sich die Review Vorgänge von Teilnehmer zu Teilnehmer unterscheiden. Vor allem bei fortgeschrittenen Teilnehmer, erkennt man eine bestimmte Lesestrategie, die sie im gesamten Studienverlauf in jeder Aufgabe anwenden und damit in den meisten Fällen die gestellte Aufgabe lösen. Bei Programmieranfänger hingegen, sieht man zwischen den Visualisierungen große Unterschiede, da keine genauen Strategien beim Codereview angewendet wurden. Durch die farbliche Unterscheidung der einzelnen AOIs kann man sogar sehen, welche Komponente

des Programms genau angeschaut wird und wie lange, im Vergleich zu anderen. Das war in den Visualisierungstools der vergangenen Studien nicht möglich. In Abbildung 8.1 wir das Visualisierungswerkzeug, das in der Uwano Studie entstanden ist, angezeigt.

Abbildung 8.1: [UNM+06]. Visualisierungswerkzeug, das in der Uwano Studie verwendet wurde. Links wird das betrachtete Programm angezeigt, und rechts wird die zugehörige Timeline angezeigt. Der Unterschied zwischen Uwanos Visualisierungswerkzeug und das in dieser Studie realisierte Tool ist, dass die definierten AOIs farblich in der Timeline unterschieden werden, die Hits der AOIs mit Hilfe eines horizontalen Balkendiagramms angezeigt werden und die Leserichtung der Teilnehmer in einem Balken unterhalb der Timeline visualisiert werden. (siehe Abb. 6.3)

Ausblick

Man muss an dieser Stelle spezifizieren, dass die Hypothesen nur auf deskriptiv statistischer Ebene evaluiert wurden, weshalb man eine weitere Studie führen sollte, um die Hypothesen zu verifizieren oder falsifizieren. Interessant wäre es auch die Effektivität von Control Struture Diagrams an einer Gruppe von ausschließlich Programmieranfängern zu testen und das über einen längeren Zeitraum, um zu schauen, ob die Control Strukture Diagrams in gewisser Weise den Verständnis Vorgang beschleunigen können.

Man könnte zusätzlich das Visualisierungswerkzeug so erweitern, dass man beispielsweise die AOIs direkt im Tool einzeichnen kann. Dadurch hat man einen gewissen Freiheitsgrad bei der Wahl der AOIs und kann evtl. eine kategorisierte komponentenweise Analyse durchführen.

A Anhang

Im Anhang befinden sich folgende Dokumente:

1. Fragebogen zur Person

2. Fragebogen zur subjektiven Empfindung

3. Stimuli (Java Pogramme)

FRAGEBOGEN:

Allgemeine Fragen zu den Probanden:

Geschlecht:

○ männlich

○ weiblich

Alter: _____

Tragen Sie eine Sehhilfe?

○ NEIN

○ BRILLE

○ KONTAKTLINSEN

Abschluss:

○ ALLGEMEINE HOCHSCHULREIFE	○ BACHELOR	○ MASTER
○ STAATSEXAMEN	○ DIPLOM	Sonstige:

Studienfach:_____

Nebenfächer:_____

Angestrebter Abschluss:

○ ALLGEMEINE HOCHSCHULREIFE	○ BACHELOR	○ MASTER
○ STAATSEXAMEN	○ DIPLOM	Sonstige:

Abbildung A.1: Fragebogen zur Person.

ID:

Vorkenntnisse:
1. Ich habe Programmiererfahrung: ○ NEIN ○ JA

Falls JA:

wenig					viel
○	○	○	○	○	○
1	2	3	4	5	6

Programmiersprachen (Java ausgeschossen):_____

2. Ich habe Programmiererfahrung in Java: ○ NEIN ○ JA

Falls JA:

wenig					viel
○	○	○	○	○	○
1	2	3	4	5	6

3. Ich habe bereits mit Control Structure Diagrams gearbeitet: ○ NEIN ○ JA

Anstrengung:
Wie sehr mussten Sie sich anstrengen, um die gestellte Aufgabe zu erfüllen?

gar nicht					viel
○	○	○	○	○	○
1	2	3	4	5	6

Verständnis:
1. Wie sehr haben die unterschiedlichen Visualisierungen dazu beigetragen, den Code besser/schneller zu verstehen?
Plain Text:

gar nicht					viel
○	○	○	○	○	○
1	2	3	4	5	6

Java Syntax Highlithing:

gar nicht					viel
○	○	○	○	○	○
1	2	3	4	5	6

Java Code inkl. Control Structure Diagrams :

gar nicht					viel
○	○	○	○	○	○
1	2	3	4	5	6

Ranking : Welche Visualisierung würden Sie bevorzugen, wenn Sie den Java-Code auf Fehlerexistenz analysieren müssten? (Ranking von 1(hoch) bis 3(niedrig)).

○ Plain Text

○ Java Highlighted Syntax (wie in IDE Eclipse)

○ Java Code mit Control Structure Diagrams

1

Abbildung A.2: Fragebogen zur subjektiven Empfindung

```
import java.util.Scanner;
public class Accumulate {
   public static int makeSum(int max) {
      int i;
      int sum = 0;
      i = 0;
      while(i < max) {
         sum = sum + i;
         i = i + 1;
      }
      return sum;
   }
   public static void main(String[] args) {
      int input, sum;
      Scanner sc = new Scanner(System.in);
      System.out.println("Input number: ");
      input = sc.nextInt();
      sum = makeSum(input);
      System.out.println("Sum from 1 to " + input + " is " + sum + ".");
   }
}
```

Abbildung A.3: Accumulate

```
import java.util.Scanner;
public class Average5 {
   public static void main(String[] args) {
      int i, input, sum;
      double ave;
      sum = 0;
      i = 0;
      while(i < 5) {
         Scanner sc = new Scanner(System.in);
         System.out.println("Input number: ");
         input = sc.nextInt();
         sum = sum + input;
         i = i + 1;
      }
      ave = sum / i;
      System.out.println("Average: " + ave);
   }
}
```

Abbildung A.4: Average5

```java
import java.util.Scanner;
public class AverageAny {
   public final static int MAX = 255;
   public static void main(String[] args) {
      int i, num;
      int[] list = new int[MAX];
      double sum = 0;
      i = 0;
      while(i < MAX) {
         Scanner sc = new Scanner(System.in);
         System.out.println("Input number: ");
         list[i] = sc.nextInt();
         if(list[i] == 0) {
            break;
         }
         i = i + 1;
      }
      num = i;
      i = 0;
      while(i < MAX) {
         sum = sum + list[i];
         i = i + 1;
      }
      System.out.println("Average of" + i + "Num is" + sum / i);
   }
}
```

Abbildung A.5: AverageAny

```java
import java.util.Scanner;
public class Prime {
   public static void main(String[] args) {
      int i, num;
      boolean isPrime = false;
      Scanner sc = new Scanner(System.in);
      System.out.print("Input Number:");
      num = sc.nextInt();
      i = 2;
      while(i < num) {
         if(num % i == false) {
            isPrime = 1;
         }
         i = i + 1;
      }
      if(isPrime == true) {
         System.out.println(num + " is prime number.");
      }
      else {
         System.out.println(num + " is NOT prime number.");
      }
   }
}
```

Abbildung A.6: Prime

```
import java.util.Scanner;
public class Swap {
   public static void swap(int a, int b) {
      int tmp;
      tmp = a;
      a = b;
      b = tmp;
   }
   public static void main(String[] args) {
      int[] list = new int[2];
      int i;
      i = 0;
      while(i < 2) {
         System.out.print("Data " + (i + 1) + ":");
         Scanner sc = new Scanner(System.in);
         list[i] = sc.nextInt();
         i = i + 1;
      }
      swap(list[0], list[1]);
      i = 0;
      while(i < 2) {
         System.out.println("Data #" + (i + 1) + ":" + list[i]);
         i = i + 1;
      }
   }
}
```

Abbildung A.7: Swap

```
import java.util.Scanner;
public class Sum5 {
   public static void main(String[] args) {
      int i, input;
      i = 0;
      while(i < 5) {
         Scanner sc = new Scanner(System.in);
         System.out.println("Input Number: ");
         input = sc.nextInt();
         sum = sum + input;
         i = i + 1;
      }
      System.out.println("Sum: " + sum);
   }
}
```

Abbildung A.8: Informationen zu den verwendeten Stimuli.

Literaturverzeichnis

[BKR+14] T. Blascheck, K. Kurzhals, M. Raschke, M. Burch, D. Weiskopf, T. Ertl. „State-of-the-art of visualization for eye tracking data“. In: *Proceedings of EuroVis*. Bd. 2014. 2014 (zitiert auf S. 20).

[Boe+81] B. W. Boehm et al. *Software engineering economics*. Bd. 197. Prentice-hall Englewood Cliffs (NJ), 1981 (zitiert auf S. 23).

[BT06] R. Bednarik, M. Tukiainen. „An eye-tracking methodology for characterizing program comprehension processes“. In: *Proceedings of the 2006 symposium on Eye tracking research & applications*. ACM. 2006, S. 125–132 (zitiert auf S. 25, 26).

[CS90] M. E. Crosby, J. Stelovsky. „How do we read algorithms? A case study“. In: *Computer* 23.1 (1990), S. 25–35 (zitiert auf S. 25).

[Duc07] A. Duchowski. *Eye tracking methodology: Theory and practice*. Bd. 373. Springer Science & Business Media, 2007 (zitiert auf S. 15, 17, 19).

[FKA+13] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, G. Saake. „Do background colors improve program comprehension in the# ifdef hell?“ In: *Empirical Software Engineering* 18.4 (2013), S. 699–745 (zitiert auf S. 11, 24, 27).

[GB09] E. Goldstein, J. Brockmole. *Sensation and perception*. Nelson Education, 2009 (zitiert auf S. 16–18, 27).

[Gul14] C. Gull. *BigData mit JavaScript visualisieren*. Franzis Verlag GmbH, 2014 (zitiert auf S. 29, 32).

[HCM+02] D. Hendrix, J. H. Cross, S. Maghsoodloo et al. „The effectiveness of control structure diagrams in source code comprehension activities“. In: *Software Engineering, IEEE Transactions on* 28.5 (2002), S. 463–477 (zitiert auf S. 26, 42, 65).

[HNS06] T. Hakala, P. Nykyri, J. Sajaniemi. „An experiment on the effects of program code highlighting on visual search for local patterns“. In: *Psychology of Programming Interest Group* (2006), S. 38–52 (zitiert auf S. 27).

[Hol03] A. F. G̃. Hole. *How to Design and Report Experiments*. SAGE Publications Ltd, 2003 (zitiert auf S. 42).

[KR91] J. Koenemann, S. P. Robertson. „Expert problem solving strategies for program comprehension“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 1991, S. 125–130 (zitiert auf S. 11).

[LV00] F. Lanubile, G. Visaggio. „Evaluating defect detection techniques for software requirements inspections“. In: *ISERN Report no. 00-08* (2000), S. 1–24 (zitiert auf S. 24).

[MMS+04] A. Moreno, N. Myller, E. Sutinen, M. Ben-Ari et al. „Visualizing programs with Jeliot 3“. In: *Proceedings of the working conference on Advanced visual interfaces*. ACM. 2004, S. 373–376 (zitiert auf S. 26).

[SFM12] B. Sharif, M. Falcone, J. I. Maletic. „An eye-tracking study on the role of scan time in finding source code defects“. In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM. 2012, S. 381–384 (zitiert auf S. 25, 39, 41, 47, 65).

[UNM+06] H. Uwano, M. Nakamura, A. Monden, K.-i. Matsumoto et al. „Analyzing individual performance of source code review using reviewers' eye movement“. In: *Proceedings of the 2006 symposium on Eye tracking research & applications*. ACM. 2006, S. 133–140 (zitiert auf S. 11, 24–26, 29, 39, 41, 47, 63, 65, 68).

[Wie96] K. E. Wiegers. *Creating a software engineering culture*. Pearson Education, 1996 (zitiert auf S. 23).

[Züh11] D. Zühlke. *Nutzergerechte Entwicklung von Mensch-Maschine-Systemen: Useware-Engineering für technische Systeme*. Springer-Verlag, 2011 (zitiert auf S. 15).

Alle URLs wurden zuletzt am 19. 05. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift