Wissenschaftliche Arbeit im Rahmen des Studiums
für das Lehramt an Gymnasien im Fach Informatik

WISSENSTRANSFER IM PAIR PROGRAMMING UND MIT GITHUB COPILOT: EIN VERGLEICH

NIKLAS SCHNEIDER
2574450

Rotenbergstraße 12
66111 Saarbrücken
s8nlschn@stud.uni-saarland.de

28. November 2023

Gutachter:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Jürgen Steimle Human Computer Interaction Lab

Chair of Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

UNIVERSITÄT
DES
SAARLANDES

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Wissenschaftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, sind unter Angabe der Quellen als Entlehnung kenntlich gemacht. Bei Zeichnungen, Skizzen oder Plänen sowie bildlichen und grafischen Darstellungen ist angegeben, wenn sie nach eigenen Angaben durch andere ausgeführt oder übernommen worden sind. Sollte ich Teile dieser Arbeit bereits für andere Prüfungen eingereicht haben, habe ich dies ebenfalls kenntlich gemacht. Die eingereichte elektronische Version der Arbeit stimmt mit der vorliegenden schriftlichen überein.

Saarbrücken, der 28. November 2023

Niklas Schneider

ABSTRACT

Assistenten, die in die Entwicklungsumgebung von Programmierern integriert sind und automatisch Code generieren können, haben in den letzten Jahren Fuß gefasst. Das Unternehmen MICROSOFT warb zu Beginn seiner Kampagne für GITHUB COPILOT mit dem Slogan ‚Your AI pair programmer' für sein Produkt GITHUB COPILOT. Dies wirft die Frage auf, ob ein menschlicher Programmierer beim Pair Programming wirklich durch solch einen Assistenten ersetzt werden kann.

Diese Arbeit ist einem Teilaspekt dieser Frage gewidmet, ob es einen Unterschied im Auftreten von Wissenstransfer zwischen Mensch mit GITHUB COPILOT und Pair Programming gibt. Dazu erstellen wir zunächst ein Framework, welches es ermöglicht das Konzept Wissenstransfer mit Hinblick auf vier Aspekte zu untersuchen und GITHUB COPILOT am bereits erprobten Konzept von Pair Programming zu messen. Diese Aspekte sind Häufigkeit, Struktur und Schwierigkeit der Inhalte, die Inhalte an sich und die Qualität von Wissenstransfer.

Dazu wird eine empirische Studie konzipiert, bei deren Durchführung 8 Einzelpersonen und 7 Paare jeweils die gleichen Programmieraufgaben lösen. Die aufgezeichneteten Sitzungen werden transkribiert und durch das Framework analysiert.

Die Daten legen nahe, dass sogenannte Wissenstransfer-Episoden zwischen zwei Menschen länger andauern als bei einem Menschen und GITHUB COPILOT. Zwischen Menschen können lange Diskussionen entstehen, während mit GITHUB COPILOT eher kürzer interagiert wird. Außerdem zeigt sich, dass Menschen, die mit GITHUB COPILOT arbeiten, weniger vom Thema abgelenkt werden, jedoch gleichzeitig dazu neigen, Code-Vorschläge einfach hinzunehmen ohne sie zu verstehen.

DANKSAGUNG

An dieser Stelle möchte ich mich bei alljenigen bedanken, die mich während der Anfertigung dieser wissenschaftlichen Arbeit unterstützt haben.

Zuerst möchte ich mich bei Kallistos Weis und Christof Tinnes, die meine wissenschaftliche Arbeit betreut haben, für ihre konstruktive Kritik und ihre hilfreichen Anregungen in zahlreichen Diskussionen bedanken. Außerdem danke ich Marvin Weyrich, der durch seine Expertise auch wertvolle Anregungen für die Studie gegeben hat.

Ebenfalls möchte ich mich bei Herrn Prof. Sven Apel und Herrn Prof. Jürgen Steimle für die Möglichkeit über dieses spannende Thema zu schreiben und die Begutachtung dieser Arbeit bedanken.

Ein besonderer Dank gilt den 22 Kommilitonen, die sich die Zeit genommen haben, um an meiner Studie teilzunehmen. Ohne sie hätte diese Arbeit nicht entstehen können.

Außerdem gilt mein Dank Annalena Zenner und Nils Alznauer für das Korrekturlesen meiner Arbeit.

Abschließend möchte ich mich bei meiner Familie, bei meinen Freunden und bei meiner Freundin Annalena bedanken, die mich während des Studiums immer unterstützt und motiviert haben.

Niklas Schneider

Saarbrücken, der 28. November 2023

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

1 EINLEITUNG

Pair Programming werden in vielen Studien allerlei Vorteile attestiert [1, 5, 11], jedoch ist der Widerstand die Technik in Unternehmen einzusetzen immer noch hoch. Sie wird auch in Großunternehmen wie zum Beispiel MICROSOFT eingesetzt [5, 30], aber Faktoren wie erhöhte Personalkosten oder Unstimmigkeiten zwischen den Entwicklern führen dazu, dass sich Unternehmen gegen Pair Programming entscheiden und die Programmierer alleine arbeiten [9].

Jüngste Ansätze im Bereich des maschinellen Lernens, insbesondere im Bereich der Sprachmodelle, ermöglichen mittlerweile eine automatischen Codegenerierung. So entstanden Modelle wie CODEX[1] [7] und, darauf aufbauend, GITHUB COPILOT[2]. Dabei kann GITHUB COPILOT direkt in Entwicklungsumgebungen (IDEs) integriert werden, was eine schnelle und einfache Nutzung durch den Entwickler ermöglicht.

Dies wirft die Frage auf, ob und wie solche Modelle den Menschen bei der Programmierung ersetzen können. Sicherlich sind GITHUB COPILOT und andere verwandte Modelle (noch) nicht in der Lage, selbständig zu handeln [7]. Sie brauchen immer noch einen Menschen, der ihnen eine Aufforderung gibt und dann entscheidet, ob sie die Vorschläge des Modells annehmen oder ablehnen. Aber reicht das Potenzial der Modelle bereits aus, um den zweiten Menschen in einem Pair Programming Setting zu ersetzen? Schon andere Arbeiten haben sich zum Ziel gesetzt, herauszufinden, ob GITHUB COPILOT in verschiedenen Bereichen an die Fähigkeiten von Menschen anknüfen oder diese sogar übertreffen kann [2, 8, 23]. In dieser Arbeit wird im Speziellen der Aspekt des Wissenstransfers betrachtet, da dieser ein elementarer Bestandteil des Pair Programming ist [16, 23, 31, 32]. Zu diesem Zweck wird eine empirische Studie durchgeführt, um den Wissenstransfer zwischen traditionellem Pair Programming und GITHUB COPILOT-gestützter Programmierung zu vergleichen.

Mit der Studie soll untersucht werden, ob bei der Benutzung von GITHUB COPILOT im Vergleich zum Mensch-Mensch-Setting ähnliche Effekte bezüglich des Wissenstransfer s sichtbar sind. Können Programmierer, die GITHUB COPILOT verwenden, ebenso von ihrem ‚Programmierpartner' lernen, wie jene, die zu zweit zusammensitzen und miteinander reden und diskutieren können?

Die wesentlichen Beiträge der Arbeit umfassen ein Framework zur Formalisierung von Wissenstransfer, ein Tool, um das Framework auf Daten effizient anzuwenden und das Design und die Durchführung der empirischen Studie, in deren Zuge das Framework angewendet wird.

Das Framework formalisiert das Konzept von Wissenstransfer basierend auf früheren Arbeiten (von Zieris und Prechelt [31] und Kuttal et al. [16]) macht und Wissenstransfer messbar, sodass es in der konzipierten Studie zur Anwendung gebracht werden kann. Diese Arbeit ist nach bestem Wissen die erste, die einen Vergleich des Wissenstransfers

1 https://openai.com/blog/openai-codex
2 https://github.com/features/copilot

zwischen einem Mensch-GITHUB COPILOT-Setting und einem Mensch-Mensch-Setting anstrebt. Somit ist das Studiendesign eine neue Herangehensweise, die in dieser Arbeit gestellten Fragen zu beantworten. Des Weiteren wurde ein Werkzeug erstellt, mit dem eine Auswertung der einzelnen Sitzungen übersichtlich und schnell mit dem gegebenen Framework durchzuführen ist, sodass es für zukünftige Arbeiten einfacher ist, die Studie zu wiederholen. Außerdem kann das Tool weiterführend angepasst werden, sodass es im Allgemeinen fähig ist, beliebige Daten in Transkripten zu annotieren.

Die Schlussfolgerungen am Ende der Arbeit geben einen Einblick in typische Arbeitsweisen und wie diese sich zwischen den zwei Szenarien unterscheiden. Dadurch bietet die Arbeit auch wertvolle Ergebnisse, die im Hinblick auf die rapide Entwicklung von Programmierwerkzeugen wie GITHUB COPILOT auch genutzt werden können, um zu vergleichen, ob sich dahingehend eine Veränderung oder Verbesserung einstellt.

Als erstes wird das Konzept von Wissenstransfer beleuchtet. Dazu wird zunächst in Kapitel 2 der notwendige theoretische Hintergrund erarbeitet, wonach in Kapitel 3 ein Überblick über die aktuelle Forschung zum Thema GITHUB COPILOT folgt. Anschließend werden in Kapitel 4 die Forschungsfragen gestellt und erläutert sowie der Aufbau und die Durchführung der empirischen Studie beschrieben. Kapitel 5 enthält die Auswertung der gesammelten Daten und die Beantwortung der Forschungsfragen. Zuletzt werden in Kapitel 6 die Ergebnisse diskutiert und Schlussfolgerungen für zukünftige Arbeiten in diesem Feld gezogen.

2 HINTERGRUND

In diesem Kapitel werden wichtige Begriffe definiert und Konzepte erklärt, die im Rest dieser Arbeit von Bedeutung sein werden.

2.1 MENSCHEN

Der erste Begriff, der näher erläutert wird, ist das Wort *Mensch*. In dieser Arbeit und insbesondere in der konzipierten Studie wird zwischen GITHUB COPILOT und Menschen unterscheiden, wenn es darum geht, wer den Code produziert und wie dieser Prozess abläuft. Das Wort Mensch allein spezifiziert jedoch nicht genau, wie der Arbeitsablauf aussehen soll. Dies umfasst zum Beispiel Faktoren wie die Erlaubnis, bestimmte Hilfsmittel während der Programmierung zu verwenden oder welche Art von Einschränkungen dem Menschen gegeben wurden.

Deshalb folgt eine Definition dessen, was im Kontext dieser Arbeit unter Menschen verstanden wird: Ein Mensch wird als Programmierer betrachtet, dem eine Aufgabe gestellt wurde, die es zu lösen gilt; dabei handelt es sich nicht unbedingt um eine einzelne, unabhängige Aufgabe, sondern um eine, die in einen größeren Kontext eines Softwareprojekts eingebettet ist, damit in der Studie realitätsnahe Szenarien abgebildet werden können. Natürlich können Menschen verschiedene Hilfsmittel verwenden, wenn sie vor einem Problem stehen. Sie können andere Menschen konsultieren, Dokumentationen lesen oder im Internet nach Lösungen suchen. Solche Hilfsmittel, einschließlich solcher bei denen ein Computer genutzt wird, werden auch berücksichtigt, wenn sich auf die Arbeit eines Menschen bezogen wird. All dies wird relevant, wenn darüber gesprochen wird, wie die Studie konzipiert ist und was die genauen Umstände sind, mit denen die Teilnehmer bei der Durchführung der Studie konfrontiert werden.

Begriffe wie ‚Programmierer' oder ‚Entwickler' werden im weiteren Verlauf der Arbeit synonym zu Mensch verwendet.

2.2 PAIR PROGRAMMING

Pair Programming kann beschrieben werden als das Schreiben von Quellcode mit zwei Personen, die an einer Maschine mit einer Tastatur und einer Maus arbeiten („with two people looking at one machine, with one keyboard and one mouse" [4]). Sie sind in verschiedenen Rollen involviert: einer wird *Driver* und der andere *Navigator* genannt; von Zeit zu Zeit werden die Programmierer ermutigt, die Rollen zu tauschen, um sicherzustellen, dass beide Teilnehmer ständig konzentriert und engagiert sind und beide die Möglichkeit haben, ihre eigenen Ideen einzubringen [1].

Der Driver ist die Person, die die Tastatur und die Maus bedient und als solche den Code schreibt. Der Navigator hingegen hat die Aufgabe, die Arbeit des Drivers aktiv zu beobachten, nach taktischen und strategischen Fehlern zu suchen, über Alternativen

nachzudenken, zu erledigende Dinge aufzuschreiben und Referenzen nachzuschlagen [1]. Besonders wichtig ist der Aspekt, dass beide Partner aktiv an Vorschlägen, der Entwicklung und der Formulierung von Codeteilen beteiligt sind. Es ist hingegen nicht vorgesehen, dass der Navigator dem Driver den Code diktiert, der ihn wiederum einfach aufschreibt, wie es vielleicht durch die alleinige Kontrolle des Drivers über den Computer suggeriert wird. Auch Beck [4] umreißt diesen sehr wichtigen Aspekt: Pair Programming ist ein Dialog zwischen zwei Menschen, die versuchen, gleichzeitig zu programmieren und gemeinsam zu verstehen, wie man besser programmieren kann („Pair Programming is a dialog between two people trying to simultaneously program [...] and understand together how to programm better" [4]).

Wie von Arisholm et al. [1] erörtert, kann Pair Programming auch in früheren Phasen der Softwareentwicklung eingesetzt werden, zum Beispiel in der Entwurfsphase. Diese Arbeit beschränkt sich jedoch auf die Phase der Codeerzeugung.

2.3 GITHUB COPILOT

GITHUB COPILOT wurde im Oktober 2021 als Plugin für mehrere häufig verwendete IDEs (wie unter anderem VISUAL STUDIO, VISUAL STUDIO CODE und verschiedene JETBRAINS-Produkte) veröffentlicht und wird auf seiner Website[1] als ‚Your AI pair programmer' beworben. Es wurde von Chen et al. [7] als eine spezielle Version des GENERATIVE PRE-TRAINED TRANSFORMER (GPT)-Modells CODEX[2] vorgestellt, das von OPENAI entwickelt wurde. CODEX wurde mit öffentlich verfügbarem Code von GITHUB speziell weitertrainiert (mit Fine-Tuning des Modells) und ist daher in der Lage, Code in den meisten gängigen Programmiersprachen zu erzeugen. Es bietet eine Vervollständigung mittels automatischer Codegenerierung, die durch die Texteingabe des Anwenders, also des Menschen, ausgelöst wird. Diese Eingabe kann eine Codezeile (zum Beispiel die Definition einer Funktion oder eine bedingte Anweisung) oder aber auch ein Codekommentar sein, der beschreibt, welche Funktion von GITHUB COPILOT implementiert werden soll. Je nach Prompt wird das begonnene Konstrukt (Funktion, Schleife, und so weiter) vervollständigt oder von Grund auf implementiert.

Dabei generiert GITHUB COPILOT zunächst einen Vorschlag, der direkt an der aktuellen Cursorposition kursiv angezeigt wird und als Vorschau dient. Der Benutzer kann ihn schnell durch Drücken der Tab-Taste annehmen. Der Vorschlag kann auch einfach mit der Esc-Taste verworfen werden. Zusätzlich generiert GITHUB COPILOT neun weitere Vorschläge, die der Benutzer in einer separaten Liste einsehen kann. Diese kann je nach IDE durch verschiedene Buttons oder Tastenkombinationen aufgerufen werden. In JETBRAINS PYCHARM, welches auch in der später beschriebenen Studie Anwendung finden wird, gibt es (in der verwendeten Version) beispielsweise am rechten Bildschirmrand einen Knopf mit dem Symbol von GITHUB COPILOT, welcher die Liste öffnet.

Während dies nur die Einbettung in IDEs beschreibt, die derzeit sehr verbreitet sind und in der Zukunft auch andere Verwendungsarten auftreten können, soll GITHUB COPILOT in genau dieser Umgebung evaluiert werden, da es mit dem ausdrücklichen Ziel entwickelt wurde, einen Menschen beim Pair Programming zu ersetzen.

1 https://github.com/features/copilot
2 https://openai.com/blog/openai-codex

Natürlich muss, ähnlich wie beim Menschen, definiert werden, was es für GITHUB COPILOT bedeutet, Code zu erstellen, das heißt, wie er im Kontext der Arbeit und der Studie verwendet wird, da er ein integraler Bestandteil der Studie ist. Wie später in Kapitel 3 dargelegt, wird GITHUB COPILOT zumindest in anderen empirischen Studien oft verwendet, indem ihm ein Prompt in Form einer textuellen Beschreibung oder einen Funktionsnamen vorgegeben wird und er ausschließlich anhand der Ergebnisse ausgewertet wird, die die generierte Funktion liefert, wobei das Maß meist die Korrektheit ist, die durch die Durchführung von Tests ermittelt wird. Das Ziel ist es jedoch, die Fähigkeiten von GITHUB COPILOT als Ganzes zu erforschen und insbesondere, wie Menschen mit ihm interagieren, und als solches kann nicht nur den Code betrachtet werden, den er ohne weitere Interaktion mit einem menschlichen Entwickler erzeugt.

2.4 WISSENSTRANSFER

Diese Arbeit beschäftigt sich mit dem Konzept von Wissenstransfer, im speziellen mit der Übertragung von Wissen zwischen zwei Menschen. Wegen dieser zentralen Bedeutung sind dem Begriff eine Reihe von Abschnitten gewidmet, die Schritt für Schritt verschiedene Aspekte beleuchten. Zuerst werden die verschiedenen Bedeutungen des Begriffs Wissenstransfer allgemein in verschiedenen wissenschaftlichen Feldern erklärt. Anschließend folgt eine Eingrenzung auf den für die die Studie relevanten Bereich des Software Engineerings, wo Wissenstransfer konkret inhaltlich eingegrenzt wird. Dabei werden zuerst zwei verschiedene Sichtweisen aus der Literatur betrachtet und darauf aufbauend ein Framework erstellt, welches als Grundlage für diese Arbeit und die durchgeführte Studie dienen soll.

2.4.1 *Begriffserklärung*

Der Begriff Wissenstransfer hat in verschiedenen wissenschaftlichen Bereichen unterschiedliche Bedeutungen, weshalb zunächst verschiedene Definitionen erläutert und diese mit der für diese Arbeit relevanten verglichen werden.

DIDAKTIK In der Didaktikforschung hat der Begriff *Transfer* oder auch *Wissenstransfer* die Bedeutung Wissen zu verallgemeinern und zu reflektieren [14]. Hierbei umfasst der Begriff Tätigkeiten wie zum Beispiel mit vorhandenem Wissen neue, „anspruchsvolle, komplexe oder offen formulierte Probleme“ [15] zu lösen „u. a. mit dem Ziel, zu eigenen Problemformulierungen, Lösungen, Begründungen, Folgerungen, Interpretationen oder Wertungen zu gelangen“ [14]. Diese Definition stammt aus den Bschlüssen der Kultusministerkonferenz [15] und ist somit in die Bildungsstandards für die Hauptfächer, wie sie für den Unterricht an deutschen Schulen bindend sind, fest integriert. Es handelt sich hierbei um den dritten und somit anspruchsvollsten von insgesamt drei Anforderungsbereichen („Anforderungsbereich III: Verallgemeinern und Reflektieren“ [15]), die jeweils die nötigen Kompetenzen zur Aufgabenlösung angeben [15].

Hier bezieht sich der *Transfer* darauf vorhandenes Wissen auf neue Problemstellungen zu übertragen. Diese Sicht findet sich auch in der Kognitionswissenschaft wieder [18, 19, 29].

WIRTSCHAFTSWISSENSCHAFTEN Im Bereich der Wirtschaftswissenschaften ist *Wissenstransfer* Teil des Wissenmanagements. Dies beschreibt eine Managementstrategie, bei der Wissen als „Resource“ [25] angesehen wird, die als so wichtig angesehen wird, dass „ihr im operativen und strategischen Management des Unternehmens besondere Aufmerksamkeit geschenkt werden muß.“ [20, 25]

Der *Transfer* ist dabei als die Weitergabe von Wissens einerseits innerhalb einer Organisation, um Arbeitsabläufe möglichst effizient zu gestalten, andererseits nach außen als Produktionsgut definiert [20, 25].

SOFTWARE ENGINEERING Im Gegensatz zu den vorigen beiden Sichtweisen wird nun eine andere Auffassung etabliert, die für das Verständnis im Kontext von Softwareentwicklung sinnvoll ist. Dazu wird der Transfer von Wissen zwischen zwei Personen betrachtet.

Laut Beck [4] verbessert der Aspekt, dass Pair Programming zum Großteil auf Kommunikation und Konversation basiert, den Prozess der Softwareentwicklung („The conversational nature of pair programming [...] enhances the software development process“ [4]). In den nächsten Abschnitten liegt der Fokus auf diesem Aspekt und wie solche Pair Programming-Gespräche zu verstehen sind.

Wann immer zwei oder mehr Menschen in irgendeiner Weise interagieren (in dieser Arbeit wird eine Sitzung betrachtet, an der zwei Menschen teilnehmen), indem sie Kommunikationskanäle wie gesprochene oder geschriebene Sprache nutzen, gibt es wahrscheinlich einen bestimmten Zweck für diese Interaktion. Das heißt, eine Person könnte der anderen Person eine Frage stellen und dadurch eine bestimmte Information anfordern. Diese Information wird dann von der zweiten Person empfangen, die eine Antwort auf die Frage gibt. Dies ist eine gute Erklärung dafür, was es bedeutet, *Wissen zwischen Personen zu übertragen*. Natürlich gibt es auch andere Zwecke von Interaktionen als die Weitergabe von Informationen, zum Beispiel die Kommunikation über einen Entscheidungsfindungsprozess [32]. Jedoch ist nicht nur für den Wissenstransfer-Aspekt von Interaktionen interessant, weil die Dynamik solcher Prozesse in klassischen Pair Programming-Sitzungen mit zwei Menschen mit einer Situation verlichen werden soll, in der nur ein Mensch, der GITHUB COPILOT benutzt, beteiligt ist. Das dient dazu herauszufinden, ob im zweiten Szenario Wissenstransfer überhaupt beobachtbar ist und, wenn ja, in welche Richtungen und welche Art von Wissen übertragen wird.

Im Folgenden werden Formulierungen wie ‚Informationen‘/‚Wissen‘ ‚teilen‘/‚weitergeben‘/‚transferieren‘ synonym verwendet und beziehen sich alle auf das gleiche Konzept von Wissenstransfer.

Das abstrakte Konstrukt von Wissenstransfer kann in verschiedene Aspekte unterteilt werden. Einerseits gibt es die Struktur von Wissenstransfer, die beschreibt, welche einzelnen Schritte und Kriterien eine Interaktion als Wissenstransfer ausmachen, wie Wissenstransfer formalisiert und kategorisiert werden kann, und welche verschiedenen Arten von Wissenstransfer-Interaktionen unterschieden werden können. Auf der anderen Seite gibt es die Perspektive, dass der Inhalt oder das Thema des weitergegebenen Wissens im Vordergrund steht. Hier liegt besonderes Interesse darauf, wie oft Wissen über bestimmte Themen weitergegeben wird.

Nachfolgend werden diese beiden Aspekte näher beleuchtet und einigen zugrunde liegenden Theorien erklärt, die später bei der Operationalisierung und Auswertung der

empirischen Studie helfen werden. Dazu wird in den folgenden Abschnitten zuerst zusammentragen, welche Formalisierungen es in der Literatur schon gibt und anschließend darauf basierend ein eigenes Modell für Wissenstransfer entwickeln, welches schließlich der durchzuführenden Studie zugrunde gelegt werden.

2.4.2 *Struktureller Aspekt*

Eine detaillierte Theorie über diesen Aspekt von Wissenstransfer in einem Pair Programming Setting wurde zuerst von Zieris und Prechelt [31, 32] aufgestellt, aus deren Arbeiten die notwendigen Grundlagen, die für diese Arbeit und die damit verbundene empirische Untersuchung benötigen werden, entnommen und zusammengefasst werden.

2.4.2.1 *Allgemeine Begriffe*

Bevor die Struktur selbst selbst thematisiert wird, werden einige Begriffe erklärt, die beschreiben, was eine konkrete Instanz eines Wissenstransfers ausmacht und die die Teilnehmer eines Wissenstransfers genauer konkretisieren. Die Definitionen dieser Begriffe gehen von einem traditionellen Pair Programming Setting mit zwei Menschen und ohne GITHUB COPILOT (oder ähnliche Hilfsmittel) aus. Falls zutreffend, werde die Begriffe verwendet, die sich auf Menschen beziehen, in ihrer ursprünglichen Bedeutung auch für GITHUB COPILOT, und falls es einen nennenswerten Unterschied gibt, wird dieser im Folgenden angesprochen. In dieser Arbeit werden diese von Zieris und Prechelt [31] geprägten Begriffe in der gleichen Bedeutung benutzt, wie sie in ihren Arbeiten definiert sind, sofern nicht anders angegeben. Diese Fachbegriffe sind gesondert kursiv hervorgehoben. Wenn es eine treffende deutsche Übersetzung gibt, wird diese verwendet, ansonsten werden die Begriffe in ihrer Originalsprache English verwendet. In den folgenden Abschnitten wird ein kurzer Überblick über die wichtigsten dieser Begriffe gegeben.

Zieris und Prechelt [31] führen drei Kernkonzepte von Wissenstransfer ein: der *need for knowledge* (Wissensbedarf), das *topic* (Thema) und den *target content* (Zielinformation). Der *need for knowledge* ist ein abstrakter Begriff, der weder formalisiert noch weiter erforscht wird. Daher liegt der Fokus auf den beiden letztgenannten Begriffen. Mit dem Konzept *topic* wird versucht abstrakt zu erfassen, worum es bei der Interaktion geht. Diesbezüglich geht Abschnitt 2.4.3 weiter ins Detail. Der *target content* ist definiert als eine Information, die den *need for knowledge* befriedigt [31].

Nach Zieris und Prechelt [31] wird jede einzelne Wissenstransferepisode fast immer von nur einem Paarmitglied vorangetrieben, das sie *propellor* (Treiber) nennen. Sie bezeichnen die Person, die den *need for knowledge* hat, als *customer* (Kunde), während die Person, die etwas über den *target content* weiß, als *supplier* (Anbieter) bezeichnet wird. Wenn der *propellor* zugleich der *customer* ist, wird von einer Interaktion im *pull mode* gesprochen, da die treibende Kraft im Wissenstransfer das Wissen zu sich ‚zieht'. Wenn hingegen der *propellor* der *supplier* ist, wird von einer Interaktion im *push mode* gesprochen, weil der *propellor* in diesem Fall Informationen ‚von sich weg schiebt', und zwar zum *customer* [31]. Wenn keiner der Partner Kenntnisse über den *target content* hat, das heißt, beide *customer* sind, sprechen Zieris und Prechelt [31] von einer *produce mode* Session, in der die Partner gemeinsam oder einzeln an der Schaffung des erforderlichen Wissens arbeiten, deren Ergebnisse als

co-production (Koproduktion) beziehungsweise *pioneering production* (Pionierproduktion) bezeichnet werden [31].

2.4.2.2 *Zeitliche Struktur*

Zieris und Prechelt [31] führen das Konzept der *Episoden* ein, die durch ein einzelnes Thema definiert sind, das in einem konstanten Modus verfolgt wird („defined by a single Topic pursued in a constant [m]ode" [31]). Wenn während der aktuellen Episode ein anderes Thema auftaucht und geklärt werden muss, wird eine neue Episode auf die aktuelle Episode aufgesetzt und ist somit die neue *aktuelle Episode*. Sobald diese *aktuelle Episode* beendet ist (auf Arten, auf die später eingegangen wird), wird die alte *Episode* fortgesetzt. Eine *Episode* kann nur dann enden, wenn der *propellor* den Zielinhalt als vermittelt betrachtet, die Übertragung als unnötig erkennt, den Versuch aufgibt oder das Ziel aus den Augen verliert [31].

2.4.2.3 *Äußerungseigenschaften*

Zusätzlich zu dieser zeitlichen Struktur schlagen Zieris und Prechelt [31] eine Reihe an Eigenschaften vor, die sie *Äußerungseigenschaften* nennen. Sie klassifizieren die *Äußerungen* einer Person. Hierbei ist eine *Äußerung* definiert als ein Satz, der einen zusammenhängenden Gedanken ausdrückt. Beim Sprechen, insbesondere im Monolog (darauf wird in Kapitel 4 genauer eingegangen) bilden Menschen häufig keine ganzen Sätze oder unterbrechen sich selbst, um mit einem neuen Gedanken fortzufahren. In solchen Fällen wird auch eine neue *Äußerung* einleitet.

Im Folgenden werden die wichtigsten *Äußerungseigenschaften* erläutert.

Die erste *Äußerungseigenschaft* ist der sogenannte *information type* (Informationstyp). Er beschreibt das *topic* einer *Äußerung*, welches nicht mit dem *topic* einer *Episode* verwechselt werden darf. Obwohl Zieris und Prechelt [31] mehrere Beispiele für *information types* angeben, existiert keine feste Menge, die alle möglichen Werte für dieses Attribut definiert. Dieses Problem wird in Abschnitt 4.1.1 disktutiert.

Als nächstes gibt es das boolsche Attribut *scope change* (Änderung des Themenbereichs). Es gibt an, ob der Sprecher mit der *Äußerung* beabsichtigt, den Gegenstand der *aktuellen Episode* zu verändern. Das kann zum Beispiel ein Indikator dafür sein, dass eine neue *Episode* nötig ist, um ein neues Thema zu besprechen [31].

Ein weiteres Attribut ist das *medium* (Medium) einer *Äußerung*. Die meisten *Äußerungen* werden wahrscheinlich durch reine Verbalisierung (*verbalisation*) gemacht, aber Zieris und Prechelt [31] schlagen auch zwei andere Medien vor: Personen können ankündigen, dass sie Code schreiben und dann fortfahren, ohne verbal zu erklären, was sie tun. In diesem Fall sprechen sie von *typing*. Außerdem können beide zuvor genannten Medien kombiniert werden, das heißt, wenn eine Person etwas erklärt oder fragt, während sie Quellcode schreibt, um ihren Standpunkt zu deutlich zu machen, wird dies *verbalisation plus demonstration* genannt. Dieses Attribut verliert seinen Sinn, wenn es nur einen Menschen gibt, der GITHUB COPILOT benutzt, denn diese Personen müssen in der Studie laut sprechen (damit die *Äußerungen* ausgewertet werden können). Das macht die Klassifizierung schwierig, da die Menschen in einem realen Szenario dazu neigen, nicht mit dem Computer zu sprechen, und daher in diesem Fall die *verbalisation* eher einen Gedanken widerspiegelt, der nicht

Teil einer Interaktion ist. Deshalb wird diese Eigenschaft in der vorliegenden Studie nicht berücksichtigt, sie kann aber in zukünftigen Arbeiten für interessante Erkenntnisse sorgen.

Das boolsche Attribut *termination attempt* (Beendigungsversuch) soll bestimmen, ob eine *Äußerung* darauf abzielt eine *Episode* zu beenden [31]. Diese Eigenschaft ist für diese Arbeit nicht interessant, denn, wenn man die zwei verschiedenen Szenarien betrachtet, die untersucht werden sollen, stellt man fest, dass im Mensch-GITHUB COPILOT-Setting diese Versuche immer vom Menschen ausgehen werden. Daher bietet der Vergleich hier keinen Mehrwert.

Eine weitere boolsche Eigenschaft ist die *hasted reply* (überhastete Antwort). Sie signalisiert, dass die entsprechende *Äußerung* die vorhergehende unterbricht, das heißt, dass eine Erklärung beginnt, bevor die Frage zu Ende gestellt worden ist oder die nächste Frage gestellt wird, bevor die vorherige Erklärung beendet wurde („an explanation starts before the respective question was complete or the next question starts before the previous explanation was complete“ [31]). Auch diese Eigenschaft kann als möglicher Auslöser für geschachtelte *Episoden* oder beendete *Episoden* gesehen werden, da auf überhastete Antwort eine Nachfrage oder das (vorzeitige) Beenden einer Konversation folgt.

Die boolsche Eigenschaft *redundantizes* zeigt an, ob die *Äußerung* Informationen wiederholt, die schon einmal transferiert worden sind. Das kann zum Beispiel passieren, wenn Personen eine Frage erneut stellen oder Fakten nennen, die bereits bekannt sind.

Des Weiteren ist es interessant zu betrachten, ob eine *Äußerung* durch Selbstvertrauen oder Unsicherheit geprägt ausgesprochen wird, da dies widerspiegelt, ob die Person die Information verstanden hat und ihrem Gesprächspartner folgen konnte, wobei letzteres natürlich nur im Mensch-Mensch-Setting funktioniert. Deshalb führen Zieris und Prechelt [31] das boolsche Attrribut *uncertain* (unsicher) ein, welches genau diesen Aspekt fassen soll.

Zuletzt gibt es noch die Eigenschaft *explanation trigger type* (Auslöser von Erklärungen). Nach Zieris und Prechelt [31] passiert Wissenstransfer schrittweise, sodass meistens der *supplier* versucht, den *customer* durch verschiedene Themen zu führen, bis sie beim *target content* ankommen. Dies wird als die schrittweise Einschränkung eines Themas beschrieben („the successive narrowing-down of a [t]opic“ [31]). In ihren Studien fanden sie mehrere dieser Auslöser, die jeweils entweder einen *scope change*, eine neue *Episode* oder einen *termination attempt* verursachen. Nachfolgend werden diese *explanation trigger types* aufgelistet und kurz erklärt.

- *Finding* beschreibt das Identifizieren eines Themas, das Klärungsbedarf hat. Dies erfolgt zum Beispiel durch das laute Vorlesen von (Variablen-) Bezeichnungen, Paraphrasieren von Code-Teilen oder das Signalisieren von Verwirrung („reading identifiers aloud, paraphrasing a snippet of code, or signaling and somewhat locating confusion“ [31]).

- *Direct Question* bezieht sich auf *Äußerungen*, die eine offene Frage stellen, welche eine komplexe Antwort benötigt und nicht einfach mit ja oder nein zu beantworten ist.

- *Stating Known Facts* beschreibt *Äußerungen*, bei denen die Person Fakten oder Informationen wiederholt, die alle Beteiligten schon kennen. Dieser Auslöser korrespondiert mit dem *redundantizes* Attribut.

- *Simple Step* beschreibt *Äußerungen*, bei denen eine Person versucht, den Fokus der Konversation auf einen bestimmten Punkt zu lenken. Vorausgehen kann hier oft eine *Äußerung* mit dem Auslöser *Stating Known Facts*, da nun eine darauf aufbauende Folgeerklärung gegeben wird.
- Eine *proposition* ist eine *Äußerung*, die die Erwartung äußert, dass der Partner sie entweder aktzeptiert oder ablehnt („with the expectation that the partner will either accept or refuse it“ [31]). Das trifft auch auf implizite (ja-nein-) Fragen zu.

Aufgrund des hohen zusätzlichen Aufwands bei der Auswertung werden die *explanation trigger types* in der Studie nicht berücksichtigt werden. Sie sind jedoch spannend für zukünftige Arbeiten, um besser zu verstehen, wie man automatische Systeme anpassen kann, um gezielt Wissenstransfer zu fördern.

2.4.2.4 *Ende und Ausgang von Episoden*

Abschließend wird disktuiert, auf welche Arten *Episoden* beendet werden können. Hier nennen Zieris und Prechelt [31] vier Möglichkeiten für den sogenannten *finish type*:

- *Transferred* beschreibt *Episoden*, bei denen am Ende der *target content* erfolgreich vermittelt und verstanden wurde. Bei einer Epidose im *produce mode* bezeichnet dies den Umstand, dass beide Partner sich über die erarbeiteten Informationen, die *co-production* einig sind und sie als Fakt akzeptieren. Hierbei spielt es keine Rolle, ob der Inhalt der Information sachlich korrekt ist, da in beiden Fällen ein Wissenstransfer stattgefunden hat. Dieser Ausgang ist also als erfolgreich einzustufen.
- *Gave Up* bezieht sich auf *Episoden*, in denen der *customer* signalisiert, dass das *topic* nicht mehr weiterverfolgt werden soll oder muss. Da hier der Wissenstransfer abgebrochen wird, wird dieser Ausgang als Fehlschlag gewertet.
- *Lost Sight* beschreibt *Episoden*, die dadurch enden, dass die Konversation das *topic* aus den Augen verliert und somit kein Wissenstransfer stattfindet, weshalb auch dieses Szenario als Fehlschlag anzusehen ist.
- *Unnecessary* werden *Episoden* genannt, bei denen einer der Partner das *topic* für irrelevant für den weiteren Verlauf der Zusammenarbeit erklärt und es deshalb nicht mehr weiterverfolgt wird. Ähnlich wie bei *gave up* ist dieser Ausgang als Fehlschlag zu werten.

2.4.3 *Inhaltlicher Aspekt*

Kuttal et al. [16] haben in ihren Arbeiten über einen computergesteurten Pair Programmer (Agent genannt) einen anderen Ansatz zur Quantifizierung von Wissenstransfer vorgeschlagen. Sie waren vielmehr interessiert an den Themen und Problemen, über die die Programmierpartner und die Menschen, die ihren Agenten benutzten, kommunizieren. Abbildung 2.1 zeigt die sechs Kategorien, die sie bezüglich des Inhalts der Konversationen unterscheiden. Dazu sind exemplarisch Ausschnitte aus Protokollen ihrer Studie gegeben, die helfen, die Unterschiede zwischen den Kategorien zu verdeutlichen.

Table 7: Knowledge Transfer Definitions and Examples

Knowledge Transfer	Definitions	Examples (from our study)
Tool	Knowledge about the IDE or how to use the tool.	*"there's a key bind for going back a tab"*
Program	Knowledge about the programming language itself or it's syntax.	*"I could also just aggregate the chars. I think they work like strings in that way."*
Bug	An error in the code.	*"I think we are missing code in this function."*
Code	The code itself; ie. what they are programming.	*"So I'm gonna need to use a for loop as well."*
Domain	The task (in this case, tic-tac-toe game)	*"The goal is to get three of...marker...in a row."*
Technique	About the techniques being used (i.e., test driven development, pair programming)	*"So if it's test driven development, I want to start by writing a test, I think."*

Abbildung 2.1: Definition von Wissenstransfer nach Kuttal et al. [16]

Die erste Kategorie namens *tool*, die Information über Software- oder Hardwaretools betrifft, die während der Entwicklung benutzt werden, zum Beispiel Wissen über die IDE (wie bestimmte Tastenkombinationen oder andere Funktionalitäten, die nicht offensichtlich sind).

Die Kategorie *program* bezieht sich auf Informationen über die Programmiersprache zum Beispiel Syntaxregeln oder die Semantik eines bestimmten Befehls.

Die nächsten zwei Klassen, *bug* und *code*, sind eng miteinander verwandt. Beide betreffen Wissen über den Code, den ein oder mehrere Programmierer geschrieben haben oder gerade schreiben. Aber die Kategorie *bug* wird auf Konversationen abgegrenzt, die sich mit dem Finden, Disktutieren und dem Beheben von Fehlern im Code beschäftigen, während *code* eher auf Konversationen abzielt, die sich im Allgemeinen mit dem Code oder Strategien beschäftigt, wie Ideen in Code umgesetzt werden.

Die Klasse *domain* zielt auf Wissen über die gestellt Aufgabe oder das gesetzte Ziel ab. Zuletzt gibt es noch die Kategorie *technique*, welche sich auf Informationen über abstraktere Entwicklungskonzepte bezieht, wie aus dem entsprechenden Beispiel in Abbildung 2.1 deutlich wird.

3 RELEVANTE FORSCHUNG

Das Konzept Pair Programming ist seit Jahrzehnten in der Forschung relevant; deshalb ist der erste Teil dieses Kapitels disemem Thema gewidmet. Der zweite Teil gibt einen kurzen Einblick in aktuelle Forschung zu GITHUB COPILOT, welche zum Teil schon aus vorigen Abschnitten bekannte Konzepten behandelt, andererseits aber auch Ausblicke gibt, wofür diese Technologie in der Zukunft noch genutzt werden kann.

3.1 PAIR PROGRAMMING

Die Meta-Analyse von Dybå et al. [11] untersucht die Effektivität von Pair Programming, indem die Autoren die Ergebnisse von 15 Studien zwischen 1998 und 2007 zusammenfassen, die jeweils zwischen 12 und 295 Teilnehmer (mit einem Median von 24) umfassen. Diese Studien fokussieren sich auf verschiedene Kriterien wie die Arbeitsgeschwindigkeit, Anstrengung und die Qualität, wobei Qualität mit Hilfe von Tests und die Anstrengung mithilfe der insgesamt benötigten Personenstunden gemessen wird. Insgesammt wird resümiert, dass die Zeit, die für eine Aufgabe benötigt wird, kürzer ist und dass die Qualität höher ist, wenn Pair Programming angewendet wird. Jedoch ist die Anstrengung beim Pair Programming höher, das heißt, es ist teurer, da mehr Personenstunden benötigt werden. Außerdem hängt der Erfolg von Pair Programming von dem Niveau der Expertise der beiden Programmierer ab: Je erfahrener die Programmierer und je schwieriger die Aufgabe, desto weniger profitiert man vom Pair Programming Ansatz [11].

Die Studie von Jones und Fleming [13] schlägt konkrete Kriterien vor, bei denen Pair Programming im Vergleich zum Programmieren alleine vorteilhaft ist. Zu den Kriterien gehören die Produktqualität, Aufgabeneffizienz, Selbstwirksamkeit der Programmierer und der Wissenstransfer zwischen ihnen. Insbesondere das Kriterium Wissenstransfer ist für diese Arbeit relevant, da es zur Beantwortung der unten gestellten Forschungsfrage herangezogen wird (siehe Kapitel 2). Die Studie von Jones und Fleming [13] misst diese schwer fassbaren Konzepte, indem sie ihre Teilnehmer Fragebögen ausfüllen lässt. Die Autoren schlagen auch mögliche Gründe vor, warum ein Pair Programming Ansatz die zuvor formulierten Erwartungen nicht oder zumindest nicht sofort erfüllen könnte. Dazu gehören eine Phase des Zusammenwachsens der Paare zu Beginn des Pair Programmings, die allgemeine Zusammensetzung der Paare (ob die Partner miteinander harmonieren), das Engagement, insbesondere das des Navigators, und ob das Team einen Zustand des Flow erreicht, in dem die Partner produktiv zusammenarbeiten.

Kuttal et al. [16] untersuchen einen Ansatz, bei dem ein Programmierpartner durch einen Computer ersetzt wird, aber anders als in dieser Arbeit dient das Tool nicht nur zur Codegenerierung, sondern soll dem menschlichen Programmierer eine ähnliche Erfahrung vermitteln wie ein menschlicher Partner. Sie schlagen einen Agenten vor, der mit einem simulierten Gesicht auf dem Bildschirm dargestellt wird und mit dem menschlichen Programmierer spricht. Im Gegensatz zu GITHUB COPILOT kann der Agent also die Rolle

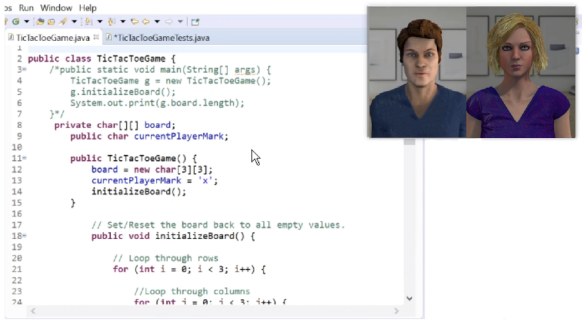


Abbildung 3.1: Das Interface des Agents in der Wizard of Oz Studie („A screenshot of the agent's interface from the Wizard of Oz study" [16])

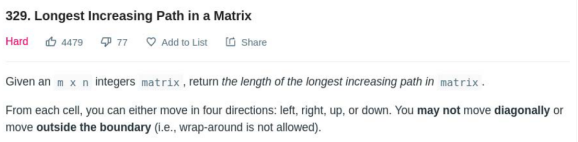
wechseln und vor allem als Navigator fungieren, der den Menschen mit Fragen und Ideen anregt. Die Autoren führen eine Wizard-of-Oz-Studie durch, bei der der Agent von einem anderen Menschen simuliert wird (da ein solch komplexer Agent nicht zur Verfügung steht), der strenge Verhaltensregeln befolgte, damit alle Teilnehmer die gleichen Erfahrungen mit dem Agenten machen konnten. Abbildung 3.1 zeigt die Sicht des Programmierers auf die Entwicklungsumgebung zusammen mit dem simulierten Agenten.

Sie konzentrieren sich auf die Fragen, ob der Einsatz eines solchen Agenten die bereits erwähnten Vorteile des Pair Programming aufrechterhalten kann, wie der Wissenstransfer zwischen Menschen sowie zwischen einem Agenten und einem Menschen funktioniert, und schließlich, ob Menschen einen Agenten als Partner akzeptieren würden. Bei der Auswertung der Studie finden die Autoren keine signifikanten Unterschiede bei den Kriterien, die nachweislich für das Pair Programming von Vorteil sind (wie Produktivität, Qualität und Selbstwirksamkeit). Allerdings erweist sich der Wissenstransfer als unausgewogen, da der Agent nicht in der Lage war, dem Menschen die Logik zu erklären oder seine eigenen Ideen einzubringen. Dennoch zeigen die Menschen Vertrauen und Akzeptanz als Partner des Agenten, indem sie mit ihm wie mit einem Menschen interagierten, Fragen stellten und Pronomen wie „we" verwendeten, wenn sie über erfolgreiche Teile der Aufgabe sprachen. Das Ziel der Studie geht über den Rahmen dieser Arbeit hinaus, indem gefragt wird, was nötig ist, um nicht nur den Code-Aspekt des Pair Programming durch einen Agenten (das heißt GITHUB COPILOT) zu ersetzen, sondern auch alle Funktionen zu liefern, die den Agenten menschlich erscheinen lassen (zum Beispiel Gespräche, ein animierter Avatar). Sie teilen jedoch das Teilziel, mit dieser Studie wissen zu wollen, ob ein menschlicher Programmierpartner durch einen Computer ersetzt werden kann.

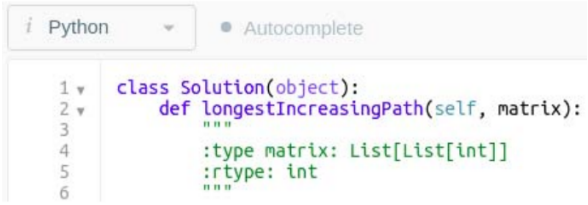
3.2 GITHUB COPILOT

Verschiedene empirische Studien analysieren den Code, den GITHUB COPILOT produziert, anhand unterschiedlicher Kriterien.

Nguyen und Nadi [17] verwenden LEETCODE-Fragen als Prompts in vier verschiedenen Programmiersprachen (JAVA, JAVASCRIPT, PYTHON und C) für GITHUB COPILOT und bewerten die Korrektheit mit Hilfe der Testfälle, die ebenfalls von LEETCODE bereitgestellt wurden. LEETCODE ist eine Website, auf der Programmierübungen zusammen mit einer Online-Entwicklungsumgebung zu deren Lösung und automatisierten Tests zur Überprü-



(a) Beschreibung der Frage



(b) Die Frage in der Entwicklungsumgebung

Time Submitted	Status	Runtime	Memory	Language
11/30/2021 22:06	Runtime Error	N/A	N/A	javascript
11/30/2021 22:06	Runtime Error	N/A	N/A	python
11/30/2021 22:06	Accepted	5 ms	39 MB	java
11/30/2021 22:05	Compile Error	N/A	N/A	c

(c) Submission History der Frage

Abbildung 3.2: Eine LeetCode-Frage namens Longest Increasing Path in a Matrix („An example LeetCode question, named Longest Increasing Path in a Matrix“ [17])

fung der eigenen Lösung angeboten werden. Die Autoren verwenden auch Maße wie zyklomatische und kognitive Komplexität, um festzustellen, wie verständlich der generierte Code tatsächlich ist. Ein Auszug aus einer beispielhaften Frage in der entsprechenden Umgebung, die für GITHUB COPILOT geeignet ist, ist in Abbildung 3.2 zu sehen.

Die Autoren finden heraus, dass die Vorschläge in JAVA am wahrscheinlichsten korrekt waren, da 57% der Aufgaben alle Tests bestanden, während GITHUB COPILOT bei der Generierung von JAVASCRIPT Code am schlechtesten abschnitt, da nur 27% der Lösungen korrekt waren [17]. Darüber hinaus stellten sie fest, dass der Code im Allgemeinen gut verständlich ist, wobei beide Komplexitätsmaße bei allen vier Programmiersprachen niedrige Werte aufweisen. Dennoch nennen die Autoren auch Schwächen von GITHUB COPILOT, nämlich die Verwendung von undefinierten Hilfsfunktionen und Code, der vereinfacht werden kann [17]. In dieser Studie wird der Schwerpunkt darauf gelegt, dass GITHUB COPILOT kleine Aufgaben selbständig löst. Im Gegensatz dazu soll diese Arbeit die Leistung von Mensch und GITHUB COPILOT untersuchen, die gemeinsam an einem größeren Projekt arbeiten.

Tang et al. [28] und Drori und Verma [10] untersuchen, wie gut OPENAI'S CODEX (auf dem GITHUB COPILOT basiert) in der Lage ist, mathematische Probleme anhand eines Prompts aus einem Mathematiklehrbuch zu lösen. Diese Aufgaben bestehen entweder darin, ein Ergebnis zu berechnen, ein Diagramm zu erstellen oder beides mit Hilfe von PYTHON-Code. Die Autoren haben diese Prompts in den Fällen angepasst, in denen CODEX keine

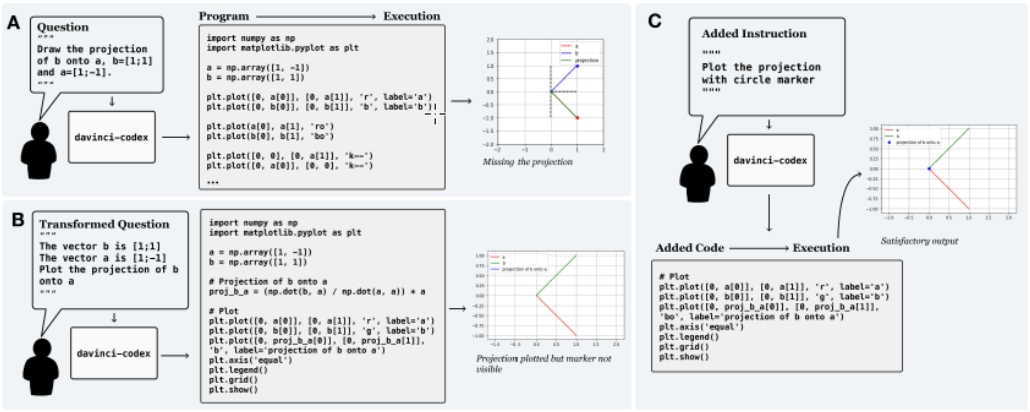


Abbildung 3.3: Pipeline von Drori und Verma [10], um eine Aufgabe auszuführen

zufriedenstellenden Ergebnisse liefert (dazu gehören *gutaussehende* („visually pleasing" [10]) Plots und die korrekte Lösung einer Berechnung). Ihr Arbeitsablauf ist in Abbildung 3.3 ausführlicher dargestellt.

Es wird in beiden Studien von Tang et al. [28] und Drori und Verma [10] von hoher bis perfekter Genauigkeit berichtet; allerdings verlassen sie sich auf Korrektheitsmaße, die nicht objektiv sind. Außerdem wird CODEX wie in der vorangegangenen Arbeit allein bewertet, im Gegensatz zu dieser Arbeit, die darauf abzielt, seine Fähigkeiten als Paarprogrammierer zu untersuchen.

In einer von Imai [12] durchgeführten Studie sollen die Teilnehmer ein Minesweeper-Spiel in PYTHON implementieren. Es gibt drei Bedingungen (in ihrer Reihenfolge randomisiert): die Rolle des Drivers oder des Navigators mit einem anderen menschlichen Partner im Pair Programming oder mit GITHUB COPILOT als Paarprogrammierer. Ziel ist es, die Produktivität und die Codequalität zu messen. Ersteres wird durch die Anzahl der pro Zeiteinheit hinzugefügten Zeilen erreicht, letzteres durch die Anzahl der gelöschten Zeilen, wobei weniger gelöschte Zeilen als höhere Codequalität interpretiert werden [12]. Die Ergebnisse zeigen, dass die produktivste Bedingung diejenige ist, in der GITHUB COPILOT verwendet wird, aber auch diejenige mit der niedrigsten Qualität, da mehr Zeilen gelöscht werden als in den Fällen, in denen zwei Menschen beteiligt waren. Die Arbeit beantwortet nicht die Frage, warum die vorgestellten Maße zur Bewertung der Produktivität geeignet sind.

Barke, James und Polikarpova [3] untersuchen, wie Programmierer mit GITHUB COPILOT interagieren. Durch eine Befragung der Teilnehmer ihrer Studie finden sie zwei allgemeine Interaktionsmodi, wobei der erste der Beschleunigungsmodus ist, bei dem der Programmierer den Code, den er zu schreiben plant, bereits im Kopf hat und GITHUB COPILOT lediglich als Autovervollständigungswerkzeug verwendet wird. Der Code-Vorschlag, der in solchen Fällen meist die Vervollständigung einer einzelnen Zeile betrifft, wird nur kurz geprüft und dann vom Programmierer akzeptiert, ohne dass er seinen Schreibfluss unterbrechen muss [3]. Der andere Modus wird als Explorationsmodus bezeichnet. Hier ist sich der Programmierer in der Regel nicht sicher, wie ein bestimmtes Problem zu lösen ist, und verwendet GITHUB COPILOT, um mögliche Ansätze zu erkunden, wobei hier auch mehr Vorschläge als nur der eine, der während der Eingabe im Code erscheint, berücksichtigt

werden. In diesem Modus befindet sich der Programmierer nicht in einem Flow und die Interaktion mit GITHUB COPILOT ist im Vergleich zum Beschleunigungsmodus [3] eher langsam. Diese Art der Interaktion kann nicht mit der Interaktion mit einem menschlichen Partner verglichen werden, da Menschen nicht in der Lage sind, den Code in einer Art und Weise aufzuschreiben, wie es GITHUB COPILOT im Explorationsmodus tut, aber nichtsdestotrotz präsentieren die Autoren einen Weg, um zu verstehen, wie Menschen durch GITHUB COPILOT beeinflusst werden.

4 EXPERIMENT

In diesem Kapitel wird zuerst basierend auf den im vorigen Kapitel erläuterten Modellen ein eigenes Framework entwickelt, welches im Experiment angewandt werden soll. Danach werden die Forschungsfragen dieser Arbeit genannt und erläutert und und anschließend erörtert, mit welchen Mitteln die Fragen beantwortet werden können und wie die dafür relevanten Faktoren gemessen und ausgewertet werden können. Außerdem wird der Aufbau und die Durchführung der Studie beschrieben.

4.1 WISSENSTRANSFER FRAMEWORK

Ein wichtiger Beitrag dieser Arbeit stellt das basierend auf den Arbeiten von Zieris und Prechelt [31] sowie von Kuttal et al. [16] entwickelte Framework für Wissenstransfer dar, welches für die in den folgenden Abschnitten beschriebene Studie zur Auswertung benutzt wird. Des Weiteren wird in Abschnitt 5.1.2 ein entwickeltes Programm beschrieben, mit dem das Framework konkret auf Transkripte von Programmiersessions angewandt werden kann.

Um aus den beiden verschiedenen Perspektiven ein eigenes, logisch zusammenhängendes Framework zusammenzustellen, werden die beiden Aspekte so kombiniert, dass sie sich jeweils sinnvoll ergänzen. Außerdem werden eigene Aspekte hinzugefügt, die in keinem der beiden erläuterten Sichtweisen repräsentiert sind.

4.1.1 *Kombination von strukturellem und inhaltlichem Aspekt*

Zunächst wird untersucht, wie sich struktureller und inhaltlicher Aspekt ergänzen lassen.

Es wurde bereits im vorigen Kapitel erwähnt, dass beim strukturellen Aspekt das *topic* größtenteils vom *information type* der einzelnen *Äußerungen* bestimmt wird; Zieris und Prechelt [31] geben jedoch keine konkrete Liste mit möglichen Klassen für die beiden Konzepte vor. Dort fehlt also eine Formalisierung für diesen wichtigen Teil von Wissenstransfer. An dieser Stelle setzt der zuvor disktutierte inhaltliche Ansatz von Kuttal et al. [16] an, der genau diese Lücke zu füllen versucht. Zwar kann die Liste unmöglich vollständig sein, aber immerhin gibt sie einen abstrakten Überblick über mögliche Werte für das *information type* Attribut. Dazu bauen wird ein eigenes Modell für das Konzept von Wissenstransfer entwickelt, indem die beiden Ansätze sinnvoll kombiniert werden.

Von nun an wird bei *Äußerungen* das Attribut *information type* umbenannt zu *topic*, da sowohl bei *Äußerungen* als auch *Episoden* dieselben Kategorien verwendet werden sollen. Das *topic* einer *Episode* wird dann durch das am häufigsten vorkommende *topic* der umfassten *Äußerungen* bestimmt. Dies passt zum Modell von Kuttal et al. [16], da sich die von ihnen vorgeschlagenen Kategorien auch auf einzelne Sätze beziehen, sodass diese Zuordnung auf *Äußerungen* übernommen werden kann, was die zwei Modelle miteinander kompatibel macht.

Tauchen in einer *Episode* mehrere *Äußerungen* zu anderen *topics* auf, ist das ein Anzeichen dafür, dass hier eine neue, geschachtelte *Episode* beginnt, denn nach der Definition von Zieris und Prechelt [31] kann eine *Episode* nur ein bestimmtes *topic* behandeln. Jede folgende Interaktion, die ein anderes *topic* betrifft, muss demnach als neue *Episode* betrachtet werden. Das folgende Beispiel zeigt jedoch auf, dass diese Herangehensweise noch problembehaftet ist.

> **Beispiel 4.1:**
> In einem Mensch-Mensch-Setting geht es in einer Diskussion gerade um einem Fehler, den eine Person im Code gefunden hat, also wäre das zugewiesene *topic* die Klasse *bug*. Aber dann entdecken die Entwickler beim Beheben des Fehlers einen weiteren Fehler und der Fokus ändert sich zu einem inhaltlich anderen Problem, während das zugewiesene *topic* immer noch als *bug* zu klassifizieren ist.

Das zeigt, dass die abstrakte Klassifizierung der *topics* allein nicht ausreicht und auch beim Annotieren von solchen Daten der konkrete Gegenstand der Konversation berücksichtigt werden muss.

4.1.2 *Zusätzliche Ergänzungen zum Framework*

In Abschnitt 2.4.2.4 wurden vier verschiedene Arten genannt, auf die *Episoden* enden können. Was Zieris und Prechelt [31] jedoch nicht berücksichtigen, in der Praxis jedoch häufig auftritt, ist, dass manche Programmierer Fakten akzeptieren ohne sie nachzuvollziehen oder in der Hoffnung, dass die Information wahr oder hilfreich ist. Um diesem Phänomen Rechnung zu tragen, wird die Liste mit *finish type*s um einen weiteren Punkt ergänzt:

- *Trust* bezieht sich auf *Episoden*, an deren Ende mindestens einer der Partner die Hoffnung äußert, dass die Information sinnvoll ist, sie aber nicht weiter nachvollzogen oder verifiziert wird. Hier hat zwar ein Wissenstransfer stattgefunden, aber die Informationen wurden vom *customer* nicht verstanden oder akzeptiert, weshalb dieser Ausgang neutral gewertet wird.

4.1.3 *Zusammenfassung des Frameworks*

Tabelle 4.1 zeigt eine kurze Zusammenfassung des finalen Evaluationsschemas für den Wissenstransfer, das in den vorhergegangenen Abschnitten erarbeitet wurde und welches im Verlauf dieser Arbeit auch für die empirische Studie verwenden werden wird.

Des Weiteren wird eine Klassifikation von *Episoden* vorgeschlagen, die darauf abzielt zu bestimmen, ob der Wissenstransfer erfolgreich war. Auf dieser Grundlage kann über die Qualität von Wissenstransfer gesprochen werden. Dazu wurde schon elaboriert, wie eine *Episode* enden kann. Tabelle 4.2 zeigt die Attribute für *Episoden*.

Tabelle 4.1: Evaluationsschema für *Äußerungen*

Äußerungseigenschaft	Mögliche Werte
Topic	*Tool*, *Program*, *Bug*, *Code*, *Domain*, *Technique*
Scope Change	TRUE, FALSE
Hasted Reply	TRUE, FALSE
Repetitive	TRUE, FALSE
Uncertain	TRUE, FALSE

Tabelle 4.2: Evaluationsschema für *Episoden*

Episodeneigenschaft	Mögliche Werte
Topic	*Tool*, *Program*, *Bug*, *Code*, *Domain*, *Technique*
Finish Type	*Transferred* (success), *Trust* (neutral), *Unnecessary* (fail), *Lost Sight* (fail), *Gave Up* (fail)

4.2 FORSCHUNGSFRAGEN

In dieser Arbeit sind liegt der Fokus hauptsächlich darauf den Unterschied des Wissenstransfers zwischen dem klassischen Pair Programming Setting und einem Mensch-GITHUB COPILOT-Setting zu vergleichen. Deshalb lautet die Forschungsfrage:

> **RQ:** Können Menschen, die mit GITHUB COPILOT programmieren, ähnliche Wissenstransfer-Effekte erfahren wie Menschen, die Pair Programming anwenden?

Diese Frage ist abstrakt formuliert, weshalb vier untergeordnete Forschungsfragen gestellt werden, die verschiedene Aspekte von Wissenstransfer instrumentalisieren und messbar machen.

> **RQ1:** *Wie vergleicht sich die Anzahl an Wissenstransfer-*Episoden *zwischen einem Mensch-Mensch-Setting und einem Mensch-*GITHUB COPILOT*-Setting*?

Wie schon in Abschnitt 2.4.2 dargestellt, trägt nicht jede einzelne *Äußerung* einer Konversation zum Wissenstransfer bei. Für die Frage ist es relevant, ob es einen signifikanten Unterschied in der Anzahl der *Episoden* gibt, weil diese ein Indikator dafür ist, wie viel und welche Teile der Konversation zum Wissenstransfer beitragen.

> **RQ2:** *Wie unterscheidet sich die Tiefe und Länge von* Episoden *zwischen den Szenarien?*

Mit der Tiefe von Wissenstransfer ist gemeint, dass *Episoden* gestapelt werden können, wenn der *customer* Wissen (zum Beispiel in Form einer Erklärung) braucht, welches sich auf ein anderes als das Thema der *aktuellen Episode* bezieht. Voraussetzung für das Stapeln ist dabei, dass das *topic* der vorherigen *Episode* wieder aufgefriffen wird und das transferierte Wissen

im weiteren Verlauf verwendet wird. Zuerst sollen häufig auftretende Muster in Tiefe und Länge der *Episoden* innerhalb der Szenarien gefunden werden. Diese können anschließend zwischen den Szenarien verglichen und nach Ähnlichkeiten und Unterschieden untersucht werden. Es könnten sich zum Beispiel einzelne lange *Episoden* über ein *topic* häufen, die keine weiteren *Episoden* öffnen (was eine *flache Episode* wäre) oder aber viele sehr kurze aufeinanderfolgende *Episoden* auftreten.

Mit diesen zwei Dimensionen, Tiefe und Länge, sollen Aussagen über die Schwierigkeit des *topics* gemacht werden, das diskutiert wird, denn schwierige *topics* brauchen möglicherweise mehr Argumente und Erklärungen, was also mehr *Äußerungen* benötigt als leichte Fragen, die meistens mit eher weniger Sätzen schon zufriedenstellend beantwortet sind.

> **RQ3:** *Gibt es einen Unterschied in der Verteilung der* topics *der* Episoden *zwischen den beiden Szenarien?*

Hier sollen die Häufigkeiten, mit denen die verschiedenen *topics* auftreten, verglichen werden, um herauszufinden, über welches *topic* am meisten disktutiert wird und ob sich dieses in den zwei Szenarien unterscheidet.

> **RQ4:** *Wie unterscheidet sich die Qualität des Wissenstransfers zwischen den zwei Szenarien bezüglich des Erfolgs des Wissenstransfers?*

Zum Schluss werden die Ergebnisse der Wissenstransfer-Prozesse analysiert, das heißt, ob sie erfolgreich abgeschlossen werden konnten oder nicht (bezüglich der Terminologie, die in Abschnitt 4.1.1 eingeführt wurde). Das ist der wichtigste Aspekt, weil Wissenstransfer für den *customer* nur nützlich ist, wenn er in der Lage ist, die Informationen, die an ihn weitergegeben werden, zu verarbeiten und zu verstehen.

4.3 OPERATIONALISIERUNG

Um die Forschungsfragen zu beantworten, muss zunächst erarbeitet werden, wie man die Konzepte, die im Kapitel zum theoretischen Hintergrund (Kapitel 2) erklärt wurden, sinnvoll erfassen kann.

4.3.1 *Konversationen klassifizieren*

Am wichtigsten und strukturgebend sind die *Episoden*, die wiederum aus *Äußerungen* bestehen. Nun muss defniert werden, wie das Transkript eines Dialogs oder Monologs sinnhaft in *Episoden* und *Äußerungen* aufgeteilt werden kann, denn dies stellt den ersten Schritt dar, um die empirische Studie auszuwerten. Dabei ist ein vorsichtiges Vorgehen notwendig, denn nicht alle *Äußerungen* werden zu einem Wissenstransfer beitragen; dies sollte in den Daten zum Ausdruck gebracht werden.

Zudem wuss zwischen Monologen und Dialogen unterschieden werden, da in einem Mensch-Mensch-Setting zwei Menschen eine Konversation haben, wohingegen es im Mensch-GITHUB COPILOT-Fall nur einen Monolg der Person gibt, die ihre Gedanken

laut ausspricht. Deshalb muss es für beide Fälle sinnvolle Definitionen geben, die aber trotzdem äquivalent sein müssen, damit die Evaluation vergleichbar bleibt.

4.3.1.1 *Dialoge*

Das Produkt zweier miteinander sprechender Menschen wird als Dialog bezeichnet; Im Kontext der Studie bezieht sich dies immer speziell das Transkript dieses Dialogs. Dieses besteht aus Sätzen, die jeweils von den zwei Gesprächspartnern stammen.

Eine *Äußerung* kann nur aus ganzen, aufeinanderfolgenden Sätzen bestehen und umfasst mindestens einen Satz. Für jeden solchen Satz wird entschieden, ob er zu der letzten oder ob er zu einer neuen *Äußerung* gehört. Dabei ist entscheidend, ob der Satz gedanklich an den vorherigen Satz anknüpft (das heißt, ob er den selben Inhaltsgegenstand hat) und ob zwischen diesem und dem vorherigen Satz eine längere Gesprächspause oder eine Unterbrechung liegt. Es können also mehrere verschiedene *Äußerungen* einer Person hintereinander vorliegen, sobald aber die jeweils andere Person zu sprechen beginnt, ist dies auch Teil einer neuen *Äußerung*.

4.3.1.2 *Monologe*

Als Monolog wird das Produkt eines Menschen, der mit sich selbst redet, bezeichnet, das heißt, der im Kontext der Studie seine Gedanken immer sofort laut ausspricht. Auch hier bezieht sich die Bezeichnung immer auf das Transkript.

Ähnlich zu Dialogen besteht der Monolog ebenfalls aus Sätzen, die aber alle von demselben Programmierer kommen. Daher fällt die Möglichkeit weg, eine neue *Äußerung* zu beginnen, wenn der Sprecher wechselt. Ansonsten gelten dieselben Kriterien wie beim Dialog. Hier ist insbesondere vermehrt damit zu rechnen, dass Menschen, die ihre Gedanken laut aussprechen, dazu tendieren, sich oft selbst im Satz zu unterbrechen und mit einem anderen Gedanken fortzufahren. Das kommt nach Charters [6] daher, dass ein Sprecher oft längere Zeit braucht, um einen Gedankengang zu vollenden. Im Kopf ist der Gedanke zwar vollumfänglich präsent, beim Aussprechen muss jedoch Wort für Wort entwickelt werden [6]. Somit kann es passieren, dass während ein Gedanke noch verbalisiert wird, ein neuer Gedanke schon entstanden ist, dessen Aussprache die des vorherigen Gedankens unterbricht. Diese neuen Gedanken führen auch im Transkript zu einer neuen *Äußerung*. Außerdem passiert es nach Charters [6] häufig, dass Menschen bei dieser Methode denselben Gedanken hintereinander in leicht verschiedenen Tonlagen oder Formulierungen aussprechen. Solche Wiederholungen werden als eine einzige *Äußerung* aufgefasst, da es sich immer noch um denselben Inhaltsgegenstand handelt.

4.3.2 *Äußerungen und Episoden klassifizieren*

Für jede *Äußerung* muss entschieden werden, ob sie zu einem Wissenstransfer beiträgt. Das ist nicht trivial, da es keine rein objektive Möglichkeit gibt dies zu entscheiden. Deshalb wird wie folgt definiert, dass eine *Äußerung* zu einem Wissenstransfer beiträgt:

Zuerst wird die Klassifizierung aus Tabelle 4.1 an. Eine *Äußerung* kann nur zu einem Wissenstransfer beitragen, wenn

(1) jeder der Eigenschaften ein eindeutiger Wert zugewiesen werden kann,

(2) die vermittelte Information Produkt der Interaktion ist.

Das schließt unter anderem aus, dass *Äußerungen*, die zum Beispiel aus einer Dokumentation Vorgelesenes wiedergeben, als Wissenstransfer gewertet werden. Denn hier ist die Information nicht durch den Transfer von Person zu Person (oder von GITHUB COPILOT zu Person) erfolgt, sondern von einer anderen Quelle. Da der Fokus aber ausdrücklich auf den beiden genannten Arten von Transfer liegt, wird beim Annotieren diese Regel angewendet.

Um die *Episoden*-Struktur zu finden, werden *Äußerungen* gesucht, die einen *need for knowledge* deklarieren (was auf *Äußerungen* zutrifft, denen ein *explanation trigger type* zugewiesen werden kann). Diese können den Beginn einer *Episode* markieren. Diese *Episode* geht bis zu derjenigen *Äußerung*, bei der einer der fünf diskutierten Ausgänge klar zu erkennen ist (zum Beispiel durch eine Bestätigung).

Wenn zwischen Anfang und Ende einer *Episode* ein anderes *topic*, beziehungsweise ein anderer Gegenstand diskutiert wird, sich anschließend jedoch wieder auf das ursprüngliche *topic* fokussiert wird, wird dadurch eine gestapelte *Episode* nötig. Auch solche sind an *Äußerungen* mit einem *explanation trigger type* zu erkennen.

4.4 METHODE

Dieser Abschnitt beschäftigt sich mit dem Aufbau der empirischen Studie, die zur Beantwortung der Forschungsfragen durchgeführt wird.

Es handelt sich hierbei um ein kontrolliertes Experiment mit zwei Gruppen. Beide Gruppen lösen die gleiche Programmieraufgabe. Die Kontrollgruppe besteht aus Paaren, die das jeweils mittels Pair Programming tun (ohne GITHUB COPILOT zu benutzen); die Experimentalgruppe aus einzelnen Teilnehmern, die GITHUB COPILOT verwenden. Bevor die Teilnehmer jedoch programmieren, füllen sie zuerst einen pre-test Fragebogen aus (dieser wird in den folgenden Abschnitten näher beschrieben).

Die Programmiersprache, in der die Aufgabe gestellt ist und gelöst wird, ist PYTHON[1]. PYTHON ist eine bekannte und verbreitete Programmiersprache, die zudem leichten und schnellen Zugriff auf eine Vielzahl an Bibliotheken erlaubt, was sie vielfältig einsetzbar macht. Außerdem ist sie sowohl für die Teilnehmer als auch die Person, die die Studie durchführt und das Beispielprojekt mit der Aufgabe aufsetzt, leicht zu benutzen, weil die Einrichtung und das Arbeiten mit der Entwicklungsumgebung unkompliziert ist. Diese Wahl stellt möglicherweise einen Threat to Validity dar, da einerseits nicht vorausgesetzt werden kann, dass alle Teilnehmer dasselbe Vorwissen über die Sprache mitbringen und andererseits annehmen müssen, dass sich die Leistung sowohl der Teilnehmer als auch von GITHUB COPILOT in verschiedenen Programmiersprachen unterscheidet [17]. Um die Studie jedoch durchführen zu können und damit die Ergebnisse vergleichbar sind, muss eine Programmiersprache festgelegt werden.

Um dies auszugleichen, füllen die Teilnehmer zuerst den Fragebogen aus. Auf diesem sollen sie sich bezüglich ihrer Erfahrung und ihrer Fähigkeit allgemein und speziell in PYTHON zu programmieren auf mehreren Likert-Skalen selbst einschätzen. Diese Fragen sind nach den Erkenntnissen von Siegmund et al. [26] gewählt: Demnach sollen die

1 https://www.python.org/

Teilnehmer ihre Programmiererfahrung zunächst im Verhältnis zu ihren Mitstudierenden einschätzen und als Kontrollfrage auch versuchen ihre Erfahrung absolut einzuschätzen. Dazu sind Skalen von eins (niedrige Erfahrung) bis zehn (hohe Erfahrung) gegeben. Für die zweite Frage schlagen Siegmund et al. [26] zwar nur eine Skala von eins bis fünf vor, aus Gründen der Konsistenz mit dem restlichen Fragebogen wurde sich hier auch für die Zehn-Punkte-Skala entschieden. Außerdem enthält der Fragebogen Fragen zu Alter und Geschlecht, die freiwillig zu beantworten sind. Dies dient dazu mögliche Effekte, die durch eine ungleiche Verteilung in diesen beiden Aspekten im Publikum, aus dem die Teilnehmer stammen, zu erkennen und so eine Bedrohung der externen Validität auszuschließen. Der Fragebogen ist im Anhang (Abbildung A.1) zu finden. Vor dem Fragebogen muss noch eine Einverständniserklärung ausgefüllt werden; diese ist ebenfalls im Anhang hinterlegt (Abbildung A.2).

4.5 DURCHFÜHRUNG

4.5.1 *Ablauf*

Sobald sich die Teilnehmer zur Studie angemeldet haben, werden sie zufällig einer der beiden Gruppen (Kontroll- oder Experimentalgruppe) zugeteilt. An einem festgelegten Termin finden sie sich im Studienraum ein, in dem der entsprechende Arbeitsplatz eingerichtet ist. Dieser besteht aus einem Rechner, auf dem das im folgenden beschriebene Projekt in einer IDE geöffnet ist und alle für die Aufgabe nötigen Pakete und Programme installiert sind. Für Teilnehmende der Experimentalgruppe ist in der IDE ein Konto hinterlegt, das den Zugriff auf GITHUB COPILOT erlaubt und das Plugin ist in der IDE aktiviert; bei der Kontrollgruppe ist das nicht der Fall.

Zuerst füllen die Teilnehmer den Fragebogen aus. Anschließend erhalten sie von der betreuenden Person (in der Regel der Autor) eine kurze verbale Instruktion, die für alle gleich abgelesen wird (mit den jeweiligen Besonderheiten der beiden Gruppen). Sie ist dem Anhang (Abbildung A.4, Abbildung A.5) zu entnehmen. Zu den Rahmenbedingungen gehört ein Zeitlimit von 45 Minuten, damit die Ergebnisse der Gruppen vergleichbar bleiben. Hierbei ist es nicht schlimm, sondern ausdrücklich erwünscht, dass nicht alle Paare und Einzelpersonen den gleichen Fortschritt erzielen, beziehungsweise das Projekt abschließen, weil das auch ein interessantes Merkmal sein kann, in dem sich die Kontroll- und die Experimentalgruppe signifikant voneinander unterscheiden. Dies muss den Teilnehmenden natürlich kommuniziert werden, da sonst Stress durch Zeitdruck aufgebaut werden könnte, der das Ergebnis verfälscht, weshalb dieser wichtige Punkt auch Teil der Instruktion ist. Außerdem sind alle Hilfsmittel (außer künstlicher Intelligenz in der Kontrollgruppe) erlaubt, das heißt, es darf im Internet nach Dokumentationen und ähnlichem gesucht werden.

Den Teilnehmern der Experimentalgruppe wird zusätzlich ein kurzes Einführungsvideo gezeigt, in dem erklärt wird, wie GITHUB COPILOT zu benutzen ist. Die Kontrollgruppe erhält hingegen als Teil der Instruktion eine kurze Erklärung über Pair Programming.

Dann erhalten die Teilnehmer noch die Gelegenheit Fragen zur Instruktion zu stellen, bevor die Bearbeitungszeit beginnt und die betreuende Person keine Äußerungen mehr tätigt. Ab diesem Zeitpunkt wird das Geschehen auf dem Bildschirm und das Gespräch,

beziehungsweise das Laut-Denken aufgezeichnet, bis die Teilnehmer sich dazu entscheiden abzugeben oder die Zeit abgelaufen ist.

4.5.2 *Aufgabe*

4.5.2.1 *Projektbeschreibung*

Wie schon in Kapitel 3 diskutiert wurde, soll im Gegensatz zum Großteil der Literatur, die sich mit GITHUB COPILOT beschäftigt, indem dessen Leistung (gemessen durch Testergebnisse) anhand einzelner kurzer und voneinander unabhängigen Aufgaben ausgewertet wird, ein eher realitätsnahes Szenario erforscht werden. Dazu wird als Kontext der Programmieraufgabe ein zusammenhängendes Projekt konstruiert, in dem bestimmte Features implementiert werden müssen, wobei aber auch schon essentielle Bestandteile vorhanden und funktionstüchtig sind. So kann auch die Interaktion zwischen Mensch und GITHUB COPILOT mit einbezogen werden, denn hierbei gilt es nicht nur ohne Kontext einen Algorithmus zu schreiben (wie zum Beispiel bei Drori und Verma [10]), sondern auch den Kontext zu verstehen, in dem die geforderte Funktionalität eingebunden werden soll. Hierzu müssen die menschlichen Programmierer Programmverständnis aufweisen, um GITHUB COPILOT sinnvoll einsetzen zu können.

Die Teilnehmer erhalten ein kleines Projekt, in dem ein Passwort-Manager implementiert ist. Dieser Passwort-Manager ist als Konsolenanwendung konzipiert und unterstützt das Hinzufügen von Benutzern, von denen jeder einzeln Einträge anlegen, ändern, einsehen und löschen kann. Hierzu verwendet das Programm eine Datenbank, worauf auch der Fokus bei der Implementierung der Teilnehmer liegt.

Sie haben die Aufgabe, die Funktionen, die mit einem `# TODO` Kommentar versehen sind, auszufüllen. Diese Funktionen beschränken sich auf eine Datei und haben alle mit dem Zugriff auf die Datenbank und ihrer Objekte zu tun.

4.5.2.2 *Aufbau des Projektordners*

Im Folgenden wird kurz der Aufbau des Projektordners beschrieben, in dem die Teilnehmer während der Studie die relevanten Dateien finden und welcher in der IDE geöffnet ist, da in Kapitel 5 darauf Bezug genommen wird.

Die Datenklassen befinden sich in der Datei `model.py`. Die Klasse `User` beschreibt einen Benutzer des Passwortmanagers. Ein solcher Benutzer hat eine eindeutige Id, einen Namen und eine Liste von Einträgen.

Diese Einträge werden durch die Klasse `Entry` abgebildet. Ein `Entry` beseitzt ebenfalls Id und Name, sowie ein Feld für weitere Informationen über den Eintrag (beispielsweise eine E-Mail Adresse) und das Passwort. Außerdem wird eine Referenz auf den Benutzer gespeichert, der den Eintrag angelegt hat.

Diese zwei Klassen sind über die Library `sqlalchemy`[2] direkt mit der Datenbank verbunden. Das heißt, dass `sqlalchemy` über die sogenannte `session` eine Schnittstelle bereitstellt, die das Hinzufügen, Ändern und Löschen mit konkreten Objekten der zwei Klassen erlaubt.

2 https://www.sqlalchemy.org/

Da mit dem Projektgerüst auch schon die Initialisierung der Datenbank implementiert ist, bleibt den Teilnehmern nur noch das Vervollständigen der Funktionen, die mit der Datenbank oder mit Objekten daraus interagieren, zum Beispiel das Hinzufügen von Benutzern und Einträgen und das Anzeigen, Bearbeiten und Löschen von Einträgen. Auch das Interagieren mit der Konsole ist schon vorgegeben und muss von den Teilnehmern nicht verändert werden. Dies beinhaltet die Ein- und Ausgabe sowie eine konkrete Menüführung, über die die einzelnen Funktionen des Programms abgerufen werden können.

4.5.2.3 *Beschreibung der Funktionen und Aufgaben*

Die konkreten Aufgaben und der Aufbau des Projekts sind der Aufgabenstellung zu entnehmen, die im Anhang (Abbildung A.3) der Arbeit beigefügt ist. Die Funktionen sind in zwei Arten zu unterteilen. Zwei Funktionen sollen Strings manipulieren, beziehungsweise zusammenstellen, während die anderen sechs sich mit Zugriffen auf die Datenbank, in der Benutzer und Einträge gespeichert sind, beschäftigen. Hier ist auch oft notwendig, dass Bedingungen geprüft werden, bevor die Datenbank verändert wird. Außerdem sind die Rückgabewerte der Funktionen in solchen Fällen von der Bedingung abhängig.

Beispiel 4.2:
Zum Start des Programms kann ein neuer Benutzer erstellt werden oder ein existierender sich einloggen. Eine der Funktionen mit Datenbankzugriff, die vervollständigt werden soll, ist das Einloggen des Benutzers. Hierzu gibt der Anwender des Programms im Willkommensbildschirm, welcher in Abbildung 4.1 dargestellt ist, `1` ein und drückt die Enter-Taste. Nun wird der Anwender nach Benutzername und Passwort gefragt. Nach der Eingabe wird die zu implementierende Funktion aufgerufen, die entscheidet, ob die eingegebenen Daten valide sind und entsprechend das `User`-Objekt oder `None` zurückgibt. Die Funktion (inklusive Lösungsvorschlag) ist in Listing 4.1 zu finden.

Listing 4.1: `login`-Funktion aus `db.py`

```
def login(session, name, password):
        """
        Checks whether a user with a given name and password exists
        :param session: The db session object
        :param name: The user's name
        :param password: The user's password as a hash
        :return: The user object, if the user exists and, thus, can be logged in; else
             None
        """
        # Loesungsvorschlag
        users = session.query(User)

        for user in users:
                if name == user.name and password == user.password:
                        return user
        return None
```

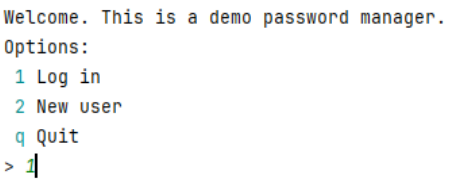


Abbildung 4.1: Willkommensbildschirm des Passwortmanagers in einer Konsole

Im nächsten Beispiel ist eine der beiden Funktionen beschrieben, in der es um String-Manipulation geht.

Beispiel 4.3:
Die Funktion `pad_str` wird mit einem String und einer gewünschten Länge aufgerufen. Sie gibt einen String zurück, der den ursprünglichen String enthält und auf die angegebene Länge mit Leerzeichen aufgefüllt ist. Dabei gibt der optionale Parameter `front` an, ob diese Leerzeichen hinten oder vorne eingefügt werden. Die Funktion soll anschließend in `get_entry_view` aufgerufen werden, um eine ansprechend formatierte Tabelle als String zu erzeugen. Die Funktion `pad_str` ist in Listing 4.2 inklusive Lösungsvorschlag dargestellt. Es ist anzumerken, dass es auch hier mehrere Arten gibt, die Funktion zu implementieren. Die Leerzeichen können wie in Listing 4.2 manuell angefügt werden oder es können die in PYTHON vordefinierten Funktionen `ljust` und `rjust` verwendet werden.

Listing 4.2: `pad_str`-Funktion aus `db.py`

```
1  def pad_str(text, total_length, front=True):
       """
       Extends a given string with spaces
       :param text: The text to extend
5      :param total_length: How long the final string should be
       :param front: True, if spaces are added at the front; False if spaces are
                     added at the back
       :return: The padded text
       """
10     # Loesungsvorschlag
       padding = " " * (total_length - len(text))

       if front:
           return f"{padding}{text}"
15     else:
           return f"{text}{padding}"
```

Es gibt einen Sonderfall in der Funktion, nämlich dass der eingegebene Text schon länger ist als die angegebene gewünschte Länge. In diesem Fall ist der Text unverändert zurückzugeben. Das geschieht in der dargestellten Lösung in Zeile 10 automatisch, da die $*$-Operation eine leere Liste zurückgibt, wenn die Zahl rechts (in diesem Fall `total_length - len(text)`) kleiner oder gleich null ist. Dadurch entstehen in diesem Fall keine Leerzeichen.

5 EVALUATION

5.1 DATENGEWINNUNG

Wie schon im vorigen Kapitel erklärt wurde, werden für die Beantwortung der Forschungsfragen verschiedene Daten verwendet. Das Hauptaugenmerk liegt hierbei auf den Aufzeichnungen der Sitzungen, die die Teilnehmer im Rahmen der Studie absolviert haben. Diese wurden für die weitere Verarbeitung in Textform umgewandelt. Erläuterungen zu diesem Prozess sind in Abschnitt 5.1.1 zu finden. In Abschnitt 5.1.2 wird beschrieben, wie aus dem so gewonnenen Transcript der Sitzung die Daten für die Beantwortung der Fragen gewonnen werden. Dabei wird sich an dem Rahmen orientieren, der am Ende von Kapitel 2 erarbeitet wurde. Des Weiteren werden die Antworten des Fragebogens ausgewertet, den die Teilnehmer zu Beginn einer Sitzung ausfüllen mussten. Diese Daten werden in Abschnitt 5.2.1 diskutiert.

Der Ablauf der Evaluation ist in Abbildung 5.1 dargestellt und wird in den folgenden Abschnitten detailliert erklärt.

5.1.1 *Transkript*

Von den Sitzungen wurden Ton- und Videoaufnahmen (des Bildschirms) mitgeschnitten. Um diese Aufnahmen in Textform zu konvertieren, waren mehrere Schritte nötig.

5.1.1.1 *Umwandlung zu Text*

Für die Umwandlung von Ton in Sprache wurde das KI-Modell WHISPER[1] [22] von OPENAI verwendet, da das manuelle Transkribieren zu viel Zeit in Anspruch genommen hätte. Es wurden noch andere Modelle in Betracht gezogen (zum Beispiel der Transkriptionsdienst von MICROSOFT, der über MICROSOFT OFFICE verfügbar ist), jedoch produzierte WHISPER die genausten Ergebnisse, insbesondere bei Hintergrundgeräuschen (wie dem Tippen auf der Tastatur) und Passagen, in denen die Teilnehmer leise oder undeutlich sprachen. Ein Vorteil des Modells von MICROSOFT gegenüber WHISPER ist das Bereitstellen von Zeitstempeln für jeden gesprochenen Satz, mit deren Hilfe man die zeitliche Dauer von *Episoden* und die Anteile der Redebeiträge einfacher hätte messen können. Jedoch fiel die Wahl aufgrund der deutlich höheren Genauigkeit der Transkribtion auf das Modell WHISPER.

Um die Tonaufzeichnungen automatisch zu transkribieren, wurden die Audiodateien in drei kleinere Dateien zerlegt, da WHISPER nur Dateien mit einer maximalen Länge von 20 Minuten verarbeiten kann. Die Output-Dateien wurden zu einer zusammengefügt und wie im Folgenden beschrieben nachbearbeitet.

1 https://openai.com/research/whisper

Abbildung 5.1: Eigene schematische Darstellung der Transformation und der anschließenden Auswertung der Daten

5.1.1.2 *Nachbearbeitung des Texts*

Um die Annotation zu erleichtern, wurden zuerst Zeilenumbrüche in das Transkript eingefügt. Der Output von WHISPER enthält zwar keine; dafür aber erkennt es an der Stimme zuverlässig das Satzende und setzt dementsprechend korrekt Satzzeichen wie Komma, Punkt, Fragezeichen und Ausrufezeichen. Nach dem Vorkommen eines Punkts, Frage- oder Ausrufezeichens wurde ein Zeilenumbruch gesetzt.

Außerdem wurden Sätze, die nur aus Füllwörtern (wie ‚ähm', ‚okay' oder ‚so') bestehen, automatisch entfernt, weil sie im Kontext des Transkripts keinen Mehrwert für den Inhalt des ausgesprochenen Gedankens enthalten.

5.1.2 *Annotation*

Im Folgenden wird der Prozess der Annotation eines Transkripts beschrieben. Ein Transkript wird zeilenweise annotiert, wobei jede Zeile aus einem (nicht notwendigerweise) vollständigem Satz besteht.

Um das Annotieren leichter und schneller zu gestalten, wurde im Zuge der Arbeit das Programm TRANSCRIPTOR entwickelt, welches es ermöglicht, durch das Transkript zu navigieren und die Zeilen mit Daten zu versehen. Es ist ebenfalls in PYTHON geschrieben.

Die erste Funktion ist das Zuordnen von Zeilen zu *Äußerungen*. Jede Zeile muss dabei zu genau einer *Äußerung* gehören. Dann können die *Äußerungseigenschaften* der einzelnen *Äußerungen* gesetzt werden (mit den Werten, die in Tabelle 4.1 aufgelistet sind). Des Weiteren können *Episoden* erstellt werden, zu denen sich die *Äußerungen* zuordnen lassen. Abbildung 5.2 zeigt einen Screenshot, in dem sich das Programm im *Episoden*-Modus befindet. Hier kann eine *Äußerung* einer *Episode* zugeordnet werden. Dazu muss zu der entsprechenden *Äußerung* navigiert werden (sodass die Klammer oder der Strich links neben dem Text fett markiert ist) und dann die Taste A gedrückt werden. Die aktuell ausgewählte *Episode* ist wie links im Bild zu sehen blau markiert. Navigiert wird durch die *Episoden* mit der rechten und linken Pfeiltaste und durch die *Äußerungen* mit den Tasten + und -. Die Ansicht kann davon unabhängig mit den Pfeiltasten nach oben und unten verschoben werden. Schließlich lassen sich ähnlich wie bei den *Äußerungen* die Attribute von *Episoden* setzen, wie in Tabelle 4.2 dargestellt. Der TRANSCRIPTOR erlaubt es aus aufgrund der Übersichtlichkeit außerdem zu annotieren, welche *Äußerungen* von welchem Sprecher getätigt wurden. Der Sprecher ist jeweils links neben dem Text mit A oder B angegeben. Am unteren Bildschirmrand sind Informationen über die aktuell ausgewählte *Episode* und *Äußerung* zusammengefasst. Hier ist auch erkennbar, dass es möglich ist, *Episoden* und

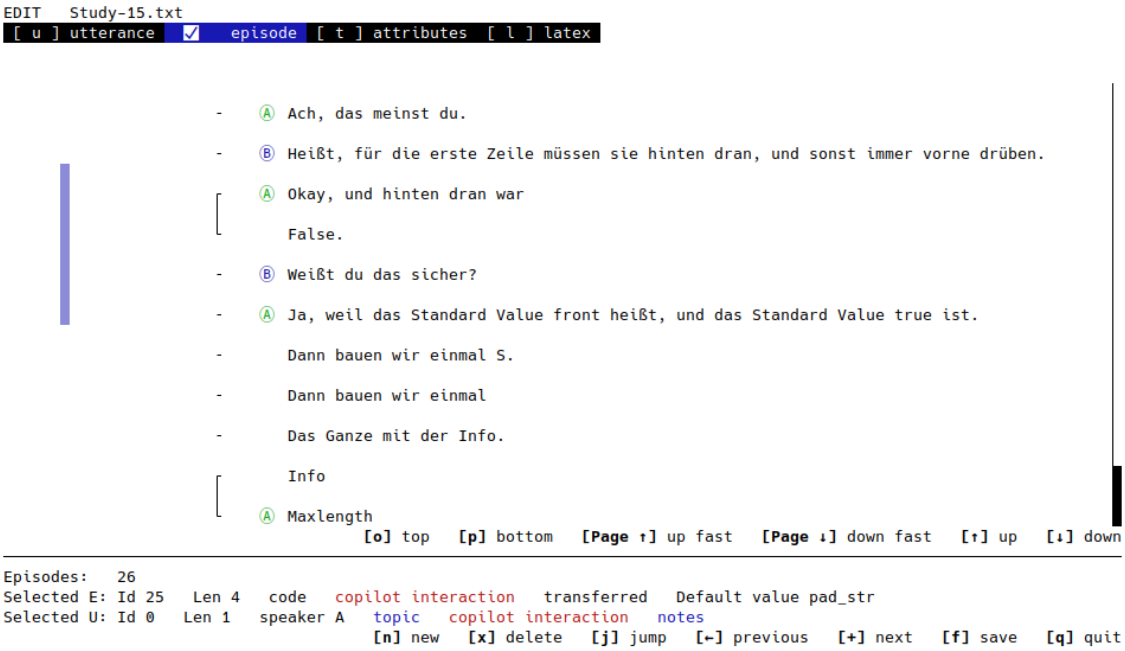


Abbildung 5.2: Screenshot des Tools TRANSCRIPTOR im *Episoden*-Modus. Ein Ausschnitt aus dem hier gezeigten Transkript ist auch in Abbildung 5.3 schematisch dargestellt

Äußerungen auch zu beschriften (durch das `notes`-Feld); ebenfalls um eine bessere Übersicht und leichtere Navigation im Transkript zu gewährleisten.

Schließlich bietet der TRANSCRIPTOR noch einen Modus zum LaTeX-Export. Hier kann eine Spanne von *Äußerungen* ausgewählt werden, die automatisch in LaTeX-Code umgewandelt wird. Das Ergebnis dieser Funktionalität wird im nächsten Abschnitt gezeigt und erklärt, da die Beispiele in dieser Arbeit damit erzeugt wurden.

Mit dem TRANSCRIPTOR werden die Transkripte nach den in Kapitel 2 erarbeiteten Kriterien annotiert. Durch die Bedienung mit ausschließlich der Tastatur kann die Annotation sehr schnell erfolgen, was die Bearbeitungszeit deutlich reduziert. Das Programm erzeugt eine Ausgabe im `json`-Format, sodass das Auswerten einfach möglich ist.

Ferner ist das Programm modular aufgebaut, sodass es leicht an andere Modelle angepasst werden kann. Somit ist der TRANSCRIPTOR ein allgemeines Werkzeug zum Annotieren von Transkripten mit beliebigen Datenformaten.

5.1.3 *Darstellung der Daten*

Wenn in dieser Arbeit Auszüge aus einem annotierten Transkript dargestellt werden sollen, geschieht dies in folgender Form:

An der Farbe des Texts ist der Sprecher der Zeile zu erkennen. Zeilen in schwarzer Farbe wurden vom ersten und Zeilen in hellblauer Farbe vom zweiten Programmierer gesprochen. Im Mensch-Mensch-Setting ist die Reihenfolge von erstem und zweitem Sprecher dabei zufällig; im Mensch-GITHUB COPILOT-Setting gibt es nur den ersten.

Die kurzen Striche oder eckige Klammern, die sich links des Texts befinden, deuten an, welche Zeilen zu einer *Äußerung* zusammengefasst wurden. Dies war notwendig, weil

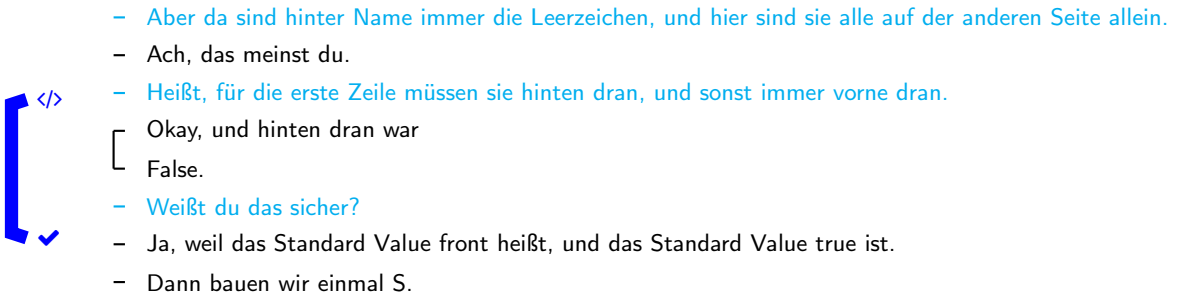


Abbildung 5.3: Beispielhafter Auszug aus einem Transkript der Kontrollgruppe aus Sitzung 15. Dieser Ausschnitt ist auch in Abbildung 5.2 dargestellt

das Sprachmodell WHISPER in manchen Fällen einen gesprochenen Satz in mehrere Sätze unterteilt, die eigentlich zusammengehören. Zeilen, die mit einem Strich versehen sind, bilden alleine eine *Äußerung*, eckige Klammern umfassen alle Zeilen, die zusammen als *Äußerung* zu verstehen sind.

Die farbigen Balken, in diesem Fall ein blauer, markieren die *Episoden*. Dabei dienen die Farben nur der Unterscheidbarkeit und haben ansonsten keine Bedeutung. Eine *Episode* umfasst alle *Äußerungen*, die von dem jeweiligen Balken eingeschlossen werden. Am Anfang und am Ende einer *Episode* sind verschiedene Symbole zu sehen. Das obere zeigt das *topic* der *Episode* an, das untere wiederum den *finish type*. Die Symbole für das *topic* (links) und den *finish type* (rechts) haben dabei die folgenden Bedeutungen:

- *tool*
- *program*
- *bug*
- *code*
- *domain*
- *technique*

- ✔ erfolgreich
  transferred
- ○ neutral
  trust
- ✖ fehlgeschlagen
  gave up, *lost sight*, *unnecessary*

Bei den *finish type*s wurden *gave up*, *lost sight* und *unnecessary* zu einer Kategorie (fail) zusammengefasst, da für die Auswertung nur relevant ist, ob eine Wissenstransfer-*Episode* erfolgreich war oder nicht.

5.2 AUSWERTUNG

In diesem Abschnitt werden zunächst die Daten statistisch ausgewertet und anschließend die Forschungsfragen auf dieser Grundlage beantwortet. Die Interpretation der Ergebnisse erfolgt in Kapitel 6.

Tabelle 5.1: Verpflichtende Angaben der Fragebögen der Experimentalgruppe (8 Teilnehmer)

Programmiererfahrung		Erfahrung mit		
relativ	**absolut**	**Python**	**Copilot**	**Pair Programming**
8	8	8	1	5
7	6	6	1	3
8	8	4	5	7
7	5	6	9	7
8	6	8	7	4
6	8	7	6	4
4	4	3	7	2
7	9	8	1	2

Tabelle 5.2: Verpflichtende Angaben der Fragebögen der Kontrollgruppe (7 Paare, 14 Teilnehmer)

Programmiererfahrung		Erfahrung mit		
relativ	**absolut**	**Python**	**Copilot**	**Pair Programming**
7	6	5	1	4
6	7	8	1	4
8	7	6	2	4
5	5	3	1	2
4	4	2	3	7
5	3	6	1	7
4	6	7	1	3
8	7	2	1	7
7	5	5	1	6
6	5	3	2	7
8	7	7	1	4
6	5	6	1	2
7	8	9	2	5
5	7	1	1	5

5.2.1 *Fragebogen*

Es fanden insgesamt 15 Sitzungen statt, acht davon im Mensch-GITHUB COPILOT-Setting und sieben im Mensch-Mensch-Setting. Es nahmen also 22 Personen an der Studie teil (zwei pro Pair Programming Setting und eine im Mensch-GITHUB COPILOT-Setting).

Der Großteil der Teilnehmer ist männlich, lediglich drei sind weiblich. Dies entspricht in etwa dem Frauenanteil in der Informatik, der an der Universität bei etwa 15% liegt [24]. Das mittlere Alter aller Teilnehmer liegt bei 22,8 Jahren und alle studieren in einem informatiknahen Studiengang im Bachelor und im Master.

Tabelle 5.1 und Tabelle 5.2 zeigt die Antworten der Fragebögen für die ersten fünf Fragen, die jeder der 22 Teilnehmer ausgefüllt hat.

Wie auch schon an den Angaben für die Programmiererfahrung zu erkennen ist, haben die Teilnehmer insgesamt eine hohe Programmiererfahrung. Das Publikum der Studie

waren Studierende im Fach Informatik, die sich in einem fortgeschrittenen Stadium ihres Studiums befinden oder während ihres Studiums bei einer Veranstaltung als Teaching Assistant engagiert waren. Dies wurde so entschieden, weil der Erfolg der Teilnehmer nicht von der Fähigkeit zu programmieren abhängen sollte, sondern an den gewählten Methoden (Pair Programming oder GITHUB COPILOT).

In Tabelle 5.3 sind die Durchschnittswerte der verpflichtenden Angaben der Fragebögen aufgelistet.

Wie bereits dargestellt, fällt auf, dass die Teilnehmer im Durchschnitt eine Programmiererfahrung haben, die über dem Mittel liegt. Hierbei ist auch anzumerken, dass absolute und relative Programmiererfahrung mit einem Koeffizienten von $0,56$ bei einem p-Wert von $0,006$ korrelieren (dafür wurde der Spearman Korrelationskoeffizient benutzt [27], weil die Skalen auf dem Fragebogen von 1 bis 10 ordinalskaliert sind). Bestätigt durch die Ergebnisse von Siegmund et al. [26] kann gefolgert werden, dass die Aussagen der Teilnehmer über ihre Programmiererfahrung mit ihrer tatsächlichen Erfahrung positiv korrelieren.

Außerdem korreliert die absolute Programmiererfahrung mit der Erfahrung in PYTHON mit einem Spearman-Koeffizienten von $0,48$ mit einem p-Wert von $0,023$. Das heißt, dass aus einer hohen Programmiererfahrung wahrscheinlich auch eine hohe Erfahrung mit der Srpache PYTHON folgt. Hier ist anzumerken, dass die Sprache im Rahmen von Universitätsveranstaltungen erst spät gelehrt wird, was erklären könnte, warum die Erfahrung damit im Durchschnitt niedriger ist als die allgemeine Programmiererfahrung. Jedoch lässt sich keine Korrelation zwischen Alter und Programmiererfahrung oder Alter und Erfahrung mit PYTHON feststellen.

Insgesamt weichen die Durchschnittswerte in den Gruppen (zu sehen in den ersten zwei Zeilen von Tabelle 5.3) nicht signifikant vom Mittelwert der Gesamtheit ab (nach einem Einstichproben-t-Test mit p-Werten von $0,36$ für die relative, beziehungsweise $0,39$ für die absolute Programmiererfahrung in der Experimentalgruppe; in der Kontrollgruppe liegen die p-Werte bei $0,49$ und $0,40$)

Auffällig sind die Ergebnisse der Frage nach der Erfahrung mit der Benutzung von GITHUB COPILOT. Hier ist in Tabelle 5.3 zu sehen, dass die Teilnehmer der Experimentalgruppe eine deutlich höhere Selbsteinschätzung $(4,6)$ abgegeben haben als die Teilnehmer der Kontrollgruppe $(1,4)$. Ein Einstichproben-t-Test zeigt, dass der Mittelwert der Kontrollgruppe hier hoch signifikant (mit einem p-Wert von $0,10$) vom allgemeinen Mittelwert abweicht. Für die Experimentalgruppe ist die Abweichung mit einem p-Wert von $9 \cdot 10^{-6}$ nicht statistisch signifikant.

Die Teilnehmer wurden zwar zufällig einer der Gruppen zugeordnet, jedoch birgt diese Tatsache einen Threat to Validity, da hier ein Vorteil für die Teilnehmer bestanden haben könnte, die Erfahrung mit GITHUB COPILOT hatten und es benutzten, während der Großteil an Teilnehmern ohne Erfahrung damit nicht in die Situation kam mit GITHUB COPILOT zu arbeiten. Das könnte die interne Validität verletzen.

Beim Pair Programming zeigt sich dieses Problem nicht. Das Konzept scheint allgemein bekannt zu sein und einige hatten schon Erfahrung mit der Technik, wie die Durchschnittswerte in Tabelle 5.3 nahelegen. Hier ist des Weiteren keine problematische Aufteilung auf die Gruppen festzustellen; sowohl in der Experimentalgruppe $(4,25)$ als auch in der Kontrollgruppe $(4,8)$ stimmen die Werte fast mit dem allgemeinen Durchschnitt $(4,6)$ überein. Das

Tabelle 5.3: Durchschnittswerte der verpflichtenden Angaben der Fragebögen

	Gesamt	Experimentalgruppe	Kontrollgruppe
Programmiererfahrung (Relativ)	6,4	6,9	6,1
Programmiererfahrung (Absolut)	6,1	6,8	5,9
Erfahrung mit Python	5,5	6,3	5,0
Erfahrung mit GitHub Copilot	2,5	4,6	1,4
Erfahrung mit Pair Programming	4,6	4,25	4,8

lässt sich durch einen Einstichproben-t-Test bestätigen (keine signifikante Abweichungen zum Mittelwert in den beiden Gruppen, p-Werte 0,64 und 0,69).

Zusammenfassung der Auswertung des Fragebogens: Die Auswertung des Fragebogens legt nahe, dass es bei der Verteilung der Teilnehmer auf die zwei Gruppen keine statistischen Bias gibt, bis auf die Erfahrung mit GITHUB COPILOT. In dieser Hinsicht könnte die interne Validität verletzt sein.

5.2.2 *Häufigkeit von Wissenstransfer*

Die zu beantwortende Forschungfrage lautet

RQ1: *Wie vergleicht sich die Anzahl an Wissenstransfer*-Episoden *zwischen einem Mensch-Mensch-Setting und einem Mensch-*GITHUB COPILOT*-Setting*?

Damit soll ermittelt werden, wie viele Teile der gesamten Konversation zum Wissenstransfer beitragen. Um diese Frage zu beantworten, werden alle *Episoden* in den beiden Gruppen gezählt. In den Pair Programming Sitzungen sind 226 *Episoden* aufgetreten und in den Sitzungen, in denen ein Teilnehmer GITHUB COPILOT verwendet hatte, kam es zu 127 *Episoden*. Absolut traten also im Mensch-Mensch-Setting mehr *Episoden* auf.

Pro Sitzung liegt der Durchschnitt für die Kontrollgruppe (die Pair Programming benutzt) bei 32,3 und für die Experimetalgruppe bei 15,9. In Abbildung 5.4 ist ein Histogramm zu sehen, anhand dessen die Häufigkeiten der *Episoden* pro Sitzung in den beiden Szenarien verglichen werden kann. Es ist zu beobachten, dass 50% der Teilnehmer mit GITHUB COPILOT zwischen 13 und 20 (mit Median 16,5) *Episoden* haben, während die Hälfte der Teilnehmer der Kontrollgruppe zwischen 26 und 35 (Median 33) *Episoden* haben. Die Abweichung der Mittelwerte ist statistisch nur für die Experimentalgruppe signifikant (p-Wert 0,04).

Zusammenfassung RQ1: Es ist ein deutlicher Unterschied in der Anzahl der *Episoden* zwischen den beiden Szenarien festzustellen. Beim Pair Programming treten siginifikant mehr Wissenstransfer-*Episoden* auf als mit GITHUB COPILOT.

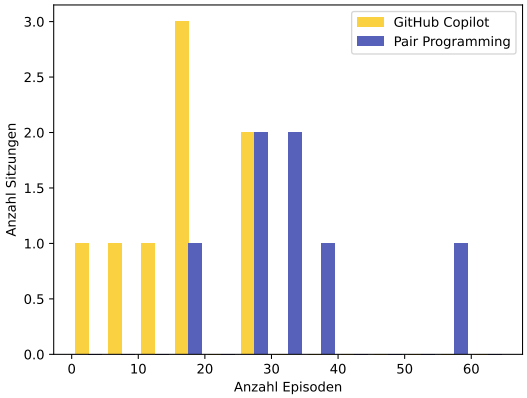


Abbildung 5.4: Vergleich der Anzahl der *Episoden*. Das Histogramm stellt dar, wie viele Sitzungen (auf der y-Achse) es mit einer bestimmten Anzahl an *Episoden* (auf der x-Achse) gibt. Die Bin-Größe beträgt 5 *Episoden*

5.2.3 *Aufbau und Schwierigkeit von Wissenstransfer*

Hier galt es die Frage

> **RQ2:** *Wie unterscheidet sich die Tiefe und Länge von* Episoden *zwischen den Szenarien?*

zu beantworten, um Aussagen über die Schwierigkeit der Inhalte der Episoden zu treffen. Dazu wurden die Länge und Tiefe der *Episoden* gemessen. Bei der Länge weden die *Äußerungen* gezählt, aus denen die entsprechende *Episode* besteht. Ein sinnvolleres Maß wäre die Zeit, die eine Wissenstransfer-*Episode* dauert, gewesen, weil die Dauer so ein absolutes Maß wäre. Da beim Transkribieren allerdings keine Zeitmarker für die einzelnen *Äußerungen* erstellt werden, kann die zeitliche Dauer einer *Episode* nicht ermittelt werden. Da *Äußerungen* unterschiedlich lang sind, können keine Rückschlüsse auf die tatsächliche Dauer geschlossen werden; jedoch erlaubt diese Anzahl trotzdem einzuschätzen, wie viele Gedanken, (Nach-) Fragen, Antworten, und so weiter nötig waren, um einen Wissenstransfer zu vollziehen.

5.2.3.1 *Länge*

In Abbildung 5.5 sind die beiden Gruppen hinsichtlich der Länge der *Episoden* gegenübergestellt. Im Durchschnitt hat eine *Episode* in einer Pair Programming Sitzung eine Länge von $13,8$ und bei Teilnehmern mit GITHUB COPILOT eine Länge con $9,3$ *Äußerungen*. Der Durchschnitt ist hier aber nicht aussagekräftig, weil es, wie in Abbildung 5.5 zu sehen, vor allem bei den *Episoden* der Kontrollgruppe viele Ausreißer nach oben gibt, die durch die niedrigen orangefarbenen Balken am rechten Rand abgebildet werden. Auch ist der Median mit $13,0$ beim Pair Programming höher ist als $7,8$ in der GITHUB COPILOT Gruppe. Außerdem unterscheiden sich die beiden Verteilungen siginifikant nach der Durchführung eines Chi-Squared-Tests (p-Wert $6 \cdot 10^{-129}$).

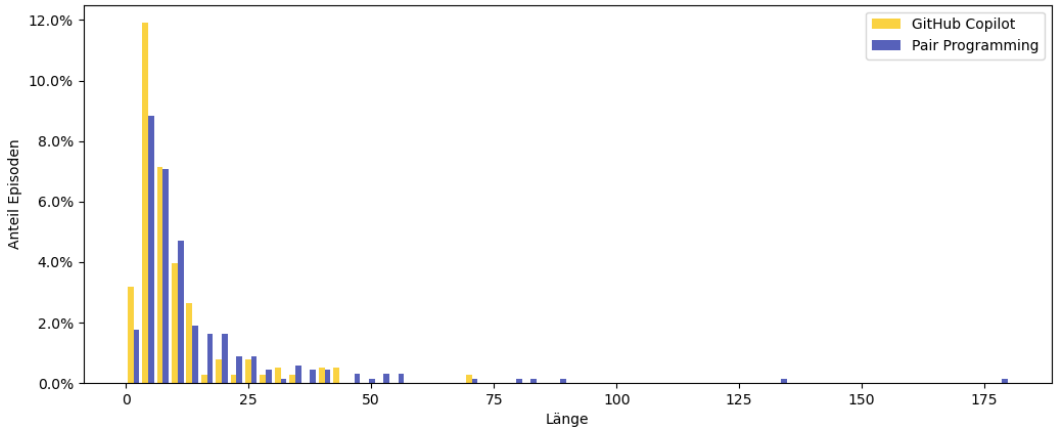


Abbildung 5.5: Vergleich der Länge der *Episoden* (gemessen in der Anzahl der *Äußerungen*). Auf der x-Achse ist die Länge, auf der y-Achse der Anteil der *Episoden* mit dieser Länge aufgetragen. Die Bin-Größe des Histogramms beträgt 3 *Äußerungen*

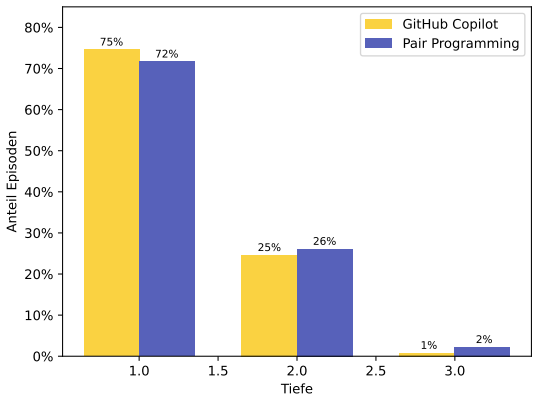


Abbildung 5.6: Vergleich der Tiefe der *Episoden*

5.2.3.2 *Tiefe*

In Abbildung 5.6 ist die Tiefe der *Episoden* nach den zwei Gruppen getrennt im Histogramm aufgetragen. Es ist zu erkennen, dass sich Experimental- und Kontrollgruppe hier nicht wesentlich unterscheiden.

In beiden Gruppen sind die meisten *Episoden* auf oberster Ebene (sie haben eine Tiefe von 1), sie betreffen also einen *need for knowledge*, der sich nicht aus einer laufenden *Episode* ergibt. Nur rund ein Viertel der *Episoden* ist auf eine laufende *Episode* gestapelt und 1%, beziehungsweise 2% sind auf zwei *Episoden* gestapelt (sie haben also eine Tiefe von 3).

Aus diesen Daten lassen sich nicht, wie erhofft, Aussagen über die Schwierigkeit der disktutierten Inhalte treffen, weil dafür zu wenige *Episoden* mit größerer Tiefe existieren. Denn es kann nicht behauptet werden, dass die diskutierten Themen, bei denen es lediglich *Episoden* mit Tiefe 1 oder 2 gibt, grundsätzlich trivial sind. Ein Gegenbeispiel dafür sind zum Beispiel die *Episoden* der Experimentalgruppe, in denen der Mensch den Code von GITHUB COPILOT erfolgreich nachvollzieht und sogar durch eine Recherche bestätigt. Hier

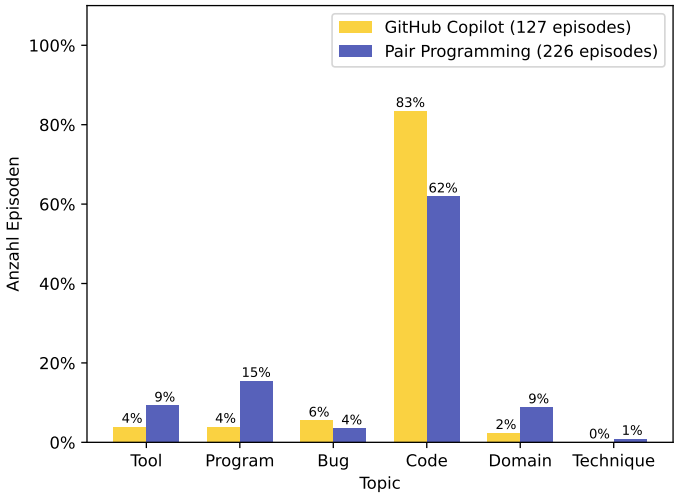


Abbildung 5.7: Relative Verteilung der *topics* der *Episoden*

liegt in den meisten Fällen zwar eine Tiefe von 1 vor, jedoch macht dies den Gegenstand des Wissenstransfers nicht weniger anspruchsvoll. Insofern ist der Aspekt der Tiefe für die Beantwortung der von RQ2 implizierten Frage nach den Schwierigkeitsunterschieden nicht zielführend. Dies wird in Kapitel 6 weiter ausgeführt.

Zusammenfassung RQ2: In der Länge der *Episoden* gibt es signifikante Unterschiede zwischen den beiden Gruppen, bei der Tiefe unterscheiden sie sich nicht, da hier die Verteilung der Tiefe in den beiden Gruppen fast gleich ist.

5.2.4 *Inhalt von Wissenstransfer*

Bei dieser Forschungsfrage lag der Fokus auf den *topics* der *Episoden*:

RQ3: *Gibt es einen Unterschied in der Verteilung der* topics *der* Episoden *zwischen den beiden Szenarien?*

Die Verteilung der *topics* ist in Abbildung 5.7 dargestellt. Die y-Achse zeigt den Anteil der *Episoden* der jeweiligen Gruppe mit dem *topic*, auf welchem der entsprechende Balken auf der x-Achse steht.

Da in den beiden Gruppen unterschiedlich viele *Episoden* vorkommen (127 in der Experimental-, 226 in der Kontrollgruppe), wurde für die Darstellung hier statt der absoluten Werte relative bevorzugt.

Die Verteilungen unterscheiden sich nach dem Chi-Squared Test [21] signifikant mit einem p-Wert von $0,0001$. Dieser statistische Test vergleicht zwei diskrete Verteilungen, indem er die Nullhypothese prüft, ob sich eine Verteilung nicht signifikant von der anderen unterscheidet. Dabei müssen die beiden Verteilungen nicht notwendigerweise gleich viele Samples (hier *Episoden*) haben, was hier wie oben angemerkt der Fall ist.

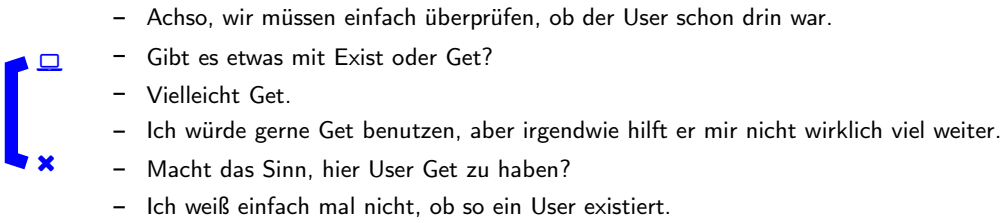


Abbildung 5.8: *Episode* über die Frage, welche Methode zum Abfragen einer Query genutzt werden soll (aus Sitzung 5)

Zunächst fällt auf, dass in der Experimentalgruppe das mit Abstand am häufigsten vorkommende *topic code* ist. Alle anderen *topics* sind mit maximal 6% vertreten. Bei der Kontrollgruppe ist *code* zwar auch das häufigste *topic*, jedoch sind die anderen hier stärker vertreten.

Über (abstrakte) Programmiertechniken und Vorgehensweisen wurde sich im Allgemeinen selten ausgetauscht (mit 0% und 1% *Episoden* mit *topic technique*). Ebenfalls ist der Anteil von *bug* in beiden Gruppen sehr niedrig. Dies könnte auf die Ähnlichkeit zu *code* zurückzuführen sein; die Teilnehmer fokussierten sich generell eher darauf, den Fehler zu beheben, indem sie neuen Code produzieren, anstatt sich lange über die Ursache des Fehlers auszutauschen.

Während die Teilnehmer beim Pair Programming auch über die Aufgabenstellung diskutiert haben, ist diese Quote bei der Experimentalgruppe ebenfalls sehr niedrig. Das kann damit begründet werden, dass die Ausgabe von GITHUB COPILOT meistens Code ist, der ein Problem löst, woraus aber nicht explizit hervorgeht, was die Fragestellung war. Trotzdem konnte GITHUB COPILOT in manchen Fällen dem Menschen durch einen Vorschlag vermitteln, dass zum Beispiel an einer Stelle in der Funktion noch eine wichtige Überprüfung fehlt.

Außerdem kam es mit GITHUB COPILOT selten zu *Episoden*, in denen GITHUB COPILOT dem Menschen Wissen über die Syntax von PYTHON vermitteln konnte. Im Fall der in Abbildung 5.8 dargestellten *Episode* ist der Wissenstransfer fehlgeschlagen, da der Programmierer den Fokus bezüglich des Themas verloren hat. In der Kontrollgruppe kam es deutlich häufiger zu *Episoden* über das *topic program* als in der Experimentalgruppe.

Zusammenfassung RQ3: Der Unterschied in den Verteilungen der *topics* sind signifikant. In Sitzungen mit GITHUB COPILOT ist das *topic* der *Episoden* vorrangig *code*, während beim Pair Programming die *topics* breiter gefächert sind.

5.2.5 *Qualität von Wissenstransfer*

Die letzte Frage, die zu beantworten ist, lautet:

RQ4: *Wie unterscheidet sich die Qualität des Wissenstransfers zwischen den zwei Szenarien bezüglich des Erfolgs des Wissenstransfers?*

Die Verteilung der *finish type*s ist in Abbildung 5.9 dargestellt.

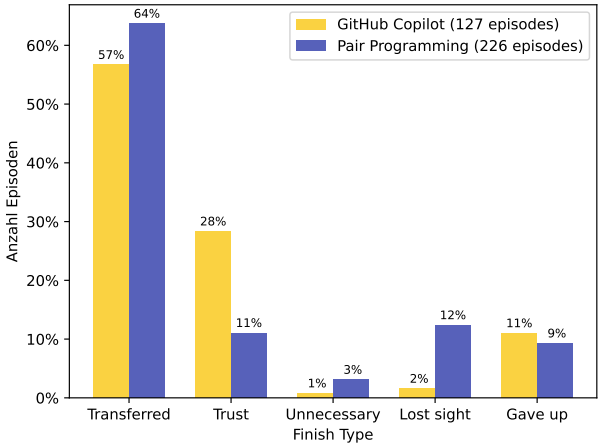


Abbildung 5.9: Relative Verteilung der *finish type*s der *Episoden*

- Ich würd's mal einfach probieren.
- Wenn GitHub Copilot das sagt, wird's schon stimmen.

Abbildung 5.10: Das Ende einer *Episode* mit dem *finish type trust* aus Sitzung 5

Die y-Achse zeigt den Anteil der *Episoden* der jeweiligen Gruppe mit dem *finish type*, auf welchem der entsprechende Balken auf der x-Achse steht. Ähnlich zu Abbildung 5.7 wurden hier relative Werte zur Darstellung verwendet, da die Anzahl der *Episoden* unterschiedlich zwischen den beiden Gruppen ist. Auch hier ist der Unterschied zwischen den Verteilungen nach dem Chi-Squared-Test mit einem p-Wert von $1,2 \cdot 10^{-5}$ statistisch hoch signifikant.

Die verschiedenen *finish type*s sind in Abschnitt 2.4.2.4 und Abschnitt 4.1.2 definiert. Es ist zu sehen, dass in der Kontrollgruppe der Anteil an erfolgreichen *Episoden* mit einer Differenz von 7% nur leicht höher ist als in der Experimentalgruppe. Es werden also ähnlich viele *Episoden* mit einem erfolgreichen Wissenstransfer abgeschlossen.

Interessanter ist der Unterschied in der Kategorie *trust*. Hier ist eine Differenz von 17% zu beobachten. Dies kann durch eine qualitative Analyse der Sitzungen begründet werden. In den GITHUB COPILOT Sitzungen war es häufig der Fall, dass der Mensch Vorschläge von GITHUB COPILOT nicht vollständig nachvollzog oder bloß die Hoffnung äußerte, dass der Vorschlag richtig sei, beispielhaft dargestellt in Abbildung 5.10. Dieses Phänomen trat im Mensch-Mensch-Setting seltener auf. Das zeigt, dass die Qualität von *Episoden* im Mensch-Mensch-Setting eher höher ist, da die Programmierer eher interessiert daran sind, den Inhalt nachzuvollziehen, bis er verstanden wurde, während es im Mensch-GITHUB COPILOT-Setting dafür spricht, dass die Menschen dazu geneigt sind, den produzierten Code zu akzeptieren ohne ihn zu hinterfragen.

Eine Kategorie, in der sich die Gruppen auch maßgeblich voneinander unterscheiden, ist der *finish type lost sight*. Daraus folgt, dass sich zwei Menschen im Gespräch leichter vom wesentlichen Inhalt ablenken lassen als ein Mensch mit GITHUB COPILOT, was wiederum für die Qualität des Wissenstransfers in der Experimentalgruppe spricht.

Zusammenfassung RQ4: Die Verteilung der *finish type*s unterscheidet sich signifikant zwischen den Gruppen. Es ist außerdem festzustellen, dass die Qualität des Wissenstransfers in der Kontrollgruppe dahingehend höher ist, dass es mehr erfolgreich und weniger neutral endende *Episoden* als in der Experimentalgruppe gibt. Jedoch werden Programmierer, die GITHUB COPILOT verwenden, weniger vom Inhalt abgelenkt als zwei Menschen, die sich miteinander über solche Inhalte unterhalten. Bei GITHUB COPILOT ist der Anteil der *Episoden* mit dem *finish type trust* besonders hoch.

5.3 ZUSAMMENFASSUNG

Abschließend lässt sich nach der Auswertung der Daten festhalten, dass alle Forschungsfragen dahingehend positiv beantwortet werden können, dass sich die zugrundeliegenden Verteilungen statistisch signifikant voneinander unterscheiden (außer bei der Tiefe von *Episoden*). Die Ergebnisse der statistischen Analyse konnten außerdem durch qualitative Einblicke in die Transkripte verifiziert und auch erklärt werden.

Bisher wurden aber noch keine maßgeblichen Schlussfolgerungen daraus gezogen. Dies soll im nächsten Kapitel geschehen, wo auch die Validität der Daten noch einmal diskutiert wird.

6 DISKUSSION

Das letzte Kapitel macht deutlich, dass sich Wissenstransfer im Mensch-Mensch-Setting deutlich von solchem im Mensch-GITHUB COPILOT-Setting in den meisten Belangen unterscheidet. Dieser Erkenntnis liegt aber auch die Tatsache zugrunde, dass bei der Benutzung von GITHUB COPILOT durchaus Wissenstransfer stattfindet. Die zentrale Forschungsfrage

> **RQ:** Können Menschen, die mit GITHUB COPILOT programmieren, ähnliche Wissenstransfer-Effekte erfahren wie Menschen, die Pair Programming anwenden?

ist also differenziert zu beantworten, da sich Häufigkeit, Ausprägung, Themen und auch Erfolg in den zwei Gruppen zwar unterscheiden, grundsätzlich aber auch bei Menschen, die mit GITHUB COPILOT programmieren, Wissenstransfer-Effekte feststellbar sind.

Im Folgenden wierden die einzelnen Aspekte, die sich von den Forschungsfragen (wie in Abschnitt 4.2 beschrieben) ableiten, im Detail interpretiert. Anschließend werden mögliche Threats to Validity erläutert, wonach ein kurzer Ausblick gegeben wird.

6.1 HÄUFIGKEIT VON WISSENSTRANSFER

Eine Quantifizierung von Wissenstransfer wurde in RQ1 durch das Zählen von *Episoden* vorgenommen. Zwar ist festzuhalten, dass in der Experimentalgruppe deutlich weniger *Episoden* stattfinden, dies jedoch mitunter damit zu begründen ist, dass in diesem Setting nur *Episoden* auftreten können, wenn der Mensch mit GITHUB COPILOT interagiert. Dies passiert nicht kontinuierlich über die ganze Sitzung, sondern vereinzelt. Große Teile der Sitzung werden unter anderem mit Lesen von Dokumentation verbracht. In der Experimentalgruppe ist das eine Zeitspanne, in der keine Interaktion mit GITHUB COPILOT stattfindet, während die Programmierer in der Kontrollgruppe Wissen auch über die Inhalte der Dokumentation austauschen können.

Daher lässt sich das Ergebnis von RQ1 dahingehend zusammenfassen, dass zwar mit GITHUB COPILOT weniger Wissenstransfer stattfindet, dies jedoch dem Tool selbst geschuldet ist. In dieser Arbeit wurde eine Version von GITHUB COPILOT genutzt, die nur das Generieren von Code unterstützte. Während der Durchführung der Studie erschien eine neue Version, die es dem Anwender nun auch erlaubt, per Chat direkt in der IDE Fragen zu stellen oder Diskussionen mit GITHUB COPILOT anzustoßen. Diese Änderung könnte im gleichen Studiendesign den Anteil an Wissenstransfer deutlich steigern, da jetzt auch ähnlich zum Mensch-Mensch-Setting eine Interaktion über andere Inhalte als den generierten Code möglich ist, beziehungsweise die Wissenstransfer-*Episoden* nicht mehr an die Interaktion bei der Code-Generierung gebunden sind. Dies könnte Gegenstand zukünftiger Arbeiten zu diesem Themenbereich sein.

6.2 AUFBAU UND SCHWIERIGKEIT VON WISSENSTRANSFER

RQ2 sollte die Schwierigkeit der behandelten Themen und den diskutierten Inhalten messen. Dies sollte durch die Länge und die Tiefen von *Episoden* gemessen werden. Die Begründung hierfür war, dass schwierige Inhalte mehr Argumente und Erklärungen und damit mehr *Äußerungen* und gestapelte *Episoden* brauchen als leichte, die sich zum Beispiel durch eine kurze Antwort auf eine Nachfrage lösen lassen.

In Abschnitt 5.2.3 wurde dargelegt, dass sich die Länge zwar zwischen den Gruppen unterscheidet, die Tiefe jedoch nicht. Deshalb ist festzustellen, dass darauf basierend keine Aussagen über die Schwierigkeit der Gegenstände des Wissenstransfers wie in Abschnitt 4.2 beschrieben gemacht werden können. Wenn auch die Tiefe keine Aussagen zulässt, so lassen sich dennoch aus den Daten der Länge Schlussfolgerungen ziehen.

Beispiel 6.1:
Die insgesamt längste *Episode* mit 179 *Äußerungen* stammt aus Sitzung 13, einer Pair Programming Sitzung. Sie beschäftigt sich mit dem Zusammensetzen und Ausführen einer Datenbankabfrage, welche sich in der `sqlalchemy`-Library in mehrere Befehle aufteilen lässt. Die Programmierpartner brauchten hierbei mehrere geschachtelte *Episoden*, um schließlich diese lange *Episode* abzuschließen. Sie ist dabei von der Art *co-production*, da die beiden Programmierer hier gemeinsam Wissen generieren und diskutieren, sodass sie zusammen den Code schreiben können, für den das Verständnis der Funktionsweise nötig ist.

Ein Ausschnitt dieser *Episode* ist in Abbildung 6.1 dargestellt. Zu diesem Zeitpunkt wird die Frage geklärt, welches Attribut einen `User` einzigartig macht (also welches Feld in der Datenbank als Key hinterlegt ist).

Die Antwort, dass es sich dabei nicht, wie vom ersten Sprecher zuerst vermutet, um den Namen, sondern, wie vom zweiten Programmierer aufgeklärt, um die Id handelt, wird in der darauffolgenden *Episode* verwendet, um eine Query zu erstellen, mit der dann ein `User` abgefragt werden kann. Anschließend lesen die Programmierer noch einmal in der Online-Dokumentation nach und schreiben zum Schluss Code, von dem sie aber nicht überzeugt sind, dass er funktioniert. Aus diesem Grund endet die lange Episode mit dem *finish type trust*.

Kurze *Episoden* bilden beim Pair Programming eher Frage-Antwort-Abfolgen ab, wie die im folgenden Beispiel beschriebene, die auch in Abbildung 6.2 dargestellt ist.

Beispiel 6.2:
Hier erklärt ein Programmierer dem anderen die Funktionsweise von Keyword-Args in PYTHON. Deshalb ist der *Episode* das *topic program* zugeordnet.

Im Histogramm von Abbildung 5.5 ist weiterhin gut zu erkennen, dass die meisten *Episoden* in der Experimentalgruppe nicht länger als 25 *Äußerungen* sind.

Die meisten dieser *Episoden* stellen eine Art *co-production* dar, das heißt, der Mensch schreibt mit Hilfe von GITHUB COPILOT code und verifiziert, dass dieser tatsächlich die gewünschte Funktionalität implementiert. Ein Grund, dass solche *Episoden* nicht so lange dauern wie beim Pair Programming, ist, dass in der Experimentalgruppe die Programmierer

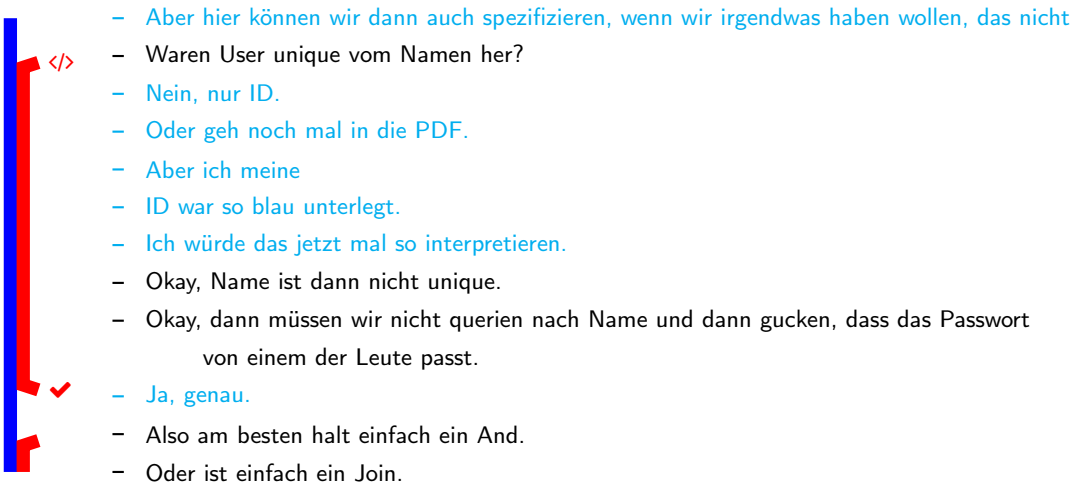


Abbildung 6.1: Auszug aus der längsten *Episode* insgesamt (Sitzung 13)

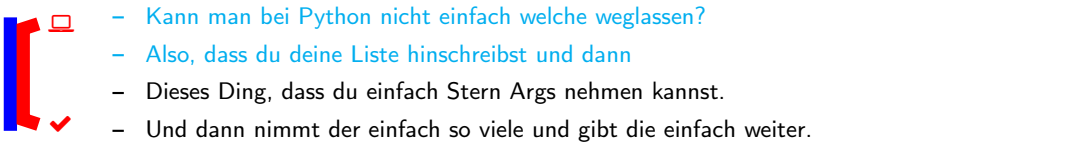


Abbildung 6.2: *Episode* über die Funktionsweise von Keyword-Args aus Sitzung 15

schneller aufgeben oder sich dazu entscheiden den generierten Code nicht verstehen zu müssen. Dieses Phänomen wird im Abschnitt 6.4 genauer diskutiert.

Aber auch hier gibt es deutlich kürzere *Episoden*, in denen zum Beispiel ein Codevorschlag von GITHUB COPILOT den Menschen auf etwas hinweist.

Beispiel 6.3:
Dieses Szenario trat am häufigsten auf, wenn die Programmierer die letzte Funktion implementierten. Sie sollte einen Eintrag bearbeiten und in die Datenbank speichern. Hierbei schlug GITHUB COPILOT am Ende vor, dass die Änderung mit dem Befehl `session.commit()` auch tatsächlich in die Datenbank geschrieben wird, damit sie persistent ist. Jedoch war diese Zeile meist in keiner der vorherigen Funktionen, wo dies auch notwendig gewesen wäre, vorhanden, sodass viele Programmierer erst ganz am Schluss der Sitzung von GITHUB COPILOT darauf aufmerksam gemacht wurden, dass es den Befehl gibt und dass er vermutlich in anderen Situationen auch angewandt werden muss.

Über die Länge lässt sich also zusammenfassend sagen, dass für Pair Programming eher längere *Episoden* typisch sind mit wenigen kurzen dazwischen, während im Mensch-GITHUB COPILOT-Setting vorwiegend *co-productions* stattfinden, die länger als einfache Frage-Antwort-Sequenzen sind, aber trotzdem im Vergleich zum Pair Programming schneller enden.

6.3 INHALT VON WISSENSTRANSFER

Die *Episoden*, die in der Experimentalgruppe stattfanden, beschäftigen sich zum Großteil auf das Verstehen und Nachvollziehen von Vorschlägen, die GITHUB COPILOT dem Menschen macht, das war in Abschnitt 5.2.4 deutlich zu sehen. Obwohl GITHUB COPILOT implizit Zugriff auf die Aufgabenstellung in Form der Docstrings hatte, fand nur in sehr wenigen Fällen ein Wissenstransfer statt, der die Aufgabe thematisierte.

Auch waren nur wenige *Episoden* zu finden, die die Syntax oder Semantik der Programmiersprache selbst betreffen. Dies könnte auch daran liegen, dass die meisten Teilnehmer eine eher hohe Erfahrung mit PYTHON hatten, was sich auf beide Gruppen des Experiments bezieht. Trotzdem zeigt sich hier ein deutlicher Unterschied zwischen den Gruppen, da Menschen sich im Mensch-Mensch-Setting Fragen stellen können und dazu auch ermutigt sind, während bei der Benutzung von GITHUB COPILOT Vorschläge, die beispielsweise Syntax enthalten, die der Programmierer nicht kennt, einfach schnell und ohne Äußerung ignoriert werden können.

Auch haben sich die Teilnehmer, wie in Abbildung 5.7 zu sehen, fast gar nicht über Programmierkonzepte oder -techniken geäußert, was der Tatsache geschuldet sein könnte, dass dies im Rahmen des vorgegebenen Programmierprojekts nicht für nötig erachtet wurde. Der Entwurf war dadurch schon festgelegt und es mussten nur noch Methoden vervollständigt werden. Die wenigen *Episoden*, die dem *topic technique* zuzuordnen sind, befassen sich mit dem Testen des Codes.

Dass auch insgesamt die Kategorie *bug* eher selten auftritt, liegt wahrscheinlich daran, dass bei auftretenden Fehlern im Allgemeinen wenig über die Ursache geredet wurde, sondern vielmehr nach einer Lösung gesucht wurde. In manchen Fällen führte das auch dazu, dass der betreffende Code gelöscht und neu geschrieben wurde, womit sich der Bezug auf den Fehler meistens auflöste.

Somit steht fest, dass bei der Benutzung von GITHUB COPILOT Wissenstransfer durch die Vorschläge passiert und meistens auch konkrete Vorgehensweisen thematisiert, während die Themen beim Pair Programming breiter gefächert sind.

6.4 QUALITÄT VON WISSENSTRANSFER

Die Daten aus Abschnitt 5.2.5 zeigen, dass es auch bei der Qualität des Wissenstransfers Unterschiede gibt. Zum Einen ist die Quote erfolgreicher Wissenstransfer-*Episoden* beim Pair Programming höher. Zum Anderen muss gesagt werden, dass die Ablenkung vom Thema bei Verwendung von GITHUB COPILOT geringer ist.

Was bei der Experimentalgruppe jedoch hervorsticht, ist der deutlich höhre Anteil an *Episoden*, die damit enden, dass der Mensch, der GITHUB COPILOT benutzt oder die beiden Programmierer ein Thema ruhen lassen, weil sie nicht bereit sind die Information zu verarbeiten. So wird nur die Hoffnung geäußert, dass die gegebene Information richtig sei, ohne die Gründe dafür nachzuvollziehen.

Das lässt darauf schließen, dass GITHUB COPILOT dazu verleitet mit weniger Konzentration zu programmieren, da Menschen dazu neigen sich darauf verlassen, dass die Vorschläge richtig sind (in dem Sinne, dass sie eine gewisse Anforderung, zum Beispiel gegeben durch das Design oder eine Dokumentation erfüllen) und sie somit einfach akzeptieren. Dies führt

im Falle eines Vorschlags, der nicht allen Details der geforderten Funktionalität entspricht dazu, dass sich der Programmierer nochmal von Neuem in den Code einarbeiten und ihn nachvollziehen muss, da durch das geschilderte Verhalten kein Programmverständnis entsteht oder sogar wichtige Details im Code (zum Beispiel Vertauschung von Variablen, Off-by-one-Fehler, und so weiter) gar nicht beachtet wurden. Somit bedeutet das einen erheblichen Mehraufwand, da eine Fehlersuche in unbekanntem Code durch das zuerst nötige Einarbeiten länger dauert.

Dieser Befund sollte dazu genutzt werden, um die vorgesehene und die tatsächliche Arbeitsweise mit GITHUB COPILOT zu reflektieren und gegebenenfalls bei der Entwicklung sowie der Verwendung des Tools anzupassen.

6.5 VALIDITÄT

In diesem Abschnitt werden mögliche die Studie betreffende Threats to Validity diskutiert.

Insgesamt ist zu beobachten, dass die meisten Teilnehmer wenig bis sehr wenig Erfahrung im Umgang mit GITHUB COPILOT haben, denn 13 von 22 Teilnehmern gaben bei der Frage eine 1 (sehr niedrige Erfahrung) und drei Teilnehmer eine 2 an. Dass fast zwei Drittel der Teilnehmer noch nie bis sehr wenig mit GITHUB COPILOT gearbeitet haben, lässt sich auf verschiedene Weisen erklären. Zum einen könnte die Hürde, dass die Nutzung von GITHUB COPILOT nicht kostenlos ist, viele Menschen davon abhalten das Werkzeug auszuprobieren und langfristig zu benutzen, insbesondere im Kontext der Universität. Zum anderen scheint es größtenteils unbekannt unter den Teilnehmern, dass es für Studierende ein Angebot von GITHUB gibt, durch welches sie GITHUB COPILOT kostenlos nutzen können. Dieses Angebot überraschte viele Teilnehmer, als sie nach der Sitzung darauf hingewiesen wurden.

Die Teilnehmer waren in dieser Hinsicht ungünstig auf die Gruppen aufgeteilt, sodass alle Teilnehmer, die Erfahrung mit GITHUB COPILOT haben auch in der Experimentalgruppe waren, obwohl die Zuteilung zufällig war. Damit ist die interne Validität gefährdet, da bei einer anderen Zuteilung die Experimentalgruppe diesen Vorteil nicht gehabt hätte.

Ein weiterer Threat to Validity ist, dass die Annotation von einer Person allein, dem Verfasser dieser Arbeit, durchgeführt worden ist. Dadurch ist die Objektivität der Daten nicht vollständig gewährleistet. Hier wäre es für zukünftige Arbeiten, in denen die Methodik ebenfalls verwendet wird, sinnvoll, jede Sitzung zuerst unabhängig von verschiedenen Personen annotieren zu lassen, bevor die Daten verglichen werden und es zu einer finalen Annotation kommt. So kann die Objektivität bei der Auswertung erhöht werden.

Die externe Validität ist insofern gefährdet, als dass von der ausgewählten Gruppe der Teilnehmer nur schwer auf die Allgemeinheit geschlossen werden kann. Während zwar die Geschlechterverteilung ungefähr mit der an der Universität übereinstimmt, kann das Leistungsniveau der Teilnehmer höher eingeschätzt werden als der Durchschnitt, da die Teilnehmer dieses, wie schon in Abschnitt 5.2.1 beschrieben, durch einen schon errungenen Abschluss oder eine Anstellung als Teaching Assistant erhöhen konnten. Außerdem hilft der Gesamtkontext von Studierenden an einer Universität nicht dabei, die Fragen mit Hinblick auf die Situation in der Arbeitswelt, also zum Beispiel in einem Unternehmen zu untersuchen, da die Situationen von Grund auf unterschiedlich sind. Somit sollten Folgestudien nicht nur an Universitäten, sondern auch mit Teilnehmern durchgeführt

werden, die GITHUB COPILOT und Pair Programming auch alltäglich im Beruf verwenden, um ein realistischeres Bild zu erhalten.

Außerdem stellt das laute Denken einen Threat to Validity dar, denn es besteht die Gefahr, dass die Teilnehmer nicht alle Gedanken aussprechen oder durch den Druck, der möglicherweise dadurch erzeugt wird, die Konzentration oder die Leistung sinkt. Außerdem könnte es die Personen langsamer machen, da das Formulieren und Aussprechen der Gedanken zusätzlich Zeit in Anspruch nimmt [6]. Somit ist es möglich, dass die Transkripte der Experimentalgruppe nicht alle Gedanken der Teilnehmer enthalten und somit Wissenstransfer-*Episoden* nicht erfasst werden können.

6.6 AUSBLICK

Diese Arbeit schlägt eine Methodik vor, wie Wissenstransfer beim Programmieren messbar gemacht werden kann. Dieses Framework kann in zukünftigen Arbeiten verwendet werden, um mehr Datenpunkte in vergleichbaren Szenarien (in demselben Umfeld der Teilnehmer) zu sammeln und um die Ergebnisse mit denen in anderen Umfeldern zu vergleichen (zum Beispiel in Unternehmen).

Außerdem wäre es interessant, wenn auch außerhalb des Umfangs dieser Arbeit, das Framework auf die erst zum Zeitpunkt der Abgabe der Arbeit verfügbare Version von GITHUB COPILOT anzuwenden. Diese neuere unterstützt im Gegensatz zu der in der Studie genutzten Version einen KI-gestützten Chat, in dem Entwickler Fragen an GITHUB COPILOT stellen können, die in Echtzeit (ähnlich zu CHATGPT von OPENAI) beantwortet werden. Somit könnten auch andere *topics* Gegenstand in der Konversation mit GITHUB COPILOT werden, was sich auch auf die Qualität des Wissenstransfers auswirken könnte.

Des Weiteren wäre es vorstellbar zusammen mit anderen Fachbereichen, die sich auf Psychologie oder Didaktik spezialisiert haben, zusammenzuarbeiten. Somit könnten basierend auf den Ergebnissen didaktische Konzepte für die Verwendung mit GITHUB COPILOT erarbeitet werden, die Arbeitsabläufe und Lernprozesse beim Wissenstransfer mit GITHUB COPILOT optimieren.

Es bleibt zu sagen, dass es vielfältige Möglichkeiten gibt, die erarbeitete Methodik in Zukunft einzusetzen.

7 ZUSAMMENFASSUNG

Dieser Arbeit hat sich mit dem Wissenstransfer beim Programmieren zwischen zwei Menschen auf der einen und zwischen Mensch und GITHUB COPILOT auf der anderen Seite beschäftigt. Dazu wurde zunächst den Begriff Wissenstransfer erklärt und beleuchtet, dass er in verschiedenen Kontexten verschiedene Bedeutungen haben kann. Auch im Kontext von Software Engineering haben sich mehrere Arbeiten dem für diese Studie relevanten Begriff von Wissenstransfer gewidmet und dabei verschiedene Sichtweisen betont. Diese wurden im Rahmen dieser Arbeit zusammengeführt. Damit wurde ein eigenes Framework erstellt, welches darauf abzielt Wissenstransfer hinsichtlich Anteil, Struktur, Inhalt und Qualität von Wissenstransfer messbar zu machen und damit die Forschungsfragen zu beantworten.

Dieses Framework wurde in einer empirischen Studie mit 22 Teilnehmern benutzt, um die insgesamt 15 Programmiersessions mit 8 Einzelpersonen und 7 Paaren dahingehend auszuwerten. Dazu wurde ein Programm erstellt, welches erlaubt diesen Prozess einfach, übersichtlich und schnell durchzuführen.

Die Auswertung bestand im Markieren von *Äußerungen* und *Episoden*, welche mit weiteren Daten (wie dem *topic* oder den *finish type*s) angereichert wurden, die zum Beantworten der Forschungsfragen relevant waren. Dabei wurde festgestellt, dass der Anteil, den Wissenstransfer an einer Sitzung ausmacht, in einem Mensch-Mensch-Setting höher ist als in einem Mensch-GITHUB COPILOT-Setting. Außerdem unterscheiden sich die beiden Szenarien in der Struktur des Wissenstransfers, sowie in den Inhalten, die dem Wissenstransfer zugrundeliegen. Bei der Qualität ließ sich feststellen, dass hier mehrere Phänomene zu beobachten waren. Zum einen ließen sich in den Pair Programming Sitzungen die Teilnehmer häufiger durch ihr Gespräch vom Thema abbringen, wohingegen GITHUB COPILOT dazu verleitete sich blind auf seine Vorschläge zu verlassen ohne sie nachzuvollziehen oder sie zu überprüfen.

Es wurde festgestellt, dass insbesondere letzteres interessante Punkte sind, welche bei der weiteren schnellen Entwicklung von GITHUB COPILOT (und anderen vergleichbaren Tools) lohnenswert erscheinen, weiter zu verfolgen und immer wieder zu überprüfen, ob sich solche Verhaltensweisen verändern oder neue entstehen. Die übergeordnete Forschungsfrage, ob es mit GITHUB COPILOT zu ähnlichen Wissenstransfer-Effekten wie beim Pair Programming kommt, kann deutlich mit ja beantwortet werden.

Somit leistet diese Arbeit einen wichtigen Beitrag zur Beantwortung der Frage, die sich aus MICROSOFT's Werbung für GITHUB COPILOT ergibt: Lässt sich in einem Paar zweier Programmierer einer davon durch den Assistenten GITHUB COPILOT ersetzen? Diese Arbeit zeigt, dass dies für den Aspekt des Wissenstransfers zwar bereits möglich ist, aber noch Verhaltensweisen gefördert werden, die dem Wissenstransfer in manchen Situationen hinderlich sind.

LITERATUR

[1] Erik Arisholm, Hans Gallis, Tore Dybå und Dag I. K. Sjøberg. „Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise“. In: *IEEE Trans. Software Eng.* 33.2 (2007), S. 65–86.

[2] Owura Asare, Meiyappan Nagappan und N. Asokan. „Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?“ In: *Empir. Softw. Eng.* 28.6 (2023), S. 129.

[3] Shraddha Barke, Michael B. James und Nadia Polikarpova. „Grounded Copilot: How Programmers Interact with Code-Generating Models“. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), S. 85–111.

[4] Kent L. Beck. *Extreme programming explained - embrace change*. Addison-Wesley, 1990.

[5] Andrew Begel und Nachiappan Nagappan. „Pair programming: what's in it for me?“ In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 2008, S. 120–128.

[6] Elizabeth Charters. „The use of think-aloud methods in qualitative research an introduction to think-aloud methods“. In: *Brock Education Journal* 12.2 (2003).

[7] Mark Chen et al. „Evaluating Large Language Models Trained on Code“. In: *ArXiv* abs/2107.03374 (2021).

[8] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais und Zhen Ming (Jack) Jiang. „GitHub Copilot AI pair programmer: Asset or Liability?“ In: *J. Syst. Softw.* 203 (2023).

[9] Prashika Dhoodhanath und Rosemary Quilling. „Case study: Factors that hinder and support the adoption of Pair Programming in an agile software development company“. In: *2020 International Conference on Artificial Intelligence, Big Data, Computing and Data Communication Systems (icABCD)*. 2020, S. 1–7.

[10] Iddo Drori und Nakul Verma. „Solving Linear Algebra by Program Synthesis“. In: *CoRR* abs/2111.08171 (2021).

[11] Tore Dybå, Erik Arisholm, Dag I. K. Sjøberg, Jo Erskine Hannay und Forrest Shull. „Are Two Heads Better than One? On the Effectiveness of Pair Programming“. In: *IEEE Softw.* 24.6 (2007), S. 12–15.

[12] Saki Imai. „Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study“. In: *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings*. ACM/IEEE, 2022, S. 319–321.

[13] Danielle L. Jones und Scott D. Fleming. „What use is a backseat driver? A qualitative investigation of pair programming“. In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. Hrsg. von Caitlin Kelleher, Margaret M. Burnett und Stefan Sauer. IEEE Computer Society, 2013, S. 103–110.

[14] Kultusministerkonferenz. „Bildungsstandards im Fach Mathematik für den Mittleren Schulabschluss“. In: *Beschlüsse der Kultusministerkonferenz* (2003).

[15] Kultusministerkonferenz. „Bildungsstandards für das Fach Mathematik Erster Schulabschluss (ESA) und Mittlerer Schulabschluss (MSA)“. In: *Beschlüsse der Kultusministerkonferenz* (2022).

[16] Sandeep Kaur Kuttal, Bali Ong, Kate Kwasny und Peter Robe. „Trade-offs for Substituting a Human with an Agent in a Pair Programming Context: The Good, the Bad, and the Ugly“. In: *CHI '21: CHI Conference on Human Factors in Computing Systems*. Hrsg. von Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, Takeo Igarashi, Pernille Bjørn und Steven Mark Drucker. 2021, 243:1–243:20.

[17] Nhan Nguyen und Sarah Nadi. „An Empirical Evaluation of GitHub Copilot's Code Suggestions“. In: *19th IEEE/ACM International Conference on Mining Software Repositories, MSR*. ACM, 2022, S. 1–5.

[18] Timothy J. Nokes-Malach und Jose P. Mestre. „Toward a Model of Transfer as Sense-Making“. In: *Educational Psychologist* 48.3 (2013), S. 184–207.

[19] Timothy J. Nokes. „Mechanisms of knowledge transfer“. In: *Thinking & Reasoning* 15.1 (2009), S. 1–36.

[20] Daniel E O'Leary. „Enterprise Knowledge Management“. In: *Computer* 31.3 (1998), S. 54–61.

[21] R. L. Plackett. „Karl Pearson and the Chi-Squared Test“. In: *International Statistical Review / Revue Internationale de Statistique* 51.1 (1983), S. 59–72.

[22] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey und Ilya Sutskever. *Robust Speech Recognition via Large-Scale Weak Supervision*. Hrsg. von Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato und Jonathan Scarlett. 2023.

[23] Peter Robe, Sandeep Kaur Kuttal, Jake AuBuchon und Jacob C. Hart. „Pair programming conversations with agents vs. developers: challenges and opportunities for SE community“. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. Hrsg. von Abhik Roychoudhury, Cristian Cadar und Miryung Kim. ACM, 2022, S. 319–331.

[24] Universität des Saarlandes. *Studierendenstatistiken Sommersemester 2023*. https://www.uni-saarland.de/universitaet/portraet/zahlen/studierendenstatistik.html. Aufgerufen: 19.11.2023.

[25] Georg Schreyögg und Peter Conrad. *Wissensmanagement*. Bd. 6. Walter de Gruyter GmbH & Co KG, 2021.

[26] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel und Stefan Hanenberg. „Measuring and Modeling Programming Experience“. In: *Empirical Softw. Eng.* 19.5 (2014).

[27] C. Spearman. „The Proof and Measurement of Association between Two Things“. In: *The American Journal of Psychology* 15.1 (1904), S. 72–101.

[28] Leonard Tang, Elizabeth Ke, Nikhil Singh, Bo Feng, Derek Austin, Nakul Verma und Iddo Drori. „Solving Probability and Statistics Problems by Probabilistic Program Synthesis at Human Level and Predicting Solvability“. In: Lecture Notes in Computer Science 13356 (2022). Hrsg. von Maria Mercedes T. Rodrigo, Noburu Matsuda, Alexandra I. Cristea und Vania Dimitrova, S. 612–615.

[29] Christopher K. Flynn Trina C. Kershaw und Leamarie T. Gordon. „Multiple paths to transfer and constraint relaxation in insight problem solving“. In: *Thinking & Reasoning* 19.1 (2013), S. 96–136.

[30] Titus Winters, Tom Manshreck und Hyrum Wright. *Software engineering at google: Lessons learned from programming over time*. O'Reilly Media, 2020.

[31] Franz Zieris und Lutz Prechelt. „On knowledge transfer skill in pair programming“. In: *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM*. Hrsg. von Maurizio Morisio, Tore Dybå und Marco Torchiano. ACM, 2014, 11:1–11:10.

[32] Franz Zieris und Lutz Prechelt. „Observations on knowledge transfer of professional software developers during pair programming“. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE*. Hrsg. von Laura K. Dillon, Willem Visser und Laurie A. Williams. ACM, 2016, S. 242–250.

A ANHANG

A.1 FRAGEBOGEN

Knowledge Transfer Study – Pre-test Questionnaire

Saarland University

Before you get the assignment, please fill out the following questionnaire. *The second part is voluntary to fill out.*

Part 1: Programming Experience

(A) How do you estimate your programming experience compared to your fellow students?

very inexperienced | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | *very experienced*

(B) How do you estimate your programming experience?

very inexperienced | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | *very experienced*

(C) How do you estimate your experience specifically in the Python programming language?

very inexperienced | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | *very experienced*

(D) How do you estimate your experience regarding GitHub Copilot?

very inexperienced | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | *very experienced*

(E) How do you estimate your experience regarding pair programming?

very inexperienced | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | *very experienced*

Part 2: General Information

These questions are voluntary to answer. We will use the information to get a better understanding of the participants of the study. For each question, you can decide whether you like to answer it or not.

(A) How old are you? _____ years

(B) What is your gender? ○ *Female* ○ *Male* ○ *Other*

(C) What is your mother tongue? _____

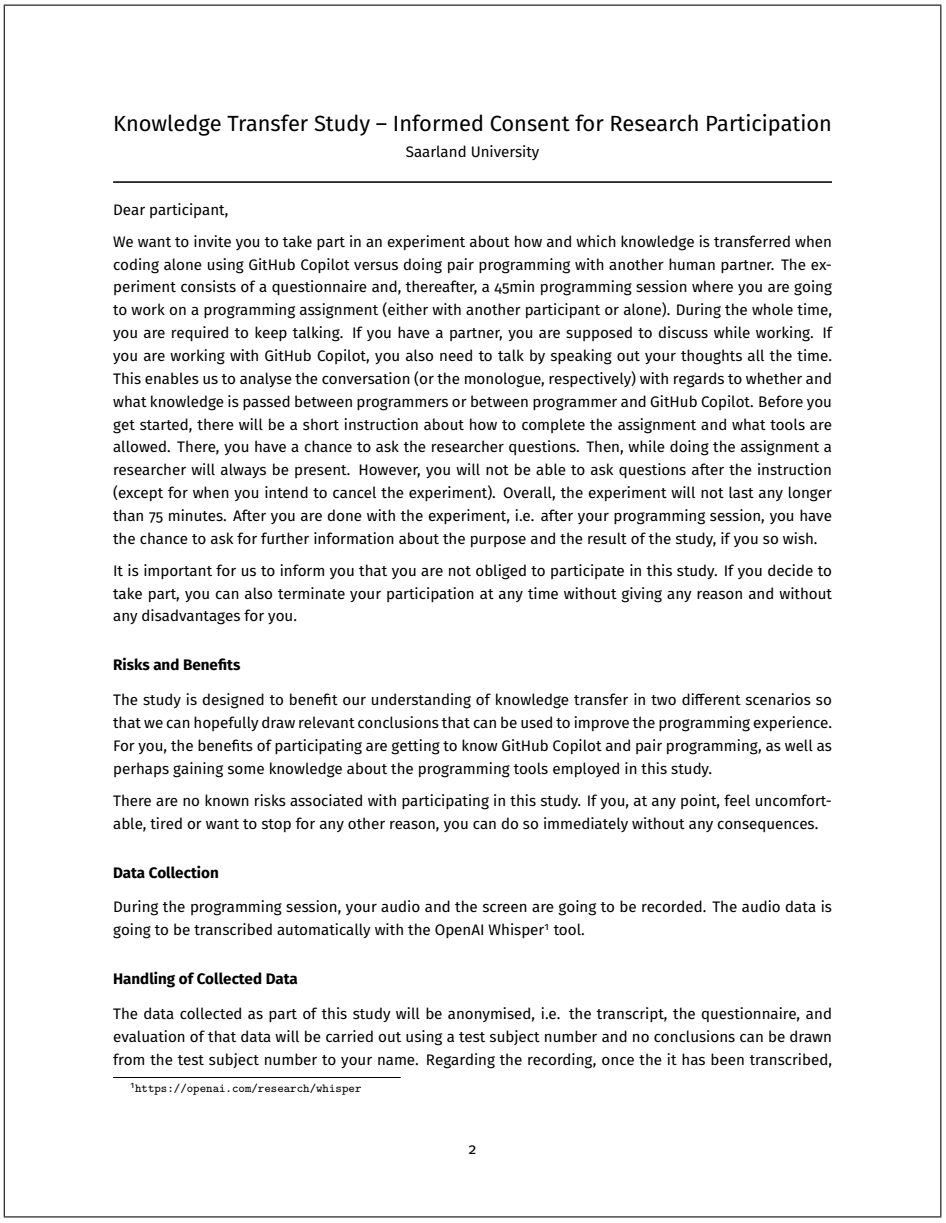
(D) How do you rate your proficiency with the English language?

novice | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | *fluent speaker*

(E) What is your study programme? _____

Abbildung A.1: Fragebogen der Studie

A.2 EINVERSTÄNDNISERKLÄRUNG



Knowledge Transfer Study – Informed Consent for Research Participation

Saarland University

Dear participant,

We want to invite you to take part in an experiment about how and which knowledge is transferred when coding alone using GitHub Copilot versus doing pair programming with another human partner. The experiment consists of a questionnaire and, thereafter, a 45min programming session where you are going to work on a programming assignment (either with another participant or alone). During the whole time, you are required to keep talking. If you have a partner, you are supposed to discuss while working. If you are working with GitHub Copilot, you also need to talk by speaking out your thoughts all the time. This enables us to analyse the conversation (or the monologue, respectively) with regards to whether and what knowledge is passed between programmers or between programmer and GitHub Copilot. Before you get started, there will be a short instruction about how to complete the assignment and what tools are allowed. There, you have a chance to ask the researcher questions. Then, while doing the assignment a researcher will always be present. However, you will not be able to ask questions after the instruction (except for when you intend to cancel the experiment). Overall, the experiment will not last any longer than 75 minutes. After you are done with the experiment, i.e. after your programming session, you have the chance to ask for further information about the purpose and the result of the study, if you so wish.

It is important for us to inform you that you are not obliged to participate in this study. If you decide to take part, you can also terminate your participation at any time without giving any reason and without any disadvantages for you.

Risks and Benefits

The study is designed to benefit our understanding of knowledge transfer in two different scenarios so that we can hopefully draw relevant conclusions that can be used to improve the programming experience. For you, the benefits of participating are getting to know GitHub Copilot and pair programming, as well as perhaps gaining some knowledge about the programming tools employed in this study.

There are no known risks associated with participating in this study. If you, at any point, feel uncomfortable, tired or want to stop for any other reason, you can do so immediately without any consequences.

Data Collection

During the programming session, your audio and the screen are going to be recorded. The audio data is going to be transcribed automatically with the OpenAI Whisper[1] tool.

Handling of Collected Data

The data collected as part of this study will be anonymised, i.e. the transcript, the questionnaire, and evaluation of that data will be carried out using a test subject number and no conclusions can be drawn from the test subject number to your name. Regarding the recording, once the it has been transcribed,

[1] https://openai.com/research/whisper

2

Abbildung A.2: Einverständniserklärung der Studie (Seite 1)

it will neither be used any further for the evaluation, nor will it be part of any publication resulting from this experiment.

Within the framework of the legal data protection regulations of the European Data Protection Regulation (DSGVO), your data will be used exclusively for research purposes and stored for at least ten years. This also includes the publication of your data anonymised in this way as well as further use by third parties for research purposes, the exact purpose and scope of which cannot be foreseen at this point in time.

Access to research data is granted to employees, interns of the Chair of Software Engineering at Saarland University and students assigned to the project for the periods described above and for exclusively scientific purposes.

In accordance with the DSGVO, you have the right to request information about your personal data at any time and to demand that it be corrected or deleted. Furthermore, you can revoke your consent at any time, provided that the storage of the data is not required for legal or contractual purposes (e.g. as part of the declaration of consent).

Handling of Personal Data

Your personal data (first name and surname) will be processed in three documents as part of the experiment. Your first and last name will be collected for the purpose of your consent to participate in this study on the following consent form. This will be kept for a period of at least 10 years.

Your first and last name will continue to be kept in the form of a receipt until the time of settlement for the purpose of settling the trial subject fees and will be forwarded to the department of Saarland University responsible for settlement.

Your first and last name will continue to be kept in the form of a list of participants in the study until the end of the overall project (maximum 5 years). The list only assigns you to the study itself and not to your individual study record.

If you have any further questions about this study, you can contact the investigator or project leader at any time (see the contact details below).

Responsible for the project:

Niklas Schneider
Lehrstuhl für Software Engineering
Saarland Informatics Campus
Campus Geb. E 1.1 2. Stock
66123 Saarbrücken
T: +49 (0) 681 302 57210
www.se.cs.uni-saarland.de

(Seite 2)

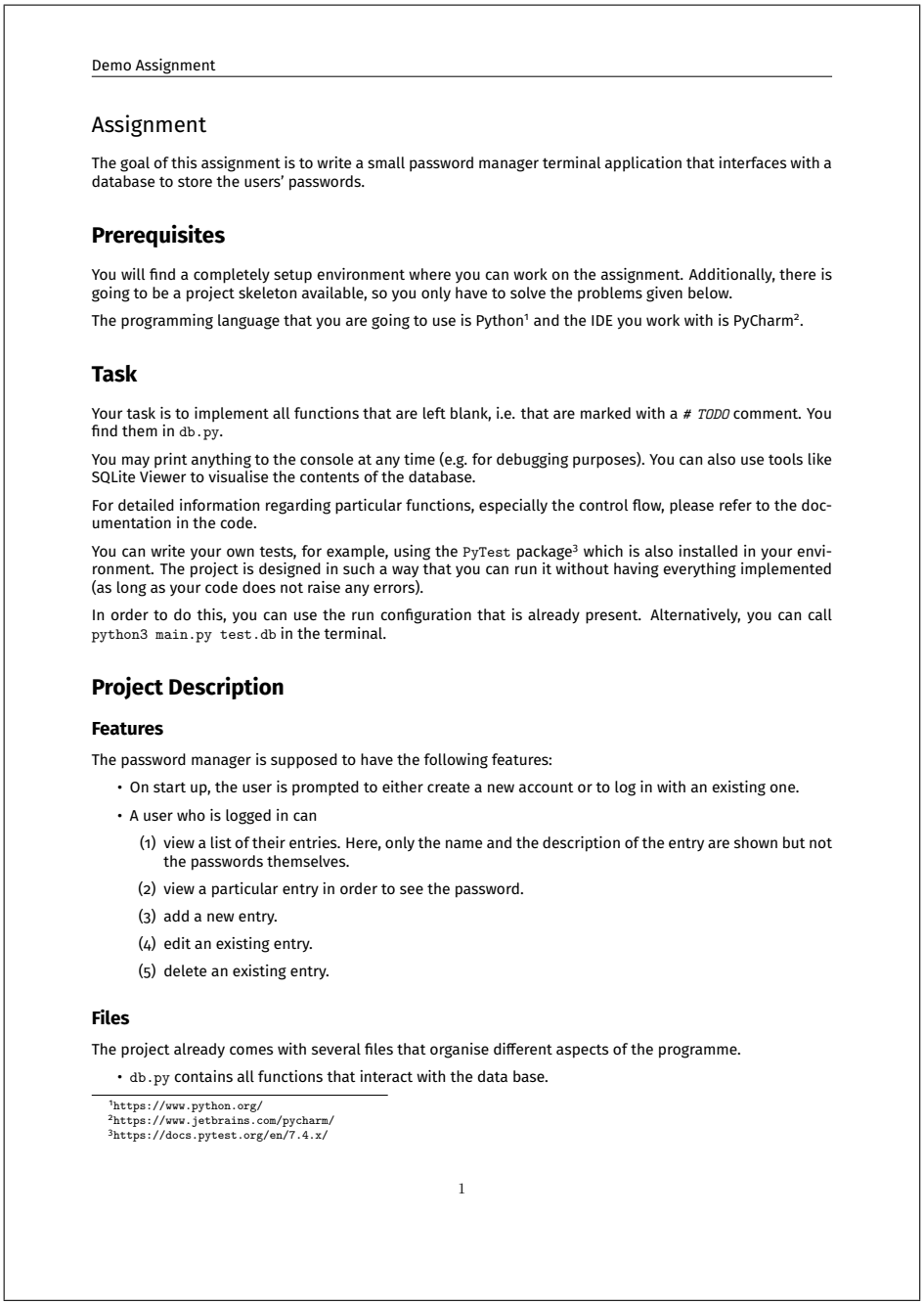
Knowledge Transfer Study – Declaration of Consent

Saarland University

- I have read the information sheet about the study: about the processes and tasks of this experiment and the voluntary nature of my participation, as well as the information sheets on handling collected data.
- I know that I am taking part in a research project in which I am working on a programming assignment.
- Furthermore, I know that I have to keep talking with my programming partner or think out loud for the whole programming session.
- I have been informed that the experiment will last no longer than a total of **75 minutes**.
- I understand that the information obtained through my participation in the study will be used for scientific purposes only.
- I have been informed that my participation in this study is completely **voluntary**.
- I can **cancel at any time without giving reasons** and without any disadvantages for me.
- If any inconvenience or discomfort arises during the study, I can stop and/or discuss my concerns with the investigator.
- Except for the two reasons above, I know that I cannot ask any questions after the programming session has begun.
- I have been informed that all information obtained through my participation in the study, including how I completed the tasks, will be **stored anonymously** so that no conclusions can be drawn about my person.
- I have been informed that my anonymised data may be stored for at least ten years for research purposes. This also includes the publication of my data anonymised in this way as well as further use by third parties for research purposes, the exact purpose and scope of which cannot be foreseen at this point in time.
- I have been informed that my personal data in the form of first and last names will be kept for at least 10 years as part of this consent form.
- In addition, my first and last name will be kept in the form of a list of participants in the study until the end of the overall project or for a maximum of 10 years.
- I know that access to all these described data will be granted to employees, interns of the Chair of Software Engineering at Saarland University and students assigned to the project in the context of their final thesis.
- I have been informed about my rights in terms of the GDPR.
- I have also been informed that I can receive additional information after the study if I so wish.
- I also know that I must treat the information given to me about the course of the experiment as confidential and must not pass it on to potential subjects.

(Seite 3)

A.3 AUFGABENSTELLUNG



Demo Assignment

Assignment

The goal of this assignment is to write a small password manager terminal application that interfaces with a database to store the users' passwords.

Prerequisites

You will find a completely setup environment where you can work on the assignment. Additionally, there is going to be a project skeleton available, so you only have to solve the problems given below.

The programming language that you are going to use is Python[1] and the IDE you work with is PyCharm[2].

Task

Your task is to implement all functions that are left blank, i.e. that are marked with a `# TODO` comment. You find them in `db.py`.

You may print anything to the console at any time (e.g. for debugging purposes). You can also use tools like SQLite Viewer to visualise the contents of the database.

For detailed information regarding particular functions, especially the control flow, please refer to the documentation in the code.

You can write your own tests, for example, using the `PyTest` package[3] which is also installed in your environment. The project is designed in such a way that you can run it without having everything implemented (as long as your code does not raise any errors).

In order to do this, you can use the run configuration that is already present. Alternatively, you can call `python3 main.py test.db` in the terminal.

Project Description

Features

The password manager is supposed to have the following features:

- On start up, the user is prompted to either create a new account or to log in with an existing one.
- A user who is logged in can
  (1) view a list of their entries. Here, only the name and the description of the entry are shown but not the passwords themselves.
  (2) view a particular entry in order to see the password.
  (3) add a new entry.
  (4) edit an existing entry.
  (5) delete an existing entry.

Files

The project already comes with several files that organise different aspects of the programme.

- `db.py` contains all functions that interact with the data base.

[1] https://www.python.org/
[2] https://www.jetbrains.com/pycharm/
[3] https://docs.pytest.org/en/7.4.x/

1

Abbildung A.3: Aufgabenstellung im Repository der Teilnehmer (Seite 1)

- `main.py` contains the entry point as well as the initialisation code.
- `models.py` contains the classes of the data model as described in the paragraph above.
- `option.py` contains code that capsules some of the logic of terminal menu screens.
- `terminal.py` contains the actual control flow and user interaction in form of a state machine.

Model

The data model consists of two different types of objects:

- `User` describes an account that can be used to log in to the password manager.
  - **id**: unique identifier for the data base
  - `name`: the name of the user
  - `password`: the password *hash* of the user
  - `entries`: a list of `Entry` objects that the user has created
- `Entry` contains information about a password a user wants to store.
  - **id**: unique identifier for the data base
  - `name`: a short description of the password
  - `info`: a longer description of the password
  - `password`: the password in plain text
  - `user`: the `User` object the entry belongs to
  - `user_id`: the id of the user the entry belongs to (needed for the data base model)

Packages

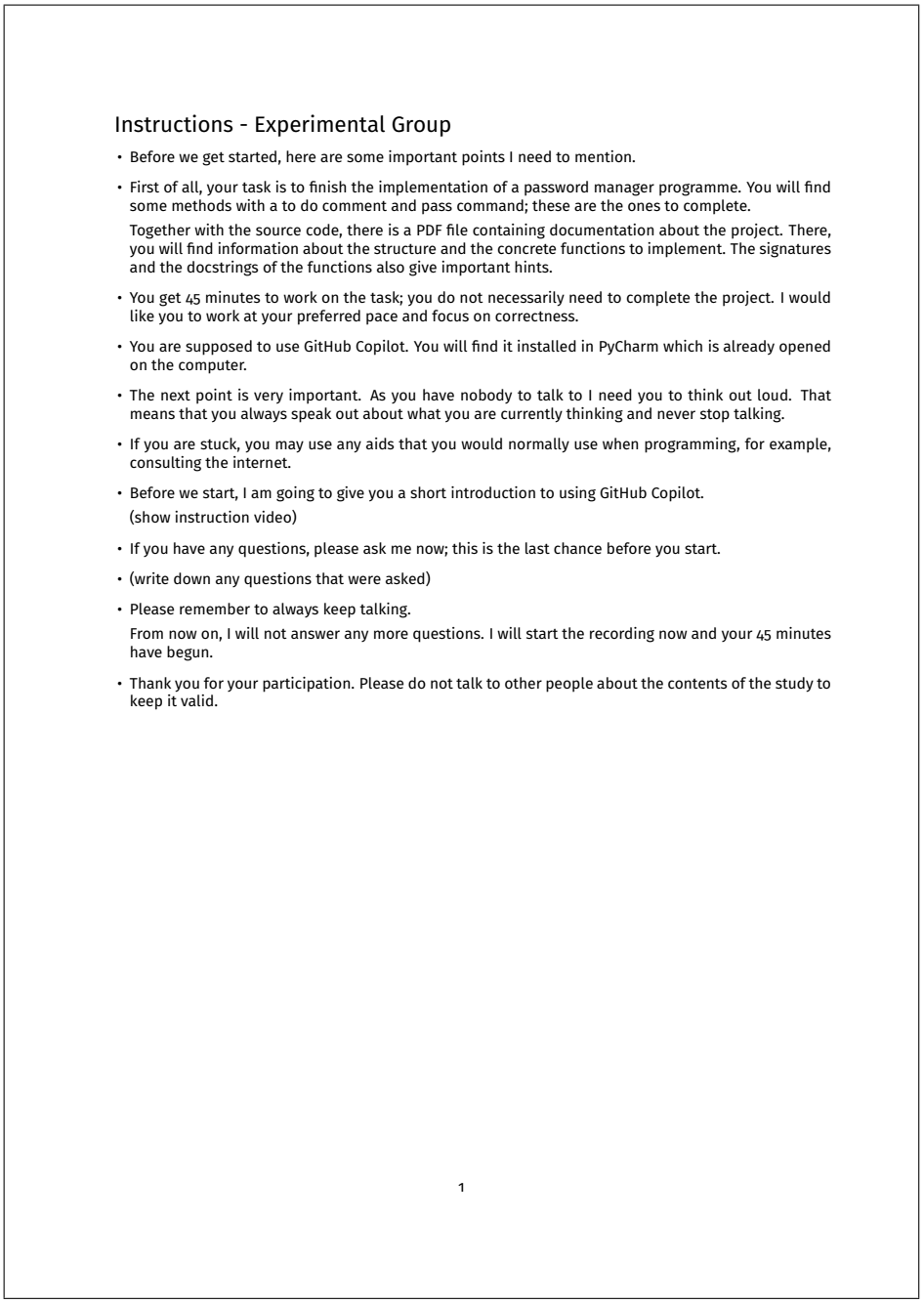
The project skeleton already uses some packages, these should suffice in order to complete the assignment. However, if you deem necessary, you may add additional packages to your liking. Packages already in use are

- `click` is used to pass command line parameters to the application.
- `sqlalchemy` is used to interface with a database.
- `colorama` is used for colourful terminal text.
- `functools` is used to access useful helper functions.
- `hashlib` is used to create password hashes.

All these packages are already installed in your environment.

(Seite 2)

A.4 INSTRUKTIONEN



Instructions - Experimental Group

- Before we get started, here are some important points I need to mention.
- First of all, your task is to finish the implementation of a password manager programme. You will find some methods with a to do comment and pass command; these are the ones to complete.

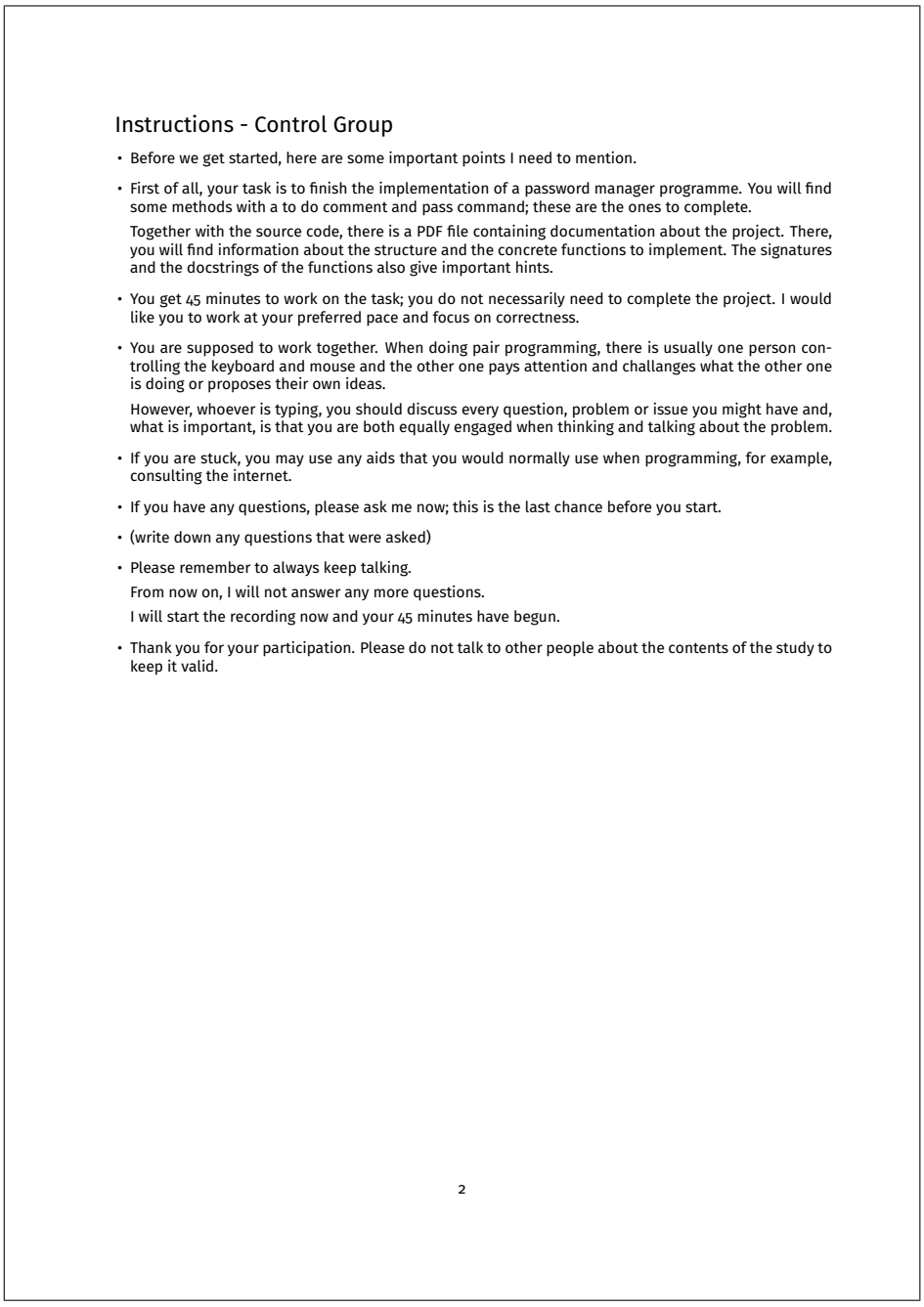
  Together with the source code, there is a PDF file containing documentation about the project. There, you will find information about the structure and the concrete functions to implement. The signatures and the docstrings of the functions also give important hints.
- You get 45 minutes to work on the task; you do not necessarily need to complete the project. I would like you to work at your preferred pace and focus on correctness.
- You are supposed to use GitHub Copilot. You will find it installed in PyCharm which is already opened on the computer.
- The next point is very important. As you have nobody to talk to I need you to think out loud. That means that you always speak out about what you are currently thinking and never stop talking.
- If you are stuck, you may use any aids that you would normally use when programming, for example, consulting the internet.
- Before we start, I am going to give you a short introduction to using GitHub Copilot.

  (show instruction video)
- If you have any questions, please ask me now; this is the last chance before you start.
- (write down any questions that were asked)
- Please remember to always keep talking.

  From now on, I will not answer any more questions. I will start the recording now and your 45 minutes have begun.
- Thank you for your participation. Please do not talk to other people about the contents of the study to keep it valid.

1

Abbildung A.4: Instruktionen für die Experimentalgruppe



Instructions - Control Group

- Before we get started, here are some important points I need to mention.
- First of all, your task is to finish the implementation of a password manager programme. You will find some methods with a to do comment and pass command; these are the ones to complete.

  Together with the source code, there is a PDF file containing documentation about the project. There, you will find information about the structure and the concrete functions to implement. The signatures and the docstrings of the functions also give important hints.
- You get 45 minutes to work on the task; you do not necessarily need to complete the project. I would like you to work at your preferred pace and focus on correctness.
- You are supposed to work together. When doing pair programming, there is usually one person controlling the keyboard and mouse and the other one pays attention and challanges what the other one is doing or proposes their own ideas.

  However, whoever is typing, you should discuss every question, problem or issue you might have and, what is important, is that you are both equally engaged when thinking and talking about the problem.
- If you are stuck, you may use any aids that you would normally use when programming, for example, consulting the internet.
- If you have any questions, please ask me now; this is the last chance before you start.
- (write down any questions that were asked)
- Please remember to always keep talking.

  From now on, I will not answer any more questions.

  I will start the recording now and your 45 minutes have begun.
- Thank you for your participation. Please do not talk to other people about the contents of the study to keep it valid.

2

Abbildung A.5: Instruktionen für die Kontrollgruppe