Wissenschaftliche Hausarbeit im Rahmen der Ersten Staatsprüfung für das Lehramt an Gymnasien im Fach Informatik, eingereicht der Hessischen Lehrkräfteakademie – Prüfungsstelle Darmstadt – .

Thema: Erklären können – Fachdidaktische Schulung von Tutor*innen für Programmieranfänger*innen

Verfasser/in: Svana Esche, Kiesstraße 29, 64283 Darmstadt
Gutachter/in: Prof. Dr. Karsten Weihe, Dr. Guido Rößling

TECHNISCHE UNIVERSITÄT
DARMSTADT

WISSENSCHAFTLICHE HAUSARBEIT
ERSTE STAATSPRÜFUNG

Erklären können – Fachdidaktische Schulung von Tutor*innen für Programmieranfänger*innen

---

Autorin:
Svana ESCHE

Gutachter:
Prof. Dr. Karsten WEIHE
Dr. Guido RÖSSLING

4. Dezember 2018

TECHNISCHE UNIVERSITÄT DARMSTADT

Zusammenfassung

Fachbereich Informatik

Erste Staatsprüfung

Erklären können – Fachdidaktische Schulung von Tutor*innen für Programmieranfänger*innen

von Svana ESCHE

Diese Arbeit handelt von der vollständigen Beschreibung einer konkreten Tutor*innenschulung, welche das Ziel verfolgt, das Erklären können zu fördern. Die Tutor*innenschulung beruft sich auf ein breites wissenschaftliches Fundament in der Fachliteratur zur Fachdidaktik der Informatik. Hierbei sind insbesondere die Vorstellungen, Fehler und informatikspezifischen Strategien wegweisend.
Zuerst wird die bisherigen Situation im Fach *Funktionale und Objektorientierte Programmierung* im 1. Semester an der TU Darmstadt – welche als Repräsentant dient – analysiert. Danach wird sich dem systematischen Vorstellen der zugehörigen Fachliteratur zu Programmieranfänger*innen gewidmet. Anschließend wird die methodische Umsetzung der Tutor*innenschulung mit im Kontext dieser Arbeit erstelltem Material vorgestellt. Als Abschluss wird der Transfer von den Erkenntnissen dieser Arbeit auf den Informatikunterricht und insbesondere auf die Verwendung durch die Lehrer*innen gezogen.

Danksagung

An erster Stelle gilt mein Dank meinen Betreuern Herr Prof. Dr. Karsten Weihe und Dr. Guido Rößling für ihre wissenschaftliche und methodische Unterstützung während der gesamten Bearbeitungsphase meiner Wissenschaftlichen Hausarbeit.
Frau Sabine General aus der hochschuldidaktischen Arbeitsstelle und Frau Jacqueline Gölz, die für die tutorielle Lehre am Fachbereich Informatik zuständig ist, danke ich für die fachlichen Gespräche, Ratschläge und Anmerkungen, die mich auf dem Weg zur fertigen Arbeit immer wieder neue Aspekte und Ansätze entdecken ließen.
Des Weiteren bin ich Herrn Florian Kadner für die gute und zahlreiche Unterstützung, sowie die konstruktive und angenehme Zusammenarbeit, sehr dankbar.
Besonders möchte ich an dieser Stelle auch meiner Familie und meinen Freunden für die unermüdliche Stärkung und Förderung danken, sowie für das stets offene Ohr für meine Gedanken.

Inhaltsverzeichnis

Zusammenfassung **i**

1 Erste Annäherung **1**
1.1 Eingrenzung des Themas 1
1.1.1 Programmieranfänger*innen 1
1.1.2 Erklären können – Fachdidaktische Schulung . . 2
1.1.3 Tutor*innen 3
1.2 Analyse der Situation 3
1.2.1 Übergang von Schule zur Hochschule 3
1.2.2 Methodische Entscheidungen 5

2 Wissenschaftliches Fundament **7**
2.1 Diagnose und diagnostische Kompetenz 8
2.2 Programmieranfänger*innen 10
2.3 Mentale Modelle 11
2.3.1 Variablen 14
2.3.2 Iteration und Schleifen 16
2.3.3 Rekursion 17
2.3.4 Objektorientierung 20
2.4 Fehler von Programmieranfänger*innen 23
2.4.1 Syntaktisches Wissen 24
2.4.2 Konzeptuelles Wissen 25
2.4.3 Strategisches Wissen 29
2.4.4 Häufigkeit von Fehlern 30
2.5 Problemlösestrategien der Informatik 32
2.5.1 Ursprung des Problemlösens 32
2.5.2 Patterns und Strategien 33

3 Tutor*innenschulung **39**
3.1 Analyse der Lerngruppe 39
3.2 Lernziele 40
3.3 Methodischer Ablauf – vormittags 41

3.3.1 Eingangsdiagnose 42
3.3.2 Einstieg 47
3.3.3 Visualisierungen und Mentale Modelle 49
3.3.4 Typische Fehler und Fehlvorstellungen 60
3.3.5 Simulation 66
3.4 Methodischer Ablauf - nachmittags 70
3.4.1 Problemlösestrategien 70
3.4.2 Rekursion 76
3.4.3 Rückschau 82

4 Transfer zur Institution Schule **84**
4.1 Vorerfahrungen der Subjekte 84
4.2 Verwendbarkeit der Schulung 85

A Grundlagen zum Material **88**
A.1 Problemklassen nach Garner u. a. (2005) 88
A.2 Strategien nach Kaplan (2012) 93

B Material zur Tutorenschulung **96**
B.1 Übersichtsplan 97
B.2 Folien 98
B.2.1 Einstieg 98
B.2.2 Visualisierungen und Gedankliche Modelle 104
B.2.3 Typische Fehler und Fehlvorstellungen 115
B.2.4 Problemlösestrategien 126
B.2.5 Rekursion 136
B.2.6 Rückblick 142
B.3 Arbeitsblätter 144
B.3.1 Wissensstand diagnostizieren 144
B.3.2 Typische Fehler nach Robins et al. (2003) 148
B.3.3 Simulationen 151
B.3.4 Problemlösestrategien 156
B.3.5 Test zur Rekursion 161

Literaturverzeichnis **166**

Eidesstattliche Erklärung **175**

Abbildungsverzeichnis

2.1 Graphische Beschreibung des Kopien-Modells (Kahney, 2009, 210) 18
2.2 Absolute Anzahl der jeweiligen Problemen nach Nummer (Garner u. a., 2005, 4) 31
2.3 Kontinuum der informatikspezifischen und allgemeinen Problemlösestrategien (Kaplan, 2012, 4) 35

3.1 Beispiel zu allgemeinen Techniken der Visualisierung von Programmcode Gailer (2018b) 54
3.2 Kopien bei der rekursiven Methode `printFun`, angelehnt an GeeksforGeeks (2018) 78

Kapitel 1

Erste Annäherung

Zuerst wird sich in diesem Kapitel dem Titel der Arbeit auf erste Weise angenähert, indem das zugehörige Thema eingegrenzt und auf die Situation an der TU Darmstadt bezogen wird. Die Annäherung geschieht über eine Aufteilung der Bestandteile des Titels und eine erste Bezugnahme auf die zugehörige Fachliteratur. Anschließend wird die bisherige Situation der Programmieranfänger*innen auf ihre Besonderheiten, methodische Entscheidungen und deren Wirksamkeit analysiert. Diese Situation steht unmittelbar in Beziehung zur Tutor*innenschulung und erhält dadurch besondere Bedeutung.

1.1 Eingrenzung des Themas

Von der Spezifikation des Titels her wird dieser zuerst in seine Bestandteile aufgeteilt, um dann die Elemente einzeln einzugrenzen und zu konkretisieren. Zusammen ergibt sich dann die spezifische Eingrenzung des Themas.

1.1.1 Programmieranfänger*innen

In Rahmen des Bachelor-Studiengangs Informatik an der TU Darmstadt werden die Programmieranfänger*innen durch Teilnehmer*innen der Lehrveranstaltung des Moduls *Funktionale und objektorientierte Programmierkonzepte* – im Nachfolgenden mit *FOP* abgekürzt – repräsentiert. Dies begründet sich durch die Formulierung der Lerninhalte des Moduls *FOP*, welche nach dem Modulhandbuch unter Anderem die „Vermittlung grundlegender Begriffe der Informatik, sowie Entwicklung einfacher Programmierfähigkeiten“ umfassen

(TU Darmstadt, 2015, 7). Somit kann dieses Modul als analog zu Kursen an anderen Universitäten angesehen werden, die international unter den folgenden Stichworten gefasst werden: *Computer Science 1 – CS1*, *novice programming* bzw. *novice teaching* oder auch *computer science introductory programming course* siehe (Anthony u. a., 2003, 137–138) bzw. *introductory computer programming* oder *entry-level programming course* siehe (Qian und Lehman, 2017, 3).

1.1.2 Erklären können – Fachdidaktische Schulung

Der inhaltliche Fokus der Schulung liegt in der Vermittlung der Fähigkeit *Erklären können* unter fachdidaktischen Gesichtspunkten. Um gut erklären zu können, benötigen die Tutor*innen „gegenstandsspezifische[s] fachdidaktische[s] Hintergrundwissen“ (Heinrichs, 2015, 43), da dieses *typische Fehler, Schwierigkeiten, Grund- und Fehlvorstellungen* umfasst. Erst unter dieser Vorausetzung kann eine passende Erklärung für die Programmieranfänger*innen erfolgen, welche sich auf professionelles Hintergrundwissen stützt. Um den Tutor*innen eine Grundlage für ein erfolgreiches *Erklären können* zu liefern, müssen genau die genannten Inhalte nach Heinrichs wesentlicher Teil der Schulung sein. Welche konkreten Inhalte aus Sicht der Informatikdidaktik die Fehler und Vorstellungen der Programmieranfänger*innen sind, wird in Kapitel 2 – Wissenschaftliches Fundament – erläutert. Des Weiteren ist es vonnöten, dass das eigentliche Erklären seitens der Tutor*innen und das zugehörige Einüben seinen Platz innerhalb der Schulung findet. Die genaue Realisierung wird in Kapitel 3 präzisiert. Durch den Fokus auf eine *fach*didaktische Schulung, wird in der Arbeit der Bereich der allgemeinen Methodik und Didaktik außen vor gelassen. Darunter würden nach Klafki (1986) bei der Didaktik die Ziele und die zugehörigen Inhalte – das *Was* – und bei der Methodik die verwendeten Methoden und Medien – das *Wie* – fallen. Beispielhaft gesprochen werden die Tutor*innen nicht darin geschult, die Inhalte als Bereiche selbst auszuwählen und die Methoden und Medien wie Tafelanschrieb und Prinzipien der Kommunikation kennen zu lernen und einzuüben. Letzteres wird bereits in der allgemeinen Tutor*innenschulung unter Leitung von Frau Dr. Jaqueline Gölz am Fachbereich Informatik aufgegriffen.

1.1.3 Tutor*innen

Die Veranstaltungsform der *FOP* wird als „integrierte Lehrveranstaltung“ (TU Darmstadt, 2015, 7) angegeben, und bezeichnet demnach eine Mischform zwischen Vorlesung, Übung und Projekt. Diese wird aktuell von Prof. Dr. Karsten Weihe gehalten und beinhaltet knapp drei Dutzend zugehörige Übungen in Kleingruppen. Die Kleingruppen werden von die Tutor*innen gehalten, welche sich selbst noch studieren. Aufgrund der Tatsache, dass der Rahmen der Tätigkeit der Tutor*innen hauptsächlich die Betreuung der Übungen umfasst, soll sich inhaltlich auf diese konzentriert werden. Auf die Vorlesung und das zugehörige Projekt der *FOP* wird daher in dieser Arbeit nicht weiter eingegangen.

Die Lehrveranstaltung *FOP* findet zu jedem Wintersemester statt. Daher wird die in dieser Arbeit entwickelte Schulung auch an einer konkreten Lerngruppe von Tutor*innen zu Beginn des Wintersemesters von mir durchgeführt werden. Somit beziehen sich die Materialien zur Durchführung der Tutor*innenschulung auf eine konkrete Lerngruppe. Diese Lerngruppe steht jedoch exemplarisch für jede Gruppe von Tutor*innen, die Übungen, Sprechstunden bzw. Tutorien zu einer Veranstaltung für Programmieranfänger*innen betreuen. Die Beschreibung der Lerngruppe wird im Abschnitt 3.1 vorgenommen.

1.2 Analyse der Situation

Anschließend wird die Situation der Teilnehmer*innen der *FOP* unter den folgenden Gesichtspunkten untersucht: der Studieneingangsphase – dem Wechsel der Schule zur Hochschule – und den bisherigen methodischen Entscheidungen – Anzahl der Teilnehmer*innen der Übungen und deren Vorkenntnisse.

1.2.1 Übergang von Schule zur Hochschule

Die Veranstaltung *FOP* ist für Student*innen des ersten Semesters geplant. Die jeweiligen Studiengänge der Student*innen sind vielfältig, daher werden hier nur die häufigsten genannt: BSc. Informatik, BSc. Wirtschaftsinformatik, BSc. Mathematik mit Nebenfach Informatik, BSc. Wirtschaftsmathematik, Lehramt an Gymnasien Informatik, BSc. Psychologie in IT, Joint Bachelor of Arts Informatik. Die wesentliche

Gemeinsamkeit der Student*innen ist, dass sie sich in der Studieneingangsphase befinden. Diese Studieneingangsphase und damit der Übergang von Schule zur Hochschule hat eine herausragende Stellung innerhalb der Bewältigung eines Studiums inne. Dies zeigt sich dadurch, dass aus Retrospektive gesehen, das Risiko eines Studienabbruchs sich bereits zu Studienbeginn abzeichnet (Heublein u. a., 2017, 124) und die meisten Studienabbrüche im ersten Studienjahr erfolgen (Heublein u. a., 2017, 47).

Bei der Didaktik der Informatik wird sich in vielen Teilen an die bereits bestehende Didaktik der Mathematik angelehnt – als Beispiel sollen hier die fundamentalen Ideen und der Problemlöseprozesse genannt werden. In der Studieneingangsphase ist das Übertragen der spezifischen Erkenntnisse der Mathematikdidaktik nicht sinnvoll. So ist der Übergang von Schule zur Hochschule im Fach Mathematik durch die Vorerfahrungen im Fach geprägt. Jede*r der Student*innen hat bereits meist zwölf oder dreizehn Jahre Erfahrung mit Mathematik in Form des Mathematikunterrichts erfahren und verfügt damit zumindest über ein Mindestmaß an Vorwissen verfügt. In der Informatik hingegen ist dies nicht unbedingt der Fall. So gibt es beispielsweise im Bundesland Hessen kein Pflichtfach in der Schule, welches informatische Inhalte und Kompetenzen vermittelt. Daher ist das Vorwissen der Student*innen äußert heterogen – vom Beherrschen von mehreren Programmiersprachen bis hinzu gar keinen Erfahrungen im Bereich des Programmierens.

Im Wintersemester 2017/18 haben unter allen Teilnehmer*innen der Veranstaltung *FOP* ein Drittel gewissermaßen keine Vorkenntnisse im Bereich des Programmierens. Keine Vorkenntnisse entsprechen bei der Umfrage, dass die Anzahl der vor dem Studium mit Programmieren verbrachten Stunden sich auf 0 bis 10 belief. Dies ergab sich durch eine Umfrage, welche von Prof. Dr. Karsten Weihe durchgeführt wurde. Die Daten beruhen auf einer persönlichen Kommunikation mit Prof. Dr. Karsten Weihe. Hier wird sich auf das Programmieren bezogen, da die Veranstaltung *FOP* das Vermitteln dessen im Fokus hat. Die Tatsache, dass Informatik mehr als Programmieren ist, soll an dieser Stelle keinesfalls bestritten, aber auch nicht weiter ausgeführt werden, da die Veranstaltung für *Programmieranfänger*innen* gedacht ist.

1.2.2 Methodische Entscheidungen

Die von den Tutor*innen betreuten Kleingruppenübungen sollen die Vorlesung ergänzen. So erhalten die Student*innen die Möglichkeit, ihre ersten, konkreten Programmiererfahrungen in einem begleiteten Rahmen zu erleben. Die Begleitung wird maßgeblich durch die Qualität der Tutor*innen, die zur Verfügung stehende Zeit und die Auswahl der Übungsaufgaben beeinflusst.

Die erlebte Wirklichkeit weicht jedoch an mehreren Punkten vom angestrebten Ziel ab. Die Masse der Student*innen – 1178 im zugehörigen Moodle-Kurs im Wintersemester 2017/18 – vermindern die individuelle Zeit zur Betreuung durch die Tutor*innen erheblich. Zu Beginn des genannten Semesters hat die durchschnittliche Teilnehmer*innenzahl einer Übung rechnerisch bei circa 36 Student*innen gelegen. Die angesetzte Zeitdauer einer Übung beträgt 90 Minuten. Dies entspricht für jede einzelne Person ungefähr 2,5 Minuten Zeit für ein individuelles Gespräch mit dem*r Tutor*in. Daher ist anzunehmen, dass die Tutor*innen nur wenige Möglichkeiten haben, individuelle Hilfestellungen zu leisten, um die Programmiererfahrungen der Student*innen zu begleiten. Die rechnerischen 2,5 Minuten umfassen ja nicht nur das Erklären. Darunter fallen auch der Austausch, welche Aufgabenstellung bearbeitet wird, welche Lösungsideen bereits vorliegen und in welcher Frage an den*die Tutor*in dies resultiert. Das Erleben der überfüllten Übungen unterscheidet sich für Student*innen nicht viel vom Erleben der ungesehenen Anonymität einer Vorlesung. Dies kann als einer der möglichen Gründe angeführt werden, warum weniger Student*innen als durch reine Rechnung die Übungen besuchen: Sie werden nicht von allen Student*innen als hilfreich und unterstützend angesehen. Dies zu ändern soll ein Effekt der Schulung sein.

Die Gesamtheit der Student*innen, welche die Veranstaltung *FOP* besuchen, ist bezüglich ihres Vorwissens extrem heterogen. Die durchgeführten Umfragen von Herrn Weihe aus dem Wintersemester 2017/18 haben ergeben, dass sich die Stundenanzahl der Programmierleistungen vor dem Studium zwischen 0 und über 1000 Stunden befindet. Hierbei lassen sich ein Drittel der Student*innen dem Bereich der nicht existenten Vorerfahrungen zuordnen, welche zwischen

0 und 10 Stunden im Vorfeld programmiert hatten wie bereits dargestellt. Die Bedeutung der Vorerfahrungen begründet sich unter Anderem dadurch, dass aus der Umfrage eine Korrelation zwischen den Vorerfahrungen und den Leistungen in der Klausur abzuleiten war. Die bestehende Heterogenität der Student*innen wurde bei der Konzeption der Übungen bisher nicht berücksichtigt. Dies kann als mögliche Erklärung für folgende, auftauchende Probleme angeführt werden. Die Student*innen mit Vorwissen besuchen die Übungen nicht. Für diejenigen Student*innen, die die Qualität ihres Vorwissens überschätzen, kann das fatal sein. Möglicherweise verpassen sie den Zeitpunkt , ab wann sie die Übungen besuchen sollten, um nicht vom Stoff überflügelt zu werden. Die Student*innen ohne Vorwissen bzw. Student*innen, die den Inhalten nicht in der vorgegebenen Geschwindigkeit folgen können, besuchen die Übungen nicht. Möglicherweise erhalten sie – aufgrund der begrenzten Zeit des*r Tutor*in – nicht die benötigten Hilfestellungen. Des Weiteren kann es einschüchternd sein, das Gefühl zu haben, dass alle anderen Student*innen, die die Übung besuchen, scheinbar mühelos den Inhalten folgen können.

Es gab bisher keine fachdidaktische Schulung am Fachbereich Informatik. Es findet lediglich eine allgemeinpädagogische Schulung für am Fachbereich beschäftigte Tutor*innen statt. Die Anzahl der Student*innen des Studiengangs Lehramt Informatik, die als Tutor*innen tätig sind, ist fast nicht existent. Daher spielen deren erworbenen, fachdidaktischen Kompetenzen keine Rolle. Ein erster Schritt zu einer möglicherweise verbreiteten fachdidaktischen Schulung soll durch diese Arbeit erreicht werden. Im Wintersemester 2018/19 wird dies von Seiten der Organisation der Kleingruppenübungen unterstützt. Die Anzahl der Student*innen in den Kleingruppenübungen wurde verringert. Zudem richtet sie sich vorwiegend an Student*innen mit wenig oder keinen Vorkenntnissen. Auf diese Weise sollen diejenigen Student*innen, die mehr Unterstützung bei der Bewältigung der Veranstaltung *FOP* brauchen, einen größeren Anteil der Zeit der Tutor*innen erhalten. Damit würde das erste Mal die Heterogenität der Teilnehmer*innen berücksichtigt werden. Daher benötigen die Tutor*innen eine entsprechende Ausbildung ihrer Fähigkeiten zum Erklären können, damit diese die Probleme der Student*innen in den Kleingruppenübungen aufgreifen können.

Kapitel 2

Wissenschaftliches Fundament

In diesem Kapitel soll die Fachliteratur aufgeführt werden, die das „gegenstandsspezifische[s] fachdidaktische[s] Hintergrundwissen“ (Heinrichs, 2015, 43) ausführt, welches die Tutor*innen – vor allem für das *Erklären können* – benötigen. Darunter versteht Prediger das Wissen über „typische Schülerfehler und Schwierigkeiten in bestimmten Inhaltsbereichen [...] und auch das Wissen über Grund- und Fehlvorstellungen in den jeweiligen Bereichen“ Prediger (2010) zitiert nach (Heinrichs, 2015, 43-44).

Daraus ergibt sich die Gliederung der Abschnitte. Nach einer einführenden Erläuterung der Bedeutung der Diagnose und der diagnostischen Kompetenz in Lehr-Lern-Prozessen (2.1) folgt eine erste Annäherung, was Programmieranfänger*innen auszeichnet (2.2). Weiter werden typische Fehler und Fehlvorstellungen von Programmieranfänger*innen (2.4) beschrieben. Die zu bildenden Grundvorstellungen bzw. mentalen Modelle der Themen der *FOP* führen die Einführung in die Programmierung und deren zugehörigen Visualisierungen (2.3) weiter aus. Abschließend folgt die Abhandlung der zugehörigen allgemeinen und informatikspezifischen Problemlösestrategien – oftmals Patterns oder Schemata genannt (2.5).

Wegweisend für das Zusammentragen der Erkenntnisse der Fachliteratur waren hier vor allem das Werk *Studying the Novice Programmer* von Soloway und Spohrer (2009) und die Übersicht der fachdidaktischen Literatur im Artikel *Learning and Teaching Programming: A Review and Discussion* von Anthony u. a. (2003).

2.1 Diagnose und diagnostische Kompetenz

Die Bedeutung der pädagogischen Diagnose und der diagnostischen Kompetenz von Lehrenden für Lernsituationen wird in der fachdidaktischen Literatur der Informatik nicht erwähnt. Daher soll an dieser Stelle auf die Erkenntnisse der Mathematikdiaktik im Bereich der schulischen Bildung verwiesen werden. Aufgrund der Verwandtschaft von Informatik und Mathematik als Disziplinen im Bereich der Wissenschaften bedient sich die Informatikdidaktik im deutschsprachigen Raum in vielen Bereichen an den Erkenntnissen der Mathematikdidaktik. Exemplarisch soll dafür die Verwendung der fundamentalen Ideen nach dem Kriterienkatalog für die Mathematikdidaktik von Schweiger (1982) zitiert nach (Schubert und Schwill, 2011, 64) genannt werden.

Die theoretische Grundannahme ist hierbei, dass der Grad der Ausprägung professioneller Kompetenz – hier insbesondere die diagnostische Kompetenz – einer Lehrkraft die Qualität der Lehre beeinflusst vgl. (Heinrichs, 2015, 47–48). Schrader und Helmke hingegen zeigen in ihrer empirischen Untersuchung, dass hohe diagnostische Kompetenz für sich alleine noch kein Qualitätskriterium darstellt zitiert nach (Heinrichs, 2015, 48–49). Die Schüler*innen schneiden am besten ab, wenn eine hohe diagnostische Kompetenz der Lehrer*in auf Hilfen treffen, die auf den aktuellen Lern- und Leistungsstand der Schüler*innen abgestimmt sind. Auf die Tutor*innenschulung bezogen entsprechen die Hilfen, die die Tutor*innen den Student*innen bereitstellen, dem *gutem Erklären*. Dass eine hohe diagnostische Kompetenz für sich stehend nicht unbedingt in einem Lernerfolg münden muss, wird ebenfalls in der Studie von Schrader und Helmke gezeigt zitiert nach (Heinrichs, 2015, 48–49). So schnitten diejenigen Schüler*innen am schlechtesten ab, deren Lehrkräfte zwar über ein hohes diagnostisches Potenzial verfügten, den Schüler*innen aber keine adäquaten Hilfen zukommen ließen. Heinrichs (2015) betrachtet sie als Katalysator für guten Unterricht.

Wissen über domänenspezifisches Denken von Student*innen wird als eine einzigartige Dimension der diagnostischen Kompetenz angesehen (Leuders u. a., 2018, 19). Dies unterstützt die Notwendigkeit dieses Hintergrundwissens. Nach Hill u.a. wird das Wissen über Inhalt und Student*innen in die folgenden Bereiche spezifiziert: Wissen über typische Fehler von Student*innen, Verständnis der Student*innen der Inhalte, Sequenz der Entwicklung von Student*innen, typische Strategien von Student*-innen zur Problemlösung. Zudem haben Hill u.a. gezeigt, dass dieses Wissen sich sowohl empirisch als auch theoretisch vom reinen Wissen über den Inhalt unterscheidet und spezifisch trainiert werden kann, siehe Leuders u. a. (2018). Diese Aufzählung deckt sich mit der zuvor aufgeführten Spezifizierung nach Heinrichs (2015) und bestimmt die Inhalte der Tutor*innenschulung.

Im Folgenden soll die englischsprachige Literatur zur Fachdidaktik der Informatik betrachtet werden. Deren examinierten Autor*innen ziehen den Bezug zu den genannten Themen jedoch nicht aus der pädagogischen Diagnostik sondern aus der Informatik. Eine Annahme hierbei ist, dass spezifische, individuelle Diagnosen und Hilfestellungen wahrscheinlich am erfolgreichsten Programmieranfänger*innen unterstützen (Robins u. a., 2001, 21). Hierfür werden jedoch keine empirischen Studien angeführt, die diese Hypothese stützen oder verwerfen lassen. Nach Robins u. a. (2001) ist voraussichtlich das Ausmaß, in welchem hochqualifizierte, auf die jeweilige Person zugeschnittene Hilfestellungen vorhanden sind, der signifikanteste Faktor, welcher den Unterrichtserfolg bei Programmieranfänger*innen beeinflusst. Die Tutor*innen zu hochqualifizierten Lehrenden auszubilden, ist unter Anderem Ziel der Tutor*innenschulung. Das Spektrum der potentiell relevanten Faktoren umfasst Motivation, Selbstvertrauen oder emotionale Reaktionen, Defizite bei allgemeinen oder spezifischen Strategien, bei allgemeinem oder spezifischem Wissen, und bei allgemeinen oder spezifischen mentalen Modellen wie Vorstellungen (Robins u. a., 2001, 42). Hierbei wird klar, dass die Tutor*innenschulung nicht auf alle Faktoren gleichermaßen Rücksicht nehmen kann. Daher wird sich auf die empirisch gestützten Erkenntnisse, wie von Heinrichs (2015) zuvor ausgeführt werden, konzentriert.

2.2 Programmieranfänger*innen

Als erste Annäherung sollen die Erkenntnisse von Studien – zusammengetragen von Winslow (1996) und Soloway und Spohrer (2009) – über die programmiertechnischen Fähigkeiten von Noviz*innen dienen. Diese Studien zeigen die unzulänglichen Fähigkeiten der Programmieranfänger*innen auf, da diese

- auf oberflächliches Wissen beschränkt sind,
- keine detaillierten mentalen Modelle besitzen,
- versagen, relevantes Wissen anzuwenden,
- allgemeine statt informatikspezifische Problemlösestrategien benutzen,
- Zeile-per-Zeile programmieren,
- sehr wenig Zeit nutzen, um zu planen und zu testen,
- häufig ziemlich schlecht darin sind, Code zu tracen und zu tracken. (Winslow, 1996, 18)

Dies zeigt einerseits auf, mit welchen Gegebenheiten die Tutor*innen in den Übungen konfrontiert werden. Andererseits lassen sich daraus die weiteren fachdidaktischen Inhalte wie die Thematisierung von Pattern und Schemata (2.5) als informatikspezifische Problemlösestrategien ableiten. Die vorher bestimmten Inhalte werden teilweise in der Aufzählung der Fähigkeiten aufgegriffen, wobei die mentalen Modelle (2.3) explizit genannt wurden. Damit die Programmieranfänger*innen nicht weiterhin ausschließlich auf oberflächliches Wissen beschränkt sind, ist es besonders wichtig, dass diese Tutor*innen gut vermitteln können. Zudem korrelieren die typischen Fehler und Fehlvorstellungen (2.4) sehr wahrscheinlich mit dem Versagen, relevantes Wissen anzuwenden, und dem Zeile-per-Zeile programmieren, da in diesen Bereichen die aufgeführten Fehler auftauchen werden.

Perkins u. a. (2009) unterscheiden zwischen drei Arten von Student*innen, die als Programmierstile bezeichnet werden.

- *Movers*: Diese versuchen selbst experimentell an ihrem Code zu arbeiten und ihn modifizieren. Feedback über Fehler und Fehlercodes können sie effektiv nutzen und sie besitzen gute Problemlösestrategien.
- *Tinkerers* als *Extreme Movers*: Diese ändern nahezu wahllos ihren Code ab. Sie können ihr Programm nicht tracen und tracken, was als Grund für die nahezu wahllosen Änderungen anzusehen ist.
- *Stoppers:* Diese geben bei jeder kleinen Hürde alle Hoffnung auf und brechen ihre gedanklichen Versuche ab. Damit hören sie schlichtweg auf, das Problem zu lösen. Bei ihnen herrschen vor allem Frust und andere negative Emotionen, wenn Fehler auftauchen.

Die Gemeinsamkeit von *Stoppers* und *Tinkerers* ist, dass sie ineffektiv arbeiten, wobei *Movers* effektiv programmieren und arbeiten. Hierbei soll jedoch angemerkt werden, dass nach Perkins u. a. (2009) *Stoppers* und *Extreme Movers* keineswegs selten vorkommen, sondern häufig sind. Dies wird nicht quantifiziert.

Die Relevanz der Thematisierung der Arten besteht für die Tutor*innen darin, eine angemessene Hilfestellung abhängig von der jeweiligen Student*in auszuwählen. Weiter wird dies in Kapitel 3 ausgeführt.

2.3 Mentale Modelle

Zur Klärung des Begriffs des *mentalen Modells* soll auf folgende Definition zurückgegriffen werden:

> A mental model is the student's understanding of the hidden information process underlying code and what happens inside the computer (Bayman and Mayer 1983). (Qian und Lehman, 2017, 8)

Weigend (2007) unterscheidet jedoch innerhalb der mentalen Modelle zwischen konzeptuellen Modellen – Gedanken eines Individuums

– und mentalen Modellen als fachlich akzeptierte Sichtweise von Expert*innen. Um jedoch adäquat auf die Gedanken der Programmieranfänger*innen einzugehen (2.1), soll sich im Rahmen der Tutor*innenschulung mit dem Verständnis der Programmieranfänger*innen auseinandergesetzt werden. Der Begriff *Vorstellungen* wird hier synonym verwendet, da in der deutschsprachigen Literatur – nebst Weigend (2007) – meist das Begriffspaar *(Fehl-)Vorstellungen* verwendet wird. Dieses beschreibt die Gedankenprozesse und Bilder, die beim Lernen auftauchen. Hierbei ist anzumerken, dass in der Fachliteratur der Begriff der Vorstellung im Gegensatz zum mentalen Modell nicht definiert, sondern schlichtweg verwendet wird. Die Unterscheidung von Weigend (2007) zwischen intuitiven und mentalen Modellen ist für diese Arbeit nicht relevant. So werden in der englischsprachigen Fachliteratur diese ebenfalls nicht unterscheiden. Zudem ist der Grund der falschen Vorstellung für die Tutor*innenschulung nicht von Bedeutung.

Weigend (2007) spricht in seiner Dissertation bei den Erläuterung der Vorstellungen nicht von mentalen sondern von intuitiven Modellen. Seine Unterscheidung ist jedoch für diese Arbeit nicht relevant. Einerseits werden in der englischsprachigen Literatur diese Vorstellungen nicht unterschieden und andererseits ist es unerheblich, ob bei einer falschen Vorstellung einer Programmieranfänger*in dieses auf einem nicht adäquaten Verständnis der Thematik oder auf einer falschen Intuition aufbaut.

Innerhalb der verschiedenen mentalen Modelle sticht eines besonders hervor – *the notional machine*. Dieses wird von insgesamt neun unterschiedlichen Quellen nach Qian und Lehman (2017) und Anthony u. a. (2003) erwähnt. Das Ziel der notional machine ist es, ein Fundament zu bilden, um das Verhalten von laufenden Programmen zu verstehen (Anthony u. a., 2003, 149). Besonders hervorzuheben ist, dass von einem idealisierten Computer ausgegangen wird (Qian und Lehman, 2017, 8). Auffällig ist, dass je nach Literatur die notional machine unabhängig oder abhängig der jeweiligen Programmiersprache aufgefasst wird (Anthony u. a., 2003, 149), (Qian und Lehman, 2017, 8). Aufgrund dessen wird sich hier bei der Definition der notional machine auf den gemeinsamen Kern konzentriert: *Eine notional machine meint ein mentales Modell eines idealisierten, konzeptuellen Computers.*

Der Sinn der Thematisierung von mentalen Modellen besteht im

„Verstehen, Erklären, Problemlösen und Kontrollieren“ (Weigend, 2007, 33). Weigend (2007) führt diese Punkte wie folgt aus: Das Modell kann als Hilfe zum Verstehen von abstrakten Konzepten verwendet werden. Beim Erklären wird das Modell benutzt, um die eigenen Gedanken anderen Personen auf einfache Art zu beschreiben, indem geeignete Repräsentationen gebraucht werden. Als Beispiele seien Programmkommentare und Dokumentationen genannt. Beim Problemlösen – bezogen auf die Programmentwicklung – kann das mentale Modell als Hilfestellung verwendet werden, indem das Problem zuerst auf Modellebene angegangen und gelöst wird. Beim Kontrollieren werden mentale Modelle verwendet, indem die Überprüfungen der Korrektheit beim Testen und Debuggen sich innerhalb des mentalen Modells abspielen. Von der Seite der Tutor*innen ist es gewinnbringend, die Erklärungen der Student*innen bewusst einzufordern. Auf diese Weise kann überprüft werden, ob deren Vorstellungen adäquat, oder eine Quelle für Fehlvorstellungen sind. Dies wird in Abschnitt 2.4.2 weiter ausgeführt. Dies ist besonders wichtig, da die Student*innen irgendeine Vorstellung mit sich bringen, wie ein Computer funktioniert und wie der Programm-Code verarbeitet wird. Diese mitgebrachten Vorstellungen werden unter dem Begriff Prä-Konzept gefasst. Prä-Konzepte aufzugreifen, fördert das Verständnis; umgekehrt tauchen nicht aufgegriffene, unpassende Prä-Konzepte später wieder auf (Westermann u. a., 2012, 940). Des Weiteren ist es für die Tutor*innen vonnöten, über eine breite Palette von Vorstellungen zur verfügen, um je nach Student*in und Situation die passende parat zu haben. Aus diesem Grund werden nach dem Absatz über die zugehörigen Visualisierungen die verschiedenen mentalen Modelle und deren Visualisierungen zu den einzelnen Konzepten aufgeführt. Da sich die englischsprachige Literatur vorwiegend auf *Fehl*vorstellungen anstatt auf Vorstellungen konzentriert, wird im Bereich der Vorstellungen hauptsächlich auf die Arbeit von Weigend (2007) verwiesen.

Einer Visualisierung liegen die darzustellenden Informationen zugrunde, welche in geometrische und damit visuell erfassbare Informationen transformiert werden (Schumann und Müller, 2013, 1). Die Vorteile der Visualisierung liegen darin, dass durch die doppelte Kodierung – verbal und nonverbal – die Erinnerungswahrscheinlichkeit erhöht wird und damit die Visualisierung als virtuelle Eselsbrücke

dient, siehe Levin et al. (1987) zitiert nach Lewalter (1997) und Paivio (1971) zitiert nach Weigend (2007). Zudem können Visualisierungen organisieren, indem sie Zusammenhänge oder zeitliche Abläufe aufzeigen (Lewalter, 1997, 31). Des Weiteren veranschaulichen Visualisierungen intuitive Modelle, die dahinter stehen, siehe Linn und Clancy (1992) zitiert nach Winslow (1996). Zudem können Visualisierungen als *konzeptuelle Modelle* fungieren, die den Lerner*innen helfen tragfähiges Wissen aufzubauen (Sorva u. a., 2013, 3). Die Typen von Visualisierungen, welche von Sorva u. a. (2013) in verschiedensten Softwares zur Programm-Visualisierung identifiziert wurden, sind für die Tutor*innenschulung nicht brauchbar. Dies liegt an dem Fokus von Sorva u. a. (2013) auf das Erstellen einer Taxonomie der verschiedenen Programme und damit auf der oberflächlichen Betrachtung der einzelnen Vorstellungen der Programmierkonzepte und deren zugehörigen Visualisierungen. Beispiele für allgemeine Techniken zur Visualisierung sind das Hervorheben durch Farben, Kästen, Umkreisen und das Verdeutlichen von Zusammenhängen durch Pfeile.

Wichtig an dieser Stelle ist anzumerken, dass es auch informatikspezifische Visualisierungen gibt, die nicht an ein bestimmtes Programmierkonstrukt gebunden sind. Stellvertretend hierfür ist das *Nassi-Shneiderman-Diagramm* bzw. *Struktugramm* zu nennen, welches von Yoder und Schrag (1978) graphisch visualisiert und erklärt wird.

2.3.1 Variablen

Bei der Thematisierung der Vorstellungen von Variablen soll von den Vorstellungen zur Benennung einer Entität ausgegangen werden. Welche Metaphern werden (un-)bewusst in Gedanken gebildet, wenn eine Variable benannt wird und diese mit einem Wert eine Verbindung eingeht? Kurz gesagt, was passiert, wenn wir `x = 1` lesen?
Das Behältermodell beinhaltet die Vorstellung, dass Variablen Behälter für Daten sind, deren Etiketten die Namen der Variablen darstellen. In diesem Sinne kann die Anweisung `x = 1` so interpretiert werden, dass der eventuell vorherige Inhalt im Behälter mit Etikett `x` gelöscht und anschließend eine Repräsentation der Zahl `1` im Behälter abgelegt wird Weigend (2007). Eine Visualisierung des Behältermodells ist die Vorstellung des Behälters als Schachtel ohne Deckel. Der Inhalt entspricht dem Text auf einem Zettel, der sich innerhalb der Schachtel befindet.

Weigend (2007) sieht die Essenz des Behältermodells in der Beständigkeit des Behälters gegenüber dem wechselhaften Ändern des Inhalts. Im Gegensatz dazu steht das Referenzierungsmodell, welches den Inhalt als Objekt und dauerhafte Entität in den Fokus rückt. Hier wird der Variablenname als Name eines Objekts aufgefasst und der Name ist an das Objekt gebunden. Mögliche Visualisierungen des Referenzierungsmodells sind, dass der Name mittels eines Pfeiles auf das gebundene Objekt zeigt, oder auch dass an das Objekt ein Zettel mit dem Namen der Variablen geklebt wird. Ein wichtiger Unterschied in den Modellen besteht im dem Fall, wenn eine Variable erstellt wird, die nicht an ein Objekt gebunden ist. Im Behältermodell stellt dies kein Problem dar, da schlichtweg ein Behälter erzeugt wird, dessen Inhalt leer ist. Nach Weigend (2007) kann im Referenzierungsmodell ein Objekt erst benannt werden, wenn es existiert. Hier lässt sich jedoch der Kunstgriff anführen – wie er auch in der Programmiersprache Java vorgenommen wird – das *Nichts* als Objekt hinzuzunehmen. Auf diese Weise kann die erstellte Variable auf das spezielle Objekt Nichts verweisen. Weder auf das Nichts noch auf den Ausweg durch den genannten Kunstgriff wird von Weigend (2007) an dieser Stelle eingegangen. Auf das Nichts wird erst im Rahmen des Kapitels *Daten* und dort bei der Repräsentation von Daten eingegangen, wobei jedoch nicht auf die Modelle von Variablen Bezug genommen wird.

Nachfolgend sollen noch einige Probleme und Vorzüge der genannten Modelle aufgeführt werden, die nicht nur von Weigend (2007) festgestellt wurden. Ein Vorzug des Behältermodells wird bei der Thematisierung von unveränderbaren Objekten wie Zahlen oder auch bei der Darstellung von Arrays deutlich. Hier spiegelt die Dauerhaftigkeit des Behälters die unveränderbaren Objekte wider (Weigend, 2007, 62). Ein Problem dieses Modells wiederum ist, dass man bei Mehrfachnamen für *ein* Objekt die Abhängigkeiten nicht gut erkennt (Weigend, 2007, 61). So müssten sich die Inhalte bei Änderung des zugeordneten Objekts auf magische Weise ändern. Beim Referenzierungsmodell werden die Abhängigkeiten durch die Visualisierung mehrerer Zeiger bzw. Pfeile besonders gut deutlich.

An dieser Stelle ist zu erwähnen, dass Weigend (2007) sich meist auf die Programmiersprache Python bezieht. Da in der Veranstaltung

FOP jedoch auch die funktionale Programmiersprache Racket behandelt wird, lassen sich nicht alle Vorstellungen eins zu eins übertragen. In rein funktionalen Sprachen gibt es keine Zustände und daher keine dynamische Veränderung und keine zustandsverwaltenden Variablen, die auf Speicherzellen referenzieren (Wagenknecht, 2016, 48-49). Bei Racket gilt das aufgrund der Befehle `define` und `set!` nur bedingt, da sie Nebenwirkungen – Seiteneffekte – haben (Wagenknecht, 2016, 16). In Java wiederum sind Variablen die Speicherplätze, die jeweils nur einen Typ umfassen können. Referenzen sind daher Verweise auf die Variable (Hölzl u. a., 2013, 55). Folglich ist für das Erklären der Funktionalität der Variablen in Java das Referenzierungsmodell besonders hilfreich.

2.3.2 Iteration und Schleifen

Unter der Iteration soll das „Durchlaufen [...] einer aufzählbaren (eventuell unendlichen) Kollektion von Items“ (Weigend, 2007, 95) verstanden werden. Weigend (2007) trennt explizit zwischen der genannten Iteration und den *Wiederholungen mit nicht antizipierbarem Ende*. Die Iteration wird durch die `for`-Schleife programmtechnisch umgesetzt (Weigend, 2007, 96), wobei aber eine `for`-Schleife a priori endlich ist. In Java besteht der Unterschied zwischen einer `for`-Schleife und einer iteratorbasierten Schleife gerade darin, dass der Iterator nicht explizit gemacht wird (Ernst u. a., 2016, 692). Daher ist die Einteilung nach Weigend (2007) für die Tutor*innenschulung nicht ohne Bedenken zu übernehmen. Im Unterabschnitt *Schleifen* der *Wiederholungen mit nicht antizipierbarem Ende* wird nur die `while`-Schleife verstanden, die sich als „kreisförmiger Kontrollfluss“vorzustellen ist (Weigend, 2007, 99). Hierbei ist anzumerken, dass auch eine `for`-Schleife sehr wohl mit einem kreisförmigen Kontrollfluss assoziiert werden kann. Die Arten von Schleifen werden jedoch von Weigend (2007) strikt voneinander getrennt. Um sich den kreisförmigen Kontrollfluss vorzustellen, werden folgende Beispiele vorgestellt.

- Der Wechsel von Tag und Nacht im Verlauf eines Tages.
- Der Zyklus der Jahreszeiten.
- Das Wechselspiel von Frage und Antwort in einem Interview. (Weigend, 2007, 99)

Inwieweit diese geeignet sind, Programmieranfänger*innen die Funktionsweise und Idee einer `while`-Schleife vorzustellen, sei dahin gestellt. Als gewinnbringende Sammlung von Metaphern für Schleifen wurde die Zusammenstellung von Wilson (2015) befunden, der sieben Alltagsbeispiele nennt, die sich relativ klar einer `for`-Schleife oder einer `while`-Schleife zuordnen lassen. Schleifen sind untrennbar mit der Vorstellung von Sprüngen im Kontrollfluss verbunden (Weigend, 2007, 99). Um diese zu visualisieren, verweist Weigend (2007) auf Programmablaufpläne bzw. Flussdiagramme, jedoch gibt es in diesen keine speziellere Darstellung der Wiederholung als einen Kreis.

Die fachliche Unterscheidung zwischen den einzelnen Schleifenarten soll hier keine weitere Erwähnung finden, da die Vorstellungen und mentalen Modelle dieser als fachdidaktische Elemente geschult werden sollen. Weitere Vorstellungen sind nicht vorhanden, daher ist dieser Abschnitt kurz gefasst.

2.3.3 Rekursion

Im Gegensatz zum vorherigen Abschnitt über Schleifen ist die fachdidaktische Literatur über Rekursion und deren mentale Modelle wesentlich ertragreicher. Wegweisend ist hier vor allem die Untersuchung zu den mentalen Modellen von Kahney (2009), welche bereits 1989 veröffentlicht wurde. Die Rekursion nimmt innerhalb einer Tutor*innenschulung eine Sonderstellung ein, da deren inhaltliche Wichtigkeit in der Informatik und ihre Schwierigkeit als besonders hoch einzuschätzen ist. Rekursion ist nach Ernst u. a. (2016) inhaltlich wichtig, da sie benutzt wird, um rekursive Datentypen wie Listen, Bäume und Graphen zu definieren. Algorithmen, die nach dem Prinzip des Teilens und Herrschens verfahren, verwenden Rekursion und primitiv rekursive und μ-rekursive Funktionen sind für die Theorie der Berechenbarkeit elementar (Ernst u. a., 2016, 459). Hierbei ist unter Anderem Church (1936) zu nennen, der für die Entwicklung des λ-Kalküls verantwortlich ist. Es kennzeichnet die Rekursion, dass sie schwer zu verstehen ist Levy und Lapidot (2000), ein abstraktes Konzept darstellt Levy (2001) und es wenige Analogien im Alltag gibt Levy und Lapidot (2000).

Nachfolgend soll die Untersuchung von Kahney (2009) und deren daraus folgende mentale Modelle knapp vorgestellt werden. Dieser

beschreibt die Ziel-Vorstellung der Rekursion wie folgt und verweist auf dieses Modell unter dem Namen *Kopien-Modell*:

> recursion as a process that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones. (Kahney, 2009, 209)

Das Ziel der Untersuchung war es, zwischen der benannten Ziel-Vorstellung und einem angenommen *Schleifen-Modell* (Abbildung 2.1) der Student*innen zu unterscheiden, bei dem die rekursive Prozedur als einzelnes Objekt statt einer Serie von neuen Instanziierungen betrachtet wird (Kahney, 2009, 209) und demnach eher als Iteration zu verstehen ist. Der *base case* meint im Deutschen den Rekursionsanker, bei dem keine neuen Kopien – Instanzen der Prozedur – erzeugt werden. Götschi u. a. (2003) schreibt in Bezug auf Kahney (2009), dass im Schleifen-Modell die Berechnug des gesamten Programms stoppt, sobald der Rekursionsanker erreicht ist und damit als stoppende Bedingung der Schleife anzusehen ist. Als Kontrollgruppe diente in der Untersuchung von Kahney (2009) eine Gruppe von neun Expert*innen.

```
|-----------------------------------|
|                                   |
|                                   |
|   |--->PROCEDURE-NAME        /PARAMETER/          = 'ENTRY POINT'
|   |
|   |       NOTE  <node>  <relation>  <node>        = 'ACTION PART'
|   |
|   |        CHECK <node>  <relation>  <wild-card>  = 'PROPAGATION-
|   |         IF PRESENT: PROCEDURE-NAME  *; EXIT       MECHANISM'
|   |                            |          |
|   |                            |          |
|   ----------------------------            |
|                                           |
-----------------------------------------
```

FIG. 11.2 A graphic description of the 'Loop' model of recursion.

ABBILDUNG 2.1: Graphische Beschreibung des Kopien-Modells (Kahney, 2009, 210)

Hervorzuheben ist, dass das Schleifen-Modell nicht per se falsch ist, sondern nur einen Spezialfall abdeckt und daher kein adäquates Modell für Rekursion im allgemeinen Fall darstellt. Im Falle der Endrekursion bzw. tail recursion – der rekursive Funktionsaufruf ist die letzte Aktion zur Berechnung der Funktion Abelson u. a. (2001) – ist dieses Modell vollends korrekt. Somit kann das Schleifen-Modell als

Fehlvorstellung betrachtet werden, falls nicht zusätzlich das Kopien-Modell vorliegt. Als weiteres mentales Modell wird das *Null-Modell* von Kahney (2009) angeführt. In diesem werden diejenigen Teilnehmer*innen der Untersuchung zusammengefasst, die aufgrund des Selbstaufrufens der Prozedur behaupten, dass ein solcher Selbstaufruf illegal ist und daher vom Computer verworfen wird (Kahney, 2009, 222). Das *Odd-Modell* verweist auf eine Gruppe von merkwürdigen Bemerkungen, die jedoch nicht einer gemeinsamen Logik und einem bestimmten mentalen Modell folgen, daher wird dieses Modell nicht weiter betrachtet.

Götschi u. a. (2003) hat in Anlehnung an Kahney eine eigene Untersuchung – bezogen auf die Programmiersprache Scheme – zu den mentalen Modellen durchgeführt und dabei weitere Modelle entdeckt. Besonders interessant ist dabei das Aktiv-Modell. Vor der Erläuterung dieses Modell sollen einige Sprachbausteine von Götschi u. a. (2003) zur Rekursion erläutert werden. Der Kontrollfluss zum Erstellen von neuen Kopien wird *active flow* – Kahney (2009) spricht von *passing forward* – genannt. Der *passive flow* – Kahney (2009) spricht von *passing backward* – beschreibt, dass bei jedem Level der Rekursionstiefe die Zwischenergebnisse berechnet werden und zu vorherigen Kopien zurückgegeben werden. Im Aktiv-Modell werden die Kopien mental korrekt gebildet, jedoch nur für den aktiven Fall bis zum Rekursionsanker und nicht für den passiven Fall.

Weigend (2007) unterscheidet zwischen offenen und geschlossenen Funktionstypen, wobei bei einer geschlossene Darstellung ein interner Funktionsaufruf als neue Schachtel innerhalb der äußeren Schachtel dargestellt wird. Als Problem dieser Darstellung nennt er, dass auf diese Weise die Schachteln schon bei geringen Rekursionstiefen nicht mehr dargestellt werden können. Als Beispiel eines offenen Modells nennt er gleiche Boxen, die sich mittels Greifern Parameter und Rückgabewerte übergeben. Weigend (2007) nennt jedoch nicht, welches mentale Modell hinter diesen Darstellungen steht. Daher ist die Verwendung problematisch zu sehen, wie auch in der Tutor*innenschulung gezeigt wird. Bei dem Abschnitt zur Rekursion wird jedoch im Wesentlichen auf die bereits bekannten Modelle von Kahney (2009) verwiesen, daher werden sie an dieser Stelle nicht erneut aufgeführt.

Der Zusammenhang zwischen verwendeter Sprache und dem zugehörigen mentalen Modell soll nun betrachtet werden. Das Ziel ist

es, dass das korrekte Modell beim Erklären verwendet wird. Die bereits verwendeten Begriffe des aktiven und passiven Kontrollflusses sind hilfreich, um explizit auf passiven Kontrollfluss aufmerksam zu machen. In den inadäquaten Modellen – Aktiv-Modell und Schleifen-Modell – ist dieser nicht vonnöten. Geeignete Metaphern für Rekursion sind *Ähnlichkeit* und *Selbstreferenz*; bei den Metaphern *Zyklus* und *Wiederholung* sollte zwingend auf den aktiven und passiven Kontrollfluss verwiesen werden (Levy, 2001, 10–11). Abeck beschreibt den aktiven und passiven Kontrollfluss mit dem „Auf- und Zuklappen der Aufrufkette“ (Abeck, 2005, 240). Abschließend soll als Negativbeispiel das Beschreiben des Rekursionsankers, der Rekursionsbasis, als „Abbruch“ (Fischer und Hofer, 2011, 756) genannt werden. Bei dem Wort Abbruch wird nicht deutlich, ob hier der Programmabbruch – inkorrekt – oder der Erzeugung neuer Instanzen der Rekursion – korrekt – beschrieben wird.

2.3.4 Objektorientierung

Es gibt im englischsprachigen Raum eine Vielzahl von Studien zu *Fehl*vorstellungen zur Objektorientierung, von denen Eckerdal und Thuné (2005) exemplarisch einige herausstellen. Diese werden im Abschnitt 2.4 spezifiziert. Zu den Vorstellungen der Konzepte von Programmieranfänger*innen von Objekten und Klassen in Java soll hier die Studie von Eckerdal und Thuné (2005) herangezogen werden. Im deutschsprachigen Raum ist die gesamte Fachliteratur zu (Fehl-)Vorstellungen auf die Dissertation von Weigend (2007) reduziert, welche sich mit den mentalen Modellen der Informatik und damit mit den Vorstellungen zur Objektorientierung beschäftigt.

Eckerdal und Thuné (2005) haben Aussagen und Beschreibungen von Student*innen des ersten Jahres zu den Konzepten Objekt und Klasse in der Programmiersprache Java zusammengetragen und jeweils in drei Niveaus von Vorstellungen von ansteigender Komplexität eingeordnet. Auf dem ersten Niveau wird ein Objekt als ein Stück Code und demnach rein textuell erfahren. Auf dem zweiten Niveau wird diese Vorstellung durch Aktivität des Objekts im Programm bei der Programmausführung ergänzt. Durch die Vorstellung des Objekts

als Modell eines Phänomens der realen Welt wird die Vorstellung eines Objekts vervollständigt, da auf diese Weise ein Verständnis der zugehörigen fundamentalen Ideen des objektorientierten Paradigmas gezeigt wird (Eckerdal und Thuné, 2005, 91). Die Vorstellungen zum Konzept Klasse und deren Niveau sind analog aufgebaut und beschrieben. Auf dem ersten Niveau wird eine Klasse als eigene Entität im gesamten Programm erfahren, welches den Code strukturiert. Auf dem zweiten Niveau wird die vorherige Vorstellung durch die Beschreibung von Eigenschaften und Verhalten der zugehörigen Objekte erweitert. Abgeschlossen wird die Vorstellung der Klasse auf dem dritten Niveau, indem sie durch die Vorstellung der Klasse als Modell eines Phänomens der realen Welt vervollständigt wird. Nur wenige Student*innen der Studie haben das dritte Niveau der Konzepte erreicht (Eckerdal und Thuné, 2005, 91). Zudem steht das Verständnis des jeweiligen Konzeptes mit dem der anderen in Korrelation (Eckerdal und Thuné, 2005, 92).
Die Anzahl der interviewten Personen war lediglich auf 14 beschränkt und alle Student*innen den gleichen verpflichtenden Einführungskurs in die Programmierung belegten. Auf das Vorwissen und die damit eventuell vorhandenen Prä-Konzepte wurde in der Studie nicht eingegangen. Es wäre gewinnbringend gewesen, die Vorstellungen dieser Teilnehmer*innen mit denen von Programmieranfänger*innen, die einen verwandten Kurs belegten, zu vergleichen oder auch die erreichten Niveaus der Teilnehmer*innen der Studie mit ihrem Vorwissen in Beziehung zu setzen. Da Eckerdal und Thuné (2005) betonen, dass nur wenige Student*innen der Studie die dritte Niveaustufe der Konzepte erreicht haben, könnte folgender Fall vorliegen. Diese Student*innen haben aufgrund ihres Vorwissens bereits vor Absolvieren des Kurses über das Niveau der dritten Stufe verfügt und sind daher gar nicht mehr als Programmieranfänger*innen im eigentlichen Sinne zu betrachten. Demnach wäre die dritte Niveaustufe kein zu erreichendes Ziel für Anfänger*innen.

Beginnend stellt Weigend als eine Besonderheit der objektorientierten Programmierung dar, „dass das mentale Modell von interagierenden Entitäten Bestandteil des Paradigmas ist“ (Weigend, 2007, 117). Nachfolgend stellt er fünf verschiedene Vorstellungen von Klassen vor und erwähnt nur knapp die Vorstellungen von Objekten im Kapitel

„Objekte“, da diese mit den Vorstellungen von Klassen korrespondieren (Weigend, 2007, 122). Als erste Vorstellung wird die *Klasse als Bauplan* dargestellt, „der spezifiziert, wie ein Objekt einer Kategorie später aussehen soll“ (Weigend, 2007, 117). Hier wird der Bauplan lediglich als Dokumentation aufgefasst und umfasst nicht das Generieren des Objekts. Das Generieren und Erzeugen hingegen wird in der Vorstellung der *Klasse als Fabrik für Objekte* aufgegriffen. Hier wird die Klasse als Akteur verstanden, die in der Lage ist, Objekte bestimmter Art zu produzieren (Weigend, 2007, 118). Beide Vorstellungen lassen sich sinnvollerweise in der Vorstellung einer *Schablone zum Erzeugen* zusammenfassen und sind so auch in Lehrbüchern zu finden (Hölzl u. a., 2013, 25,71). Im Kontrast dazu besteht Weigend (2007) auf eine strikte Trennung des Dokumentierens und Produzierens. Er stellt jedoch selbst heraus, dass bei der ausschließlichen Auffassung der Klasse als Bauplan zum Dokumentieren der Konstruktor wenig Sinn ergibt, da noch kein Objekt existiert, welches agieren kann (Weigend, 2007, 117). Im Anschluss folgen drei Vorstellungen von Klassen, die aus den dargestellten Gründen eher Spezialfälle und daher nicht auf eine Stufe mit den vorherigen Vorstellungen zu stellen sind.
Eine Klasse ließe sich als „Menge von Objekte eines bestimmten Typs“ vorstellen und dabei seien als Beispiel „die numerischen Klassen wie `int` oder `float` (Python)“ (Weigend, 2007, 118) genannt. In Java hingegen stellen Ganzzahlen und Fließkommazahlen eingebaute Datentypen für Zahlen dar und sind demnach keine Klassen. Zudem könnte so `int` mit der Wrapper-Klasse `Integer` in Java verwechselt werden. Somit ist diese Vorstellung mit Problemen behaftet. Wenn nicht jedes Mal genau unterschieden wird, mit welcher Programmiersprache diese Vorstellung verbunden ist, sollte sie nur mit Vorsicht verwendet werden. In der Vorstellung der Klasse als *Prototyp* – Stellvertreter für alle anderen Objekte der Klasse – wird zu Beginn herausgestellt, dass Java und Python nicht zu den Programmiersprachen gehören, die Prototyp-orientiert sind (Weigend, 2007, 118). Des Weiteren wird die Orientierung an einem Prototyp als Methode der Software-Entwicklung vorgestellt. Diese Sicht auf das Thema übersteigt jedoch den Rahmen der Tutor*innen-schulung und wird daher nicht weiter ausgeführt. Abschließend stellt Weigend (2007) die Vorstellung der Klasse als *Behälter für Funktionen (Toolbox)* dar. Als Beispiel wird die

Klasse `Math` genannt, die statische Funktionen im Sinne der funktionalen Programmierung umfasst. Da diese Klasse eher als Sonderfall in der Objektorientierung zu sehen ist, sollte sie innerhalb der Vorstellungen auch als solche behandelt werden.

2.4 Fehler von Programmieranfänger*innen

Da unter Fehlern sowohl die Begriffe *Fehlvorstellungen* und *Fehlern* fallen sollen deren Unterscheidung zu Beginn erläutert werden. Diese Begriffe werden weder in der Informatikdidaktik noch in anderen Disziplinen einheitlich unterschieden. Neben den *misconceptions* – Fehlvorstellungen – werden noch die folgenden Begriffe verwendet, um inkorrektes oder unvollständiges Verstehen des Programmierens zu beschreiben: *difficulties*, *errors*, *bugs*, *mistakes* (Qian und Lehman, 2017, 2) oder auch *problems* Garner u. a. (2005). Hierbei ist anzumerken, dass die verschiedenen Begriffe sich meist nur unzureichend mit dem Wort Fehler ins Deutsche übersetzen lassen. Die Unterscheidung zwischen Fehlvorstellungen und Fehlern soll im Wesentlichen auf deren qualitativen Hintergrund beruhen. Fehler sind leicht zu erkennen und auszumerzen – Flüchtigkeitsfehler. Bei Fehlvorstellungen steht ein systematisches Denken dahinter steht.

Um die Vielzahl von Fehlvorstellungen und typischen Fehlern zu kategorisieren, soll die Taxonomie der systematischen Übersichtsarbeit – *review paper* – von Qian und Lehman (2017) übernommen werden. Dort werden die Inhalte gemäß ihrem kognitiven Anspruch unterteilt: *Syntaktisches Wissen* ist das Wissen, welches Funktionalitäten und Regeln der jeweiligen Programmiersprache umfasst – bspw. Klammersetzung. *Konzeptuelles Wissen* beschreibt das Wissen über die Art und Weise, wie Programmierkonstrukte funktionieren – bspw. Schleifen – und Wissen darüber, wie der Computer arbeitet. Demnach werden Fehlvorstellungen als Fehler im konzeptuellen Wissen definiert. *Strategisches Wissen* beschreibt das Wissen, wie syntaktisches und konzeptuelles Wissen angewendet wird, um programmiertechnische Probleme zu lösen (Qian und Lehman, 2017, 3-4).

Diese Taxonomie erscheint sinnvoller gegenüber einer losen Sammlung von einzelnen Fehlern – ungeachtet ihrer Qualität – oder einer

thematisch orientierten Gruppierung, die die Fehler ihrem informatischen Konstrukt – bspw. Variable, Bedingung, Schleife – zuordnet. Die Gruppierung nach Themen hat den Vorteil, dass bei einer Vorbereitung auf ein konkretes Konstrukt alle Fehler vorhanden sind. Jedoch hat die Gruppierung nach Themen den Nachteil, dass die Wichtigkeit, die Qualität des Fehlers nicht mehr deutlich ist. Daher wird sich dafür entschieden, die Fehler innerhalb der Taxonomie nach Qian und Lehman (2017) nach ihrem zugehörigen thematischen Konstrukt zu sortieren. Bei syntaktischen Fehlern wird jedoch kein thematisches Konstrukt zugeordnet, da dieses dort nicht sinnvoll ist. Es spielt keine Rolle, ob eine Klammer bei einer Schleife oder einem Methodenaufruf vergessen wurde.

Schlechtes inhaltliches Wissen kann zu Fehlvorstellungen bei den Student*innen führen siehe Sadler et al. (2013) zitiert nach (Qian und Lehman, 2017, 10). Daher ist die Thematisierung von mentalen Modellen und Visualisierungen innerhalb der Tutor*innenschulung wichtig. Bei eigenen unvollständigen, wenn auch nicht falschen, mentalen Modellen führt das dazu, dass die Lehrenden eher regelbasiert lehren anstatt Begründungen zu liefern siehe Even (1993) zitiert nach (Qian und Lehman, 2017, 10). Damit wird die Förderung der Fähigkeit des *Erklären können* erneut legitimiert.

2.4.1 Syntaktisches Wissen

Qian und Lehman (2017) stellen die Erkenntnisse verschiedener Studien dar, die sich mit syntaktischen Fehlern in der Programmiersprache Java beschäftigen. Da in der Veranstaltung *FOP* Java ebenfalls einen großen Raum einnimmt, lassen sich die Erkenntnisse eins zu eins auf die Tutor*innenschulung übertragen.

Empirisch zeigt sich, dass die häufigsten Fehler in Java-Code von Student*innen durch falsch gesetzte Klammern – sowohl runde als auch eckige – und Anführungszeichen sind, siehe Altadmri und Brown (2015) zitiert nach (Qian und Lehman, 2017, 4). Jackson et al. zitiert nach (Qian und Lehman, 2017, 4) führen die folgenden häufigen Fehler von Programmieranfänger*innen auf: „irresolvable symbol error“, fehlende Semikolons und „illegal start of expression“. Nach Qian und

Lehman (2017) liegen die Ursachen der genannten Fehler in den folgenden Denkweisen: Der erste Fehler taucht meistens beim Verwenden einer Variable ohne vorheriges Deklarieren auf. Der dritte Fehler wird durch fehlende Vertrautheit mit den Ausdrücken von Java – wie falsche boolesche Ausdrücke – hervorgerufen. Zusätzlich können diese aber auch durch falsche oder fehlende geschweifte Klammern induziert werden. Über die Ursache des zweiten Fehlers – dem fehlenden Semikolon – wird nichts gesagt. Es kann jedoch wird vermutet werden, dass Anweisungen – *statements* – ohne das in Java benötigte Semikolon beendet wurden. Als weiterer häufig auftretender Fehler wird die Verwechslung des Zuweisungsoperator = mit dem Vergleichsoperator == aufgeführt siehe (Qian und Lehman, 2017, 4). Nach Qian und Lehman (2017) sind syntaktische Fehler meist leicht zu finden, da sie vom Compiler bzw. einer integrierten Entwicklungsumgebung erkannt werden. Analog zu deren Standpunkt wird diese Art von Fehlern hier nicht weiter ausgeführt.

2.4.2 Konzeptuelles Wissen

Probleme bezüglich des konzeptuellen Wissens können zu signifikanten Fehlvorstellungen führen. Diese Fehlvorstellungen sind mit den mentalen Modellen der Programmausführung und des Computersystems verbunden (Qian und Lehman, 2017, 5). Daher soll der Fokus der Fehlerdarstellung auf diesem Abschnitt liegen.

Zu Beginn werden Fehler, die zum thematischen Konstrukt *Variable* gehören, aufgeführt. So glauben manche Student*innen, dass Variablen mehr als einen Wert zur gleichen Zeit beinhalten können, siehe Doukakis (2007) zitiert nach (Qian und Lehman, 2017, 5). Du Boulay (2009) hat festgestellt, dass manche Programmieranfänger*innen keinen Unterschied zwischen den Zuweisungen `A = 2` und `2 = A` machen. Zudem scheitern wohl manche aufgrund der Übertragung ihrer mathematischen Vorerfahrungen an der Zuweisung `A = A + 1` (du Boulay u. a., 2009, 290). Es besteht Teilweise die Annahme, dass Sprache und damit der Name der Variable eine Bedeutung für den Wert der Variable impliziert, siehe Kaczmarcyk et al. (2010) und Sleeman et al. (1986) zitiert nach (Qian und Lehman, 2017, 5). In der Interpretation von Qian und Lehman (2017) bedeutet dies, dass der Wert

der Variable `largest` zwingend größer als der Wert der Variable `smallest` sein muss. Im Bereich der Operationen, die den Input und Output betreffen, ergeben sich die folgenden Fehler: Programmieranfänger*innen verstehen nicht, wo die Daten herkommen und wie sie gespeichert werden, siehe Bayman und Mayer (1983) zitiert nach (Qian und Lehman, 2017, 5). Zudem glauben manche Student*innen, dass ein Print-Statement wie `The value of X is 1` den Wert der Variable `X` zu `1` abändern würde, siehe Haberman und Ben-David Kolikant (2001) und Sleeman et al. (1986) zitiert nach (Qian und Lehman, 2017, 5).

Es folgen die konzeptuellen Fehler zum thematischen Konstrukt *Bedingungen*. Manche Student*in glaubt, dass bei einer Verzweigung – `if-else`-Anweisung – sowohl die Anweisungen im `if`-Block als auch im `else`-Block ausgeführt werden, siehe Sirkia (2012) und Sleeman et al. (1986) zitiert nach (Qian und Lehman, 2017, 5). Zudem wird fälschlicherweise angenommen, dass wenn die Bedingung eines `if`-Statements zu `false` ausgewertet wird, die Ausführung des gesamten Programms stoppt, siehe Sleeman et al. (1986) zitiert nach (Qian und Lehman, 2017, 5). Der Ursprung der folgenden Fehler zu *Bedingungen* liegt in der falschen Übertragung der natürlichen Sprache. So ist `and` ein boolescher Operator und eine Konjunktion in natürlicher Sprache, die auch die folgenden Worte bedeuten kann: aber, und deswegen, und dann. Dies führt bei der Verwendung des Operators in Bedingungen nach den Regeln der natürlichen Sprache zu Fehlern (Du Boulay, 2009, 288). Diese Überlegung lässt sich auf den booleschen Operator `or` übertragen, der nicht-ausschließend operiert, in der natürlichen Sprache jedoch meist ausschließend verwendet wird. Dem Ursprung dieser Fehler folgend wurde festgestellt, dass die Fähigkeit der Sprache Englisch von chinesischen Student*innen signifikant mit ihrem Erfolg im Programmieren lernen korreliert, siehe Qian und Lehman (2016) zitiert nach (Qian und Lehman, 2017, 8).

Nachfolgend werden die konzeptuellen Fehler zum thematischen Konstrukt *Schleifen* aufgeführt. Die Ursache der Fehler in der Übertragung der natürlichen Sprache wird im Bereich der Schleifen fortgeführt. So muss die Bedingung bei einer `while`-Schleife *während* des gesamten Ablaufs gültig sein, anstatt einmal pro Iteration abgefragt zu werden, siehe Bonar und Soloway (1989) zitiert nach (Anthony u. a.,

2003, 153–154), (Du Boulay, 2009, 295). Programmieranfänger*innen können oft den Sichtbarkeitsbereich einer Schleife nicht abschätzen und wissen daher nicht, welche Zeilen wie oft wiederholt werden. So wird beispielsweise angenommen, dass nur `print`-Statements wiederholt werden, siehe Sleeman et al. (1986) zitiert nach (Qian und Lehman, 2017, 5). Nach Qian und Lehman (2017) glauben manche der Programmieranfänger*innen, dass eine Schleifenart besser als die andere sei. Da es sich hier um eine Schulung für Programmieranfänger*innen handelt, werden die Bereiche zur Semi-Entscheidbarkeit und die Optimierung durch Compiler an dieser Stelle nicht ausgeführt. Diese Bereiche sind Beispiele, dass die Schleifen nicht als gleich gut anzusehen sind. Wahre Bedingungen innerhalb einer Schleife werden auch ausgeführt, wenn sich der Programmzeiger außerhalb des Programms befindet, siehe Pea (1986) zitiert nach (Qian und Lehman, 2017, 5). Abschließend folgt die Erkenntnis von Du Boulay (2009), nach der Programmieranfänger*innen oft vergessen, dass bei einer `for`-Schleife die Schleifen-Kontroll-Variable aktualisiert wird.

Die konzeptuellen Fehler zum thematischen Konstrukt *Rekursion* beruhen vor allem in der Verwendung des inadäquaten Schleifen-Modells, wie sie bei den Vorstellungen zur Rekursion bereits genannt wurden.

Abschließend werden die konzeptuellen Fehler zum thematischen Konstrukt *Objektorientierung* aufgeführt, beginnend mit den Fehlern, die *Objekte* an sich betreffen.

Programmieranfänger*innen glauben, dass Objekte automatisch instanziiert werden, siehe Détienne (1997) zitiert nach (Anthony u. a., 2003, 152). Nachfolgend sollen die typischen Fehlvorstellungen nach Holland u. a. (1997) zu Objekten und die Auseinandersetzung von Eckerdal und Thuné (2005) dazu vorgestellt werden. Objekte werden als Wrapper für Variablen betrachtet (Holland u. a., 1997, 132). Instanziierungsvariablen – Attribute – eines Objekts einer Klasse müssen alle vom gleichen Typ sein (Holland u. a., 1997, 132). Objekte sind einfache, inaktive Datensätze wie bspw. CD-Klasse, in dem jedes Objekt eine CD repräsentiert (Holland u. a., 1997, 132). Hieran wird von Holland u. a. (1997) kritisiert, dass auf diese Weise nicht realisiert wird, dass das Verhalten von Objekten sich je nach Zustand des jeweiligen

Objekts ändert. Das Arbeiten in Methoden funktioniert nur über Zuweisungen – assignments – statt auch über message passing (Holland u. a., 1997, 132). Eckerdal und Thuné (2005) entgegnen den Darstellungen von Holland u. a. (1997), dass diese sich allein dadurch vermeiden ließen, wenn die Objekte und Klassen in den Beispielen gewissen Mindestanforderungen genügen. So sollen Klassen mehr als eine Instanziierungsvariable besitzen, mehr als ein einziges Objekt einer Klasse instanziiert werden und das Verhalten der Objekte soll variieren je nach Zustand (Eckerdal und Thuné, 2005, 92). Hierbei ist anzumerken, dass Holland u. a. (1997) selbst diese Strategien zum Vermeiden in seinem Artikel aufführt und damit die Kritik von Eckerdal und Thuné (2005) nicht in diesem Maße berechtigt ist. Nachfolgend soll die Tendenz des Verwechseln des Attributs `name` mit der Identität des Objekts und deren Konsequenzen nach Holland u. a. (1997) aufgeführt werden. Diese Annahme führt zu den weiteren, genannten Fehlvorstellungen:

- Nur eine einzige Variable kann zur gleichen Zeit auf ein gegebenes Objekt referenzieren.
- Wenn eine Variable auf ein Objekt referenziert, dann verweist es immer auf das Objekt.
- Eine Variable spezifiziert das zugehörige Objekt immer auf einzigartige Weise.
- Zwei verschiedene Variablen müssen auf zwei verschiedene Objekte zeigen.
- Ein Objekt weiß, welche Variablen auf es zeigen.
- Zwei Objekte der selben Klasse mit dem gleichen Status – Belegung von Attributwerten – sind das selbe Objekt.
- Zwei Objekte mit dem gleichen Wert für Attribut `name` sind das selbe Objekt.

(Holland u. a., 1997, 133)

Zu den konzeptuellen Fehler im Bereich der *Klasse* gehört die nicht vorhandene Trennung zwischen der Klasse und ihrer Rolle und damit eine Überspezialisierung der Klasse, welche zu einer übertriebenen Form von Taxonomien und Klassifikationsschmemata ausartet siehe (Meyer, 1996, 105). Als Beispiel kann die Aufteilung der Klasse `Worker` in den Klassen `FemaleWorker` und `MaleWorker` statt dem Hinzufügen

des Attribut `sex` zur Klasse `Worker` angeführt werden. Zudem wird fälschlicherweise eine Klasse als Sammlung von Objekten oder ein Objekt als Teilmenge einer Klasse angesehen siehe (Qian und Lehman, 2017, 6). Holland u. a. (1997) führt die Fehlvorstellung auf, dass es in jeder Klasse nur ein einziges Objekt geben kann siehe (Holland u. a., 1997, 132). Bezüglich der *Vererbung* führt Meyer (1996) die folgenden Fehler auf. Es wird eine Vererbung, welche eine (ist-ein)-Beziehung repräsentieren soll, verwendet, obwohl eine Aggregation – eine (hat-ein)-Beziehung – vorliegt (Meyer, 1996, 105). Als Beispiel dafür kann man sich das Szenario vorstellen, dass die Klasse `Tür` von der Klasse `Haus` erbt, obwohl im nachzubildenden Modell ein Haus eine Tür hat, die Tür auch ohne das Haus existiert und eine Tür kein Haus ist. Zudem nennt er den Fehler, dass Vererbung verwendet wird, um die Funktionalitäten der Superklasse zu verwenden, ohne dass eine (ist-ein)-Beziehung besteht (Meyer, 1996, 105).

Die Fehler der Objektorientierung, die dem Bereich des *Designs* angehören, sollen aufgrund der Ausrichtung der Veranstaltung *FOP* nur knapp erläutert werden. Der Ausgangspunkt hierbei ist, dass die Idee des *Seperation of Concerns* nicht beachtet wird. Damit ist das Aufteilen von Concerns – Dinge von Interesse – in einzelne Module gemeint (Painter, 2006, 2), (Rito da Silva, 1999, 3). In der Objektorientierung werden die Concerns in Klassen, deren Objekte und Vererbung dazwischen aufgeteilt (Rito da Silva, 1999, 11–12). Dies wird in der *FOP* in Verbindung mit dem Konzept des *Model-View-Controller* nach Krasner u. a. (1988) gebracht. Falls die Concerns nicht separiert werden, kann das zu Fehlern bezüglich des Designs führen. Thomasson u. a. (2006) nennt hier auszugsweise, dass ein Attribut Attribut einer anderen Klasse sein sollte. Ein Attribut sollte Objekt einer Klasse statt ein `String` sein. Zwei oder mehr Attribute sollten als eigene Klasse ausgelagert werden.

2.4.3 Strategisches Wissen

Unter strategischem Wissen zählt Qian und Lehman (2017) jenes Wissen, welches Strategien zur Problemlösung (2.5) umfasst. Als Fehler

sollen hier nur einige wenige genannt werden, da der Fokus der Veranstaltung *FOP* nicht in der Vermittlung der Patterns liegt und damit diese nicht von den Tutor*innen für das Erklären können benötigt werden. Ebenso werden das Verstehen von Syntax und Semantik sowie die zugehörigen mentalen Modelle als Voraussetzung für Strategien gesehen de Raadt (2008) zitiert nach (Qian und Lehman, 2017, 6). Demnach sind die Kenntnisse über die Fehler zum syntaktischen und konzeptuellem Wissen wichtiger einzustufen. Programmieranfänger*innen sind zwar teilweise in der Lage lauffähige Programme zu schreiben, jedoch überprüfen sie nicht die zulässigen Grenzen von Bedingungen und die unerwarteten Fälle, siehe Fisler et al. (2016), Sajaniemi und Kuittinen (2005), Soloway und Spohrer (1986) zitiert nach (Qian und Lehman, 2017, 6). Eine Schwierigkeit stellt das Verschmelzen von Code-Blöcken dar, die aber gemeinsam verwendet werden sollen, siehe de Raadt (2008) zitiert nach (Qian und Lehman, 2017, 6). Beispielhaft werden Patterns konkateniert, ohne sich um den Übergang zwischen diesen zu kümmern, siehe Ginat et al. (2013) zitiert nach (Qian und Lehman, 2017, 6–7). Simon (2013) stellt heraus, dass Programmieranfänger*innen keine Wächter bzw. guards benutzen, um bspw. die Division durch Null zu vermeiden. Hierbei ist die Untersuchung bei Simon (2013) auf die Programmiersprache C# bezogen und lässt sich in diesem Punkt auf die in *FOP* verwendete Programmiersprache Java übertragen.

2.4.4 Häufigkeit von Fehlern

Abschließend soll knapp eine Studie von Garner u. a. (2005) vorgestellt werden, die sich mit der quantitativen Häufigkeit von Fehlern beschäftigt. Im Abschnitt der syntaktischen Fehler wurde bereits erwähnt, welches die drei häufigsten Fehlern sind, jedoch keine Größenordnung für die Anzahl mitgegeben. Diese Lücke soll durch den Artikel von Garner u. a. (2005) geschlossen werden. Es handelt sich bei dem Kurs, dem die Daten zugrunde liegen, um einen CS1-Kurs basierend auf der Programmiersprache Java, dessen Übungen – specific session for labs – von demonstratorn geleitet werden, die nach Prinzip der minimalen Hilfe verfahren (Garner u. a., 2005, 2–3). Aufgrund der methodischen Übereinstimmung zwischen dieser Veranstaltung

und dem Verfahren der Übungen mit Tutor*innen in der Veranstaltung *FOP* lassen sich die Quantitäten gut auf diese übertragen.

Die demonstrators haben bei jedem Gespräch mit den Student*-innen das auftauchende Problem einem von 27 vorher definierten Problemen zugeordnet. So wurden insgesamt 11240 Probleme klassifiziert. Eine graphische Zuordnung der absoluten Anzahl der Probleme zur Problemnummer wird in der Abbildung 2.2 gezeigt.

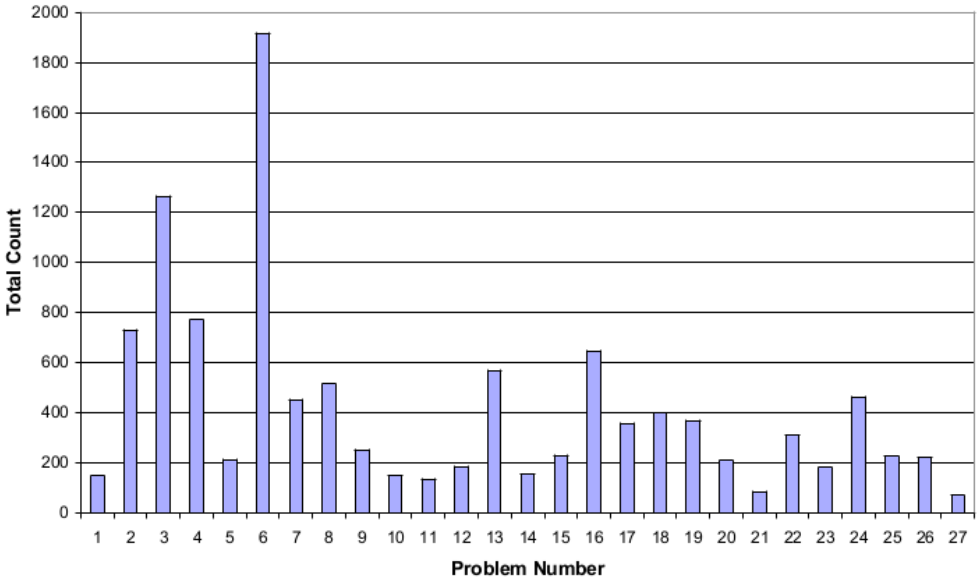


ABBILDUNG 2.2: Absolute Anzahl der jeweiligen Problemen nach Nummer (Garner u. a., 2005, 4)

Die fünf häufigsten Fehler sind in absteigender Reihenfolge Problem 6 – Basic mechanics –, Problem 3 – Stuck on program design –, Problem 4 – Problems with basic structure –, Problem 2 – Understanding the task– und als Letztes Problem 16 – Arrays –. Hierbei ist auffällig, dass die Probleme mit grundlegenden syntakischen Details – Problem 6 – deutlich häufiger als jedes andere Problem auftauchen (Garner u. a., 2005, 4). Aufgrund der nächsten drei Probleme – 3, 4 und 2 – wird die Hypothese aufgestellt, dass Probleme bezüglich dem grundlegenden Design signifikant häufiger sind als spezifische Konstrukte der jeweiligen Programmiersprache (Garner u. a., 2005, 4). Es kann auch erwidert werden, dass man zuerst eine Idee haben muss, wie man die Programmieraufgabe löst, um zu einem Code zu gelangen, bei dem die spezifischen Fehler auftauchen. Im Anhang A.1 befindet sich die komplette Übersicht der Probleme.

2.5 Problemlösestrategien der Informatik

2.5.1 Ursprung des Problemlösens

Der Ursprung des Problemlösens als solches liegt in dem Werk „Schule des Denkens“ von Polya (1994) zitiert nach (Bruder und Collet, 2011, 18). Von Polya stammt ebenfalls der folgende, gedankliche Ablauf, wenn man ein Problem löst.

> - Phase 1 ist das Verstehen der Aufgabe[...]
> - Phase 2 ist das Ausdenken eines Plans[...]
> - Phase 3 ist die Ausführung des Plans[...]
> - Phase 4 ist die Rückschau[...]
>
> Polya (1994) zitiert nach (Bruder und Collet, 2011, 18)

Hierbei ist anzumerken, dass sich das folgenden Werk explizit auf die Beschreibung des Problemlösens nach Polya bezieht (Kaplan, 2012, 2). Die Schritte, welche Winslow (1996) zur Problemlösung auf ein Programm bezogen beschreibt, beziehen sich zwar nicht explizit auf Werke von Polya. Jedoch können sie fast eins zu eins übertragen werden. Schubert und Schwill (2011) verweisen an keiner Stelle auf ein Werk von Polya. Die Eigenschaften des Problemlöseprozesses beschreiben sie wie folgt:

> Problemlösungsprozesse verlaufen nicht algorithmisch, besitzen kein strenges Schema, weisen häufig Versuch-Irrtum-Strategie in mehreren Zyklen auf, zeigen die große Be- deutung von Fehlern (Fehler sind positiv zu bewerten) und erfordern Strategien. (Schubert und Schwill, 2011, 83)

Der Begriff des Problemlösens nach Schubert und Schwill (2011) wird im Einklang des Ablaufplans nach Polya verwendet. Daher wird davon ausgegangen, dass Schubert und Schwill (2011) sich auf die Idee nach Polya beziehen, auch wenn diese es nicht thematisiert haben. Aus der Beschreibung der Eigenschaften lässt sich ablesen, dass die Phasen nicht linear zu verstehen sind. Sie sind vielmehr als Stationen auf dem Weg zum Ziel zu begreifen, die mitunter auch mehrfach durchlaufen werden.

Neben den Strategien zum Problemlösen werden noch eine Vielzahl semantisch ähnlicher Begriffe in der Literatur verwendet. Qian

und Lehman (2017) nennt hier *plans*, *schemas* und *patterns*. *Heuristiken* sind Erkenntnisse, die auf Analogien, Erfahrungen und Hypothesen anstatt wissenschaftlichem Fundament aufbauen, siehe Claus und Schwill (2006) zitiert nach (Schubert und Schwill, 2011, 82). Allgemein bezeichnet die Heuristik diejenige Wissenschaftsdisziplin, „die sich mit Problemlösemethoden“ (Bruder und Collet, 2011, 35) beschäftigt, und wurde nach dem Ausspruch – *heureka* – von Archimedes benannt. Kaplan (2012) hingegen, der ebenfalls Polya als seine Grundlage nennt, beschreibt, dass man sich *heuristics* als “ meta-skills“ bzw. “ statements of wisdom“ vorstellen kann, die durch Erfahrung gewonnen werden. Erfahrene Programmierer*innen können – bei Verwendung von metakognitiven Fähigkeiten – Programme nicht besser verstehen, wenn sie Heurismen benutzen, siehe Shaft (2005) zitiert nach (Kaplan, 2012, 2).

Nun sollen die Argumente wiedergegeben werden, warum Strategien bzw. Heurismen Programmieranfänger*innen näher gebracht werden sollten. Heurismen dienen als Werkzeuge und geben Orientierung beim Problemlöseprozess siehe (Bruder und Collet, 2011, 42–43). Hierbei geben Strategien gerade unerfahrenen Personen eine Hilfestellung an die Hand, die geistig bewegliche oder erfahrene Personen nicht benötigen, da sie bspw. implizit vollziehen. Hieraus lässt sich ableiten, dass Heurismen Ideen zur Lösung eines Problems und damit erste Ansätze bieten können. Damit lassen sie sich als ersten Anknüpfungspunkt auf das strategische Problem – Wie soll ich anfangen? – auffassen. Des Weiteren wird von die Kaplan (2012) die Hypothese aufgestellt, dass Strategien besonders für die unbekannte Situationen – Problemsituationen – besonders hilfreich sind. Wie in den Eigenschaften des Problemlöseprozesses nach Schubert und Schwill (2011) bereits beschrieben, behaupten diese, dass dieser Prozess Strategien erfordere. Für diese Aussage wird jedoch keine weitere Begründung gegeben.

2.5.2 Patterns und Strategien

Bei der Frage, was Patterns, Schemata, Pläne, Templates bzw. informatikspezifische Problemlösestrategien sind, unterscheiden sich die Definitionen untereinander. Die Definitionen sind abhängig von der betrachteten Sichtweise. Aus Sicht der Softwareentwicklung sind Patterns

> descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. Vlissides u. a. (1995)

Aus Sicht der Objektmodellierung sind Entwurfsmuster

> eine bewährte, generische Lösung für ein immer wiederkehrendes Problem, das in bestimmten Situationen auftritt. Balzert (1999) zitiert nach (Schubert und Schwill, 2011, 183)

Aus Sicht der Didaktik werden Pattern definiert als „key elements needed to write parts of the program“ (Proulx, 2000, 1).

Hierbei ist zu beachten, dass die unterschiedlichen Definitionen für eine Überlagerung des Begriffs sprechen. Dies ist jedoch nicht den Programmieranfänger*innen klar. Analog zu den verschiedenen Definitionen sind im Bereich der Patterns und Strategien vor allem Sammlungen für erfahrenere Programmierer*innen und Software-Entwickler*innen geläufig, welche jedoch für Erstere irrelevant sind (Kaplan, 2012, 2). Um jedoch die Vorteile der Strategien, für sie brauchbar zu machen, haben Kaplan (2012) und Proulx (2000) Strategien speziell für sie zusammengestellt.

Der Ausgangspunkt für Kaplan (2012) war die Einführung eines „"pre-CS1"course“ mit dem Ziel, Meta-Strategien zu erklären. So sollten die Student*innen für den Kurs CS1 gewinnbringend vorbereitet werden. Da im klassischen Kurs CS1 Meta-Strategien keine Beachtung finden, sollte diese im Vorkurs auf so kompakte Weise den Student*innen vermittelt werden, dass sie diese in der Entwicklung ihrer Programme verwenden können siehe (Kaplan, 2012, 2). Daraus entwickelte Kaplan (2012) eine Sammlung von 11 Strategien, welche er selbst als Startpunkt einer wissenschaftlichen Debatte in der Fachdidaktik betrachtet. Damit sind sie weder als vollständig noch als korrekt anzusehen. Bis heute keine Erweiterung oder Auseinandersetzung bezüglich der Auswahl der Strategien in der Literatur zu finden. Im Folgenden werden die Strategien im Originallaut zitiert siehe (Kaplan, 2012, 4).

1. BREAK IT DOWN
2. LEARN THE BASICS WELL, LOOKUP THE REST
3. FIND AND USE EXAMPLES
4. TEST TEST TEST (and test some more)
5. FOLLOW DIRECTIONS
6. SOLVE THE PROBLEM IN ENGLISH FIRST
7. REPRESENT WHAT?
8. WHAT DOES IT DO?
9. MAKE SURE YOU UNDERSTAND THE PROBLEM
10. IF YOU CAN'T SOLVE THE BIG PROBLEM SOLVE A SMALLER ONE.
11. HAVE A PLAN B

Da die Überschriften der Strategien selbsterklärend sind, wird darauf verzichtet die einzelnen Strategien weiter auszuführen, wie es Kaplan (2012) in seinem Artikel formuliert. Bei Bedarf sind diese unter A.2 nachzulesen. Hierbei ist anzumerken, dass Kaplan (2012) die genannten Strategien auf einem Kontinuum zwischen informatikspezifischen und allgemeinen Problemlösestrategien einordnet.

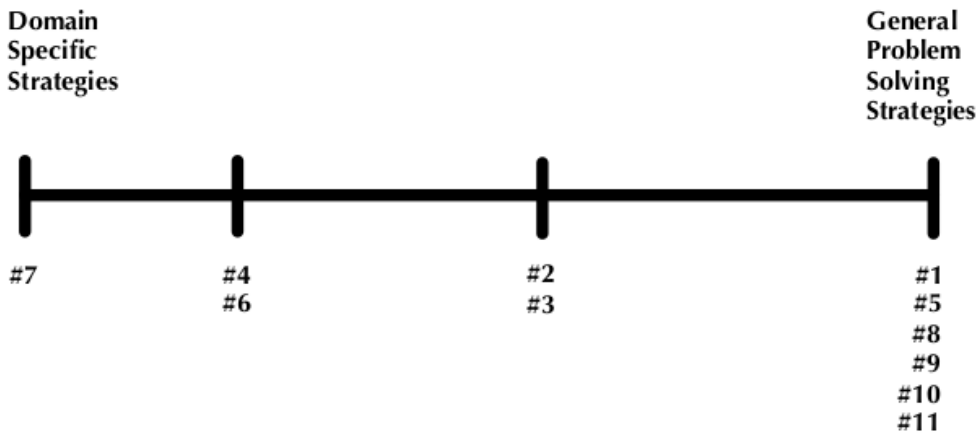


ABBILDUNG 2.3: Kontinuum der informatikspezifischen und allgemeinen Problemlösestrategien (Kaplan, 2012, 4)

Die meisten Strategien sind gerade nicht informatikspezifisch sind. Demnach entspricht die Sammlung nicht der Anforderung, dass die Programmieranfänger*innen mehr informatikspezifische Problemlösestrategien verwenden (2.2). Vielmehr bietet sie den Tutor*innen einen Ausgangspunkt, was überhaupt unter Strategien zu verstehen ist, da diese meist nur unbewusst vorhanden sind.

Im Kontrast zu den eher allgemeinen Problemlösestrategien nach Kaplan (2012) steht bei dem Artikel von Proulx (2000) die Darstellung informatikspezifischer Strategien im Vordergrund. Diese wurden speziell für Programmieranfänger*innen konzipiert und zusammengetragen. Neben einer knappen Vorstellung der Pattern wird auf die Grundlage des Artikels, einem Trainingskurs zu den Pattern verwiesen. Das Training ist aufgrund eines nicht mehr aktuellen Links nicht mehr verfügbar und auch auf persönliches Nachfragen, kam es zu keinen weiteren Informationen. Eine ähnliche Sammlung von Patterns für Programmieranfänger*innen ist jedoch auf der Webseite Bergin zu finden. Diese widmet sich unter dem Titel *Some Pattern Papers by Joseph Bergin & Others* der ausführlichen Beschreibung von verwandten Pattern. Es folgen die Titel der Pattern nach Proulx (2000) und einer Kurzbeschreibung.

1. Name Use Patterns: Umgang mit Identifiern mit dem Ablauf – deklarieren, definieren/konstruieren, benutzen, zerstören.

2. Reading Data Pattern: Niveau von verschieden Arten von Input: Rohdaten, verifizierter Input, gefilterter Input.

3. Read - Process - Write Pattern: das bekannte Pattern ICO – Input-Calculate-Output siehe Winslow (1996) – in einer anderen Formulierung.

4. Encapsulation Pattern: Auswahl von Situationen, in denen es hilfreich ist eine Hilfsfunktion zu schreiben.

5. Repetition Patterns: Methoden, wie man eine Schreibe – als Sinnbild der Wiederholung – schreibt.

6. Selection Pattern: Methoden, um boolesche Bedingungen zu designen.

7. Traversal Patterns: Methoden, um von einer Collection das nächste Element zu erhalten.

8. Cumulative Result Patterns: Beispiel für Kompositionen von verschiedenen Pattern.

9. Conversion Patterns: Beschreibung von fünf Klassen von Umrechnungen.

10. Indirect Reference Patterns: Sammlung von Array-Referenzen, expliziten Pointer-Variablen und Iteratoren.

Nach der Konzeption von Proulx (2000) baut der gesamte zugehörige einführende Programmierkurs vollends auf einem Training der Patterns auf. Die Grundlage der Veranstaltung *FOP* ist eine völlig andere und stützt sich vorwiegend auf die Vorstellung des funktionalen und objektorientierten Programmierparadigmas. Daher ist es nicht sinnvoll, alle genannten Pattern in die Tutor*innenschulung zu integrieren.

Pillay (2010) führt aus, dass die Patterns nach Proulx (2000) dafür geeignet sind, Module mit dem Fokus auf Problemlösefertigkeiten und prozedurale Programmierung in Java von Student*innen im ersten Jahr zu unterstützen. Die Design Patterns zur Objektorientierung folgen nach dem Curriculum von Pillay (2010) nach der Einführung in die objektorientierte Programmierung. Aufgrund des bestehenden Curriculums der Veranstaltung *FOP* und dem Fokus der Arbeit auf die Tutor*innenschulung werden die Design Patterns nicht behandelt, da diese in den Veranstaltung nicht explizit zur Beachtung kommen. Zur Erinnerung: Aus dem gleichen Grund wurden die typischen Fehler im objektorientierten *Design* nach Thomasson u. a. (2006) nicht in die Tutor*innenschulung übernommen.

Als Vertreter*innen des deutschsprachigen Informatikdidaktik führen Schubert und Schwill (2011) lediglich die „Top-Down-Methode“ und die „Bottom-Up-Methode“ als informatikspezifische Strategien an. Unter der Top-Down-Methode soll ein Vorgehen verstanden werden, bei dem man „mit schrittweiser Verfeinerung das Gesamtproblem durch eine Sequenz von Teilproblemen“ austauscht (Schubert und Schwill, 2011, 86). Bei Verwendung der Bottom-Up-Methode „werden existierende Teillösungen experimentell verknüpft in der Erwartung Teilprobleme zu lösen, für die noch keine Standardlösungen vorliegen.“ (Schubert und Schwill, 2011, 87). Die Nennung der beiden Strategien ist deswegen besonders interessant, da diese bei keiner der anderen fachdidaktischen Werke so genannt werden. Lediglich Kaplan (2012) scheint die gleiche Idee unter dem Titel „Break it down“ (Kaplan, 2012, 4) zu fassen. Dem entgegen ordnet Kaplan (2012) diese Strategie

keineswegs als spezifisch für die Informatik ein, vielmehr ist sie gänzlich als allgemeine Problemlösestrategie aufgeführt siehe Abbildung 2.3.

Abschließend soll knapp der Artikel *Incorporating Problem-solving Patterns in CS1* von Reed (1998) besprochen werden. Der Titel weckt den Anschein, dass die genannten Patterns genau die hier verwendeten Inhalte abdecken. Dem ist jedoch nicht der Fall. Das Pattern der binären Reduktion, welches die Grundlage der Thematisierung der Pattern zum Problemlösen bildet, wird im Artikel und im zugehörigen CS1-Kurs an mehreren Beispielen dargestellt. Die inbegriffenen Beispiele – beispielhaft Merge Sort, Rätselaufgaben zum exponentiellen Wachstum, binäre Suche, rekursive Sortieralgorithmen – sind jedoch an der TU Darmstadt nicht Inhalt der Veranstaltung *FOP*, sondern allenfalls der nachfolgenden Veranstaltung *Algorithmen und Datenstrukturen*. Daher wird das genannte Pattern und die exemplarischen Ausführungen in der Tutor*innenschulung keine Rolle spielen.

Kapitel 3

Tutor*innenschulung

Zu Beginn wird die Lerngruppe analysiert, um darauf aufbauend die Methoden zu den relevanten Inhalten festzulegen. Die relevanten Inhalte wurden in Kapitel 2 beschrieben. Die daraus entstehenden Beamer-Folien und Arbeitsblätter aus dem Anhang B erhalten hier ihre Verweise. Dazu folgen passend die Erwartungshorizonte der Fragen und Aufgabenstellungen an die Tutor*innen.

3.1 Analyse der Lerngruppe

Bei der Lerngruppe handelt es sich um eine Gruppe von Student*innen der Technische Universität Darmstadt, die als Tutor*innen für die Veranstaltung *FOP* eingestellt wurden. In der Gruppe gibt es insgesamt 3 Studentinnen und 23 Studenten, es handelt sich also um eine Tutor*innengruppe von umfangreicher Größe – verglichen mit der allgemeinen Tutor*innenschulung am Fachbereich. Die vertretenen angestrebten Abschlüsse der Lerngruppe sind: BSc. und MSc. Informatik, BSc. und MSc. Wirtschaftsinformatik, BSc. Computational Engineering und MSc. IT-Sicherheit. Die Lerngruppe ist in ihren Vorerfahrungen bezüglich der Tätigkeit einer Tutor*in ausnehmend heterogen. So betreuen 24 der 33 Student*innen zum ersten Mal die Veranstaltung *FOP*. Davon wiederum sind 21 Student*innen zum ersten Mal überhaupt als Tutor*in tätig. Zusätzlich ist einer der 24 Student*innen bereits am Fachbereich Mathematik als Tutor*in tätig gewesen. Dies wird sich voraussichtlich in den Fertigkeiten des Erklärens zu den Themen der *FOP* niederschlagen. Des Weiteren besteht ein großer Abstand zwischen den Fachsemestern der Student*innen. Ein Großteil der Student*innen befindet sich zum Zeitpunkt der Schulung am Ende ihres zweiten Fachsemesters, während die Spanne bis zu Student*innen am Ende des zweiten Semesters des Master-Studiums reicht. So haben

die Student*innen im höheren Fachsemester bereits weitere Veranstaltungen des Fachbereichs Informatik besucht und sind damit erfahrener im Programmieren und können Querverbindungen zu verwandten Themen ziehen.

In der Gruppe gibt es keine sozialen Auffälligkeiten. Diejenigen der Tutor*innen, die bereits zuvor die Veranstaltung *FOP* betreut haben, kennen sich durch das gemeinsame Arbeiten und diejenigen, die zum allerersten Mal Tutor*in sind kennen sich lose, da sie im gleichen Fachsemester sind. Diese Bekanntschaften und kollegialen Verhältnisse sind jedoch nicht weiter definierend für die Planung der Tutor*innenschulung. Im Schulungsraum sind eine Tafel, ein Beamer sowie ein Overheadprojektor vorhanden.

Alle Tutor*innen haben die Veranstaltung *FOP* bzw. deren Vorgänger *Grundlagen der Informatik 1* erfolgreich abgeschlossen. Die neuen Tutor*innen wurden persönlich angeworben und deren Fähigkeiten in Einzelgesprächen überprüft. Daraus lässt sich ableiten, dass alle Tutor*innen mit den fachlichen Inhalten der Veranstaltung *FOP* vertraut sind und diese beherrschen auf einem Level zwischen „Advanced Beginner“und „Competence“nach Dreyfus u. a. (1987) zitiert nach (Winslow, 1996, 19). Für diese Einschätzung spricht, dass die Tutor*innen die Themen der Veranstaltung *FOP* – die Grundlagen für Programmieranfänger*innen – selbst nur vier Monate plus die Zeit der Klausurvorbereitung als Teilnehmer*innen besucht haben. Daher verfügen sie maximal über das Erstellen eines Plans, nachdem sie sich mit der gegebenen Situation der Problemstellung beschäftigt haben, was für diese Einschätzung des Levels spricht. Ebenfalls spricht dafür, dass Student*innen nach Erlangen des Bachelor-Grades ein Level zwischen „Competence“und „Proficiency“nach Dreyfus u. a. (1987) zitiert nach (Winslow, 1996, 19) erreicht haben sollten.

3.2 Lernziele

Erinnernd sei angemerkt, dass die Lehrveranstaltung *FOP* der TU Darmstadt exemplarisch für eine universitäre Veranstaltung steht, die eine Einführung in die Programmierung vermittelt. Die Lernziele der Tutor*innenschulung sind die folgenden. Die Nummerierung der Ziele entspricht keiner Wertung, sondern dient der einfacheren Zuordnung.

1. Die Tutor*innen werden für einen adäquaten Umgang der Lösungen, Lösungsansätze und mentale Modellen jeder einzelnen Student*in sensibilisiert. Dies geschieht, indem die Tutor*innen bei diesen ansetzen und sie diese als Grundlage des Lernens betrachten.

2. Die Tutor*innen kennen informatikspezifische und allgemeine Strategien, um die Student*innen beim Programmieren adäquat zu unterstützen.

3. Die Diagnosekompetenz und damit die Fähigkeit des *Erklärens* seitens der Tutor*innen wird verbessert, indem sie ...

   (a) ... über eine größere Anzahl an Lösungsmöglichkeiten von typischen Aufgaben der *FOP* verfügen.

   (b) ... typische Fehler und Fehlvorstellungen zu den Inhalten der *FOP* kennen.

   (c) ... über geeignete Visualisierungen und Darstellungen zu den zugehörigen mentalen Modellen der Inhalte verfügen.

   (d) ... verschiedene Arten von Programmierstilen von Anfängern kennen und adäquat mit diesen umgehen können.

4. Die Tutor*innen sind in der Lage, ihre erworbenen Fähigkeiten bezüglich des *Erklärens* praktisch anzuwenden.

Die Ziele sollen durch die folgenden Phasen – wie sie im Methodischen Ablauf 3.3 und 3.4 beschrieben werden – während der Tutor*innenschulung erreicht werden.

3.3 Methodischer Ablauf – vormittags

In diesem Kapitel wird nun jede Phase der Tutor*-innenschulung mit eigenen Methoden und Sozialformen ausgeführt, deren Auswahl begründet und die jeweiligen Erwartungshorizonte beschrieben. Der äußere Plan der Tutor*innenschulung orientiert sich am Prozessmodell „Auf dem Weg zum kompetenzorientierten Unterricht – Lehr- und Lernprozesse gestalten“ (HKM, 2012, 74) des hessischen Kultusministeriums, da dieses einen Überblick gibt, wie das Vorwissen der Tutor*innen in Lehr- und Lernprozessen einbezogen werden kann.

Unter einem Erwartungshorizont soll hier die Gesamtheit der zu erwartenden Reaktionen und Rückmeldungen der Tutor*innen bei zu agierenden Methoden – bspw. Fragen an das Plenum und Arbeitsaufträge – ausgeführt werden. Dies umfasst sowohl die richtigen als auch falschen Lösungen, falls es dergleichen bei der entsprechenden Methode gibt. Der Sinn in der Formulierung des Erwartungshorizont besteht durch folgende Überlegung. Bereits vor der Durchführung der Tutor*innschulungen wird überprüft, ob die ausgewählten Methoden mit den Lernzielen übereinstimmen, und welche Schwierigkeiten eventuell auftreten können, um auf diese vorbereitet während der Schulung reagieren zu können.
Für alle Erwartungshorizonte, die eine Frage an die Lerngruppe der Tutor*innen innerhalb einer Präsentation aufgreifen, kann unabhängig von der jeweiligen Frage der folgende Fall eintreten: die Tutor*innen liefern keine Beiträge oder nur auf mehrfaches Nachfragen Beiträge. Das kann aufgrund der Lerngruppenanalyse auf folgende Punkte zurück geführt werden. Die erfahrenen Tutor*innen sind nicht motiviert genug zu antworten, da sie mit den Fragen und Antworten bereits vertraut sind und sie zu banal finden und sich deshalb nicht äußern. Die Tutor*innen, die zum ersten Mal Tutor*in werden, sind zu unsicher, um zu antworten. Die Tutor*innenschulung findet nur wenige Tage nach der allgemeinpädagogischen Schulung für neue Tutor*innen am Fachbereich Informatik statt. Daher ist mit einer gewissen Müdigkeit zu rechnen, auch wenn sich die Themen der jeweiligen Schulungen kaum überschneiden. Aufgrund der Qualitätssicherung ist die Tutor*innenschulung nicht freiwillig und die Tutor*innen erhalten unabhängig von ihrem Beteiligungsgrad ihre zugesicherte Vergütung. Beide Aspekte beeinflussen ebenfalls die Motivation.

3.3.1 Eingangsdiagnose

Der Einstieg mittels einer Eingangsdiagnose entspricht dem Handlungsfeld „Lernen vorbereiten und initiieren“(HKM, 2012, 74). Die Tutor*innen sollen sich bezüglich ihrer Fähigkeiten zum Verstehen der

realen Gedanken und Programme einer Programmieranfänger*in einschätzen und damit sich über ihr eigenes Vorwissen bewusst zu werden. Dieses soll mittels der Methode *Think-Pair-Share* geschehen. Diese fordert eine hohe Aktivität der Tutor*innen. Zudem können die Tutor*innen sich sowohl selbstständig mit dem Material auseinander setzen als auch ihr Vorwissen mit anderen abgleichen.

Die Tutor*innen erhalten einen Abschnitt einer Hausübung, welche der Lehrveranstaltung *FOP* aus dem Wintersemester 2017/18 entnommen ist. Das zugehörige Arbeitsblatt ist im Anhang unter B.3.1 zu finden, während die Folien zur Aufgabenstellung der *Think-Pair-Share*-Phase unter B.2.1 zu finden sind. Das Arbeitsblatt ist von den Tutor*innen in Einzelarbeit innerhalb von 20 Minuten zu bearbeiten. Dazu müssen die Tutor*innen die Aufgabenstellung der Hausübung lesen, welche die Anforderungen an die Prozedur vorgibt und Beispiel-Code für die definierte Struktur umfasst. Darauf folgt eine reale, in genau dieser Form abgegebene Lösung einer anonymen Student*in, welche von den Tutor*innen zu lesen und zu korrigieren ist. Dabei sollen die Tutor*innen sich zusätzlich ihre eigenen Gedanken zu dem Gelesenen notieren. Zudem sollen die Tutor*innen ein Feedback an die Student*in erstellen, worauf in der Rückschau (3.4.3) Bezug genommen wird.

Bei der Auswahl der Aufgabe wurde sich für eine auf der Programmiersprache Racket aufbauende Aufgabe entschieden. Dies liegt an der Kürze der Zeit, in der die Eingangsdiagnose stattfinden soll. Eine Aufgabe vom gleichen Komplexitätsgrad bezüglich der basierenden (Fehl-)Vorstellungen hätte in der Programmiersprache Java deutlich längeren Code produziert und damit den Fokus auf die Lesegeschwindigkeit der Tutor*innen gesetzt. Damit würde die Eingangsdiagnose jedoch nicht diejenigen Inhalte abfragen, die ursprünglich als Ziel angesetzt wurden. Andererseits wird in der Lehrveranstaltung *FOP* mit dem Erlernen der Programmiersprache Racket eingestiegen, jedoch nachträglich in den weiteren Lehrveranstaltungen nicht mehr verwendet. Somit ruft die Einstiegsdiagnose die fachlichen Fähigkeiten der Tutor*innen wieder hervor.

Die Auswahl einer Aufgabe aus der Hausübung statt einer aus den Kleingruppen wurde erwogen, da nur zu den Hausübungen Lösungen von Student*innen abgegeben werden mussten und damit auf

diese zugegriffen werden konnte. Innerhalb dieser Sammlung von Lösungen wurde sich für eine nicht korrekte Lösung entschieden, weil die Fehler sowohl syntaktischer als auch möglicherweise konzeptueller waren und allgemein viele unterschiedliche Fehler innerhalb weniger Zeilen zu finden waren. Es liegt die Vermutung nahe, dass die Student*in Strukturen und Prädikate in Racket nicht verstanden hatte. Um die Aufgabe dennoch zu lösen, wurde auf bekannte und verstandene Konzepte – die Listen, wie sie als Beispiele in den Zeilen 18 bis 20 der Aufgabenstellung zu finden sind – zurück gegriffen. Die einzelnen Fehler werden nachfolgend beim Erwartungshorizont beschrieben.

Es wurde sich für die Sozialform der Einzelarbeit entschieden. Da das *individuelle* Vorwissen zur diagnostischen Kompetenz 2.1 bezogen auf das Verstehen von Programmcode diagnostiziert werden soll. Bei einer gemeinsamen Bearbeitung treten die eigenen Stärken und Schwächen nicht deutlich genug hervor – bezogen auf das Verstehen und Diagnostizieren des Programmcodes. Das Vorwissen soll nicht nur hypothetisch abgefragt werden – Wie gut schätzt du dich im Korrigieren von Racket-Übungen ein? –, sondern praktisch an einer Aufgabe durchgeführt werden. Nur durch das tatsächliche Handeln durch das Code-Verstehen und Diagnostizieren der Fehler wird das Vorwissen tatsächlich sichtbar. Zudem ist diese Phase der Einzelarbeit als *Think*-Phase der angewandten Methode *Think-Pair-Share* zu sehen. Für diesen Arbeitsauftrag ist eine Bearbeitungszeit von 20 Minuten angesetzt.

Erwartungshorizont zur Korrektur der Lösung der Student*in:

- benötigte Zeitdauer der Korrektur heterogen: 10–30 Minuten siehe Lerngruppenanalyse 3.1
- Zeile 9: Vertrag ist nicht konform mit Vorgaben, richtig wäre: `product symbol -> boolean`
- Zeile 11: Bsp. passt zum richtigen Vertrag aber nicht zum geschriebenen.
  `'fruits` ist keine Kategorie aus den Beispielen der Aufgabenstellung sondern `'fruit`
- Zeilen 7 und 13 sind identisch und damit redundant. Dies führt zum Fehler beim Interpretieren, da nicht neu definiert werden darf.

Fragen sich eventuell, ob eine falsche Vorstellung vorliegt wie: „Funktionen können nur auf Hilfsfunktionen zugreifen, wenn sie im gleichen Bereich vorliegen.“ Möglicher Hinweis auf falsche Interpreter-Vorstellung von Racket.

- Zeile 15: Was soll der Kommentar?
- Zeilen 17–21 und 22–26 sind quasi Duplikate, besser: Auslagern in Hilfsmethode
- Zeilen 21+26: falscher Aufruf von `categorie=?`, da zweiter Parameter fehlt
- Zeilen 28+29: Beispiele passen zum falschen, beschriebenen Vertrag
- Mögliche, falsche Idee: *Beispiel*-Listen `fruits` und `drinks` werden rekursiv durchgeschaut und nach gesuchtem Produkt durchsucht. Großer Fehler: es sind Beispiele, keine vollständigen Listen
  Vielleicht: `define-struct` nicht verstanden und die Prädikate dazu, daher auf Bekanntes – Listen – zurück gegriffen
- Richtige Idee: Kategorie eines einzelnen Produkts über Prädikat `product-categorie` abzufragen und mit gegebenem Symbol vergleichen

Direkt an die *Think*-Phase schließt die *Pair*-Phase an. Hier sollen sich die Tutor*innen innerhalb von 5 Minuten mit ihren Sitznachbarn über ihre Gedanken aus der vorherigen Phase austauschen. Im Rahmen des Privaten zwischen zwei Tutor*innen ist es realistischer, dass diese sich relativ frei über ihre Stärken und Schwächen austauschen und letztere nicht vertuschen.

Um einen ersten Überblick über das Vorwissen der Lerngruppe zu gewinnen und von Seiten der Tutor*innen ihr Vorwissen in denen der Lerngruppe sozial einzubetten, wurde sich für die Methode des *Blitzlichts* Peterßen (2009) für die *Share*-Phase entschieden. Der wesentliche Vorteil der Methode ist, dass sie wenig Zeit kostet. Im

Vergleich zu einer Feedbackmethode wie dem Daumenfeedback liefert sie jedoch mehr Informationen über die Vorkenntnisse der Tutor*innen. Davon ausgehend kann entschieden werden, wie viel Instruktion erforderlich ist bzw. auf welches Vorwissen in den nachfolgenden Phasen zurückgegriffen werden kann. Die Kritik an der Methode, dass die Teilnehmer*innen sich nicht auf die Beiträge der anderen konzentrieren können (Wahl, 2013, 137), wird in diesem Fall relativiert. Die Tutor*innen sollen vorrangig sich selbst einschätzen und nicht im Vergleich zu anderen. Des Weiteren sind die Tutor*innen sich nicht unbekannt aufgrund der Besuche gemeinsamer Lehrveranstaltungen wie auch der vorhergehenden allgemeinpädagogischen Schulung siehe 3.1. Die Tutor*innen äußern sich kurz in einem Satz zu dem Satzanfang „Ich schätze mich . . . “.

Erwartungshorizont – Blitzlicht

Der Vielseitigkeit der erwarteten Antworten ist durch die unterschiedliche Vorerfahrungen der Tutor*innen geprägt wie sie im Abschnitt 3.1 beschrieben wurde.

Ich schätze mich . . .

- als *ungeübt* ein, daher brauche ich viel Zeit zum Korrigieren.
- als *langsam* ein, daher brauche ich viel Zeit zum Korrigieren.
- als *sicher* ein, da ich bereits öfter als Tutor*in tätig war.
- als *geübt* ein, da ich diese Aufgabe schon mal korrigiert habe.
- *besser* ein, wenn ich am *PC korrigieren* dürfte. Die Abgaben der Hausaufgaben sind doch online!
- *besser* ein, wenn es Java wäre. *Racket* ist schon so lange her.
- *gleich* gut bzw. so wie die *anderen* ein.
- so gut ein, dass ich die Schulung gar nicht nötig habe.
- *schlechter* als die anderen ein, da ich es zum ersten Mal mache.

3.3.2 Einstieg

In dieser Phase wird vorgestellt, was die Tutor*innen erwartet. Zudem erfahren diese durch Erklärungen, welches fachdidaktische Handwerkzeugs sie benötigen. Dadurch wird der Tagesablauf der Schulung begründet. Das zugehörige Material ist unter B.2.1 zu finden.

Die meisten der Tutor*innen bringen kein Vorwissen mit, was Programmieranfänger*innen und damit die Adressaten der Schulungsteilnehmer*innen kennzeichnet. Vielmehr sind sie selbst noch Novizen nach Einteilung der Level nach Winslow (1996). Daher wurde sich entschieden die Frage was Programmieranfänger*innen können und welche Arten von Programmieranfänger*innen (2.2) in einer Präsentation frontal zu vermitteln.

Die unzulänglichen Fähigkeiten der Programmieranfänger*innen geben vor allem denjenigen Tutor*innen ein Bild von den zu betreuenden Student*innen, die zum ersten Mal Tutor*in sind. Die Unterteilung der Arten der Programmieranfänger*innen ist deshalb interessant, da die Tutor*innen so die individuell verschiedenen Student*innen besser einordnen und so adäquat auf sie reagieren können. Dies folgt der Idee, dass nicht jede der Student*innen gleich viel und die gleiche Art von Aufmerksamkeit und Hilfestellung benötigt, aber diese sich zu Gruppen bündeln lassen. Diese werden durch die folgenden Frage ans Plenum zurück gegeben, sodass die erfahrenen Tutor*innen die anderen an ihrer Erfahrung teilhaben lassen. Dennoch können sich die anderen Tutor*innen bereits in die entsprechende zukünftige Situation herein versetzen. Hierbei stellen die zugehörigen Erwartungshorizonte mögliche Antworten da, welche jedoch keineswegs vollständig sind.

Erwartungshorizont – Wen erwartet ihr in den Übungen, Sprechstunden, . . . ?

- Übungen: *stopper* und *tinkerer*, unsichere, sehr motivierte Student*-innen, um Hausübungspunkte diskutieren
- Sprechstunden: ganz schlechte Student*innen wie *stopper*, die die Sprechstunden als letzten Anker nutzen; sehr motivierte Student*-

innen, die die Richtigkeit ihrer Lösungen gerne persönlich bestätigt hätten; sehr gute Student*innen wie *mover*, die gerne Sonderfälle diskutieren und weiterführende Fragen stellen möchten; unzufriedene Student*innen, die die bisherigen Bewertungen wie bspw. die Punktevergabe diskutieren wollen

- Emails: Student*innen, die die Hausübungen verspätet aus Gründen X per Mail nachreichen möchten; Student*innen, die nachfragen, ob neben der Sprechstunden noch weiter Zeit für Erklärungen sei

Erwartungshorizont – Wie kann man das Wissen über die Arten zum besseren Lehren nutzen?

- Durch die Kenntnis der verschiedenen Bedürfnisse kann man mehr Geduld zeigen.
- Jede*r braucht andere Hilfestellungen inhaltlicher, methodischer und emotionaler Art.
- Student*innen behutsam auf ihre Schwächen aufmerksam machen und gleichzeitig Lösungsmöglichkeiten gemeinsam besprechen.

Erwartungshorizont – Wer braucht welche Art von Zuwendung bzw. Hilfe?

- *stopper:* Mut machen, von eigenen Fehlern und Schwierigkeiten und deren Bewältigungen erzählen, Erfolgserlebnisse generieren durch kleine Aufgaben.
- *mover:* minimale Hilfe, kleine Denkanstöße, Anregung zum Weiterdenken, zusätzliche Problemstellungen.
- *tinkerer:* gemeinsam innehalten und reflektieren beim Debuggen, damit die Veränderungen nicht beliebig werden; auf den wesentlichen Kern zurückführen.

Die nachfolgende Folie zum *Übungsbetrieb* klärt den thematischen Rahmen, das *Erklären können*, auf den die Schulung sich konzentriert. Erneut sollen die Tutor*innen ihr Vorwissen dazu einbringen, was man denn bräuchte, um gut erklären zu können. Die Tutor*innen schöpfen hierbei aus ihren Erfahrungen aus den bereits besuchten Lehrveranstaltungen und denen aus der Schule.

Erwartungshorizont - Welche fachlichen Fähigkeiten muss ein Tutor*in besitzen, um gut erklären zu können?

- Das Thema selbst gut verstanden haben – hohe Fachkompetenz.
- Viele Beispiele kennen, um ein passendes parat zu haben.
- Bilder zu Beispielen wie Array, Liste, Pointer, ... kennen.
- Probleme durch Nachfragen herausfinden können.
- Probleme von Student*innen mit den Inhalten kennen.
- Individuelle Probleme fachlicher Art kennen.

Anschließend werden diese Punkte, welche mit den gegenstandsspezifischen Hintergrundwissen (2.1) assoziiert sind, den Tutor*innen vorgestellt. Damit sollen die Inhalte des Tagesablaufs legitimiert werden. Danach wird die Wichtigkeit der hochqualifizierten und individuellen Hilfestellungen präsentiert. Sie sind wahrscheinlich der signifikanteste Faktor für den Lernerfolg (Robins u. a., 2001, 42). Die Vorerfahrungen der Tutor*innen, welche sich auf pädagogische Fachbegriffe beziehen, sind bestenfalls rudimentär vorhanden. Daher soll der Begriff Diagnose knapp und praxisnah – bezogen auf ihre Tätigkeit als Tutor*innen – vorgestellt werden. Hierbei wird auch der Bezug zur allgemeinpädagogischen Tutor*innenschulung des Fachbereichs Informatik gezogen, da die W-Fragen ein wesentlicher Teil des Diagnostizierens sind. Anschließend wird darauf basierend der Tagesablauf inklusive der inhaltlichen Abschnitte und Pausen vorgestellt.

3.3.3 Visualisierungen und Mentale Modelle

Dieser thematische Abschnitt wird dreigeteilt. Zu Beginn wird den Tutor*innen eine Reihe von kurzen Videos zur Visualisierung von Code-Schnipseln zur Bewertung präsentiert. Anschließend folgt eine theoretische Einbettung der Thematik Visualisierungen und Mentale Modelle und die Gründe, warum sie gerade für die Tutor*innen von Bedeutung sind. Danach werden den Tutor*innen zu den verschiedenen thematischen Konzepten (2.4.2) die zugehörigen mentalen Modelle und Visualisierungen vorgestellt. Die Präsentation wird durch kurze Abschnitte zur Diskussion aufgelockert. Falls sich bei den Diskussionsfragen niemand oder nur weniger Tutor*innen melden, wird die Methode der *Murmelphase* – im Englischen *buzz group* genannt –

angewandt. Bei dieser sind die Lerner*innen dazu angehalten sich mit ihren Sitznachbar*innen murmelnd über die aktuelle Frage zu unterhalten. Dadurch können die Tutor*innen ihren Gedanken in einem geschützten Rahmen aussprechen und sich so versichern, dass andere sie ebenfalls für richtig halten. Es hat sich gezeigt, dass durch diese Methode eine höhere Absolutzahl an Wortbeiträgen bzw. Meldungen gefördert wird und sich mehr Lerner*innen gedanklich aktiviert fühlen (Barzel, 2011, 36). Ein weiterer Vorteil der Methode liegt in ihrer Wiederverwendbarkeit im Verlaufe der Tutor*innenschulung.

Die Videos der Code-Schnipsel wurden der *Python Visual Sandbox* von Weigend (2007) entnommen, welche unter Weigend (2005) zu finden ist. Die Idee der *Python Visual Sandbox* wird wie folgt beschrieben.

> Die Python Visual Sandbox (PVS) ist eine Sammlung von interaktiven Online-Applikationen (Spielen) mit insgesamt etwa 150 Animationen, die informatische Konzepte veranschaulichen. Im Unterschied zu freien Visualisierungsübungen [...] werden hier vorgefertigte Repräsentationen intuitiver Modelle vorgeführt, von denen vermutet wird, dass sie Schülerinnen und Schülern (bewusst oder unbewusst) verwenden. Die PVS setzt darauf, dass die Nutzer (Spieler) interne mentale Modelle wieder erkennen, die sie vielleicht selbst – mangels Visualisierungskompetenz – nicht explizieren können. (Weigend, 2007, 46)

Aus Ermangelung an vergleichbaren Video-Dateien zur Visualisierung von elementaren Programmierkonzepten wurde die Python Visual Sandbox als Grundlage herangezogen. Da nicht davon ausgegangen werden kann, dass die Tutor*innen Kenntnisse in Python haben und die Unterschiede zu Java nicht unerheblich sind, wurde sich auf elementarste Code-Schnipsel bezogen. Dies sind einfache Zuweisungen, welche sich bis auf fehlende Semikolons und Typdeklarationen nicht von dem Pendant in Java unterscheiden. Diese sind im Weigend (2007) unter *Python Quiz – assign* zu finden. Hieraus wurden zehn Videos entnommen. Diese behandeln das Behälter- und dem Referenzmodell und zeigen sowohl adäquate als auch nicht adäquate Visualisierungen.

Aufgrund der Heterogenität der Tutor*innen kann vermutet werden, dass ein Teil von ihnen ihre internen mentalen Mentale bisher

nicht expliziert hat. So ist es für diese leichter zu entscheiden, ob eine gegebene Darstellung adäquat ist oder nicht, als selbst eine Visualisierung für den entsprechenden Code-Schnipsel innerhalb kürzester Zeit zu entwerfen. Zudem stellen die Video-Dateien durch den Medienwechsel eine Abwechselung dar und die Tutor*innen werden aktiviert selbst tätig zu werden, indem sie die Darstellungen bewerten müssen. Die Bewertung geschieht durch Handzeichen, falls die jeweilige Tutor*in die Darstellung für adäquat befindet. Die Video-Dateien werden einzeln in Dauerschleife abgespielt gezeigt. Da dieser Einsteig in die Thematik nur knapp ausfallen soll, kommt es nur zu einer Bewertung, jedoch nicht zur Diskussion, warum sich für die jeweilige Bewertung entschieden wurde.

Erwartungshorizont – Sind die Darstellung der Videoausschnitte geeignet?

Die neuen Tutor*innen sind wahrscheinlich unsicherer mit unbekannten Darstellungen. So könnte es sein, dass gerade diese Tutor*innen sich nicht trauen, eine Darstellung als geeignet zu bewerten, selbst wenn sie der Überzeugung sind, dass dies der Fall sei. Somit würden sie sich als Konsequenz nicht melden. Zudem könnte es passieren, dass diese Tutor*innen sich an den erfahrenen Tutor*innen orientieren und sich erst mit Verzögerung melden, um sich in ihrer Einschätzung bestätigt zu fühlen.

- Bei den Darstellungen zu den Strings taucht jeweils der gleiche Tippfehler auf, da bei `Tuesday"` die Anführungszeichen fehlen.
- *assign_4*: Diese Darstellung ist nicht geeignet, da nach der Zuweisung `b = a`, der Inhalt der Variable leer wäre, was aber nicht der Fall ist.
- *assign_5*: Diese Darstellung ist geeignet, da auf magische Weise sich der Inhalt der Variable `a` dupliziert hat und so auch Variable `b` den gleichen Inhalt angenommen hat.
- *assign_6*: Diese Darstellung ist nicht geeignet, da durch die Zuweisung `b = a` die Variable `b` auf den *Inhalt* der Variable `a` verweisen soll und nicht auf die Variable selbst.

- *assign_10*: Diese Darstellung ist geeignet, da beide Variablen den geforderten Wert `3` angenommen haben und bei jeder Veränderung der Variable `a` sich auch der Inhalt der Variable `b` ändern würde.
- *assign_11*: Diese Darstellung ist geeignet, da die Variable `a` auf den Wert `3` zeigt, und die Variable `b` nachschaut, auf was die Variable `a` zeigt und anschließend selbst auf diesen Wert zeigt.
- *assign_12*: Diese Darstellung ist nicht geeignet, da durch die Zuweisung `b = a` die Variable `b` auf den Inhalt der Variable `a` zeigen soll und nicht auf die Variable selbst.
- *string_5*: Die Darstellung ist geeignet, da der Variable `today` jeweils nur ein einzelner Wert gleichzeitig zugewiesen wird und sie auf die vorherigen Werte keinen Zugriff mehr hat.
- *string_7*: Die Darstellung ist nicht geeignet, da bei der sukzessiven Zuweisung der Variable `today` nicht bei jeder Zuweisung ein neuer String erstellt wird.
- *string_8*: Die Darstellung ist nicht geeignet, da bei der sukzessiven Zuweisung der drei Werte die Variable `today` nicht auf alle Werte gleichzeitig zeigt.
- *string_9*: Die Darstellung ist geeignet, da die Variable `today` auf genau einen der drei String-Werte zeigt und dies in der richtigen Reihenfolge erledigt.

Nun folgt die Erklärung der Methodik zum zweiten Abschnitt, dem Darstellen der Gründe, warum mentale Modelle und Visualisierungen wichtig sind und was sie zu leisten vermögen. Hier wird sich auf die Fachliteratur (2.4.2) bezogen und diese knapp vorgestellt. Da die Tutor*innen keinen Mehrwert davon haben, zu wissen, welche Autor*in für die dargestellten Gründe verantwortlich ist, wird auf eine korrekte Zitation verzichtet. Neben dem Darstellen der Gründe sollen auch Handlungsanweisungen an die Tutor*innen erfolgen, damit diese eine

Anleitung erhalten, wie diese Konzepte konkret in ihrer Tätigkeit umzusetzen sind. Gerade aufgrund der mangelnden pädagogischen Vorbildung der Tutor*innen stellen konkrete Handlungsempfehlungen sicher, dass diese in der Lage sind, in neu erworbenes Wissen umzusetzen. Die Handlungsempfehlungen sind bewusst knapp gefasst. Beispielsweise sei das Nachfragen, was die Student*innen sich unter etwas vorstellen, um dann in den Erklärungen darauf einzugehen, aufgeführt. Die Nennung der Gründe soll außerdem sicherstellen, dass die Tutor*innen den Mehrwert dieses Abschnitts der Schulung erfahren. Dies ist insbesondere für die erfahrenen Tutor*innen relevant. Diese verfügen teilweise über jahrelange Erfahrung und fühlen sich auch ohne eine weitere Schulung kompetent genug. Es wurde verzichtet, eine konkrete Definition der Begriffe Visualisierung und mentales Modell zu lehren, da aus diesen sich kein Mehrwert für die Tutor*innen ergibt. Dieser liegt vielmehr in dem Kennenlernen verschiedener mentaler Modelle und ihren Darstellungen, damit sie diese in ihren Erklärungen verwenden können. Damit die Tutor*innen mentale Modelle und deren Visualisierungen nicht als Allheilmittel wahrnehmen, soll ein Ausblick vorgestellt werden, was alles misslingen kann. Dort wird unter Anderem auf die folgenden Module verwiesen.

Die methodische Umsetzung des zweiten Abschnitts des Moduls erfolgt in einer Präsentation, da hier die zielgerichtete, knappe Darstellung der Fachliteratur den Fokus darstellt.

Nun erfolgt eine exemplarische Präsentation der Visualisierungen. Aufgrund der Beschränktheit der Zeit kann schlichtweg nicht alles vorgestellt werden. Die Darstellung erfolgt mittels eines dozierendes Vortrags, welcher durch Diskussionsfragen aufgelockert werden soll. Durch die Diskussionsfragen, die ans Plenum herangetragen werden, sollen die Tutor*innen aktiviert werden, sich zu beteiligen. Damit ist deren Aktivität nicht auf das reine Zuhören beschränkt.

Zu Beginn sollen den Tutor*innen allgemeine Techniken zur Visualisierung siehe 2.3 in Erinnerung gerufen werden, da davon ausgegangen werden kann, dass sie diese bereits kennen. Hier wurde ein gelungenes Beispiel zur allgemeinen Visualisierung – siehe Abbildung 3.1 – von Programm-Code von Gailer (2018b) übernommen, welcher das Tutorial von Kjell und Gailer (2004) ins Deutsche übersetzt hat. Dies bezieht sich auf die folgenden Programmzeilen.

```
1 String alpha = "Dempster Dumpster";
2 alpha = null;
```

An dieser Stelle sei auch erwähnt, dass auf den Folien sich die Verweise der jeweils aufgegriffenen Thematik auf die Materialien der Veranstaltung *FOP* befinden. Einerseits dienen sie der Einbettung in die Themen der Tutor*innenschulung in die spezielle Lehrveranstaltung. Andererseits können sie zum Nachlesen der Tutor*innen zu einer bestimmten Thematik verwendet werden. Diese Verweise lassen sich auch in den folgenden Modulen finden. Die Handlungsanweisung für die Tutor*innen zielt darauf ab, dass Visualisierungen sich nur verwenden lassen, wenn man über geeignetes Material verfügt und dieses auch benutzt.

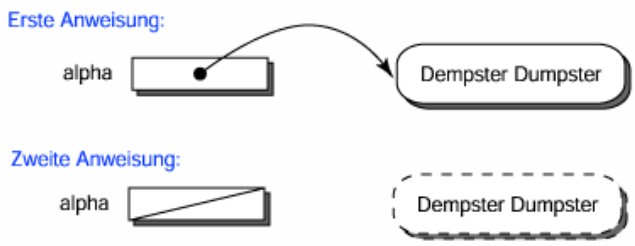


ABBILDUNG 3.1: Beispiel zu allgemeinen Techniken der Visualisierung von Programmcode Gailer (2018b)

Als Nächstes folgen zwei Beispiele, wie sich eine bedingte Anweisung bzw. Schleifen mittels einem Nassi-Shneiderman-Diagramm – auch Struktogramm genannt – darstellen lassen. Diese sollen der Vollständigkeit dienen, wie man Programm-Code visualisieren kann. Damit erweitern die Beispiele den Fundus der Tutor*innen, auf die sie zum Erklären zurückgreifen können.

Die folgenden Folien basieren auf den einzelnen Konstrukten (2.3). Hierbei wurde auch die Reihenfolge übernommen. Beginnend werden das Behälter- und Referenzierungsmodell der Variablen behandelt (2.3.1) aufgegriffen wurden. Zuerst wird dieses am Beispiel vorgestellt und erläutert. Hierauf schließt eine Diskussionsfrage an das Plenum gerichtet an, welche die Verwendung der Darstellung sowie deren Probleme zur Diskussion bringt. Die Bilder der Modelle sind Screenshots aus Weigend (2005) entnommen. Die folgende Folie stellt die möglichen Antworten zum Nachlesen für die Tutor*innen bereit.

Der folgende Erwartungshorizont bezieht sich darauf. Auf dieser Folie wird auf die Thematik der Programmiersprache Racket eingegangen, welche zuvor nicht explizit angesprochen wird. Dadurch soll eine reine Fokussierung auf Java vermieden werden.

Erwartungshorizont – Welche Darstellung ist in welcher Situation besser geeignet? Welche Probleme können auftauchen?

- Behältermodell ist gut geeignet für unveränderbare Objekte wie Zahlen oder auch für die Darstellung von Arrays, da diese eine unveränderbare Länge haben, siehe Weigend (2007, 62).
- Probleme beim Referenzmodell: man kann Objekte erst benennen, wenn sie existieren, siehe Weigend (2007, 61).
- Probleme beim Behältermodell mit Mehrfachnamen für *ein* Objekt, da sieht man die Abhängigkeiten nicht gut wie bei den Zuweisungen `x = 1`, `y = x`, siehe Weigend (2007, 61).
- Wenn die Tutor*innen keine Antworten dazu wissen, kann die nächste Folie als Tipp gegeben werden.

Erwartungshorizont – Unterscheidung nach Programmierparadigma: Was passt bei funktionalen Sprachen besser? Was bei prozeduralen?

- In rein funktionalen Sprachen gibt es keine Zustände und daher keine dynamische Veränderung und keine zustandsverwaltenden Variablen, die auf Speicherzellen referenzieren (Wagenknecht, 2016, 48-49)
- Bei Racket gilt es nur bedingt aufgrund der Befehle `define` und `set!`, die Nebenwirkungen – Seiteneffekte – haben (Wagenknecht, 2016, 16)
- In Java sind Variablen die Speicherplätze, die nur einen Typ umfassen können. Referenzen sind Verweise auf die Variable. Objekt mit Referenz ist wie Fernseher mit Fernbedienung. (Hölzl u. a., 2013, 55)

Da das Nichts beim Erklären von Variablen eine Sonderrolle einnimmt, soll in der Präsentation explizit darauf eingegangen werden. Zuerst wird das Beispiel zur leeren Liste von (Weigend, 2007, 74) übernommen. Dieser hat festgestellt, dass Schüler*innen Probleme bei der

Visualisierung des Nichts aufweisen. Dies zeigt sich darin, dass sie die Modelle für gleich gut befunden haben. Circa 60% haben sie jeweils als geeignet empfunden, wobei Mehrfachnennungen erlaubt waren (Weigend, 2007, 75). Für die Tutor*innen dienen die Darstellungen sowohl als Positiv- als auch als Negativbeispiele. Durch die Diskussion im Plenum erfahren sie, wie andere Tutor*innen die Darstellungen bewerten. Hierbei ist davon auszugehen, dass sich vor allem die erfahrenen Tutor*innen einbringen. Das ist nicht unbedingt negativ, da die anderen Tutor*innen von ihren Vorerfahrungen profitieren können.

Erwartungshorizont – Darstellungen der leeren Liste bewerten

- *Darstellung 1* ist nicht geeignet, da es nicht die leere Liste sondern eine Liste mit einem leeren Element umfasst (Weigend, 2007, 74).
- *Darstellung 2* ist nicht geeignet, da es eine fünf-elementige Liste bzw. Array darstellt, deren Element die leeren Elemente sind (Weigend, 2007, 74).
- *Darstellung 3* ist geeignet, da es einen Behälter mit keinem Fach darstellt. Somit kann nur *nichts* in dieses Fach passen (Weigend, 2007, 74).
- *Darstellung 4* ist geeignet, da es ein Zeiger auf *nichts* darstellt.
- Weitere Darstellungen für die leere Liste: nur außerhalb des Behälter- und Referenzmodells mit weiteren Analogien: beschriebene Blätter, etc.

Es wird an die Definition von `NULL` in Java erinnert (Evans und Flanagan, 2014, 76). Danach folgt an einem Beispiel,wie man sich das Nichts vorstellen kann. Dieses können die Tutor*innen als Anregung übernehmen. Hierbei wurde sich für die Unterscheidung des leeren Strings und dem `null`-String mittels einem Blatt Papier entschieden. Diese Visualisierung wurde bereits mündlich zwischen den Tutor*innen weitergetragen, wie aus eigener Erfahrung zu berichten ist. Die einfache und klare Darstellung spricht für diese Visualisierung. Daher kennt ein Teil der Tutor*innen dieses kanonische Beispiel. Somit ist die aufgeführte Frage eher als rhetorische Frage zu sehen.

Auf das Thema der Datenstrukturen soll in der Tutor*innenschulung nur knapp eingegangen werden, da es thematisch vielmehr

in der nachfolgenden Lehrveranstaltung *Algorithmen und Datenstrukturen* verankert ist. Da jedoch besonders Listen durch die Thematisierung der Programmiersprache Racket eine wichtige Rolle spielen, soll kurz auf diese eingegangen werden. Zudem soll den Tutor*innen vermittelt werden, dass das Erklären zielgerichtet ist und damit auch die Darstellungstiefe der Visualisierungen verbunden ist.

Erwartungshorizont – Darstellungstiefe von Datenstrukturen

- *Geringe Darstellungstiefe*: ersten Überblick geben, zum Problemlösen bei Aufgaben, die sich auf Datenstrukturen beziehen.
- *Hohe Darstellungstiefe:* Fokus auf dem Verstehen der Datenstruktur an sich, Fehler finden – Fehler könnte in der falschen Vorstellung einer Datenstruktur liegen.

In den folgenden Folien wird die Fachliteratur zum Programm-Konstrukt (2.3.2) den Tutor*innen knapp dargelegt. Hierbei liegt der Fokus darauf ein Gleichgewicht zwischen dem Vorstellen der mentalen Modelle und deren Vorstellungen sowie der Bewertung verschiedener Darstellungen zu finden. Beginnend wird eine `for`-Schleife mit allgemeinen Visualisierungstechniken gezeigt, wie sie bei Thytos (2014) exemplarisch zu vielen, ähnlichen Darstellungen zu finden ist.

Erwartungshorizont – Schleife mit allgemeinen Visualisierungstechniken

- zusätzliche Erklärungen sind hilfreich
- erleichtern das Verstehen der Bedeutung der Fachwörter
- Farben verdeutlichen einzelne Aspekte
- Farben lenken ab
- Schlechtere Lesbarkeit durch die Farben

Nach diesem Einstieg zu den Schleifen wird die Zielvorstellung der Schleife als kreisförmiger Kontrollfluss nach Weigend (2007) vermittelt. Hierbei sollen die Tutor*innen aktiv werden, indem sie verschiedene Alltagsbeispiele – sowohl nach Weigend (2007) als auch nach Wilson (2015) – bezüglich ihrer Eignung bewerten. Zusätzlich sollen sie entscheiden, welche Art von Schleife durch die Bilder suggeriert wird. Hier wurde sich bewusst dafür entschieden, sowohl nicht-passende

Beispiele als auch Beispiele für die `for`-Schleife und für die `while`-Schleife zu wählen. Auf diese Weise wird sicher gestellt, dass die Tutor*innen zumindest über ein passendes Beispiel zu jeder Art verfügen und auf dieses in ihren Erklärungen zurückgreifen können. Die Diskussionsfragen werden wie zuvor auch im Plenum besprochen, da so die Tutor*innen voneinander profitieren können. Ein besonderer Augenmerk liegt hier auf der expliziten Thematisierung von Sprüngen, da die Sprünge essentiell für das Verstehen von Schleifen sind.

Erwartungshorizont – Welche Art von Schleife wird durch die Bilder suggeriert?

- *Jahreszeitenzyklus:* Vorstellung einer unendlichen Schleife, da die Abfolge der Jahreszeiten nicht endet
- *Emails checken:* Vorgehen: checken, ob neue Emails vorhanden sind, wenn ja, lesen und abarbeiten $\rightsquigarrow$ while-Schleife
- *Pixel eines Bildes verändern:* Man weiß vorher wie viele Pixel das Bild hat, daher ist die Anzahl der Durchläufe bekannt $\rightsquigarrow$ for-Schleife
- *Treppe hochlaufen:* Man weiß vorher meist nicht, wie lang eine Treppe ist. Man läuft solange hoch bis die Treppenstufen aufführen $\rightsquigarrow$ while-Schleife

Abschließend zur Thematik der Schleifen und Iteration werden vier verschiedene Darstellungen zu einer datengesteuerten Wiederholung nach Weigend (2007) zur Diskussion und Bewertung präsentiert. Da die Einzelheiten der Bilder nur schwierig zu erkennen sind, sollen diese sprachlich vorgestellt werden. Wie zuvor auch, soll es die Tutor*innen dazu befähigen, über eine größere Vielfalt an Vorstellungen zum Erklären zu verfügen. Zudem können nicht-adäquate Vorstellungen mit bekannten abgeglichen werden. So gestaltet sich die Fehlersuche, was durch die Student*innen falsch verstanden wurde, einfacher.

Erwartungshorizont – Iteration durch Collection

- *Darstellung oben links:* Collection wird während der Iteration verändert (Weigend, 2007, 96).
- *Darstellung oben rechts:* Iterator wird nicht expliziert (Weigend, 2007, 96).

- *Darstellung unten links:* Hier passt alles: der Iterator wird durch den roten Punkt dargestellt und die Collection wird durch die Iteration nicht verändert (Weigend, 2007, 96).
- *Darstellung unten rechts:* Das Modell des Abhakens ist ein völlig neues und passt nicht zu den vorherigen Modellen. Bei einem Iterator wird nicht abgehakt (Weigend, 2007, 96-97).

Da die Rekursion aufgrund ihrer Wichtigkeit (2.3.3) noch in einem eigenen Modul behandelt wird, soll an dieser Stelle nur das Thema am Rande behandelt und vor allem die Problematik bei bestimmten Darstellungen aufgegriffen werden. Beispielhaft für problematische Darstellungen werden die Visualisierungen von Weigend (2007) zu den offenen und geschlossenen Modellen gezeigt. Den Tutor*innen werden diese zur Diskussion gestellt.

Erwartungshorizont – Problematik bei Visualisierungen der Rekursion

- *Offenes Modell:* Funktionen und nicht Daten erscheinen als Akteur.
- *Geschlossenes Modell:* Darstellung ist nur für endrekursive Prozeduren gültig, die Methode `mirror` ist aber nicht endrekursiv.
- Bei beiden Modellen sind die Kopien und damit der essentielle Teil der Rekursion – der Selbstbezug – nur schlecht deutlich.

Nachfolgend werden Vorstellungen bezüglich der Objektorientierung zu den Konzepten Klasse und Objekt vorgestellt. Aufgrund der mangelnden Visualisierungen zu den Themen werden diese rein sprachlich beschrieben. So ergeben sich die Präsentationsfolien als Zusammenfassung der verschiedenen Vorstellungen (2.3.4) genannt werden. Wie zuvor sollen durch die Sammlung der verschiedenen Vorstellungen die internen Vorstellungen der Tutor*innen expliziert, benannt und gegebenenfalls erweitert werden.

Da die Vorstellungen zum funktionalen Programmierparadigma in der Fachliteratur nur unzureichend behandelt werden, tauchten sie unter 2.3 nicht auf. So wird an dieser Stelle nur eine Anregung über

das Operator- und Maschinenmodell von Funktionen gegeben. Dieses wurde jedoch noch nicht auf seine Tauglichkeit und Tragfähigkeit in Verbindungen mit einer funktionalen Programmiersprache getestet. Diese Unvollständigkeit muss auch zwingend den Tutor*innen transparent gemacht werden.

3.3.4 Typische Fehler und Fehlvorstellungen

Das Modul *Typische Fehler und Fehlvorstellungen* ist zweigeteilt. Zuerst folgt ein Austausch der eigenen Vorerfahrungen der Tutor*innen mit der bestehenden Literatur. Anschließend werden verschiedener typischer Fehler und Fehlvorstellungen präsentiert, welche nach den Kategorien von Qian und Lehman (2017) unterteilt sind. Das zugehörige Material ist im Anhang unter B.3.2 und B.2.3 zu finden.

Da das Themenfeld der typischen Fehler und Fehlvorstellungen den Tutor*innen durch eigene Erfahrungen gut bekannt ist, bietet es sich an dieser Stelle die Vorerfahrungen dieser miteinzubeziehen. Dafür bietet sich aus folgenden Gründen die bereits bekannte Methode *Think-Pair-Share* in einer abgewandelten Weise an.

Zunächst sollen die Tutor*innen innerhalb 10 Minuten ohne Kenntnis der Fachliteratur selbst typische Fehler und Fehlvorstellungen formulieren. Auf diese Weise werden ihre Vorerfahrungen nicht durch die Kenntnis der Fachliteratur beeinflusst. Da davon ausgegangen wird, dass die Tutor*innen ein hohes Bedürfnis nach Absicherung und Mitteilung mit anderen Tutor*innen haben, soll die Phase in Partnerarbeit stattfinden. Somit bezieht sich die übliche personale Trennung bei der Methode Think-Pair-Share hier nicht auf einzelne Tutor*innen sondern auf die Paare von Tutor*innen.

Erwartungshorizont – Liste von typischen Fehlern erstellen

Es wird von folgender Annahme ausgegangen. Die erfahrenen Tutor*innen nennen, dass zu den typischen Fehler gehört, dass die Student*innen nicht wissen, wie sie anfangen sollen. Als Folge daraus haben die Student*innenund Probleme mit dem Programm-Design. Die

Probleme und damit Fehler beim Programmieren, die das Programm-Design und des Verstehen des Zwecks des Programms betreffen, sind relativ häufig unter den Fehlern Garner u. a. (2005).

- Auflistung von sehr speziellen Fehlern in mehreren Bereichen oder nur Nennung von Bereichen bzw. Metafehler
- Bereich der Metafehler: Dokumentation nicht ordentlich, Tests vergessen
- Bereich der Strukturen: Variablen nicht initialisiert, syntaktische Fehler wie Tippfehler, falsche Namen – `stop` statt `break` –, Klammern oder Satzzeichen.
- Bereich der Rekursion: Rekursionsanker vergessen, kein rekursiver Aufruf vorhanden.
- Bereich der Schleifen: Schleife terminiert nicht bzw. Endlosschleife, Schleife wird nur ein einziges Mal durchlaufen (Extremfälle).
- Bereich der Objektorientierung: sehr ausführlich – Klasse vs. Objekt – oder Tutor*innen vergessen, dass das Thema zu *FOP* gehört; Objekte werden in Funktionen verwendet, ohne sie vorher zu erstellen.
- Fehlende Vorbereitung: Begriffe wurden nicht nachgeschlagen bzw. verstanden und daher falsch verwendet, bspw. Zugriffsmodifikatoren – `private` usw. – oder Schlüsselwörter wie `static`.
- Die Bereiche der Planung und des Designs von Programmen werden wahrscheinlich vernachlässigt werden, da diese in den bisher besuchten Lehrveranstaltungen auch meist keine Rolle gespielt hat.

Es muss nicht unbedingt sein, dass die erfahrenen Tutor*innen mehr Fehler aufzählen können. So sind die Tutor*innen, die erst am Ende des zweiten Fachsemester angelangt sind, inhaltlich noch näher an den Fehlern der Programmieranfänger dran. Nach der Taxonomie von Dreyfus und Dreyfus zitiert nach Winslow (1996) bewegen sich Tutor*innen – je nach Erfahrungen – zwischen den Stadien *Novice*, *Advanced Beginner* und bestenfalls in *Competence*. So sind sie selbst meist noch als Anfänger*innen zu bezeichnen.

In der nachfolgenden zweiten Phase des *Think* findet der Austausch der Vorerfahrungen des Tutor*innen-Paars mit der Fachliteratur statt. Das Ziel ist hierbei, dass die Tutor*innen überprüfen können, inwieweit sie über mehr oder weniger fundierte Kenntnisse bezüglich der Thematik verfügen. Um den Tutor*innen nicht nur einen qualitativen sondern auch einen quantitativen Vergleich zu ermöglichen, wurde sich dafür entschieden die Sammlung der Fehler nach Garner u. a. (2005) zu verwenden. Zudem ist die geschriebene Sammlung der Fehler – siehe A.1 – auch im englischsprachigen Original knapp und gut verständlich formuliert. Die Relevanz des quantitativen Vergleichs wird dadurch unterstützt, dass Fehler schlichtweg nicht gleich häufig auftauchen siehe 2.2. Daher ist es wichtig für die Tutor*innen zu wissen, welche die häufigsten Fehler sind, damit man diese besser erkennen und die Student*innen dabei unterstützen kann. Hierbei sollen die Tutor*innen nicht zu jedem denkbaren Fehler ein gutes Vorgehen und eine gute Erklärung parat haben. Zu den häufigsten Fehlern sollten sie jedoch darüber verfügen. Um die Tutor*innen in einem effizienten und zielgestützten Vergleich zu fördern, wurde sich dafür entschieden, dass diese unter drei vorgegeben Aspekten vergleichen sollen.

1. Welche Probleme hattet ihr auch so aufgeschrieben?
2. Welche hattet ihr nicht?
3. Was hattet ihr auch so gemeint, habt es anders formuliert?

Ebenso lautet die Arbeitsanweisung zum Lesen der Sammlung von Fehlern *Überfliegt*, damit die Tutor*innen sich den Fokus auf das Vergleichen und nicht auf das ausführliche Studium der Literatur legen. Für diesen Arbeitsauftrag haben die Tutor*innen 10 Minuten Zeit, wobei hiervon 5 Minuten für das Überfliegen der Literatur und 5 Minuten für den Vergleich gedacht sind.

Erwartungshorizont – Vergleich der Liste nach Garner u. a. (2005) mit der eigenen

- Die meisten Tutor*innen-Teams werden aufgrund der begrenzten Zeit – 10 Minuten zum Erstellen – maximal die Hälfte der genannten Aspekte von Garner u. a. (2005) selbst aufgeführt haben. Da dies aber immerhin 13 Aspekte wären, kann diese Leistung dennoch als gut eingeschätzt werden.

- Wie bereits beim Erstellen angedeutet, werden die Tutor*innen die Bereiche Planung und Design vernachlässigen. Daher haben sie mit den Punkten *Tools*, *Problems with basic structure*, *Understanding the task* und *Stuck on program design* keine Übereinstimmung mit ihrer erstellten Liste.
- Durch den Austausch stellen die beiden Tutor*innen-Teams fest, dass sie ähnliche Fehler aufgelistet haben oder in ganz andere Richtungen gedacht haben. Jedoch ist es sehr unwahrscheinlich, dass sie identische Aspekte aufgeführt haben. Daher ist der Austausch für alle Tutor*innen gewinnbringend, da sie von dem Vorwissen der anderen profitieren. Durch den Austausch werden nicht zwingend neue Aspekte von Fehler gelernt, sondern die bereits bekannten Fehler wieder ins Bewusstsein gerufen.
- Beim Vergleich mit der Liste nach Garner u. a. (2005) werden die Tutor*innen ähnliche Erfahrungen machen; sie kennen die genannten Fehler durchweg, aber sie sind nicht unbedingt in ihrem Bewusstsein vorhanden, sodass sie sie aktiv abrufen und formulieren können.

In der anschließenden *Pair*-Phase von 10 Minuten werden die vorherigen Zweier-Teams zu einer Vierergruppe gepaart. In diesen soll zunächst ein Austausch über die zwei vorherigen Phasen untereinander stattfinden. In dieser Phase ist einerseits die Möglichkeit gegeben sich zu versichern, dass das andere Paar auch nicht in der knappen Zeit alle Fehler gewusst und genannt hat. So können aber auch die Tutor*innen-Teams sich gegenseitig in ihren Ideen und Formulierungen bereichern und voneinander lernen. Dies stellt stellvertretend, dass die Tutor*innen nicht alleine agieren und lehren müssen. Andererseits bietet die Phase den Raum kritisch mit der gegebenen Literatur umzugehen, und für sich selbst zu überprüfen inwieweit diese Sammlung von Fehler auf die betreute Lehrveranstaltung *FOP* übertragbar ist.

Um den bereits angesprochenen quantitativen Vergleich zu fördern, sind die Vierer-Gruppen aufgefordert innerhalb der gleichen Phase die nach ihrer Meinung fünf häufigsten Fehler in der Liste von B.3.2 zu bestimmen. Hierbei sehen die Tutor*innen die Graphik 2.2, damit sie die ungleiche Verteilung der Fehler in ihrer Entscheidung berücksichtigen können. Hierbei stimmt die Nummer des Problems nicht mit der

Nummerierung auf dem Arbeitsblatt B.3.2 überein.

Erwartungshorizont – Zuordnen der Fehler zur Häufigkeitstabelle nach Garner u. a. (2005)

- Wenn den Tutor*innen die Bereiche der Planung und des Designs schon in den Phasen zuvor bewusst waren oder sie ihnen im Austausch mit den anderen Tutor*innen in der Vierergruppe bewusst wurden, ist es wahrscheinlich, dass sie diese zu den häufigeren Fehlern einordnen. In diesem Fall kommt es zu keiner Überraschung bei der Auflösung der korrekten Reihenfolge der Häufigkeit. Andernfalls kommt nun spätestens an dieser Stelle die Erkenntnis, dass diese Bereiche nicht zu gering geschätzt werden können.
- Keine*r der Tutor*innen hat sich wahrscheinlich im Vorfeld mit der quantitativen Auswertung von Fehlern von Programmieranfängern beschäftigt. Somit wird die Entscheidung zur Bestimmung der fünf häufigsten Items aus einer diffusen Mischung aus Bauchgefühl und persönlichen Vorerfahrungen getroffen. Aufgrund des intuitiven Vorgehens bei der Entscheidungsfindung sollte die Zuordnung in kurzer Zeit vonstatten gehen.

Den Vergleich der Vorerfahrungen der Tutor*innen mit der Fachliteratur schließt die *Share*-Phase ab. In dieser werden die Ergebnisse – die Nummern der fünf häufigsten Fehler – der Vierer-Gruppen an der Tafel zusammengetragen. Erst danach wird über die Präsentation das Ergebnis nach Garner u. a. (2005) präsentiert. Dies hat den Zweck, dass einerseits ein Bild von den Ideen der Lerngruppe eingefangen werden kann, und andererseits können die Tutor*innen ihre Ideen mit denen der anderen vergleichen. Es ist davon auszugehen, dass einige Nummern von mehreren Vierer-Gruppen genannt werden, da sie ähnliche Vorerfahrungen teilen. Anschließend wird das Ergebnis kurz besprochen, wobei sich an den Impulsen der Lerngruppe orientiert wird. Falls es von einigen Tutor*innen Diskussionsbedarf dieser stumme Impuls reicht, sich zu äußern, soll dieses auch so hingenommen werden. Andernfalls, kann dies über die Frage, ob jemand ein Aha-Erlebnis hatte oder, ob das Ergebnis überraschend sei angeleitet werden.

Als Überleitung zur Präsentation und zum Herstellen von Zusammenhängen erfährt die darauf folgende Folie – Wie geht es weiter? – ihre Bedeutung. Es soll in der Schulung tunlichst vermieden werden, dass die einzelnen Module ohne Bezüge zueinander isoliert stehen. Dies soll den Tutor*innen damit verdeutlicht werden. Zudem wird so der Übergang zur nachfolgenden Präsentation von spezifischen Fehlern geschaffen.

Die Präsentation der typischen Fehler und Fehlvorstellungen folgt in Stichpunkten der Aufzählung der Fachliteratur (2.4). Hier liegt der Fokus auf dem Vorstellen der Fachliteratur zu typischen Fehlern und dem zugehörigen rein deklarativen Wissen. Daher ist die Methode der Präsentation als Instruktion anderen Methoden vorzuziehen. Das Finden von Fehlern wurde im Einstieg bereits behandelt und wird gemeinsam mit dem Reagieren auf Fehler im nachfolgenden Modul der Simulation thematisiert. Bei der Formulierung der Stichpunkte wurde sich dafür entschieden, dies in möglichst ganzen Sätzen zu formulieren. Auf diese Weise können die Tutor*innen das Material zur Schulung zum Nachlesen für die Vorbereitung auf einzelne Kleingruppentermine bzw. Themenblöcke verwenden.

Die Aufteilung in syntaktische, konzeptionelle und strategische Fehler folgt der Aufteilung nach Qian und Lehman (2017) (2.4) und ist nach Programmierkonzepten sortiert. Dies erleichtert das Verstehen der Zusammenhänge und das Nachlesen zu einem bestimmten Thema. Da es für die Tutor*innen jedoch keine Rolle spielt, welcher Literatur die vorgestellten Fehler entnommen wurden, wurde auf Quellenverweise verzichtet. Auf den allgemeinen Bezug zur Fachliteratur soll jedoch mündlich hingewiesen werden. Zur weiteren Kenntnisnahme schlichtweg auf die Folien im Anhang B.2.3 verwiesen.

Das Modul abschließend wird eine allgemeinpädagogische Forderung an die Tutor*innen gestellt. Bei der Präsenzphase – bestehend aus den Kleingruppen und Sprechstunden – handelt es sich um Lernphasen, nicht um Leistungsphasen. Dieser Unterschied ist explizit und transparent zu machen. Dies sollen auch die Tutor*innen gegenüber

den Student*innen vertreten. So kommt es bei permanenter Leistungssituation zu Vermeidungsstrategien (Mischke 2004) zitiert nach Paradies u. a. (2011), wie sie auch auf der Folie exemplarisch präsentiert werden. Relevant ist hier vor allem die Haltung der Tutor*innen. Wenn jedoch die Student*innen Fragen stellen anstatt ihre Fehler zu vertuschen, erhalten die Tutor*innen die Möglichkeit *erklären zu können*. Andernfalls sind die hier vorgeschlagenen Maßnahmen, damit Tutor*innen besser erklären können, irrelevant.

3.3.5 Simulation

Das Ziel dieses Moduls *Simulation* ist das Anwenden der bisher gelernten Inhalte und das Erleben der Rolle der Tutor*in. Letzteres ist vor allem für die große Anzahl der Tutor*innen wichtig, da sie zuvor noch nicht diese Rolle eingenommen haben. Als Methode des Anwendens wurde sich für ein pädagogisches Rollenspiele entschieden.

Der Sinn des pädagogischen Rollenspiels – hier Simulation genannt – ist, dass die Teilnehmer*innen auf die Wirklichkeit vorbereitet werden (Günther, 2019, 2). Die Wirklichkeit der Tutor*innen meint hier ihre Rolle als Tutor*innen und den Umgang in Bezug auf das *Erklären können* mit den Student*innen. Da das pädagogische Rollenspiel „ein effektives Gruppenverfahren mit handlungsorientierten Lösungen“(Günther, 2019, 14) ist, ist es für die Schulung als Methode besonders geeignet. So können besonders diejenigen Tutor*innen, die bisher nicht als Tutor*in tätig waren, im Vorfeld die Gespräche – auf handelnde und reflektierende Weise – üben. Zudem können die Tutor*innen in einem geschützten Rahmen die benötigten Fertigkeiten in der Praxis erlernen (Günther, 2019, 19). Die positiven Merkmale dieser Methode sind, dass die erlebten Inhalte – das Erklären können – besser im Gedächtnis bleiben und bewusster gemacht werden (Günther, 2019, 13). Dies wurde von (Walker u. a., 2007, 218) zitiert nach (Günther, 2019, 13) gezeigt, wobei Techniken des Rollenspiels auch in Programmen aufgenommen wurden, die das Erlernen von Werten und Standards betreffen.

Neben den speziellen Vorteilen der Methode des pädagogischen Rollenspiels, wie sie bereits zuvor erläutert wurden, ist durch die Betrachtung des Lernlevels nach Bings und Tang zitiert nach (Ulrich,

2016, 44) ein Nutzen gegeben. Durch diese Methode kommt es zum Anforderungsbereich des *Anwendens* (Ulrich, 2016, 47), welches das höchste Lernlevel darstellt, was durch Lehrmethoden erreicht werden kann. Die höchste Stufe des Lernlevels, das *Theoretisieren*, kann nur in Qualifikationsarbeiten, guten Hausarbeiten Student*innen höherer Semester und Masterarbeiten im Rahmen der Hochschullehre erreicht werden (Ulrich, 2016, 45).

Der Ablauf der Simulation folgt dem einer Spielsitzung, wie sie im Abschnitt *Rollenspiel als Weiterbildungsbaustein* in Günther (2019) beschrieben ist. Hierfür kennzeichnend ist, dass die Tutor*innen „keinen so großen fachlich-sachlichen Vorsprung vor ihrer unmittelbaren Zielgruppe“(Günther, 2019, 19) – den Student*innen – haben. Dies trifft für die meisten der Tutor*innen zu, da sie selbst erst das zweite Fachsemester vollendet haben. Zuerst findet eine Einleitung in die Simulation statt, welche eine kurze, mündliche Begründung umfasst, was der Sinn und Zweck dieser Phase ist. Die Auswahl der Themen ist bereits im Vorfeld erfolgt, da die Tutor*innen die Module und Inhalte der Schulung nicht vorher kennen. So sind sie wahrscheinlich nicht in der Lage sind, passende Aufgaben unmittelbar zu generieren. Des Weiteren konnten die Aufgaben durch die Auswahl meinerseits nicht nur auf die zuvor besprochenen Themen angepasst werden. So konnte die Aufgaben der Übungsblättern der *FOP* benutzt werden, sodass die Tutor*innen an realen Aufgabenstellung der Lehrveranstaltung im Wintersemester 2018/2019 üben können. Die Themen behandeln sowohl Aufgaben zu beiden Programmiersprachen – Racket und Java –. Sie sollen unterschiedlich aufwendige und schwierige Aufgaben beinhalten und zudem die Möglichkeit geben, die bisher besprochenen Visualisierungen, mentale Modelle und Fehler aufzugreifen.

Damit die Simulation zügig abläuft, wurde sich entschieden, sie mittels Rollen zu strukturieren. Diese Rollen sind Tutor*in, Student*in und Beobachter*in. Die ersten beiden Rollen sind selbstverständlich. Die Rolle der Beobachter*in ist wichtig, damit vor allem die Person in der Rolle der Tutor*in Feedback nach der Erprobung erhält. Das Feedback soll dem Zweck dienen, dass die erfahrenen Tutor*innen eine Rückmeldung auf ihre bisherige Art des Erklärens erhalten. Die Tutor*innen, die zum ersten Mal in dieser Rolle sind, erhalten eine

Gelegenheit sich in einem geschützten Rahmen auszuprobieren. Zudem erfahren sie durch das Feedback eine direkte Rückmeldung und können dieses direkt in den Erprobungen umsetzen. Damit jede der Tutor*innen sich erproben kann, wurde sich dafür entschieden, dass die Rollen rotieren, sodass sich jede in jeder Rolle einmal befunden hat. Damit die Gespräche zwischen den Tutor*innen nicht allzu oberflächlich sind, wurde sich für eine Vorbereitungszeit entschieden. Die Tutor*in kann sich so in Erinnerung rufen, welche Vorstellungen und typischen Fehler mit der ausgewählten Aufgabe einhergehen. Die Student*in kann sich überlegen, was die Fragen sind bzw. eine eventuell fehlerhafte (Teil-)Lösung anfertigen. Die Beobachter*in kann sich überlegen, auf welchen Aspekte – wie in den Fragen beschrieben wurde – sich besonders konzentriert werden soll.

Erwartungshorizont – Simulationen

Für alle aufgeführten Punkte gilt, dass sie eintreten können aber nicht müssen. So zeigen nicht alle Tutor*innen mit den jeweiligen Voraussetzungen das entsprechende Verhalten; jedoch ist es für einen Teil von ihnen wahrscheinlich.

- Tutor*innen mit Vorerfahrungen als studentische Mitarbeitende kennen diese Art von Simulationen schon aus vorherigen Semestern und fangen zügig mit den eigentlichen Simulationen an.
- Tutor*innen mit Vorerfahrungen als studentische Mitarbeitende simulieren zu schwierige *Student/innen*, die entweder noch gar nichts verstanden haben, nicht verstehen wollen oder stellen absichtlich schwierige, weiterführende Fragen.
- Tutor*innen, die zum ersten Mal überhaupt Tutor*in sind, könnte die Rolle der *Tutor/in* schwer fallen, da sie diese Rolle bisher noch nicht eingenommen haben.
- Tutor*innen, die zuvor keine Lehr-Lern-Gespräche simuliert hatten, könnte die Rolle der *Beobachter/in* schwer fallen. So wissen sie nicht, was genau sie beobachten sollen und geben kein spezifischeres Feedback als richtig oder falsch zurück.
- Tutor*innen, die bisher noch nie Simulationen durchgeführt haben, neigen zu extremen Verhalten als *Student/in*, welches nicht

als realitätsnah eingestuft werden kann: übermäßig frech, provozierend, geben sich keine Mühe den Erklärungen zu folgen und auf diese einzugehen oder sprechen völlig am Thema vorbei.

- Tutor*innen mit Vorerfahrungen in der Betreuung der Veranstaltung *FOP* kennen die Aufgaben zur Simulation bereits und benötigen daher weniger Zeit zum Vorbereiten bzw. nutzen die Zeit effektiver für Gespräche.
- Tutor*innen, die sich persönlich nahe stehen, können innerhalb der Simulation nicht davon abstrahieren und neigen zum Scherzen und nehmen damit die Simulation nicht ernst.
- Tutor*innen, die schnell informatische Aufgaben und deren Lösungen kognitiv aufnehmen können, werden sich eher für längere Aufgabenstellungen entscheiden.
- Unsichere, ungeübte bzw. unmotivierte Tutor*innen werden eher diejenigen Aufgaben aussuchen, bei denen die Erklärungen oder die Fehler und Fehlvorstellungen explizit durch die vorgehenden Module angesprochen wurden. Auf diese Weise kann die eigene Arbeitsleistung reduziert werden.
- Gerade bei ungeübten Tutor*innen – unabhängig, ob die Themen der *FOP* schon länger zurück liegen oder nicht – kann es vorkommen, dass sie etwas falsch oder nicht vollständig richtig erklären. Im Idealfall wird dies auf höfliche, konstruktive Art durch die *Beobachter/in* aufgegriffen.

Aufgrund des Umstandes, dass nur unspezifisches Feedback an die *Tutor*in* zurückgegeben wird bzw. Unsicherheit bei den *Beobachter*innen* besteht, wurde sich für folgende Ergänzung entschieden. Am Ende des Aufgabenblattes wurden sieben *Tipps zum Beobachten und Feedback geben* aufgeführt. Hierbei wurden sowohl die Visualisierungen und die Exemplarität als auch die W-Fragen der Diagnose siehe 3.3.2 und der Verweis auf die grundlegenden Regeln zum Feedback geben – wie in der allgemeinen Tutor*innenschulung behandelt – aufgeführt. Die Tipps sind als Fragen ähnlich einer Checkliste aufgeführt, damit diese als Art eines Qualitätschecks seitens der Beobachter*in benutzt werden können.

3.4 Methodischer Ablauf - nachmittags

3.4.1 Problemlösestrategien

Dieses Modul der Tutor*innenschulung richtet sich an eine Vermittlung der Erkenntnisse, die im Abschnitt 2.5 des wissenschaftlichen Fundaments zusammengetragen wurden. Ähnlich zu dem Modul der typischen Fehler und Fehlvorstellungen 3.3.4 ist dieses aufgeteilt in das Aufgreifen des Vorwissens und Verbindung dessen zur Fachliteratur und einer anschließenden Vorstellung exemplarischer Strategien. Die Folien der Präsentation und die zugehörigen Arbeitsblätter sind unter B.2.4 und B.3.4 zu finden.

Im Gegensatz zu der Thematisierung von Fehlern ist den Tutor*innen nicht unbedingt klar, was unter Problemlösen und deren Strategien verstanden wird. Somit wird dem Aufgreifen des Vorwissens eine Erklärung mittels einer Präsentation ebenjener Begrifflichkeiten vorangestellt. Die frontale Instruktion eignet sich für ein effizientes Darstellen eines unbekannten Sachverhalts und Klärung von Begriffen besonders gut – beschrieben im Abschnitt 2.5.1. Der Vorteil der Darstellung des Problemlöseprozesses nach Polya – als Begriffsklärung – liegt in der guten Übertragbarkeit auf das Programmieren wie es beispielsweise von (Winslow, 1996, 19) ausgeführt wird. Was allgemein unter einem *Problem* zu verstehen ist, ist für die Tutor*innen in der Schulung und ihrer Tätigkeit nicht relevant und kann schlichtweg mit Programmieraufgabe übersetzt werden. Da die Nummerierung der Schritte des Prozesses eine lineare Lesart ermuntert, soll den Tutor*innen beispielhaft vorgestellt, dass die Phasen des Prozesses durchaus mehrfach durchlaufen werden. Die Definition der Strategien zum Problemlösen wird über die Formulierungen nach Schubert und Schwill (2011) und Kaplan (2012) aufgegriffen und vielmehr umschrieben als definiert. Zudem werden Synonyme für die Strategien vorgestellt, damit die Tutor*innen in der Lage sind, bei Lesen von Literatur mit den verschiedenen Begriffen umzugehen.

Nachfolgend sollen die Eigenschaften des Problemlöseprozesses, deren Implikationen für das Handeln der Tutor*innen und die Gründe des Lehrens von Strategien aufgezeigt werden. Die Eigenschaften

sind von Bedeutung, da die Tutor*innen daran erkennen wie Programmieren abläuft – unabhängig von der Programmiersprache, der Aufgabe und der Programmierer*in. Zudem stellt die Erforderlichkeit der Strategien die Verbindung zwischen dem Problemlösen – dem Programmieren – und den Strategien her. Die Implikationen für die Tutor*innen – abgeleitet von den Eigenschaften – sollen den Tutor*innen erneut Handlungsanweisungen vermitteln. Erst wenn einem bewusst ist, dass Programmieren wesentlich von Fehlern und deren Korrekturen geprägt ist, ist man in der Lage dies als Herausforderung zu begreifen. Genau darin sollen die Tutor*innen die Student*innen unterstützen. Durch die Eigenschaft, dass der Prozess des Problemlösens gerade nicht algorithmisch und linear abläuft, wird insbesondere deutlich, dass mehrere Lösungen zum Ziel und zur Bewältigung des Problems möglich sind. Damit soll der Verwendung von *Muster*lösungen als einzig richtigen oder schönen Weg entgegen getreten werden. Dies spielt vor allem beim Lesen und Bewerten der Lösungen der Student*innen eine große Rolle. Um die Verwendung von Strategien zu fördern, werden die Tutor*innen aufgefordert gemeinsam mit den Student*innen zu reflektieren. Dafür werden ihnen konkrete Fragen zur Rückschau vorgestellt.

Da die Tutor*innen nur wenige Kenntnisse über das Problemlösen und dessen Strategien aufweisen, ist es vonnöten die Gründe vorzutragen, welche für das Lehren von Strategien sprechen. Hierbei wird sich zunächst noch auf allgemeine Strategien bezogen, während später explizit auf informatikspezifische Strategien wie Pattern eingegangen wird. Hierbei wurden die Gründe aus 2.5.1 übernommen und besonders hervorgehoben, dass Strategien keine Leistung verbessern, sondern sie vielmehr als Hilfestellung anzusehen sind.

Nachfolgend soll knapp eine Verbindung des Vorwissens der Tutor*innen mit den Strategien der Fachliteratur aufgebaut werden. Im Idealfall sollte jede Tutor*in sein Vorwissen abfragen, sich anschließend mit der Fachliteratur beschäftigen und abschließend vergleichen.

So kann an das Vorwissen angeknüpft und mit der Fachliteratur erweitert und abgeglichen werden. Aufgrund der zeitlichen Beschränkung der Tutor*innenschulung und der Vielzahl von anderen wichtigen Themen, wurden die zwei gedanklichen Schritten auf die Lerngruppe aufgeteilt. Die verwendete Methode hierbei ist erneut eine Abwandlung der Think-Pair-Share-Methode. In der *Think*-Phase notieren die Tutor*innen entweder bereits bekannte Strategien – Arbeitsauftrag Kreis – oder lesen eine Zusammenstellung von Strategien nach Kaplan (2012) – Arbeitsauftrag Quadrat –. Beide Arbeitsaufträge sind unter B.3.4 zu finden.

Nachfolgend wird das Vorwissen der Student*innen und der daraus resultieren Erwartungshorizont zum Arbeitsauftrag Kreis beschrieben. Das Vorwissen der Student*innen wird vor allem durch die spezielle Wortwahl von einzelnen Strategien bestimmt, da bisher in den ersten Semestern des Studiums Strategien als Gesamtes nicht gelehrt wurden. Eine Ausnahme stellt die Kanonik *Software Engineering* – Bestandteil des Pflichtbereichs der Studiengänge BSc. Informatik und BSc. Wirtschaftsinformatik – dar. Diese beschäftigt sich explizit mit Entwurfsmuster (Design Patterns) TU Darmstadt (2015). Zwei Drittel der Student*innen haben zum Zeitpunkt der Schulung erst das Ende ihres zweiten Fachsemesters erreicht (3.1). So haben maximal zehn Teilnehmende bereits die Veranstaltung Software Engineering besucht und damit die entsprechenden Vorkenntnisse über Entwurfsmuster erworben. Daher soll die Think-Phase in Zweiergruppe durchgeführt werden, damit die Tutor*innen gegenseitig von ihrem Vorwissen profitieren können und es zu einem Lernzuwachs kommt. Falls die Tutor*innen nur über sehr wenig Vorwissen verfügen, fällt es ihnen in der Zweiergruppe leichter, sich an bereits behandelte Strategien aus den bisherigen Veranstaltung zu erinnern.

- *Top-Down:* explizit in der Veranstaltung *FOP* als eine Art der Programmierung genannt.
- *Bottom-Up:* explizit in der Veranstaltung *FOP* als eine Art der Programmierung genannt; dies entspricht nach Kaplan (2012) der Strategie *Break it down*, da im zweiten Teil der Erklärung zuerst die kleineren Teile gelöst werden und anschließend zur Lösung des originalen Problems zusammengesetzt werden.

- *Divide & Conquer bzw. Teile und Herrsche:* explizit in der Verbindung mit dem Algorithmus *Quicksort* in der Veranstaltung *Algorithmen und Datenstrukturen* genannt; das Aufteilen des Programms in kleinere Parts entspricht der Strategie *Break it down* nach Kaplan (2012).
- *Testen, Debuggen:* als eine Art, um auftauchende Fehler zu lösen, indem man sie findet; dies entspricht nach Kaplan (2012) der Strategie *Test, test, test*.
- *Beispiele finden:* die Prozedur an Beispielen erarbeiten und von den Vorgehen verallgemeinern, um die Lösung des Problems zu finden; dies entspricht bei Kaplan (2012) *Find and use examples*.

Erwartungshorizont – Arbeitsauftrag Quadrat

- Je nachdem wie die Student*innen der englischen Sprache mächtig sind, brauchen sie für das Bewältigen des Arbeitsauftrags unterschiedlich lange.
- Abhängig von ihren eigenen Erfahrungen mit Problemlösestrategien – egal, ob explizit oder implizit erfahren – können die Tutor*innen die elf genannten Strategien besser mit ihrem eigenen Wissen abgleichen und damit schneller aufnehmen.

In der Phase des anschließenden Austauschs – die *Pair*-Phase zwischen den Zweiergruppen – soll gerade jene Ergänzung und Verflechtung zustande kommen. So vollziehen diejenigen Student*innen, die ihr Vorwissen verschriftlicht haben, die Punkte der Fachliteratur nach; die anderen vollziehen gedanklich nach, ob sie auch über das Vorwissen der anderen verfügt hätten. Dies soll in einem Gespräch zu viert geschehen. Hierbei kommt es vermutlich zu einer Bewertung der Strategien nach Kaplan (2012). Das Vorwissen zu den Strategien wurde bereits aufgeführt und wird daher nicht mehr genannt.

Erwartungshorizont – Austausch in der Vierergruppe

- Tutor*innen haben spezifischere Strategien als die genannten nach Kaplan (2012) erwartet.
- Student*innen halten die Strategien nach Kaplan (2012) nicht für Strategien. Sie würden diese eher unter dem Begriff informatische Allgemeinbildung fassen, die man sich im Laufe des Studiums aneignet.

Die *Share*-Phase im Plenum greift die Bewertung der Problemlösestragien aus den Vierer-Gruppen auf. Die Frage, ob die Strategien spezifisch für die Informatik waren, führt zum Kontinuum der Strategien. Auf dieser Folie werden noch weitere allgemeine Problemlösestrategien aufgeführt, welche sich auf das Programmieren beziehen lassen. An dieser Stelle sollte auch auf die Strategien Top-Down und Bottom-Up eingegangen werden, wenn diese nicht selbst von den Tutor*innen genannt werden. Um der Erwartung der Tutor*innen entgegen zu kommen und informatikspezifische Strategien zu lehren, ist der Rest der Präsentation einer Vorstellung dieser gewidmet.

Zu Beginn steht eine Sammlung von Begriffen, welche im Kern die informatikspezifischen Strategien innehaben. Zuerst soll die Frage, was unter dieser Sammlung verstanden wird, an die Tutor*innen zurück gegeben werden. Dies dient dazu, die Vorerfahrungen und bekannten Begriffe der Tutor*innen aufzugreifen. Hier werden die folgende Aspekte, Nennungen erwartet: Entwurfsmuster, vorstrukturierter Code, Kochrezepte.

Die Vielzahl der Sichtweisen und Begriffen dient dazu, einerseits die Ungenauigkeit der Literatur zu vermitteln und anderseits aufzuzeigen, dass nicht immer das Gleiche darunter verstanden wird. Um die Vorerfahrungen derjenigen Tutor*innen aufzugreifen, die sich bereits mit Softwareentwicklung beschäftigt haben, wurde diese Sicht ebenfalls übernommen. Die Bedeutung der Patterns wird aus der Literatur (2.5.2) übernommen. Hierbei sollen die Tutor*innen ermuntert werden, strategisches Wissen bei Erklärungen zu verwenden und den Student*innen nahezubringen. Hierbei sollen jedoch auch die Grenzen aufgezeigt werden. Patterns sind weder ein Universalmittel beim Programmieren, noch sind sie in jeder Situation hilfreich.

Die Präsentation schließt mit einer Vorstellung von ausgewählten Patterns, die zum Erklären verwenden werden können. Dieses Modul wurde bewusst knapp gehalten, da das Lehren von Strategien nicht allein durch die Tutor*innen aufgefangen werden kann. Die vorgestellten Pattern sind aus Proulx (2000) entnommen, wobei aus Gründen der Knappheit nur teilweise oder nicht vorgestellt werden. Für interessierte Tutor*innen wird auf die Sammlung von Bergin verwiesen.

Die Auswahl besteht aus den folgenden: Read-Process-Write,

Encapsulation, Strategien für Schleifen als Repetition, Traversal und eine Kombination von Pattern. Alle ausgewählten Pattern lassen sich einerseits knapp den Tutor*innen vermitteln und können in Erklärungen für Student*innen der *FOP* Verwendung finden.

Bei dem Read-Process-Write Pattern wurde sich bewusst für ein sehr einfaches Beispiel entschieden, welches sowohl sprachlich als auch mit Code-Stücken dargestellt wird. Hieran soll deutlich werden, dass Pattern sehr wohl in einfachsten Programmieraufgaben zum Einsatz kommen.

Das Encapsulation Pattern gibt den Tutor*innen Tipps, unter welchen Bedingungen es sinnvoll ist, Hilfsfunktionen zu schreiben. Diese Tipps sind besonders für noch nicht so erfahrenen Tutor*innen sinnvoll, da sie selbst noch nicht so lange programmieren. Für alle Tutor*innen gilt, dass sie sich in diesen Situationen zwar unbewusst für eine Hilfsfunktionen entscheiden würden. Für das Erklären und Lehren ist sinnvoll, diese explizit zu machen, um sie den Student*innen zu vermitteln.

Durch die Aufteilung der Schritte, eine Schleife zu schreiben, können die einzelnen Schritte separat abgehandelt werden. Je nach Reihenfolge der Schritte kommen weitere Vorteile dazu. Dieses sehr spezifische Pattern soll den Tutor*innen eine Möglichkeit geben, wie eine Schleife in kleinere, leichter zu verstehende Schritte aufgeteilt werden kann.

Das Traversal Pattern wurde ausgewählt, da besonders die einfach linerare Traversierung in Java und die Traversierung einer Liste in Racket unabdingbar ist. Hierbei dient das Pattern vor allem, um Gemeinsamkeiten der einzelnen Arten des Durchschreitens herzustellen. Zudem sollen die Student*innen in die Lage versetzt werden, von einer konkreten Programmieraufgabe zu abstrahieren. So können sie von einer erfolgreich gelösten Traversierung das Vorgehen auf eine andere übertragen.

Die Mächtigkeit der einzelnen Pattern ist vor allem durch ihre Kombination gegeben. Dies soll an einem Beispiel deutlich gemacht werden, dass selbst mit dieser geringen Auswahl an Pattern, sich Probleme noch weiter unterteilen lassen. Auf diese Weise sollen die Student*innen durch die Tutor*innen ebenfalls dazu gebracht werden ihre Probleme aufzuteilen. Nach der Aufteilung sind jedoch gerade die informatikspezifischen Strategien von Bedeutung, um die einzelnen

Teilprobleme zu bearbeiten.

3.4.2 Rekursion

Dieses Modul beginnt mit einer Überprüfung der eigenen, mentalen Modelle der Tutor*innen. So können die Tutor*innen ihre eigenen Fähigkeiten selbst einschätzen bevor sie Erklärungen dazu erhalten. Da Kahney (2009) bereits einen Test für das Vorhandensein bestimmter mentaler Modelle der Rekursion entwickelt hat, wurde sich dafür entschieden. Nicht alle der Expert*innen und nur ein Zehntel der Noviz*innen verfügten im Test über das geeignete Kopien-Modell von Rekursion (Kahney, 2009, 216–218). Dem folgend werden nicht alle Tutor*innen die Verinnerlichung des Kopien-Modells erkennen lassen. Dies bestärkt. dass die Tutor*innen ihre mentalen Modellen zuerst selbst überprüfen sollen, bevor sie die Theorie dahinter verstehen und anschließend anwendungsbezogen erklären können.

Der Test von Kahney (2009) wurde für die Tutor*innen angepasst. Die digitale Schriftsetzung ist leichter zu lesen. Die Programmiersprache SOLO wird den Tutor*innen wahrscheinlich schwer fallen. SOLO ist zwar an LOGO angelehnt (Mulholland und Eisenstadt, 1998, 2) und ähnlich Pseudocode zu lesen, jedoch ist die Abstraktion in einer ungewohnten Sprache kognitiv aufwendiger. Um diesen Denkvorgang für die Tutor*innen zu erleichtern, werden die Tipps zu dem Platzhalter `?` und dessen Verweisen `*` zusätzlich zu der Aufgabenstellung von Kahney (2009) aufgeführt. Zudem werden auch die Befehle `NOTE`, `CHECK`, `CONTINUE`, `EXIT` und `DONE` erklärt Eisenstadt (1983). Hierbei wurde jedoch die Sprache angepasst, da die Erklärungen nur zum Bearbeiten der Aufgabe und nicht dem Beherrschen von SOLO dienen sollen. Der Test ist im Anhang unter B.3.5 zu finden.

Damit die Tutor*innen wirklich ihre individuellen Vorstellungen und mentalen Modelle überprüfen können, ist der Test in Einzelarbeit zu bearbeiten. Andernfalls würde die Idee der individuellen Vorstellungen ad absurdum geführt. Das Minimalziel für die Tutor*innen stellt das Bewerten der Lösungsvorschläge mittels Ankreuzen der entsprechenden Kästchen dar. Andernfalls kann der Test am Ende des Moduls nicht ausgewertet werden. Um sicherzustellen, dass alle Tutor*innen das Minimalziel erreichen, soll gegebenenfalls mehr Zeit zur Bearbeitung zur Verfügung gestellt werden. Damit diejenigen Tutor*-

innen, die ihre Bewertung bereits vorgenommen haben, keinen Leerlauf haben, wird von diesen noch eine Begründung für ihre Bewertung eingefordert.

Die Rückmeldung des Tests erfolgt am Ende des Moduls, damit die Tutor*innen die verwendeten Namen der Modelle kennen. Die Art der Rückmeldung erfolgt wie folgt: Die letzte Folie gibt lediglich eine Vermutung, auf welches Modell die entsprechenden Bewertungen der Lösungsvorschläge verweisen. So kann jede der Tutor*innen die Auswertung ganz persönlich vornehmen, ohne sich vor den anderen bloßgestellt zu fühlen.

Nachfolgend wird der Erwartungshorizont zur Bearbeitung des Tests zur Rekursion beschrieben. Die Tutor*innen werden sich in der benötigten Zeitdauer deutlich unterscheiden. Da die Tutor*innen aus dem zweiten Semester meist erst die Programmiersprachen *DrRacket* und *Java* kennengelernt haben, wird es ihnen noch schwerer als den anderen fallen, sich auf SOLO einzulassen. Zudem ist das Lesen und Verstehen einer vermutlich unbekannten Programmiersprache leichter, wenn man sich zuvor schon mit ähnlichen Programmiersprachen beschäftigt hat. Aufgrund der Textmenge zu den Erklärungen der einzelnen Befehle sind diejenigen Tutor*innen schneller, deren Lesekompetenzen höher sind. Auf diese Unterschiede soll durch die optional verstandene Begründung am Ende des Tests Rücksicht genommen werden.

Zu Beginn der nachfolgenden Präsentation – wie sie in B.2.5 zu finden ist – wird die Wichtigkeit von Rekursion erläutert und damit begründet, warum sie ein eigenes Modul innerhalb der Tutor*innenschulung innehat. Anschließend folgen die zwei wichtigsten Modelle (2.3.3) zur Rekursion sowie welche Bedeutung die Behandlung der Modelle auf das Erklären hat. Danach folgt die Auswertung des Tests zur Rekursion.

Die Begründung der Wichtigkeit von Rekursion folgt den wissenschaftlichen Beschreibungen (2.3.3). Die Vorstellung der Rekursion wurde im direkten Wortlaut übernommen, da dieses kurz und knapp die Idee gut wiedergibt. Hierbei sollen bei der Präsentation die einzelnen Aspekte nach einem vorherigen Vorlesen übersetzt und besprochen werden. Die graphische Darstellung der Kopien soll einerseits die Idee

veranschaulichen und andererseits die Entwicklung der graphischen Repräsentation im Vergleich zur nachfolgenden zeigen. Die nachfolgende Veranschaulichung – siehe Abbildung 3.2 folgt einer Darstellung von GeeksforGeeks (2018). An dieser soll die aufgegriffene Idee der Kopien beispielhaft veranschaulicht werden. Die bloße Zeichnung von Kahney (2009) ist hierbei nicht ausreichend, wie auch bemerkt werden soll.

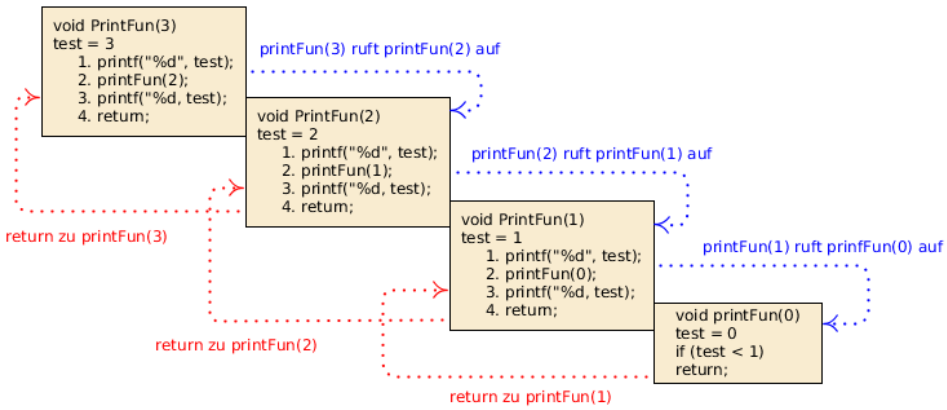


ABBILDUNG 3.2: Kopien bei der rekursiven Methode `printFun`, angelehnt an GeeksforGeeks (2018)

Dies soll in einer Partnerarbeit in einer Murmelphase und anschließend im Plenum besprochen werden. So können die Tutor*innen sich zu zweit mit den Kopien in dieser Form vertraut machen und sich die zugehörigen Sprachbausteine aneignen. Die Sprachbausteine werden beim nachfolgenden Absatz zum Erklären noch ausführlicher aufgegriffen. Die Tutor*innen sind an dieser Stelle der Präsentation selbst gefordert, die Funktionsweise des Programms zu durchdenken und das neu erlernte Konzept auf ein Beispiel anzuwenden. Durch diese Aktivität sollen sie das Konzept besser verinnerlichen und ein höheres Lernniveau nach Ulrich (2016) als bloßes Aufnehmen der Informationen erreichen.

Damit die Tutor*innen nicht nur rein visuell den Ablauf der Methode über die Folien der Beamer-Präsentation nachvollziehen können, erhält jede der Tutor*innen die farbige Darstellung quer auf einem A4-Papier. Dieses ist ebenfalls im Material unter B.3.5 zu finden, wird aber als einzelnes Blatt den Tutor*innen ausgegeben. So können die Tutor*innen den Ablauf des Programmflusses zusätzlich haptisch

nachvollziehen, sich direkt dazu Notizen machen, Zwischenergebnisse und Gedanken notieren. Zudem dient die Darstellung als Material und Anregung für die Tutor*innen bei ihrer zukünftigen Tätigkeit, wie man das Kopien-Modell den Student*innen am Beispiel vermitteln kann.

Erwartungshorizont – Murmelphase zum Methodenaufruf

1. *Was wird beim Print-Befehl in 3. bei `printFun(2)` ausgegeben?* Als korrekte Antwort wird vermerkt, dass hier die Zahl `2` ausgegeben wird. Falls noch Unsicherheit der Tutor*innen beim Kopien-Modell besteht, ist die Vorstellung bei Verwendung des Schleifen-Modells, dass die Ausführung bei Erreichen des Rekursionsankers bei `printFun(0)` stoppt. Dieser Vorstellung folgend wird der angegebene Print-Befehl gar nicht erreicht und daher auch nichts ausgegeben.
2. *Was macht die Methode `printFun`?* Die Methode gibt für den Parameter n die folgende Zeile auf dem Bildschirm aus, wobei jeweils die konkreten ganzzahligen Werte angegeben werden.

   $$n\ (n-1)\ (n-2)\ \ldots\ 3\ 2\ 1\ 1\ 2\ 3\ \ldots\ (n-2)\ (n-1)\ n$$

   Damit zählt die Methode vom Parameter n ganzzahlig bis 1 herunter und danach von 1 ganzzahlig bis n wieder hoch.
   Wenn die korrekte Verwendung des Kopien-Modells nicht zur Bearbeitung herangezogen wird, gelangen die Tutor*innen zu folgenden, verkürzten Ergebnis.

   $$n\ (n-1)\ (n-2)\ \ldots\ 3\ 2\ 1$$

3. *Wo sind hier die Kopien?* Die Methoden-Aufrufe `printFun(3)`, `printFun(2)` und `printFun(1)` gleichen sich bis auf die Werte der Variablen `test` und dem Parameter des Methoden-Aufrufs unter 2. Somit können diese als Kopien angesehen werden. Da die Kopien durch die graphischen Darstellungen als solche dargestellt werden, ist davon auszugehen, dass spätestens an dieser Stelle alle Tutor*innen diese erkennen.

Um neben den Kopien-Modell das prominenteste Beispiel der inadäquaten Modelle zur Rekursion abzudecken, soll das Schleifen-Modell nach Kahney (2009) knapp vorgestellt werden. Zudem liegt der Fokus darauf den Tutor*innen zu vermitteln, dass das Schleifen-Modell nicht einfach als falsch zu betrachten ist. Es deckt lediglich einen Spezialfall ab. Falls ein Teil der Tutor*innen die verschiedenen Modelle bereits gut durchdrungen haben, wird die folgende Frage seitens des Plenums beantwortet. Andernfalls wird die Antwort – sprich die Definition – ebenfalls vorgestellt.

Erwartungshorizont – Für welche Art von Programmen ist das Schleifen-Modell dennoch passend?

- Tutor*innen geben ein Beispiel an, statt die Art der Programme zu nennen, und versuchen den wesentlichen Kern des Beispiels – die Endrekursion – heraus zu stellen.
- Tutor*innen umschreiben die definierende Eigenschaft der Endrekursion, ohne deren Wortlaut zu gebrauchen. Ein mögliches Beispiel wäre:„Funktionen, bei denen der rekursive Aufruf am Schluss ist, sodass es nicht wieder zurückgehen muss.“
- Tutor*innen kennen den Begriff und/oder das Konzept der Endrekursion nicht.

Es wird davon ausgegangen, dass ein Großteil der Tutor*innen mit dem Konzept der Endrekursion unter diesem Begriff nicht vertraut ist. Daher wird die Endrekursion an einem Beispiel vorgestellt. Das Beispiel zur einer endrekursiven Funktion besteht aus der vertrauten Funktion, die die Fakultät des Inputs berechnet. Im Beispiel wurde der Code zur Fakultätsfunktion mittels eines Akkumulators geschrieben, sodass es sich um eine endrekursive Funktion handelt.

Nachfolgend sollen den Tutor*innen einfach umsetzbare Handlungsanweisungen mitgegeben werden. Damit sollen sie in der Lage sein, die Rekursion in Hinblick auf die Ausbildung des Kopien-Modells gut erklären zu können. Sprache ist eng mit den Vorstellungen und mentalen Modellen verknüpft. Daher soll zu Beginn ein Negativbeispiel in Bezug auf das mündliche Erklären präsentiert werden. Bei dem Wort „Abbruch“ wird schlichtweg nicht deutlich, ob es sich

um den Abbruch der Aufrufkette oder um einen Programmabbruch handelt. Einerseits ist bei dieser Wortwohl nicht klar, wie der*der Tutor*in den Begriff im Gespräch gemeint hat. Andererseits ist unklar, wie er von dem*r Student*in verstanden wurde. Beide Aspekte stellen eine zusätzliche Hürde im Verstehen dar. So vermittelt ein Programmabbruch eine Vorstellung von Rekursion, die nur mäßig mit dem Kopien-Modell vereinbar ist.

Bezogen auf die Visualisierung der Rekursion wird die Handlungsanweisung mitgegeben, die verschiedenen Kontrollflüsse – die Aufrufkette – und die Kopien explizit zu machen und farbig zu kennzeichnen. Um den Tutor*innen passende Metaphern für eine Rekursion nach dem Kopien-Modell zu liefern, wurde diese ebenfalls mit angeführt.

Nach Behandlung der Sprache und der Visualisierungen beim Erklären soll die letzte Handlungsanweisung die Auswahl eines geeigneten Beispiels betreffen. Bei diesem wird ein konkretes und bekanntes Beispiel – die Zahlenfolge der Fibonacci-Zahlen – angeführt. Dieses zeichnet sich besonders dadurch aus, dass es aus zwei rekursiven Aufrufen besteht. Aus diesem Grund ist es nicht endrekursiv und fördert vor allem ein Kopien-Modell. Diese kann man in einer Baumstruktur aus Kopien besonders gut deutlich machen.

Der Stack ist eng mit einem rekursiven Aufruf verbunden ist und findet daher seinen Platz in der Schulung. Die Mehrheit der Tutor*innen studiert als Hauptfach Informatik. So ist diesen nicht unbedingt bewusst, dass nicht jede Person den Stack und die Speicherverwaltung bereits en detail aus anderen Lehrveranstaltungen kennt. Der Kern des Stack wird jedoch in *FOP* behandelt und auf diesen soll an dieser Stelle verwiesen werden.

Für diese genannten Folien wurde sich dafür entschieden, diese in einer Präsentation mit Instruktion zu vermitteln. Dafür spricht, dass sichergestellt wird, dass alle Tutor*innen das geeignete Kopien-Modell mit den dazu verbundenen Handlungsanweisungen mit Erklärungen vermittelt bekommen. So wird eine Sicherung des Grundniveaus an Erklärungen vorgenommen.

3.4.3 Rückschau

Dieses letzte Modul dient den thematisch aufgespannten Bogen – beginnend beim Korrigieren und Verfassen eines Feedbacks – wieder aufzugreifen und abzuschließen. Daher sollen die von den Tutor*innen verfassten Feedbacks aufgegriffen werden. Dazu sollen die Tutor*innen mit ihrer Sitznachbar*in das erstellte Feedback austauschen und lesen. Durch die Fragestellungen – Wie empfindest Du das erhaltene Feedback? Was würdest Du anders machen? – werden die Tutor*innen zum Reflektieren aufgefordert. Hierbei sollen die Tutor*innen einerseits erfahren, wie frustrierend ein negatives Feedback sein kann und welche Kommentare wirklich weiter helfen. Andererseits ist es das Ziel, dass die Tutor*innen überlegen, inwieweit die Themen der Tutor*innenschulung bei den Feedbacks eine Rolle spielen. Bei den typischen Fehlern und Fehlvorstellungen ist es gut, diese zu kennen, da diese wahrscheinlich in den zu korrigierenden Hausübungen auftauchen. Die typischen Fehler und Fehlvorstellungen stellen unter Anderem eine Grundlage für die gemachten und gefundenen Fehler dar. Die behandelten Problemlösestrategien stellen eine Quelle für Tipps dar, die bei den Feedbacks formuliert werden können.

Um die Tutor*innen nicht bezüglich ihren fachlichen Fähigkeiten eventuell bloßzustellen, wurde sich dafür entschieden, die Feedbacks nicht im Plenum zu besprechen. Bei möglicherweise nicht erkannten Fehlern oder fachlich falschen Korrekturhinweisen ist es ausreichend, diese mit der Sitznachbar*in zu besprechen. Dies muss nicht zwingend im Plenum geschehen.

Die explizite Frage nach offenen Fragen soll die Tutor*innen ermuntern, Unklarheiten oder weiterführende Fragen zu formulieren. Je nach Frage reicht auch ein Verweis an die zuständige Person. Dies ist vor allem für Fragen, die den Übungsbetrieb an sich betreffen, wichtig, da Herr Florian Kadner für diese zuständig ist.

Als Feedback wurde sich für ein *Daumenfeedback* entschlossen. Bei dieser Methode wird mittels des Daumen eine Rückmeldung gegeben. Der Vorteil dieser Feedback-Methode liegt insbesondere in ihrer Schnelligkeit und, dass sie weder Vorbereitung noch Material erfordert. Als Nachteil sind jedoch sind jedoch keine anonymen Rückmeldungen möglich. Da nach einem ganzem Schulungstag die Motivation

für ein ausgedehntes Feedback mit hohem Schriftanteil eher nicht gegeben sein wird, wurde sich gegen ein solches Feedback entschieden.

In der hier ausgeführten Version des Daumenfeedbacks wird den Tutor*innen folgendes Bewertungsschema vorgestellt. Ein nach oben zeigender Daumen bedeutet, dass die Schulung gut gefallen hat und sinnvoll und gewinnbringend war. Ein zur Seite gerichteter Daumen bedeutet, dass sie als mittelmäßig einzuschätzen ist. Ein nach unten zeigender Daumen entspricht einer schlechten Bewertung und zeigt an, dass die Schulung nicht als sinnvoll und gewinnbringend empfunden wurde. Da nicht davon ausgegangen werden kann, dass die Tutor*innen die Methode des Daumenfeedbacks kennen, muss diese vor der Bitte um ein Feedback erklärt werden.

Kapitel 4

Transfer zur Institution Schule

Abschließend soll untersucht werden, inwiefern sich die Tutor*innenschulung methodisch und inhaltlich auf die Institution Schule übertragen lassen kann. Dazu werden zuerst jeweils die Vorerfahrungen der betreffenden Subjekte – Programmieranfänger*innen und Tutor*innen – verglichen. Zum Schluss wird eine hypothetische Verwendung der Tutor*innenschulung im Kontext der Institution Schule angerissen.

4.1 Vorerfahrungen der Subjekte

Das jeweilige Pendant der betreffenden Subjekte setzt sich wie folgt zusammen: Die Schüler*innen des Unterrichtfachs Informatik entsprechen den Programmieranfänger*innen. Die Tutor*innen gleichen in ihrer Rolle als Lehrpersonen und Unterstützer*innen den Lehrer*-innen.

Zu Beginn des Informatikunterrichts in der 11. Klasse ist die Gesamtheit der Schüler*innen ähnlich heterogen wie die der Programmieranfänger*innen der *FOP*. Jedoch liegt der Anteil derjenigen *mit* Vorerfahrungen bezüglich den Fähigkeiten zur Programmierung nur bei rund 18 % statt bei zwei Drittel (Magenheim und Schulte, 2005, 115) – siehe 1.2.2. Somit ist die Anzahl der Programmieranfänger*innen sehr hoch. Daher lässt sich die Tutor*innschulung mit ihrem Fokus auf Programmieranfänger*innen besonders gut auf die Schule übertragen.

Lehrer*innen, die das Schulfach Informatik unterrichten, sind oft nur unzureichend durch eine Hochschulausbildung zur Didaktik der Informatik ausgebildet worden (Müller, 2017, 127–128). Somit ergibt sich ein hoher Anteil an Unterrichtsstunden, die fachfremd – ohne Lehrbefähigung für das Fach Informatik – unterrichtet werden. Somit gleichen die Tutor*innen denjenigen fachfremden Lehrer*innen,

die zwar selbst ein fachbezogenes Hochschulstudium in der Informatik absolviert haben und dann über einen Quereinstieg in den Lehrberuf kamen. Beide Gruppen unterrichten informatische Grundlagen, ohne deren fachdidaktische Grundlagen selbst erfahren zu haben. Der Unterschied besteht jedoch darin, dass diese Lehrer*innen durch ihr abgeschlossenes Hochschulstudium im Bereich der Programmierung vermutlich ein wesentlich höheres Niveau erreicht haben als diejenigen Tutor*innen im zweiten Fachsemester. Dennoch sind die Vorerfahrungen der Tutor*innen der genannten Gruppe von Lehrer*innen sehr ähnlich zu bewerten.

4.2 Verwendbarkeit der Schulung

Die Tutor*innenschulung und deren Material kann an verschiedenen Stellen in der Institution Schule verwendet werden.

So kann die Zusammenfassung der zugehörigen Fachliteratur – wie sie in Kapitel 2 dargestellt wurde – als Grundlage für die Vorbereitung des Unterrichts zur Einführung der Programmierung dienen. Insbesondere für die Einführung in die Objektorientierung lassen sich die spezifischen Themen übertragen, da auch die Veranstaltung *FOP* eine objektorientierte Programmiersprache behandelt. Hierbei ist an zuführen, dass diese Inhalte in der gymnasialen Oberstufe in Hessen im Fach Informatik verpflichtend sind. So listet das Kerncurriculum unter Anderem verbindlich die Themenfelder „Grundlagen der Programmierung“, „Rekursion“ und „Klassen und Objekte“ in den Phasen E1 und Q1 auf (HKM, 2016, 26). Damit sind alle inhaltlichen Themen bis auf die informatikspezifischen Problemlösestrategien aus Kapitel 2 explizit vom Kerncurriculum angeführt.

Die Literaturhinweise der Hessischen Lehrkräfteakademie Hessische Lehrkräfteakademie (2015) geben zwar mögliche Titel zur Vorbereitung auf die Themenfelder an, jedoch behandelt keiner der Titel das jeweilige Thema fachdidaktisch. Eben sowenig wird beim Themenfeld *Grundlagen der Programmierung* irgendein Titel angegeben. Demnach handelt es sich bei Kapitel 2 um eine sehr gute Ergänzung oder auch als Grundlage für die Vorbereitung auf den Unterricht im jeweiligen Themenfeld.

Zudem besteht die Möglichkeit der Verwendung der Tutor*innenschulung als Grundlage einer Schulung zur Weiterbildung von Lehrkräften für das Fach Informatik. So können besonders diejenigen Lehrer*innen, die keine vollwertige Informatik-Ausbildung erhalten haben und daher Quereinsteiger in die Lehramtsberuf sind, von der kompakten Zusammenfassung der fachdidaktischen Inhalte profitieren. Dafür müssten die folgenden Aspekte der Tutor*innenschulung geändert und an die neue Lerngruppe angepasst werden.

Die Folien, die die Programmiersprache Racket betreffen, müssen nicht unbedingt komplett herausgestrichen werden. Das Kerncurriculum listet als ein mögliches Themenfeld „Deklaratives Problemlösen“auf, bei welchem entweder Prolog als deklarative bzw. logische Programmiersprache oder eine funktionale Programmiersprache behandelt wird (HKM, 2016, 40). Da es sich bei Racket um eine funktionale Programmiersprache handelt, kann je nach Vorwissen und Interesse der Lerngruppe Racket ebenfalls eine sinnvolle Verwendung in der Weiterbildung erfahren. Der Eingangstest – basierend auf einer Abgabe einer Student*in mittels Racket – stellt jedoch eine Ausnahme und sollte in eine unter der Lerngruppe weit verbreitete Programmiersprache abgeändert werden. Andernfalls scheitern die Lehrer*innen an dem eigentlichen Verstehen der Programmiersprache. So wird das eigentliche Ziel – die Auseinandersetzung mit den dahinter liegenden Vorstellungen – nicht erreicht. Da das Kerncurriculum zur Einführung in die Programmierung eine objektorientierte Programmiersprache zwingend vorschreibt, wird letztendlich die Schüler*innen-Lösung des Eingangstests mit Java oder Python geschrieben sein. Hierbei würde sich Java aus Gründen der Einfachheit anbieten, da die Tutor*innenschulung durch ihren Bezug auf *FOP* auf die Programmiersprache Java ausgelegt ist.

Analog dazu sollten die Aufgabentexte, die bei der Simulation verwendet werden, an die geänderte Lerngruppe und die Inhalte des Schulunterrichts angepasst werden. Hier könnte auch eine Mischung aus mehreren Programmiersprachen verwendet werden, um einer heterogenen Lerngruppe gerecht zu werden. So sind beispielsweise das Thema *Vererbung* im Sinne der Objektorientierung und Listen von Objekten nur für einen Leistungskurs verpflichtend (HKM, 2016, 33). Daher sollte dieses inhaltliche Thema zwar behandelt werden, jedoch einen geringeren Umfang als in der Tutor*innenschulung einnehmen.

Die inhaltlichen Punkte abstrakte Klassen und Interfaces tauchen im Kerncurriculum gar nicht auf und sollte daher aus der Aufgabensammlung der Simulation gestrichen werden. Als Fundus für die Aufgaben wären eventuell die Abituraufgaben aus den vorherigen Jahren, Sammlungen aus Schulbüchern und Wettbewerben geeignet. Diese wäre jedoch als zusätzliche Aufgabe für eine Weiterbildung zusammen zu tragen.

Das Design innerhalb der Objektorientierung hat in der Tutor*-innenschulung lediglich eine kleine Rolle gespielt, da dieses nicht im Fokus in der *FOP* steht. Das Kerncurriculum sieht im Gegensatz dazu die objektorientierte Modellierung aus Problembeschreibung als verpflichtenden Bestandteil für den Leistungskurs vor (HKM, 2016, 31,33). Daher würde es sich anbieten das Design der Objektorientierung für die Weiterbildung stärker in Fokus zu nehmen. Hierfür bietet sich eine detaillierte Lesart von Thomasson u. a. (2006) an. Zudem sind die Lehrer*innen im Gegensatz zu den Tutor*innen zusätzlich gefordert eigene Beispiele zu erstellen. Ein Teil der Fehler im Bereich der Objektorientierung stammt aus schlechten Beispielen Holland u. a. (1997). Daher ist der Bereich des Designs beim Erstellen der Beispiele von Relevanz und sollte zusätzlich Teil der Tutor*innenschulung werden.

Ebenso ist anzumerken, dass aufgrund der geänderten Lerngruppe sich selbstverständlich die Analyse der Lerngruppe und damit die Erwartungshorizonte ändern. Jedoch ist gezeigt worden, dass sich die Beschreibung der Tutor*innenschulung als sinnvolle Grundlage für das Erstellen einer Schulung zur Weiterbildung für Lehrer*innen zeigt. Ein möglicher Titel wäre *(Fehl-)Vorstellungen von Programmieranfänger*-innen – Was bedeutet das für meinen Unterricht?*.

Anhang A

Grundlagen zum Material

A.1 Problemklassen nach Garner u. a. (2005)

The problem descriptions used in the study were:

1 Tools

Problems with the Mac, OS X, directories (lost files), jEdit, Applet runner, or other basic tools. Includes being unable to find the resources described in the lab book, but not other kinds of general lab book / text book issues (do not record these). Does not include Java / file naming conventions (Problem 6).

2 Understanding the task

Problems understanding the lab exercise / task or its "solution". In other words, whatever other problems they may be having, in this case they don't actually know what it is that the program is supposed to be doing. (Does not include minor clarifications of some detail, which do not need to be recorded).

3 Stuck on program design

They understand the task / solution (its not Problem 2) but can't turn that understanding into an algorithm, or can't turn the algorithm into a program. Cases such as "I don't know how to get started" or "what classes should I have?" or "what should the classes do?".

4 Problems with basic structure

This problem number is meant to capture problems at the class / object / major structural level - they have a general design and classes (its not Problem 3), but are getting basic structural details wrong. E.g. code outside methods, data fields outside the class, data fields inside a method / confused with local variables (problems specifically with data fields or about or within methods, e.g. mixing up loops, will be some other problem number). Includes very basic problems with creating and using instance objects, e.g. how many, what they are for (but not more specific problems e.g. with class vs instance Problem 22, or constructors Problem 24).

5 Problem naming things

They have a problems choosing a name for something. Especially where this seems to suggest that they don't understand the function of the thing that they are trying to name.

6 Basic mechanics

Problems with little mechanical details (where these are not better described by some other problem). Braces, brackets, semi–colons. Typos and spelling. Java and file naming conventions. Import statements (when forgotten, when misunderstood see Problem 18). Formatting output. Tidiness, indenting, comments.

7 Control flow

Problem with basic sequential flow of control, the role of the main or init method. Especially problems with the idea of flow of control in method call and return (e.g. writing methods and not calling them, expecting methods to get called in the order they are written). (For issues with parameter passing and returned results see Problem 13). Does not include event driven model, Problem 27

8 Loops

Conceptual and practical problems relating to repetition, loops (including for loop headers, loop bodies as blocks).

9 Selection

Conceptual and practical problems relating to selection, if else, switch (including the use of {blocks}).

10 Booleans and conditions

Problems with booleans, truth values, boolean expressions (except boolean operator precedence, see Problem 19). Problems with loop or selection headers / conditions will have to be judged carefully – is this a problem formulating the boolean expression (Problem 10) or understanding how the expression / result is relevant to the loop or selection (Problems 8, 9)?

11 Exceptions, throw catch

Problems with exceptions, throw catch.

12 Method signatures and overloading

Problems related to overloading. Failure to understand how method signatures work / which version of a method gets called. (Includes problems with constructors that are really about the signatures of constructors – c.f. Problem 24).

13 Data flow and method header mechanics

Especially conceptual problems with arguments / parameters and return types / values. Includes problems with method header mechanics (incorrect or mismatching parameter specifications, incorrect return types or use of void). Includes any other problems with "data flow" that are not better described by Problem 14.

14 Terminal or file IO

Problems with terminal or file IO / data flow (not including exception handling Problem 11, or output formatting Problem 6).

l15 Strings

Strings and string functions. Does not include formatting output (Problem 6) or problems relating specifically to strings as reference types (Problem 21).

16 Arrays

Problems relating to arrays as a data structure, including array subscripts, array contents, array declaration and initialisation (cf Problem 17). Does not include failing to understand that an array as a whole is itself a reference type or may contain references (Problem 21).

17 Variables

Problems with the concept of or use of variables. Includes problems with initialisation and assignment. (Missing the distinction between a data field and a local / method variable is Problem 4). Does not include cases more accurately described as problems with reference types (Problem 21) or arrays (Problem 16) rather than the concept of a variable.

18 Visibility & scope

Problems with data field visibility, local variable scope (e.g. defining a variable in one block and trying to use it in another, problems arising from unintended reuse of identifiers), and namespace / imported package problems (but not including forgotten "import" statement, Problem 6). Includes cases confusing data fields and variables of the same name, but not where this is better described as a failure to understand "this" (Problem 21).

19 Expressions & calculations

Problems with arithmetic expressions, calculations, notation such as "++", and all forms of precedence (including boolean operator precedence, c.f. Problem 10).

20 Data types & casting

Problems caused by failing to understand different data types and casting for primitive types (reference types are Problem 21).

21 Reference types

Problems arising from a failure to understand the concept or use of reference types (references / pointers, "this", different references to the same object, etc), or that reference types behave differently from primitive types (when assigned, compared etc).

22 Class versus instance

Problems understanding the class object vs. instance object distinction, including problems with class and instance data fields (and use of "static").

23 Accessors / Modifiers

Specific problems (c.f. for example Problem 13) with the concepts of / purpose of an accessor or a modifier method.

24 Constructors

Specific problems (c.f. for example Problems 12, 13, 22) with the concept of / purpose of a constructor. 25 Hierarchies Problems relating to hierarchical structure, inheritance (extends, overriding, shadowing, super), and issues relating to the use of abstract methods or interfaces.

26 GUI mechanics

Problems with GUIs and the use of AWT, Swing etc. Includes problems with specific required methods such as actionPerfomed(), run(),

implements actionListener and so on (but some issues might general problems with the concept of interfaces, Problem 25). Does not include the underlying concepts of event driven programming. 27 Event driven programming Problems with the underlying concepts of event driven programming, and general "flow of control" type issues that arise in the transition from application to applet.

A.2 Strategien nach Kaplan (2012)

1. BREAK IT DOWN.
2. LEARN THE BASICS WELL, LOOKUP THE REST.
3. FIND AND USE EXAMPLES.
4. TEST TEST TEST (AND TEST SOME MORE)
5. FOLLOW DIRECTIONS.
6. SOLVE THE PROBLEM IN ENGLISH (OR YOUR NATIVE LANGUAGE) FIRST.
7. REPRESENT WHAT?
8. WHAT DOES IT DO?
9. MAKE SURE YOU UNDERSTAND THE PROBLEM.
10. IF YOU CAN'T SOLVE THE BIG PROBLEM, SOLVE A SMALLER PROBLEM.
11. HAVE A PLAN B.

1. BREAK IT DOWN: When you are faced with a problem that is too complex, one way to approach it is to break the problem down into smaller parts. By breaking the problem into smaller parts you may be able to more readily solve the original problem by solving its pieces.
2. LEARN THE BASICS WELL, LOOKUP THE REST: Beginning programmers often get hung up on how to do certain things. Because programs have to be very precise it is easy to make mistakes when you first start out. It is okay to look up what is needed when you become stuck.

3. FIND AND USE EXAMPLES: There are thousands upon thousands of coding samples that can be examined to better understand how a specific task is accomplished. Using examples to create programs does two things: First, you get to see how something is done, and if you pay attention by seeing how it is done you will learn how to do it. Second, you'll spend less time, at least at the start, writing your programs. Students complain about how long it takes to write a program. If you can find and use examples and you can learn from the examples you might be able to learn faster then if you tried to write a program from scratch.

4. TEST TEST TEST (AND TEST SOME MORE): Testing is part of programming. You test to get a program to work and you test to make sure the program works correctly. The better you test, the better your program will be.

5. FOLLOW DIRECTIONS: Make sure that you are doing what the directions say. The directions tell you what you can and cannot do. Don't overcomplicate or oversimplify what you have to do. Sometimes the directions will even give you part of the solution – so read carefully.

6. SOLVE THE PROBLEM IN ENGLISH (OR YOUR NATIVE LANGUAGE) FIRST: Learning a new programming language is similar to learning a new foreign language. You just don't jump right into writing the steps in the new language. That is a direct way to failure. Solve the problem in your native language. Write each of the steps in English or the language that you are most comfortable with. Translate this version into the programming language you are using. That way you will have the problem solved and you can focus on how to express your solution in the programming language.

7. REPRESENT WHAT?: An important part of solving any programming problem is to figure out how the problem will be represented in data. The variables that are needed and other structures like arrays need to be defined. This is a key aspect of creating a program to solve any programming problem.

8. WHAT DOES IT DO?: Every computer program that was ever written does something. That is the nature of programs. They make

computers do things that are desired by the programmer. Now sometimes the author of a program will get it wrong. The computer will do something that was not wanted. There can be lots of reasons for this and one of them is that the programmer really didn't understand what the computer was supposed to do for solving the problem. Make sure you understand what it is that you want the computer to do.

9. MAKE SURE YOU UNDERSTAND THE PROBLEM: If you don't understand what the problem is, how can you come up with a solution? You can't. Besides following directions, understanding what it is you must do is extremely important. If you don't understand what to do, don't even think about starting to work on the solution to the problem. Do whatever it takes to understand the problem you must solve.

10. IF YOU CAN'T SOLVE THE BIG PROBLEM, SOLVE A SMALLER PROBLEM.: This may seem like the first statement (Break It Down) but it isn't. This has to do with complexity of the problem you are to solve. A problem might be so complex that you have no idea how to approach it let alone solve it. So, instead of bashing one's head against a wall (a tried and true method for solving problems), formulate a simpler problem related to the original problem. The simpler problem will be solvable (we hope). Use the learning that came from solving the simpler problem to solve the more complex problem.

11. HAVE A PLAN B: No matter how well we plan and no matter how completely we plan, there is always something that happens to get in the way of our plan. Therefore, if you have a way to solve the problem and you find you have travelled down a path that doesn't have a good or desired ending, you need to have an alternative. Plan B is that alternative. It may be an alternate way of solving the problem. It may be a different breakdown of the problem. It may be a simpler approach. Whatever it is, have a plan B just in case.

Anhang B

Material zur Tutorenschulung

B.1 Übersichtsplan

Start	Ende	Dauer	Modul	Thema	Methode
08:30	08:35	5	-	Begrüßung	✓
08:35	09:05	30	3.3.1	Diagnose Wissensstand	Think-Pair-Share
09:05	09:20	15	3.3.2	Arten von Programmieranfänger*innen	Präsentation
09:20	09:30	10	3.3.2	Ziele der Schulung	Präsentation
09:30	09:40	10	3.3.3	Warum Visualisieren?	Präsentation
09:40	10:25	45	3.3.3	Exemplarische Vorstellungen	Präsentation
10:25	10:45	15	-	PAUSE	PAUSE
10:45	11:15	30	3.3.4	Häufigkeit von Fehlern	Think-Pair-Share
11:15	11:45	30	3.3.4	Spezifische Fehler	Präsentation
11:45	12:40	55	3.3.5	Simulation	Simulation
12:40	13:25	45	-	MITTAGSPAUSE	MITTAGSPAUSE
13:25	13:40	15	3.4.1	Übersicht Problemlösen	Präsentation
13:40	14:10	30	3.4.1	Vergleich Wissensstand zu Literatur	Think-Pair-Share
14:10	14:35	25	3.4.1	Vorstellung Pattern	Präsentation
14:35	14:45	10	-	PAUSE	PAUSE
14:45	15:00	15	3.4.2	Wissensstand Rekursion	Test zu Modellen
15:00	15:30	30	3.4.2	Modelle von Rekursion	Präsentation
15:30	15:40	10	3.4.3	Offene Fragen, Rückbezug auf Beginn	Peer- und Daumenfeedback

B.2 Folien

B.2.1 Einstieg

Tutor*innenschulung

Organisatorisches, eigener Stand und Einstieg

Svana Esche

Fachbereich Informatik – TU Darmstadt

Datum

Wo steht ihr selbst?

Los geht's!

Arbeitsauftrag A – *20 Minuten Zeit – Think*
Bearbeitet das ausgeteilte Arbeitsblatt, bei dem ihr eine reale Lösung einer Student*in korrigiert und entsprechendes Feedback dazu formuliert.

Wo steht ihr selbst?

Los geht's!

Arbeitsauftrag A – *20 Minuten Zeit – Think*
Bearbeitet das ausgeteilte Arbeitsblatt, bei dem ihr eine reale Lösung einer Student*in korrigiert und entsprechendes Feedback dazu formuliert.

Arbeitsauftrag B – *5 Minuten Pair*
Tauscht euch über eure Notizen und Gedanken dazu aus.

Wo steht ihr selbst? Was erwartet euch als Tutor*innen? Wie geht es weiter?

Wo steht ihr selbst?

Los geht's!

Arbeitsauftrag A – *20 Minuten Zeit – Think*
Bearbeitet das ausgeteilte Arbeitsblatt, bei dem ihr eine reale Lösung einer Student*in korrigiert und entsprechendes Feedback dazu formuliert.

Arbeitsauftrag B – *5 Minuten Pair*
Tauscht euch über eure Notizen und Gedanken dazu aus.

Blitzlicht – Share
Jeder sagt einen Satz als Ich-Botschaft, wie er*sie sich nach dem Korrigieren in der Rolle als Tutor*in einschätzt:
„Ich schätze mich . . . .“

Was erwartet euch als Tutor*innen?

Wo steht ihr selbst? Was erwartet euch als Tutor*innen? Wie geht es weiter?

Was können die Student*innen als Programmieranfänger*innen?

Studien zusammengetragen von Winslow (1996) und Soloway und Spohrer (1989) zeigen, dass Programmieranfänger*innen

- auf oberflächliches Wissen beschränkt sind,
- keine detaillierte mentale Modelle besitzen,
- versagen relevantes Wissen anzuwenden,
- allgemeine statt informatikspezifische Problemlösestrategien benutzen,
- Zeile-per-Zeile programmieren,
- sehr wenig Zeit nutzen, um zu planen und zu testen,
- häufig ziemlich schlecht sind, Code zu tracen/tracken.

Welche Arten von Programmieranfänger*innen gibt es?

stopper (zögernd)	*mover* (anpackend)	*tinkerer* (tüftelnd)
▶ ohne Hoffnung es alleine zu schaffen	▶ geben nicht auf	▶ extreme movers
▶ Problem oder keine Idee, was als nächstes kommt → Stop	▶ können Fehlercodes/Feedback effektiv nutzen	▶ können ihr Programm nicht tracen/tracken
▶ Frust, negative Emotionen bei Fehlern	▶ modifizieren, experimentieren mit ihrem Code	▶ Veränderungen werden zufällig gemacht
▶ ineffektiv	▶ effektiv	▶ ineffektiv

Wen erwartet ihr in den Übungen, Sprechstunden, ...?
Wie kann man das Wissen über die Arten zum besseren Lehren nutzen?
Wer braucht welche Art von Zuwendung bzw. Hilfe?

Wie hat sich der Übungsbetrieb (nicht) geändert?

- *Konzept der Kleingruppenübung bleibt bestehen!*
- Reduzierung der Kleingruppengröße:
  $>$ 30 Student*innen → circa 6 Student*innen
- Heterogenität (kein Vorwissen bis > 1000h programmiert)
  → Homogenität
- Kein Team-Teaching, sondern kleinere Zirkel von Kleingruppen

Das erfordert Änderung der Schulung, und damit eine Konzentration auf das *Erklären können*.

Ihr seid *front line teachers*, ihr fangt alles auf!

Welche fachlichen Fähigkeiten muss eine Tutor*in besitzen, um gut erklären zu können?

Wo steht ihr selbst? Was erwartet euch als Tutor*innen? Wie geht es weiter?

Welche fachlichen Fähigkeiten muss eine Tutor*in besitzen, um gut erklären zu können?

- Beobachten, diagnostizieren, was erklärt werden muss/soll
  $\rightarrow$ Frage: Wo liegt eigentlich das Problem?
- Probleme kennen und erkennen.
- Vielfalt von Lösungswegen kennen.
- Visualisierungen nutzen, um geeignete Vorstellungen zu fördern.
- Strategien kennen, erkennen und vermitteln.

Der signifikanteste Faktor für den Lernerfolg ist das Ausmaß hochqualifizierter, individueller Hilfestellungen.

Jedoch sind Hilfestellungen ohne Diagnose nicht zielführend, Diagnose ohne Hilfestellung ebenfalls nicht.

Was soll hier unter Diagnose verstanden werden?

Jegliches Beobachten, Erkennen, was das Problem ist bzw. wie vorgegangen wurde und wie man daran ansetzen kann.

Fragen für die Arbeitsphase, um zu diagnostizieren (siehe Kommunikation - Kernkompetenz - W-Fragen):

- Was machst du?
- Was denkst du dabei?
- Kannst du mir aufmalen, wie du es dir vorstellst bzw. was du machen möchtest?

Bei der Korrektur von schriftlichen Hausübungen wird selbstverständlich ebenfalls diagnostiziert.

Wie geht es weiter?

1. Einstieg
2. Visualisierungen, Vorstellungen und mentale Modelle

   Pause
3. Typische Fehler und Fehlvorstellungen
4. Simulation eines Gesprächs

   Mittagspause
5. Strategien, Pattern

   Pause
6. Rekursion
7. Rückschau

B.2.2 Visualisierungen und Gedankliche Modelle

Tutor*innenschulung

Visualisierungen und Gedankliche Modelle

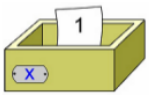
Svana Esche

Fachbereich Informatik – TU Darmstadt

Datum

Warum visualisieren?

- *Virtuelle Eselsbrücke:* Doppelte Kodierung – verbal und nonverbal – erhöht Erinnerungswahrscheinlichkeit.
- *Organisation:* Visualisierungen können organisieren, indem sie Zusammenhänge oder zeitliche Abläufe aufzeigen.
- *Darstellung:* Visualisierungen veranschaulichen intuitive Modelle, die dahinter stehen.



Behältermodell

Vorsicht! Bilder ohne Worte stellen das andere Extrem dar und sind auch nicht lernförderlich.

Besser: Visualisierungen und Worte – bpsw. Programmcode, Erklärungen, Metaphern – kombinieren.

Mentale Modelle und Vorstellungen - Warum?

Mentale Modelle werden verwendet für folgende Tätigkeiten:

- *Verstehen:* Modell als Verstehenshilfe für abstrakte Konzepte.
- *Erklären:* Kommentare, Dokumentationen schreiben.
- *Problemlösen:* Modell bringt Lösungsidee hervor.
- *Kontrollieren:* Hilfe beim Debuggen.

Erklärungen verraten etwas über das Verständnis → nachfragen, sich von den Student*innen erklären lassen.

Wunschziel: breite Palette an unterschiedlichen Modellen zum Erklären bereithalten.

Mentale Modelle - Was kann alles misslingen?

- Präkonzepte – Vorstellungen der Student*innen – aufzugreifen fördert das Verständnis, umgekehrt tauchen nicht aufgegriffene, unpassende Präkonzepte später wieder auf.
- Antropomophisieren/Vermenschlichen des Computers – „Der Computer weiß, was ich damit meine.“
- Übersetzung von natürlicher Sprache
  ⇝ Modul *Typische Fehler und Fehlvorstellungen*
- Weitere Fehlvorstellungen
  ⇝ Modul *Typische Fehler und Fehlvorstellungen*

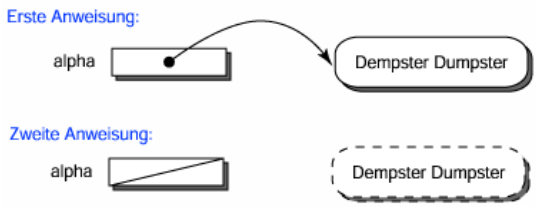
Auf zweifache Weise exemplarisch

1. Beispielhafte Auswahl von Visualisierungen.
2. Visualisierungen müssen an Beispielen erfolgen!
   – einfach, reichhaltig, Anwendungsmöglichkeit –

Allgemeine Techniken zur Visualisierung

- Hervorheben durch Farbe, Kästen, Umkreisen
- Zusammenhänge verdeutlichen durch Pfeile

```
1        String alpha = "Dempster Dumpster";
2        alpha = null;
```

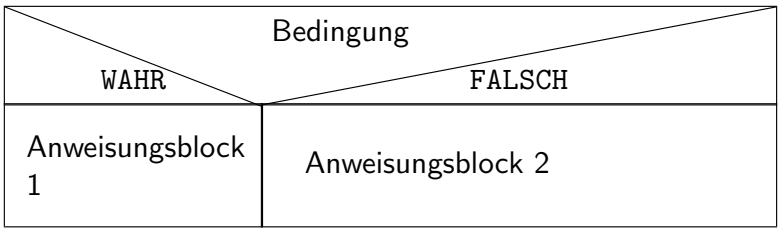


→ Stifte und Papier in den Übungen und Sprechstunden dabei haben und benutzen!

Foliensatz: Funktionale Abstraktion, Folien: 179–183

Zusammenfassung: Literale, Identifier, Strings und Symbole

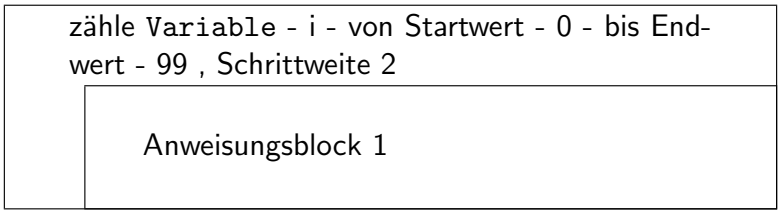
Nassi-Shneiderman-Diagramm / Struktogramm

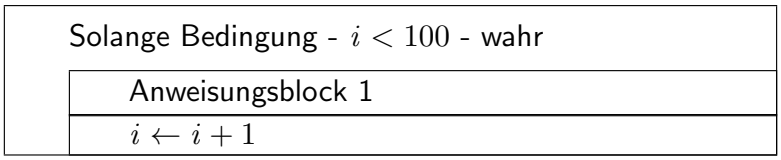


Foliensatz: Funktionale Abstraktion, Folien: 84–87

Warum visualisieren? Mentale Modelle und Vorstellungen Exemplarisch

Nassi-Shneiderman-Diagramm / Struktogramm





Variablen

Beispiel: `x = 1`
`y = 1`

Behältermodell *Referenzierungsmodell*

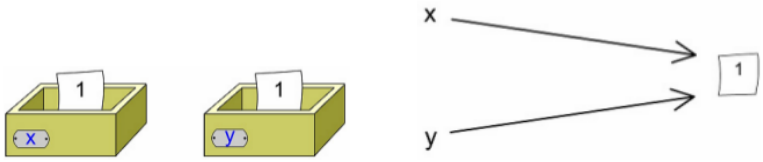


Abbildung: Quelle: [We07]

Welche Darstellung ist in welcher Situation besser geeignet?
Was für Probleme können auftauchen?

Foliensatz: Referenztypen, Folien: 5–16

Behälter- und Referenzierungsmodell

- Behältermodell ist gut geeignet für unveränderbare Objekte wie Zahlen oder auch für die Darstellung von Arrays.
- Probleme beim Referenzierungsmodell: man kann Objekte erst benennen, wenn sie existieren.
- Probleme beim Behältermodell mit Mehrfachnamen für *ein* Objekt, da man die Abhängigkeiten wie im Beispiel nicht gut erkennt.
- In rein funktionalen Sprachen gibt es keine Zustände und daher keine dynamische Veränderung und keine zustandsverwaltenden Variablen, die auf Speicherzellen referenzieren.
- Bei Racket gilt es nur bedingt aufgrund der Befehle `define` und `set!`, die Nebenwirkungen – Seiteneffekte – haben.
- In Java sind Variablen die Speicherplätze, die jeweils nur einen Typ umfassen können. Referenzen sind Verweise auf die Variable.

Nichts

Beispiel: leere Liste im Behälter- und Referenzmodell

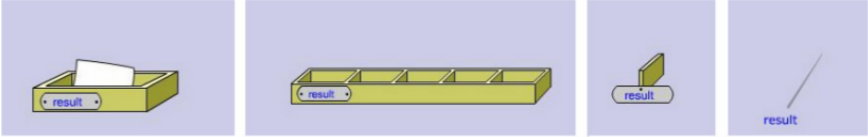


Abbildung: Quelle: [We07]

Welche Darstellung ist am geeignetsten?
Was fallen euch noch für Darstellungen für die leere Liste ein?

Nichts in Java

Wie kann man *nichts* in Java darstellen?

Erinnerung:

> *NULL refers to nothing or the absence of a reference.*

Beispiel: leerer String

Unterscheidung zwischen dem leeren String und dem `null`-String:

```
1  String a = "";
2  String b = null;
```

Die Vorstellung von Strings soll ein beschriebenes Blatt Papier sein. Wie könnte das aussehen?

Nichts in Java

Wie kann man *nichts* in Java darstellen?

Erinnerung:

> *NULL refers to nothing or the absence of a reference.*

Beispiel: leerer String

Unterscheidung zwischen dem leeren String und dem `null`-String:

```
1  String a = "";
2  String b = null;
```

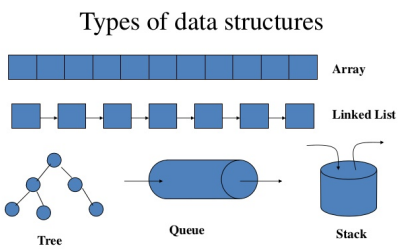
Die Vorstellung von Strings soll ein beschriebenes Blatt Papier sein. Wie könnte das aussehen?

- String `a` als leerer String ist ein unbeschriebenes Blatt Papier.
- String `b` ist gar kein Blatt.

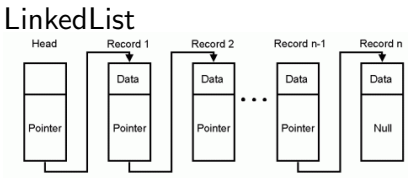
`string` – *Foliensatz*: Referenztypen, Folien: 32–39

Datenstrukturen

In welcher Situation eignet sich welche Darstellungstiefe?



There are many, but we named a few. We'll learn these data structures in great detail!



Foliensatz: Referenztypen, Folien: 378–381

Zusammenfassung: Listen

Schleife mit allgemeinen Visualisierungstechniken

Darstellung 1:

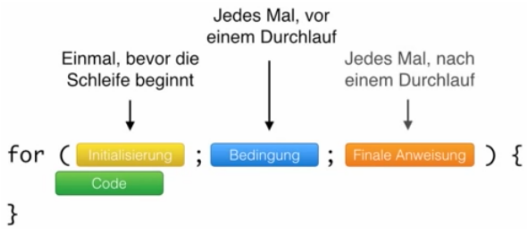


Abbildung: Quelle: [Th18]

Darstellung 2:

```
for (Initialisierung ; Bedingung ; Finale Anweisung) {
   Code
}
```

Was ist besser zum Erklären bzw. Verstehen geeignet?

Mentale Modelle und Vorstellungen zur Schleife

Ziel: Schleife (loop) als Vorstellung eines kreisförmigen Kontrollflusses (Weigend 2007)

Alltagsbeispiele

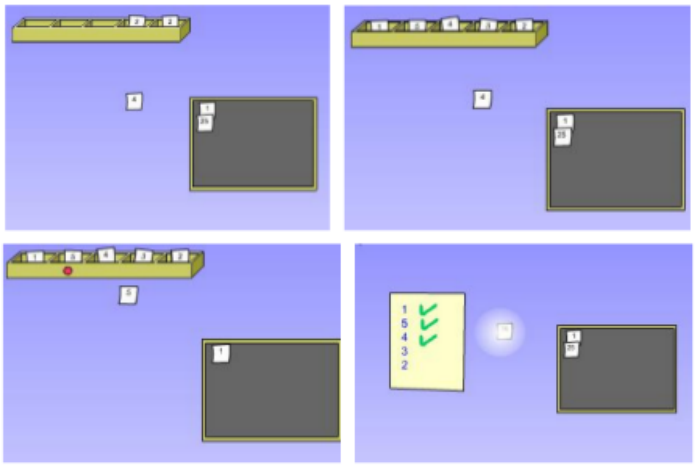
- Jahreszeitenzyklus – wirklich? –
- Emails checken
- Pixel eines Bildes verändern
- Treppe hochlaufen

Welche Art von Schleife wird durch die Bilder suggeriert?

Vorstellung von *Sprüngen* im Programmablauf ist essentiell für Schleifen. Die Sprünge sollten visualisiert werden!

Iteration – datengesteuerte Wiederholung

Eine fünf-elementige Collection soll elementweise kopiert werden. (Wortwahl: Weigend 2007)

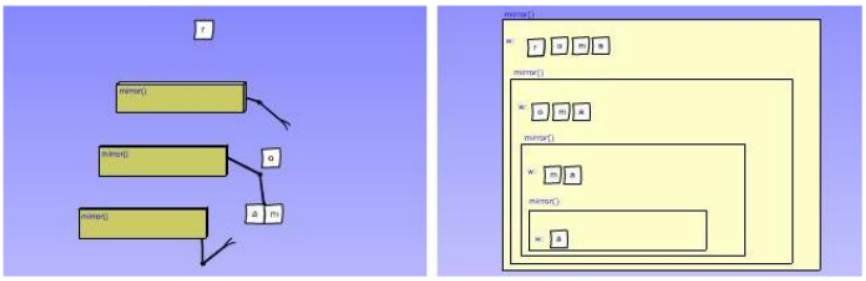


Was passt in den Darstellungen nicht?

Rekursion

Hier nur knapp, da Rekursion noch in einem eigenen Modul behandelt wird.

Beispiel:
Methode `mirror`, die rekursiv den Eingabe-String `roma` rückwärts schreibt



Was könnte bei diesen Visualisierungen problematisch sein?

Vorstellungen der OO - Klasse

- *Bauplan:* Beschreibung von Verhalten und Eigenschaften von Objekten
- *Fabrik für Objekte:* Fokus auf das Instanziieren bzw. "Produzieren".
- *Schablone zum Erzeugen* umfasst sowohl die Bauplan- als auch den Fabrik-Vorstellung einer Klasse.
- *Behälter für Funktionen:* Beispiel: `a = Math.sqrt(2)` `Math` agiert als Behälter für Funktionen, es muss kein Objekt instanziiert werden!
- *eigene Entität* im Programm, die den Code strukturiert. Vorstellung ist alleine *nicht* tragfähig, nur in Verbindung mit anderen!
- *Modell* eines Phänomens der realen Welt.

Zusammenfassung: Referenzen und Objekte

`java.lang.Math` – *Foliensatz*: Referenztypen, Folien: 175–176

Vorstellungen der OO - Objekt

- *ein Stück Code*
  Vorstellung ist alleine *nicht* tragfähig, nur in Verbindung mit anderen!
- *aktiver Teil* bzw. *Akteur* im Code, der die Programmausführung beinhaltet:

  > *Nach dem Paradigma der OOP empfängt ein Objekt eine Botschaft, wertet sie aus und führt die spezifizierte Operation aus. Der Ursprung der Aktivität liegt also im beauftragten Objekt.*

- *Modell* eines Phänomens der realen Welt.

Zusammenfassung: Referenzen und Objekte

Vorstellungen bei Racket?

Aufgrund der mangelnden Fachliteratur zu funktionalen Programmiersprachen soll an dieser Stelle auf die allgemeinen Techniken zur Visualisierung verwiesen werden.

- Funktionen als Kästchen darstellen. Stichwort: Operator- bzw. Maschinenmodell.
- Funktionen höherer Ordnung sind demnach verschachtelte Kästen.

Der mentale Kern von Racket ist die *funktionale Abstraktion*.

Foliensatz: Funktionale Abstraktion, Folien: 348–367

B.2.3 Typische Fehler und Fehlvorstellungen

Tutor*innenschulung

Typische Fehler und Fehlvorstellungen

Svana Esche

Fachbereich Informatik – TU Darmstadt

Datum

Was wisst ihr schon?

Der Sprung ins kalte Wasser!

Arbeitsauftrag A – *10 Minuten Zeit – Think*
Stellt in Partnerarbeit eine Liste in Stichpunkten von typischen Fehlern von Programmieranfängern zusammen.

Was wisst ihr schon?

Der Sprung ins kalte Wasser!

Arbeitsauftrag A – *10 Minuten Zeit – Think*
Stellt in Partnerarbeit eine Liste in Stichpunkten von typischen Fehlern von Programmieranfängern zusammen.

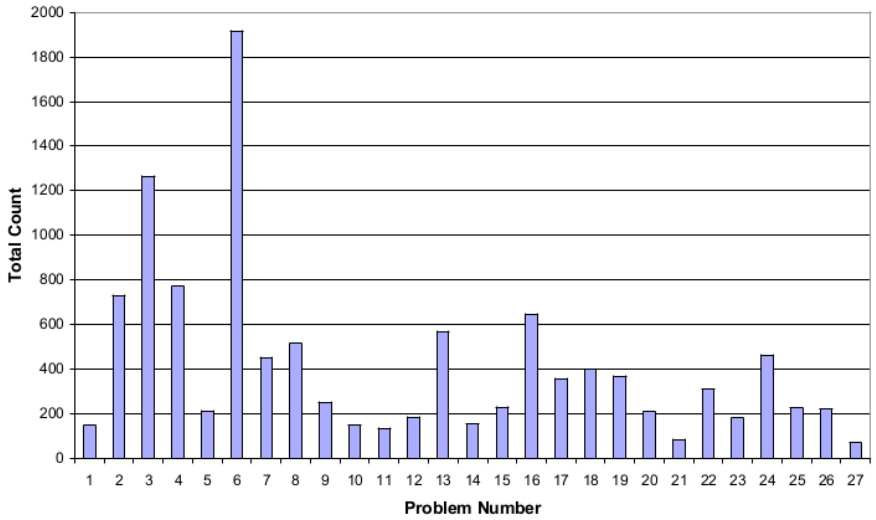
Arbeitsauftrag B – *10 Minuten Zeit*
Überfliegt die erhaltene Zusammenstellung von typischen Fehlern von Garner et al. (2005). Vergleicht sie mit eurer zuvor erstellten Liste unter den folgenden Aspekten:

1. Welche Probleme hattet ihr auch aufgeschrieben?
2. Welche hattet ihr nicht?
3. Was hattet ihr auch so gemeint, habt es aber anders formuliert?

Welche Fehler tauchen am häufigsten auf?

Arbeitsauftrag C – *10 Minuten Zeit – Pair*
Bildet Vierer-Gruppen. Tauscht euch über eure Ergebnisse zu den Arbeitsaufträgen A und B aus.
Bestimmt anschließend, welche fünf der Items nach Garner et al. (2005) am häufigsten auftraten. Eure Nummerierung entspricht *nicht* der in der folgenden Abbildung!

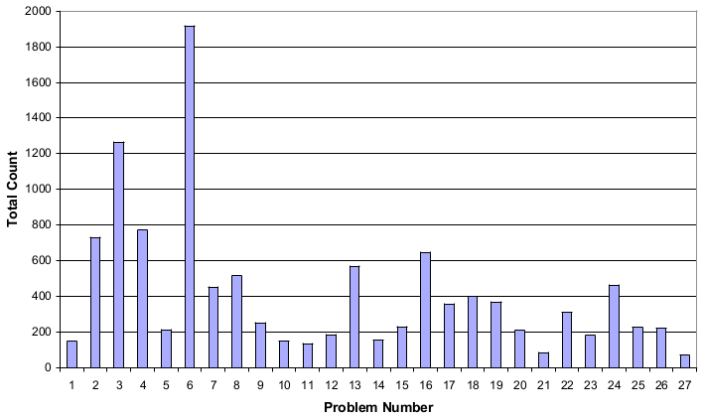
Welche Fehler tauchen am häufigsten auf?



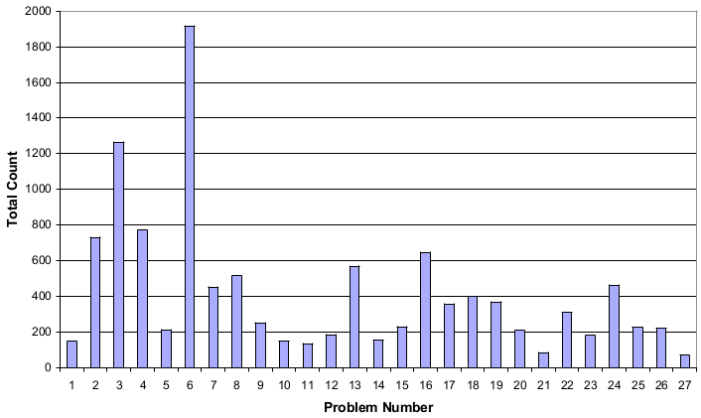
Erinnerung: Bestimmt die fünf häufigsten Fehler nach der Liste von Garner et al. (2005).

Was wisst ihr schon? Fehler und Fehlvorstellungen – exemplarisch Trennung von Lern- und Leistungssituationen

Welche Fehler tauchen am häufigsten auf?



Welche Fehler tauchen am häufigsten auf?



6 → IX Basic mechanics
3 → XI Stuck on Program Design
2 → X Understanding the task
4 → V Problems with the basic structure
16 → XVII Arrays

Wie geht es weiter?

- spezifische Fehler kennen, um sie in Student*innen-Lösungen zu *er*kennen

  ⇝ Kapitel *Fehler und Fehlvorstellungen – exemplarisch* direkt im Anschluss
- Vorstellungen vorher aufbauen anstatt Fehlvorstellungen zu ändern

  ⇝ Modul *Visualisierungen und Gedankliche Modelle*
- Spezifischen Fehler zur Frage „Wie fange ich an?“ mit Strategien und Patterns lehren begegnen

  ⇝ Modul *Problemlösestrategien: allgemein und informatikspezifisch*

  ⇝ Kapitel *Patterns* innerhalb des genannten Moduls

Syntaktische Fehler in Java

Syntaktische Fehler sind leicht zu finden und zu beheben bei Nutzung einer IDE. Sie fallen unter den Begriff der *Basic Mechanics*.

- *Irresolvable symbol error "cannot find symbol":* Variable benutzt, ohne diese vorher zu deklarieren
- Semikolon vergessen
- Klammer durcheinander: `([)]`
- Zuweisung `=` statt Vergleich `==`

== und = – *Foliensatz*: Referenztypen, Folien: 124–154

Konzeptuelle Fehler bzw. Fehlvorstellungen bei Variablen

Konzeptuelle Fehler sind schwieriger zu finden und folgen einer internen Logik bei den Student*innen und können daher zu Fehlvorstellungen führen.

Manche Student*innen glauben, dass

- Variablen mehr als einen Wert zur selben Zeit speichern können $\rightsquigarrow$ Behältermodell.
- die Reihenfolge von Statements irrelevant ist, bspw. ist `A = 5` äquivalent zu `5 = A`.
- Sprache Bedeutung impliziert, bspw. für die Variablen `smallest` und `largest` gilt: `smallest < largest`.

Konzeptuelle Fehler bzw. Fehlvorstellungen bei Variablen

Manche Student*innen

- verstehen nicht, wo bei der Verwendung von Variablen beim Input und Output die Daten herkommen und gespeichert werden.
- glauben, dass Print-Statements den Wert der Variable verändern, bspw. `System.out.print("Der Wert von X ist 1.");`.
- können Variablen nicht korrekt initialisieren und keine speziellen Variablen wie Summen- und Zählvariablen verwenden.
- übertragen mathematische Vorerfahrungen mit Variablen und scheitern damit an `A = A + 1`.

Foliensatz: Referenztypen, Folien: 5–16

Konzeptuelle Fehler bzw. Fehlvorstellungen bei Bedingungen

Konzeptuelle Fehler sind schwieriger zu finden und folgen einer internen Logik bei den Student*innen.

Manche Student*innen glauben, dass

- beide Blöcke bei `if`- und `else`-Statements ausgeführt werden.
- wenn die Bedingung eines `if`-Statements zu falsch ausgewertet wird, das ganze Programm stoppt.

Zusammenfassung: Programme und Prozesse

Fehler und Fehlvorstellungen bei Schleifen

- Eine Schleifenart ist „besser“ als die andere.
- Nur Teile im Schleifenkörper werden wiederholt, bspw. nur `print`-Befehle.
- Wahre Bedingungen innerhalb des Schleifenkörpers werden ausgeführt, auch wenn sich der Programmzeiger außerhalb der Schleife befindet.

`while` ⇝ während ⇝ Bedingung ist während des gesamten Ausführens gültig, anstatt einmaliges Abfragen pro Iteration. Sensibel mit Sprache umgehen!

`for`: Student*innen verstehen nicht bzw. vergessen, dass die Schleifen-Kontroll-Variable aktualisiert wird.

Zusammenfassung: Programme und Prozesse

Fehlvorstellungen der OO – Objekte

- Objekt als Wrapper für Variablen.
- Instanzierungsvariablen – Attribute – eines Objekts einer Klasse müssen allen vom gleichen Typ sein.
- Objekte sind einfache, inaktive Datensätze wie bspw. CD-Klasse, wo jedes Objekt eine CD repräsentiert.
- Arbeiten in Methoden funktioniert nur über Zuweisungen – Assignments – statt auch über message passing.
- Objekte werden automatisch instanziiert.

Foliensatz: Referenztypen, Folien: 42–54

Zusammenfassung: Referenzen und Objekte

Fehlvorstellungen der OO – Objekte und deren Variablen

Ausgangspunkt: Verwechseln des Attributs `name` *mit der Identität des Objekts oder einer Variable, die auf das Objekt verweist.*

- Nur eine einzige Variable kann zur gleichen Zeit auf ein gegebenes Objekt referenzieren.
- Der Verweis einer Variable auf ein Objekt ist unveränderbar.
- Eine Variable spezifiziert das zugehörige Objekt immer auf einzigartige Weise.
- Zwei verschiedene Variablen müssen auf zwei verschiedene Objekte zeigen.
- Ein Objekt weiß, welche Variablen auf es zeigen.
- Zwei Objekte der selben Klasse mit dem gleichen Status - Belegung von Attributwerten - sind das selbe Objekt.
- Zwei Objekte mit dem gleichen Wert für Attribut `name` sind das selbe Objekt.

== und = – *Foliensatz*: Referenztypen, Folien: 115–123

Fehlvorstellungen der OO – Klassen

- Keine Trennung zwischen Klasse und deren Rollen und damit eine Überspezialisierung der Klasse, sprich: übertriebene Form von Taxonomie, Klassifikationsschmema.
- Klasse ist eine Sammlung von Objekten und ein Objekt ist eine Teilmenge einer Klasse.
- Von jeder Klasse kann es nur ein einziges Objekt geben.

Klassen- und Objektmethoden – *Foliensatz*: Referenztypen, Folien: 162–174

Zusammenfassung: Referenzen und Objekte

Fehlvorstellungen der OO – Vererbung

Fehlerhafte Verwendung der *Vererbung*:

- Vererbung (ist-ein) statt Aggregation (hat-ein) wird verwendet.
- Vererbung wird verwendet, um die Funktionalitäten der Superklasse zu verwenden, ohne dass eine (ist-ein)-Beziehung besteht.

Foliensatz: Referenztypen, Folien: 55–63

Fehlvorstellungen der OO – Design

Seperation of Concerns wird nicht beachtet.

- Das Aufteilen von Concerns – Dinge von Interesse – in einzelne Module.
- Bei Java werden Concerns in Objekte und deren Klassen und Vererbung dazwischen aufgeteilt.
- *MVP – Model - View - Controller* – trennt Inhalt von Präsentation und Steuerung.

Das resultiert teilweise in folgenden Fehlern.

- Ein Attribut sollte Attribut einer anderen Klasse sein.
- Ein Attribut sollte Objekt einer Klasse statt ein String sein.
- Zwei oder mehr Attribute sollten als eigene Klasse ausgelagert werden.

Foliensatz: Korrekte Software, Folien: 314–318, 326–377

Strategische Fehler

Manche Student*innen

- überprüfen nicht die zulässigen Grenzen von Bedingungen und die unerwarteten Fälle.
- können Code-Blöcke nicht verschmelzen, die zusammen angewendet werden sollen.
- konkatenieren Patterns, ohne sich um den Übergang zwischen diesen zu kümmern.
- vergessen Wächter bzw. guards zu benutzen, um bspw. die Division durch Null zu vermeiden – bezogen auf Programmiersprachen wie Java und C#.

Foliensatz: Funktionale Abstraktion, Folie: 89

Zusammenfassung: Vertrag und Tests

Sprachlich bedingte Fehler

Grund: Befehle von Programmiersprachen basieren auf natürlicher Sprache.

- `and` ist ein boolescher Operator und eine Konjunktion in natürlicher Sprache, die auch „aber“, „und deswegen“ oder „und dann“ bedeuten kann.
- `or` ist ein boolescher Operator – nicht-ausschließend –, der in natürlicher Sprache meistens jedoch ausschließend verwendet wird.
- Fähigkeit für die Sprache Englisch korreliert signifikant mit dem Erfolg beim Programmieren lernen – Studie bezogen auf chinesische Student*innen.
- Metonymie benutzen: Irgendwas wird nicht bei dem eigentlichen Namen referenziert, sondern mit etwas Ähnlichem bspw. `temperature` statt `value` des Objekts `temperatureDarmstadt`.

Trennung von Lern- und Leistungssituationen

Fehler sind nicht per se schlecht!
Sinnvolles und produktives Lernen kann nicht passieren, ohne Fehler zu machen.
Präsenzübung ist Lernphase, Hausübung ist Leistungsphase.
Bei permanenter Leistungssituation kommt es zu Vermeidungsstrategien:

- Fehler werden vertuscht,
- Keine Nachfragen bei Nichtverstehen,
- Vermeidung von neuen Lernwegen.

Ihr sollt den Student*innen klar machen, dass sie euch alles fragen und alle Fehler und Unsicherheiten bei euch zugeben können.
Daher: nicht über Fehler lustig machen, keine personelle Abwertungen, etc. egal wie „doof“ die Fehler sind.

B.2.4 Problemlösestrategien

Tutor*innenschulung

Problemlösen

Svana Esche

Fachbereich Informatik – TU Darmstadt

Datum

Was versteht man unter Problemlösen?

Polya (1949)

1. Verstehen der Aufgabe/des Problems
2. Ausdenken eines Plans (Entwurfsskizze)
3. Plan ausführen
4. Rückschau und Ausblick

Vorsicht: nicht unbedingt linear zu verstehen!

Heurismen sind Erkenntnisse, die auf Analogien, Erfahrungen und Hypothesen aufbauen.

Strategien / Heurismen kann man sich als "meta-skills" bzw. "statements of wisdom" vorstellen.

Eigenschaften des Prozesses des Problemlösens

- verläuft nicht algorithmisch, linear,
- besitzt kein strenges Schema,
- weist häufig Irrtum-Versuchs-Strategien in mehreren Zyklen auf,
- zeigt große Bedeutung von Fehlern (Fehler sind positiv zu werten),
- erfordert Strategien / Heurismen.

Implikationen für die Tutor*innen

*Was bedeutet das für die Tutor*innen?*

- Verständnis und Motivation zeigen, sowie Mut machen im Umgang mit den Student*innen.
- Zuverlässigster Faktor für Erfolg ist die Note, die Student*innen glauben zu erreichen.
- Es gibt nicht die *eine* Lösung, den *einen* Weg zur Lösung → Vielfalt unterstützen.
- Fragen zur Rückschau anregen: Wie seid ihr vorgegangen? Kann man die Strategie auf andere Aufgaben übertragen?

Warum Strategien / Heurismen lehren?

- Das Lehren von Strategien / Heurismen vereinfacht den Prozess des Lernens.
- und gibt Orientierung,
- Strategien / Heurismen können Ideen zur Lösung geben (Wie soll ich anfangen?),
- geben gerade unerfahrenen Programmierer*innen eine Hilfestellung an die Hand, die erfahrene durch Erfahrung implizit vollziehen.

Die Verwendung von Strategien / Heurismen verbessert nicht das Verstehen und Können von erfahrenen Programmierer*innen. Aber sie *können* als Hilfe dienen!

Problemlösestrategien – ganz konkret

Welche kennt ihr schon? vs. Strategien nach Kaplan (2010)

Arbeitsauftrag A – *10 Minuten Zeit – Think*
Bearbeitet den euch ausgeteilten Arbeitsauftrag – Kreis oder Quadrat – in Partnerarbeit.

Problemlösestrategien – ganz konkret

Welche kennt ihr schon? vs. Strategien nach Kaplan (2010)

Arbeitsauftrag A – *10 Minuten Zeit – Think*
Bearbeitet den euch ausgeteilten Arbeitsauftrag – Kreis oder Quadrat – in Partnerarbeit.

Arbeitsauftrag B – *10 Minuten Zeit – Pair*
Bildet Vierer-Gruppen, indem sich jeweils ein Paar von Kreis und Quadrat zusammentun.
Tauscht euch über eure Listen aus.

Begriffsklärung Problemlösestrategien Strategisches Wissen – Patterns Exemplarische Auswahl an Patterns

Problemlösestrategien – ganz konkret

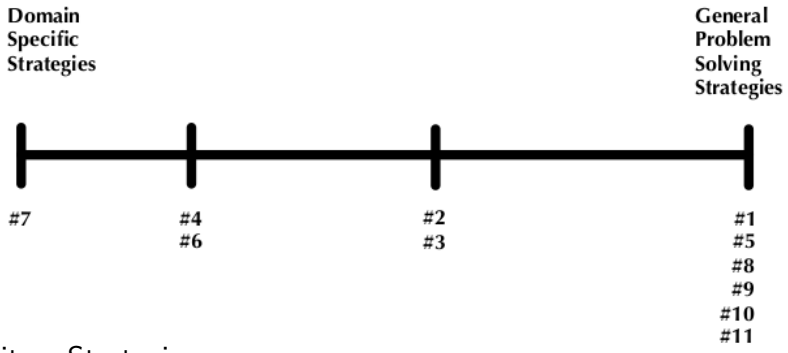
Welche kennt ihr schon? vs. Strategien nach Kaplan (2010)

Arbeitsauftrag A – *10 Minuten Zeit – Think*
Bearbeitet den euch ausgeteilten Arbeitsauftrag – Kreis oder Quadrat – in Partnerarbeit.

Arbeitsauftrag B – *10 Minuten Zeit – Pair*
Bildet Vierer-Gruppen, indem sich jeweils ein Paar von Kreis und Quadrat zusammentun.
Tauscht euch über eure Listen aus.

Share
Gab es Aha-Erlebnisse beim Bearbeiten der Arbeitsaufträge?
Waren die Problemlösestrategien informatikspezifisch oder nicht?
Wenn ja, welche?

Kontinuum der Strategien nach Kaplan (2010)



Weitere Strategien:

- Reduziere die Unterschiede zwischen Gegebenem und Ziel.
- Rückwärtsarbeiten: zentrale Kalkulation → Initialisierung, Output, etc. .
- siehe nächstes Kapitel *Pattern*.

Es gibt noch viel mehr Pattern, aber für *FOP* reicht ein kleine Anzahl.

Strategisches Wissen – Patterns / Schemata / Pläne / ...

Patterns / Schemata / Pläne / Templates sind

Strategisches Wissen – Patterns / Schemata / Pläne / ...

Patterns / Schemata / Pläne / Templates sind

- aus Sicht der Softwareentwicklung

  descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

- aus Sicht der Objektmodellierung

  eine bewährte, generische Lösung für ein immer wiederkehrendes Problem, das in bestimmten Situationen auftritt.

- aus Sicht der Didaktik

  key elements needed to write parts of the program.

Begriffsklärung Problemlösestrategien Strategisches Wissen – Patterns Exemplarische Auswahl an Patterns

Warum Patterns lehren?

Es kennzeichnet Expert*innen, dass sie

- Patterns anwenden, um Probleme zu erkennen, die auf ähnliche Weise gelöst werden.
- informatikspezifische neben allgemeinen Problemlösestrategien benutzen.

Strategisches Wissen soll vermittelt werden, weil es *nicht* ausreichend zum erfolgreichen Programmieren ist, die Syntax und Semantik der Programmiersprache korrekt verstanden zu haben.

Patterns sind jedoch keine Wundermittel, die garantiert zum Ziel führen, siehe *Eigenschaften des Problemlöseprozess*.

Bandbreiten-Genauigkeits-Dilemma:

Je allgemeiner die Werkzeuge sind, desto geringer ist ihr Nutzen bei der Lösung spezifischer anspruchsvoller Probleme.

Exemplarische Auswahl an Patterns

Es folgt eine *Auswahl* an Patterns, die für Erlernen des Programmieren hilfreich sind. Einen guten Überblick liefert

`http://csis.pace.edu/~bergin/patterns/`

Es geht hier nicht um die Pattern der Softwareentwicklung, wie sie bspw. von der Gang of Four beschrieben wurden.

Read-Process-Write Pattern – Lesen-Verarbeiten-Schreiben

Vorgehen:

1. Read: Input lesen
2. Process / Calculate: Input verarbeiten
3. Write: Output produzieren

Beispiel:

1. Arraywerte auslesen: `a[i]`
2. Aktuellen Wert mit aktuellem Maximum vergleichen und ggf. anpassen: `a[i] > max`
3. Output produzieren: `return max`

Encapsulation Pattern – Hilfsfunktionen

Typische Situationen, in denen es sinnvoll ist, Hilfsfunktionen zu schreiben:

- Umwandeln von Einheiten: Meter zu Meilen, Währungen, etc.
- Koordinatentransformation: Maßstäbe, Skalieren, Polarkoordinaten
- komplexe innere Schleife bei Verwendung von mehreren Schleifen
- komplexe Bedingungen für Wiederholungen: eigenen boolschen Wert anlegen
- aufwendige Berechnungen von Konstanten

Aufteilung einer Schleife

Wiederholung / Repetition in 5 Schritte aufgeteilt:

1. initial setup
2. verifying loop condition
3. loop body
4. loop condition update
5. post-mortem

Strategien, um Schleifen zu schreiben

- Schleifen-Body zuerst schreiben anstatt mit dem Schleifenkopf bzw. Schleifenbedingung anzufangen, Iterator-Variable sollte trotzdem benannt werden
- Wiederholte Berechnungen der selben Anzahl in Schleifen-Body zusammenfassen.

Vorteile:

- Art der Schleife – `for` oder `while` – kann später entschieden werden.
- Schleifenbedingung kann später geschrieben werden.
- Fokus auf die Lösung des Problems.

Es kann natürlich auch eine Strategie sein, den Schleifenkopf zuerst schreiben:
Dann macht man sich zuerst klar, unter welchen Bedingungen bzw. wie oft etwas getan werden soll.

Begriffsklärung Problemlösestrategien Strategisches Wissen – Patterns Exemplarische Auswahl an Patterns

Traversal Pattern – Durchschreiten

Ein Pattern, um ein Datenobjekt aus einer Datenkollektion zu übergeben. Es ist hilfreich um bspw. Daten einzulesen.

- einfache lineare Traversierung: Vektoren, Arrays, Strings
- Traversierung mittels Stream: kein Speicher, funktional bearbeitbar
- Traversierung einer vernetzten Struktur: Listen, Graphen, Bäume, Binärbäume
- Iteratorbasierte Traversierung

Beispiel für die Kombination von Pattern

Problem: Die älteste Person in einer Gruppe bestimmen.

- Gruppe als Array auffassen
- Repetition Pattern, indem Verarbeitung des Inputs wiederholt geschieht
- Traversalpattern, um in der Schleife den nächsten Eintrag des Arrays anzusprechen

B.2.5 Rekursion

Tutor*innenschulung

Rekursion

Svana Esche

Fachbereich Informatik – TU Darmstadt

Datum

Wo steht ihr selbst?

- Test zur Rekursion wie er von Kahney (1981) mit Student*innen und erfahrenen Programmierer*innen durchgeführt wurde.
- Zeitdauer: 15 Minuten
- alleine arbeiten
- Test wird nicht bewertet, sondern es gibt eine Selbstauswertung am Ende der Einheit.

Wichtigkeit von Rekursion

Rekursion ist inhaltlich wichtig in der Informatik, da

- sie benutzt wird, um rekursive Datentypen wie Listen, Bäume und Graphen zu definieren.
- Algorithmen, die nach dem Prinzip *Teile und Herrsche* verfahren, häufig Rekursion verwenden.
- primitiv rekursive und μ-rekursive Funktionen für die Theorie der Berechenbarkeit elementar sind.

Rekursion ist wichtig zu erklären, da

- sie schwer zu verstehen ist.
- sie ein abstraktes Konzept ist, und sich daher nicht selbst erklärt.
- es wenige Analogien im Alltag gibt.

Mentales Modell: Kopien

Kahney beschreibt die Vorstellung der Rekursion wie folgt:

> *recursion as a process that is capable of*
> *triggering new instantiations of itself,*
> *with control passing forward to successive instantiations*
> *and back from terminated ones.*

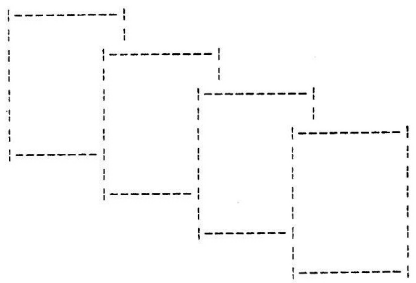
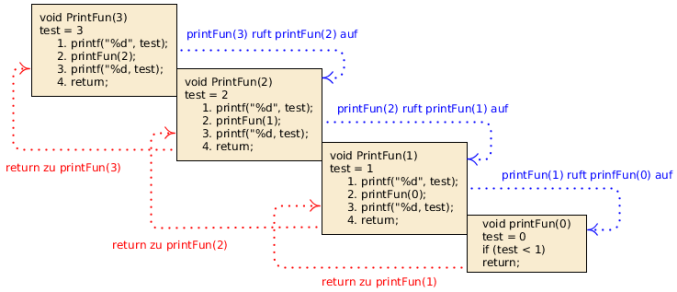


FIG. 11.1 A graphic depiction of the 'Copies' model of recursion.

Mentales Modell: Kopien



abgewandelt von `https://www.geeksforgeeks.org/recursion/`

Sprachbausteine: active flow, passive flow, base case

1. Was wird beim Print-Befehl in 3. bei `printFun(2)` ausgegeben?
2. Was macht die Methode `printFun`?
3. Wo sind hier die Kopien?

Mentales Modell: Schleifen

```
¦-----------------------------------¦
¦                                   ¦
¦                                   ¦
¦   ¦--->PROCEDURE-NAME        /PARAMETER/          = 'ENTRY POINT'
¦   ¦
¦   ¦       NOTE  <node>  <relation>  <node>        = 'ACTION PART'
¦   ¦
¦   ¦        CHECK <node>  <relation>  <wild-card>  = 'PROPAGATION-
¦   ¦         IF PRESENT: PROCEDURE-NAME  *; EXIT       MECHANISM'
¦   ¦                            ¦          ¦
¦   ¦                            ¦          ¦
¦   ----------------------------            ¦
¦                                           ¦
-------------------------------------------
```

FIG. 11.2 A graphic description of the 'Loop' model of recursion.

Für welche Art von Programmen ist das Schleifenmodell dennoch passend?

Mentales Modell: Schleifen

```
|-----------------------------------|
|                                   |
|                                   |
|   |--->PROCEDURE-NAME        /PARAMETER/          = 'ENTRY POINT'
|   |
|   |       NOTE  <node>  <relation>  <node>        = 'ACTION PART'
|   |
|   |        CHECK <node>  <relation>  <wild-card>  = 'PROPAGATION-
|   |         IF PRESENT: PROCEDURE-NAME  *; EXIT       MECHANISM'
|   |                           |         |
|   |                           |         |
|   ----------------------------          |
|                                         |
-------------------------------------------
```

FIG. 11.2 A graphic description of the 'Loop' model of recursion.

Für welche Art von Programmen ist das Schleifenmodell dennoch passend?

Stichwort: *Endrekursion* bzw. *Tail recursion*
Also wenn der rekursive Funktionsaufruf die letzte Aktion zur Berechnung von der Funktion `f` ist.

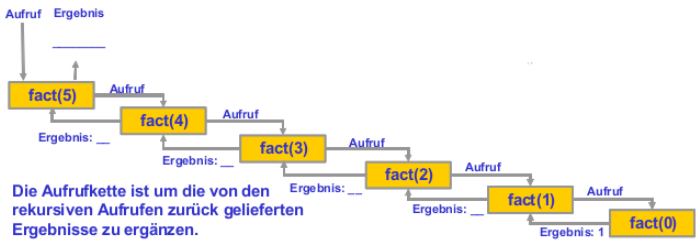
Endrekursive Funktion: exemplarisch

```
1  (define (factorial n)
2    (fact-iter 1 1 n))
3  (define (fact-iter product counter max-count)
4    (if (> counter max-count)
5      product
6      (fact-iter (* counter product)
7                 (+ counter 1) max-count)))
8  (factorial 5)
9  (fact-iter 1 1 5)
10 (fact-iter 1 2 5)
11 (fact-iter 2 3 5)
12 (fact-iter 6 4 5)
13 (fact-iter 24 5 5)
14 (fact-iter 120 6 5)
15 120
```

Was bedeutet das für das Erklären?

- Achtet auf eure Wortwahl beim Erklären!
  Negativbeispiel: Rekursionsanker, Rekursionsbasis als „Abbruch“ (Lexikon der Informatik 2011: 756)
- Pfeile in unterschiedlichen Farben für Control Flow vorwärts bzw. aktiv und rückwärts bzw. passiv und dazu mehrere Kopien am Beispiel

Erster Ansatz bzw. Darstellung zur Methode `printFun`:



Was bedeutet das für das Erklären?

- Farbiges Markieren der *Ähnlichkeit* bzw. Gleichheit der Kopien, um das Kopien-Modell zu unterstützen.
- Geeignete Metaphern: *Ähnlichkeit*, *Selbstreferenz*, bei *Zyklus* und *Wiederholung* zwingend auf das Auf- und Zuklappen der Aufrufkette verweisen.
- Beispiele favorisieren, die nicht aus einem einzigen rekursiven Aufruf bestehen: Zahlenfolge der Fibonacci-Zahlen besteht aus zwei rekursiven Aufrufen.

Foliensatz: Funktionale Abstraktion, Folien: 79–95

Zusammenfassungen: Rekursive Subroutinen und Listen, Strukturelle vs. Generative Rekursion, Programme und Prozesse

Stack

- *Call-Stack* bei Aufruf einer Subroutine – hier: rekursiver Aufruf – wird in FOP angesprochen.
- *Achtung:* Nicht jeder der FOP hört, muss Digitaltechnik – ehemals TGDI – hören.
- Je nach Kenntnis der Student*innen: Stack – last-in-first-out – und Speicherverwaltung aufgreifen.

Zusammenfassung: Programme und Prozesse

Auswertung des Tests

- Lösung 1 richtig: Keine Rekursion vorhanden, daher nicht die gewünschte Funktionalität; Hinweis, dass Rekursion möglicherweise nicht verstanden wurde
- Lösung 2 richtig und Lösung 3 falsch: Hinweis auf Schleifenmodell
- Lösung 2 und Lösung 3 richtig: Hinweis auf Kopienmodell

B.2.6 Rückblick

Tutor*innenschulung

Rückblick

Svana Esche

Fachbereich Informatik – TU Darmstadt

Datum

Feedbacks vom Beginn aufgreifen

Arbeitsauftrag - Zweierteams - 10 Minuten

1. Tausche mit deiner Sitznachbar*in das von dir erstellte Feedback zur realen Student*innenlösung von Beginn des Schulungstages aus.
2. Wie empfindest Du das erhaltene Feedback? Was würdest Du anders machen?
3. Tauscht euch zu zweit darüber aus.

Danke schön!

Gibt es noch offene Fragen?

Danke schön!

Gibt es noch offene Fragen?

Daumenfeedback!

B.3 Arbeitsblätter

B.3.1 Wissensstand diagnostizieren

WISSENSSTAND DIAGNOSTIZIEREN

Überfliege kurz die vorliegende Aufgabenstellung und lese dann die reale Lösung einer Student*in auf der letzten Seite. Korrigiere sie und notiere schriftlich in Stichpunkten *alles*, was dir beim Lesen und Korrigieren dazu durch den Kopf gegangen ist.
Formuliere anschließend auf einem extra Zettel ein Feedback an die Student*in, die die Lösung erstellt und als Hausübung abgegeben hat. Hierbei kannst Du dich an die Hilfestellungen zum Erstellen von Feedback der Tutor*innenschulung von Frau Gölz orientieren.

Aufgabenstellung 5 Warensortiment eines Supermarktes als Baum (8 Punkte)

Um mehr Ordnung in das Angebot eines Supermarktes zu bringen, soll seine Warenliste durch eine Baum-Struktur (den Supermarktbaum oder `product-tree`) dargestellt werden. Ihre Aufgabe ist es, dabei zu helfen, dass dieser Baum sinnvoll genutzt werden kann. Sie sollen es dabei ermöglichen:

- einen Baum zu erstellen,
- einen Baum nach bestimmten Kriterien zu durchsuchen,
- den Preis einer eigenen Einkaufsliste zu berechnen.

Betrachten Sie zunächst die Repräsentation eines Produkts als `product` sowie einige Testprodukte:

```
;; product models a product of a supermarket
;; id: number - the product-id
;; name: symbol - name of the product
;; price: number - price of the product in Euro
;; categorie: symbol - the categorie which the product matches
(define-struct product (id name price categorie))

;; examples
(define apple (make-product 100 'apple 0.99 'fruit))
(define melon (make-product 105 'melon 4.49 'fruit))
(define orange (make-product 109 'orange 1.59 'fruit))
(define banana (make-product 111 'banana 1.29 'fruit))
(define cherry (make-product 113 'cherry 3.49 'fruit))
(define water (make-product 55 'water 0.5 'drink))
```

```
(define tea (make-product 57 'tea 0.99 'drink))
(define cola (make-product 60 'cola 1.29 'drink))

(define fruits (list apple melon cherry orange banana))
(define drinks (list tea water cola))
(define mixed (append fruits drinks))
```

Beachten Sie insbesondere auch die Listen `fruits` , `drinks` und `mixed`. Diese Listen und Produkte dürfen Sie für Beispiele und Tests nutzen. Jedem Produkt wird eine eindeutige Produkt-Id zugeordnet, welche zwei Produkte unterscheidet. Auch wird im Folgenden davon ausgegangen, dass der Name eines Produktes ebenfalls einzigartig ist.

Aufgabenstellung - 5.2 Prädikate (1 Punkt)

Um den Baum später nach passenden Produkten durchsuchen zu können, sollen Prädikate benutzt werden. In dieser Aufgabe sollen sie zwei dieser Prädikate implementieren. Ein Prädikat bekommt jeweils ein Produkt und einen Prädikat-Wert gegeben und bestimmt, ob das Produkt das Prädikat für diesen Wert erfüllt. Bitte beachten Sie die Reihenfolge: 1. Produkt 2. Prädikat-Wert

Zuerst soll das Prädikat `categorie=?` implementiert werden. Dieses soll für ein Produkt und eine gegebene Kategorie als Prädikat-Wert bestimmen, ob das Produkt zu dieser Kategorie gehört. Beispiel:
`(categorie=? apple 'fruit)` soll `true` ergeben.

Als zweites Prädikat soll `cheaper?` implementiert werden. Dieses Prädikat soll angeben, ob ein Produkt billiger als der angegebene Prädikat-Wert ist. Beispiel:
`(cheaper? apple 2.0)` soll `true` ergeben.

Hinweis: Bei identischen Werten soll `false` zurückgegeben werden.

Reale Lösung einer Student*in aus dem WiSe 17/18

```
;; ====== Problem 5.2 ======
;; creation of predicates

;; get-name: <exp> -> symbol
;; expl.: Liest den namen des Produkts ein
;; example: apple -> 'apple
(define (get-name x) (product-name x))

;; categorie=?: symbol + symbol -> boolean
;; expl.: Prueft, ob das eingegebene Produkt mit der zugehoerigen Kategorie
   uebereinstimmt
;; example: apple 'fruits -> true

(define (get-name x) (product-name x))

(define (categorie=? list? prod);; categ)
  (cond [(empty? prod) false]
    [(symbol=? list? 'fruits)
      (cond
        [(empty? fruits) false]
        [(symbol=? (get-name (first fruits)) prod) true]
        [categorie=? (rest fruits)]
    [(symbol=? list? 'drinks)
      (cond
        [(empty? drinks) false]
        [(symbol=? (get-name (first drinks)) prod) true]
        [categorie=? (rest drinks)])])]))

(check-expect (categorie=? 'fruits 'apple) true)
(check-expect (categorie=? 'fruits 'cola) false)
```

Eigene Notizen und Gedanken

B.3.2 Typische Fehler nach Robins et al. (2003)

TYPISCHE FEHLER NACH GARNER ET AL. (2005)

I Tools
Problems with the Mac, OS X, directories (lost files), jEdit, Applet runner, or other basic tools. Includes being unable to find the resources described in the lab book, but not other kinds of general lab book / text book issues (do not record these). Does not include Java / file naming conventions [...]

II Exceptions, throw catch
Problems with exceptions, throw catch.

III Control flow
Problem with basic sequential flow of control, the role of the `main` or `init` method. Especially problems with the idea of flow of control in method call and return (e.g. writing methods and not calling them, expecting methods to get called in the order they are written). [...]

IV Loops
Conceptual and practical problems relating to repetition, loops (including for loop headers, loop bodies as blocks)

V Problems with basic structure
This problem number is meant to capture problems at the class/object/major structural level – they have a general design and classes [...], but are getting basic structural details wrong. E.g. code outside methods, data fields outside the class, data fields inside a method / confused with local variables (problems specifically with data fields or about or within methods, e.g. mixing up loops, will be some other problem number). Includes very basic problems with creating and using instance objects, e.g. how many, what they are for (but not more specific problems [...])

VI Selection
Conceptual and practical problems relating to selection – `if`, `else`, `switch` (including the use of blocks).

VII Strings
Strings and `string` functions. Does not include formatting output [...] or problems relating specifically to strings as reference types [...].

VIII Problem naming things
They have a problems choosing a name for something. Especially where this seems to suggest that they don't understand the function of the thing that they are trying to name.

IX Basic mechanics
Problems with little mechanical details (where these are not better described by some other problem). Braces, brackets, semi-colons. Typos and spelling. Java and file naming conventions. Import statements (when forgotten, not when misunderstood [...]). Formatting output. Tidiness, indenting, comments.

X Understanding the task
Problems understanding the lab exercise / task or its "solution". In other words, whatever other problems they may be having, in this case they don't actually know what it is that the program is supposed to be doing. (Does not include minor clarifications of some detail, which do not need to be recorded.)

XI Stuck on program design
They understand the task / solution [...] but can't turn that understanding into an algorithm, or can't turn the algorithm into a program. Cases such as "I don't know how to get started" or "what classes should I have?" or "what should the classes do?".

XII Booleans and conditions
Problems with booleans, truth values, boolean expressions [...]. Problems with loop or selection headers / conditions will have to be judged carefully – is this a problem formulating the boolean expression [...] or understanding how the expression / result is relevant to the loop or selection [...] ?

XIII Expressions & calculations
Problems with arithmetic expressions, calculations, notation such as "++", and all forms of precedence [...]

XIV Constructors
Specific problems with the concept of/purpose of a constructor.

XV Method signatures and overloading
Problems related to overloading. Failure to understand how method signatures work/which version of a method gets called. (Includes problems with constructors that are really about the signatures of constructors [...]).

XVI Data flow and method header mechanics
Especially, conceptual problems with arguments/parameters and return types/values. Includes problems with method header mechanics (incorrect or mismatching parameter specifications, incorrect return types or use of void). [...]

XVII Arrays
Problems relating to arrays as a data structure, including array subscripts, array contents, array declaration and initialisation. [...] Does not include failing to understand that an array as a whole is itself a reference type or may contain references [...].

XVIII Terminal or file IO
Problems with terminal or file IO/data flow (not including exception handling, [...] or output formatting [...].

XIX Visibility and scope
Problems with data field visibility, local variable scope (e.g. defining a variable in one block and trying to use it in another, problems arising from unintended reuse of identifiers), and namespace / imported package problems (but not including forgotten `import` statement, [...]). Includes cases confusing data fields and variables of the same name, but not where this is better described as a failure to understand `this` [...].

XX Variables
Problems with the concept of or use of variables. Includes problems with initialisation and assignment. [...] Does not include cases more accurately described as problems with reference types [...] or arrays [...] rather than the concept of a variable.

XXI Accessors / Modifiers
Specific problems [...] with the concepts of / purpose of an accessor or a modifier method.

XXII Data types and casting
Problems caused by failing to understand different data types and casting for primitive types [...].

XXIII GUI mechanics
Problems with GUIs and the use of AWT, Swing etc. . Includes problems with specific required methods such as `actionPerfomed()`, `run()`, implements `actionListener` and so on [...]. Does not include the underlying concepts of event driven programming.

XXIV Class versus instance
Problems understanding the class object vs. instance object distinction, including problems with class and instance data fields (and use of `static`).

XXV Hierarchies
Problems relating to hierarchical structure, inheritance (`extends`, `overriding`, `shadowing`, `super`), and issues relating to the use of abstract methods or interfaces.

XXVI Event driven programming
Problems with the underlying concepts of event driven programming, and general "flow of control" type issues that arise in the transition from application to applet.

XXVII Reference types
Problems arising from a failure to understand the concept or use of reference types (references / pointers, `this`, different references to the same object, etc.), or that reference types behave differently from primitive types (when assigned, compared etc.).

B.3.3 Simulationen

SIMULATIONEN

Aufgabenstellung

Eure Aufgabe als Dreiergruppe in dieser Phase ist, Gespräche mit Student*innen mithilfe von Simulationen zu erproben.
Die *inhaltliche Grundlage* für dieses Gespräch sind die realen Fingerübungen, die die Student*innen in der Arbeits- und Intensivphase der Kleingruppenübung bearbeiten. Für die Simulationen wurden aus dieser Sammlung einzelne Übungen ausgewählt und liegen als Material bereit.
In jeder Simulation gibt es folgende *drei Rollen*, die zu vergeben sind

1. *Tutor*in:* Eine Person der Dreiergruppe, die sich in der Rolle der Tutor*in selbst darstellt und versucht so gut es geht, die Probleme zu erklären und zu helfen. Erinnerung: W-Fragen!
2. *Student*in:* Eine Person der Dreiergruppe, die Probleme mit der Aufgabe hat, und diese mit der Tutor*in besprechen möchte. Hierzu kann sich an den typischen Fehler orientiert werden.
3. *Beobachter*in:* Eine Person der Dreiergruppe, die das Gespräch zwischen Tutor*in und Student*in beobachtet und anschließend der Tutor*in Feedback zum beobachteten Verhalten gibt.
   Wenn Tipps dazu benötigt werden → letzte Seite.

Die Zeitplanung ist wie folgt: Auswahl einer Übung durch die *Student*in*, 5 Minuten Vorbereitung auf das Gespräch, 5 Minuten Simulation des Gesprächs und anschließend 5 Minuten Feedback durch die *Beobachter*in*. Die Rollen rotieren, sodass am Ende des Blocks von 45 Minuten, jede*r der Dreiergruppe jede Rolle simuliert hat.
Erinnerung: Die Aufgaben beziehen sich auf die Programmiersprachen Racket und Java.

V8 Suche in Zahlenliste ★★☆

Definieren Sie eine Funktion `contains-x?`. Diese bekommt eine Liste und eine Zahl übergeben und liefert genau dann `true` zurück, wenn die übergebene Zahl mindestens einmal in der übergebenen Liste vorkommt.

V9 Duplikate in Zahlenliste ★★☆

Definieren Sie eine Prozedur `duplicates?` mit einem Parameter `lst`, die genau dann `true` zurückliefert, wenn eine Zahl mehr als einmal in `lst` vorkommt. *Hinweis*: Können Sie hier vielleicht Ihre Funktion aus Aufgabe V8 verwenden?

V4 Lambda-Ausdrücke ★☆☆

```
;;
(define (z x)
;;
        (lambda (y) (* x y)))
```

Ergänzen Sie den Vertrag, sowohl für die Funktion `z`, als auch für den Lambda-Ausdruck. Was liefert `((z 3) 4)` zurück?

V5 Variablen

★ ☆ ☆

Legen Sie eine Variable `int a` an und setzen Sie ihren Wert auf 127. Jetzt legen Sie eine weitere Variable `int b` an und setzen Ihren Wert auf 42. Was gibt nun der Ausdruck `int c = a % b` wieder? Beschreiben Sie in Ihren eigenen Worten, für was der `%` Operator verwendet werden kann.

V6 Bedingungen II

★ ☆ ☆

Ihr Kommilitone ist etwas tippfaul und lässt deswegen gerne einmal Klammern weg, um sich Arbeit zu sparen. Er hat in seinem Code eine Variable `int number` angelegt, in der er eine Zahl speichert. Ist diese Zahl kleiner als 0, so möchte er das Vorzeichen der Zahl umdrehen und sie anschließend um 1 erhöhen. Ist die Zahl hingegen größer als 0, so möchte er die Zahl verdoppeln. Dazu schreibt er folgenden Code:

```
1 if(number < 0) number = -number;
2 number = number + 1;
3 else number = number * 2;
```

Was passiert beim Ausführen des Codes?

Nachdem Sie ihren Kommilitonen auf den obigen Fehler hingewiesen haben, überarbeitet er seinen Versuch. Wie sieht es mit folgender Variante aus?

```
1 if(counter > 0) counter = counter * 2;
2 if(counter < 0) counter = -counter;
3 counter = counter + 1;
```

Da müssen Sie wohl selbst ran. Erstellen Sie ein korrektes Codestück, um den Sachverhalt korrekt zu implementieren.

V7 Schleifen I

★ ☆ ☆

Schreiben Sie den folgenden Ausdruck mithilfe einer `for`-Schleife:

```
1 Robot karel = new Robot(1,1,North,1);
2 int i = 5
3 while(i < 28){
4  karel.move();
5  i = i + 1;
6 }
```

V2 Zuweisen und Kopieren

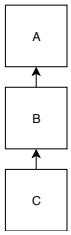
☆ ☆ ☆

Erläutern Sie in Ihren eigenen Worten den Unterschied zwischen Zuweisen und Kopieren. In welchen Fällen sind beide Aktionen synonym zu betrachten?

Wie können Sie eine Zuweisung beziehungsweise eine Kopie in Java umsetzen? Nennen Sie jeweils ein Beispiel.

V5 Typhierarchie ★★☆

Wir betrachten eine Typhierarchie (dargestellt in der Abbildung rechts) mit einer Klasse A und einem Erben B. Von B ist wiederum C abgeleitet.
Markieren Sie im folgenden Java-Code jeweils hinter `//`, ob der Compiler die Zeile akzeptiert („Okay“) oder ablehnt („Fehler“). Lösen Sie die Aufgabe zunächst durch eigene Überlegungen und überprüfen Sie erst später mittels Eclipse.




```java
class A {}
class B extends A {}
class C extends B {}

public class G {
public void m(List<B> a, List<? extends B> b,
                List<? super B> c) {

    a.add(new C()); //
    b.add(new B()); //
    c.add(new C()); //
    a.add(new B()); //
    b.add(new A()); //
    c.add(new B()); //
    a.add(new A()); //
    b.add(null);    //
    c.add(new A()); //
    b.add(new C()); //


  }

  public static void main(String args[]) {
    m(new Vector<B>(), new Vector<C>(), new Vector<A>());
  }
}
```

V1 Grundbegriffe ☆☆☆

Erklären Sie kurz in eigenen Worten die Unterschiede der folgenden Konzepte zueinander:

1. Klasse vs. Objekt
2. Objekt- vs. Klassenmethoden
3. Abstrakte Klassen vs. Interfaces
4. Überladen von Methoden vs. Überschreiben von Methoden

V3 Arrays

★ ☆ ☆

Welche Aussagen zu einem gegebenen Array `a` sind wahr?

(1) Alle Einträge des Arrays müssen vom selben Typ sein.

(2) Ein Array hat keine feste Größe und kann beliebig viele neue Einträge hinzufügen.

(3) Um die Anfangsadresse einer Komponente an Index i zu bekommen, wird i-mal die Größe einer Komponente auf die Anfangsadresse von `a` addiert.

(4) Außer den eigentlichen Komponenten des Arrays enthält das Arrayobjekt nichts weiteres.

(5) Ein Array kann nur primitive Datentypen wie zum Beispiel `int`, `char` oder `double` speichern. Somit ist insbesondere nicht möglich, Roboterobjekte in einem Array zu speichern.

Schreiben Sie die nötigen Codezeilen, um ein Array `a` der Größe 42 vom Typ `int` anzulegen. Füllen Sie danach das Array mithilfe einer Schleife, sodass an der Stelle `a[i]` der Wert `2i+1` steht. Nutzen Sie dabei zuerst eine normale, danach eine verkürzte Schleife.

V5 Schleifen

★ ☆ ☆

```
1 int x = 0;
2 while(Math.pow(x,2) <= 1000){
3 x++;  }
4 System.out.println(x);
```

(1) Welche Funktion erfüllt der oben stehende Code?

(2) Ersetzen Sie die `while`-Schleife einmal durch eine `for`- und einmal durch eine `do...while`-Schleife.

Tipps zum Beobachten und Feedback geben

1. Hat die Tutor*in den Fehler erkannt und anschließend der Student*in gesagt, wo genau der Fehler war?
2. Hat die Tutor*in am Beispiel erklärt?
3. Hat die Tutor*in die Erklärung schriftlich visualisiert oder rein mündlich kommuniziert?
4. Hat die Tutor*in erklärt, wie es korrekt gewesen wäre?
5. Hat die Tutor*in nachgefragt, ob die Erklärung verstanden wurde?
6. Hat die Tutor*in sich von der Student*in die Erklärung in deren eigenen Worten wiedergeben lassen?
7. Feedback geben: höflich bleiben, „Ich finde, dass ...“ statt „Du hast ...“ – wie bei den Feedbackregeln der allgemeinen Tutor*innenschulung!

B.3.4 Problemlösestrategien

ARBEITSAUFTRAG KREIS

Notiert auf diesem Blatt euch bekannte Strategien zum Problemlösen.

ngerman

ARBEITSAUFTRAG QUADRAT

Lest die folgende Ansammlung von Problemlösestrategien nach Kaplan (2010).

1. BREAK IT DOWN.
2. LEARN THE BASICS WELL, LOOKUP THE REST.
3. FIND AND USE EXAMPLES.
4. TEST TEST TEST (AND TEST SOME MORE)
5. FOLLOW DIRECTIONS.
6. SOLVE THE PROBLEM IN ENGLISH (OR YOUR NATIVE LANGUAGE) FIRST.
7. REPRESENT WHAT?
8. WHAT DOES IT DO?
9. MAKE SURE YOU UNDERSTAND THE PROBLEM.
10. IF YOU CAN'T SOLVE THE BIG PROBLEM, SOLVE A SMALLER PROBLEM.
11. HAVE A PLAN B.

1. BREAK IT DOWN: When you are faced with a problem that is too complex, one way to approach it is to break the problem down into smaller parts. By breaking the problem into smaller parts you may be able to more readily solve the original problem by solving its pieces.
2. LEARN THE BASICS WELL, LOOKUP THE REST: Beginning programmers often get hung up on how to do certain things. Because programs have to be very precise it is easy to make mistakes when you first start out. It is okay to look up what is needed when you become stuck.
3. FIND AND USE EXAMPLES: There are thousands upon thousands of coding samples that can be examined to better understand how a specific task is accomplished. Using examples to create programs does two things: First, you get to see how something is done, and if you pay attention by seeing how it is done you will learn how to do it. Second, you'll spend less time, at least at the start, writing your programs. Students complain about how long it takes to write a program. If you can find and use examples and you can learn from the examples you might be able to learn faster then if you tried to write a program from scratch.
4. TEST TEST TEST (AND TEST SOME MORE): Testing is part of programming. You test to get a program to work and you test to make sure the program works correctly. The better you test, the better your program will be.
5. FOLLOW DIRECTIONS: Make sure that you are doing what the directions say. The directions tell you what you can and cannot do. Don't overcomplicate or oversimplify what you have to do. Sometimes the directions will even give you part of the solution – so read carefully.
6. SOLVE THE PROBLEM IN ENGLISH (OR YOUR NATIVE LANGUAGE) FIRST: Learning a new programming language is similar to learning a new foreign language. You just don't jump right into writing the steps in the new language. That is a direct way to failure. Solve the problem in your native language. Write each of the steps in English or the language that you are most comfortable with. Translate this version into the programming language you are using. That way you will have the problem solved and you can focus on how to express your solution in the programming language.

7. REPRESENT WHAT?: An important part of solving any programming problem is to figure out how the problem will be represented in data. The variables that are needed and other structures like arrays need to be defined. This is a key aspect of creating a program to solve any programming problem.
8. WHAT DOES IT DO?: Every computer program that was ever written does something. That is the nature of programs. They make computers do things that are desired by the programmer. Now sometimes the author of a program will get it wrong. The computer will do something that was not wanted. There can be lots of reasons for this and one of them is that the programmer really didn't understand what the computer was supposed to do for solving the problem. Make sure you understand what it is that you want the computer to do.
9. MAKE SURE YOU UNDERSTAND THE PROBLEM: If you don't understand what the problem is, how can you come up with a solution? You can't. Besides following directions, understanding what it is you must do is extremely important. If you don't understand what to do, don't even think about starting to work on the solution to the problem. Do whatever it takes to understand the problem you must solve.
10. IF YOU CAN'T SOLVE THE BIG PROBLEM, SOLVE A SMALLER PROBLEM.: This may seem like the first statement (Break It Down) but it isn't. This has to do with complexity of the problem you are to solve. A problem might be so complex that you have no idea how to approach it let alone solve it. So, instead of bashing one's head against a wall (a tried and true method for solving problems), formulate a simpler problem related to the original problem. The simpler problem will be solvable (we hope). Use the learning that came from solving the simpler problem to solve the more complex problem.
11. HAVE A PLAN B: No matter how well we plan and no matter how completely we plan, there is always something that happens to get in the way of our plan. Therefore, if you have a way to solve the problem and you find you have travelled down a path that doesn't have a good or desired ending, you need to have an alternative. Plan B is that alternative. It may be an alternate way of solving the problem. It may be a different breakdown of the problem. It may be a simpler approach. Whatever it is, have a plan B just in case.

B.3.5 Test zur Rekursion

TEST ZUR REKURSION

Der Wissenschaftler Hank Kahney brauchte kürzlich ein Programm, welches die folgende Veränderung ausführt: Wenn jemand `X` Fieber hat – `HAS FLU` –, dann bekommt jeder, der von X geküsst wird – `X KISSES ?` –, ebenfalls Fieber. Jeder, der infiziert ist, verbreitet die Infektion zu der Person, den er oder sie küsst, und so weiter.
Dabei sind die Datenbanken in Abschnitt 2 gegeben. Hank Kahney benötigt ein Programm, welches die Datenbank A in die Datenbank B umwandelt.

- `NOTE` erlaubt das Hinzufügen zur Datenbank.
- `CHECK` überprüft die Präsenz des Tripels – `X KISSES ?` – in der Datenbank.

  `?` wird hier als *Wildcard* bzw. als *Platzhalter* verstanden. Im Falle der *Präsenz* – `Present` – wird der Wert dem Platzhalter zugewiesen. Auf diesen kann mittels `*` zugegriffen werden. Bei *Abwesenheit* – `Absent` – wird an dieser Stelle im Programm fortgefahren.
- `CONTINUE` macht mit dem nächstem Hauptschritt weiter, sprich die nächste Zeile, die einfach eingerückt ist.
- `EXIT` beendet die Prozedur und geht zurück zu der vorherigen aufrufenden Prozedur.
- `DONE` ist das Schlüsselwort, um die Prozedur, die mit `TO` begonnen wurde, zu beenden.

1: Programmlösungen

Code - Lösung 1

```
TO INFECT X
        CHECK X KISSES ?
                If Present: NOTE * HAS FLU; EXIT
                If Absent: EXIT
        DONE
```

Code - Lösung 2

```
TO INFECT X
        NOTE X HAS FLU
        CHECK X KISSES ?
                If Present: INFECT * ; EXIT
                If Absent: EXIT
        DONE
```

Code - Lösung 3

```
TO INFECT X
        CHECK X KISSES ?
                If Present: INFECT * ; CONTINUE
                If Absent: CONTINUE
        NOTE X HAS FLU
        DONE
```

2: Datenbanken

Datenbank A

```
JOHN
|
|...KISSES MARY

MARY
|
|...KISSES TIM

TIM
|
|...KISSES JOAN

JOAN
```

Datenbank B

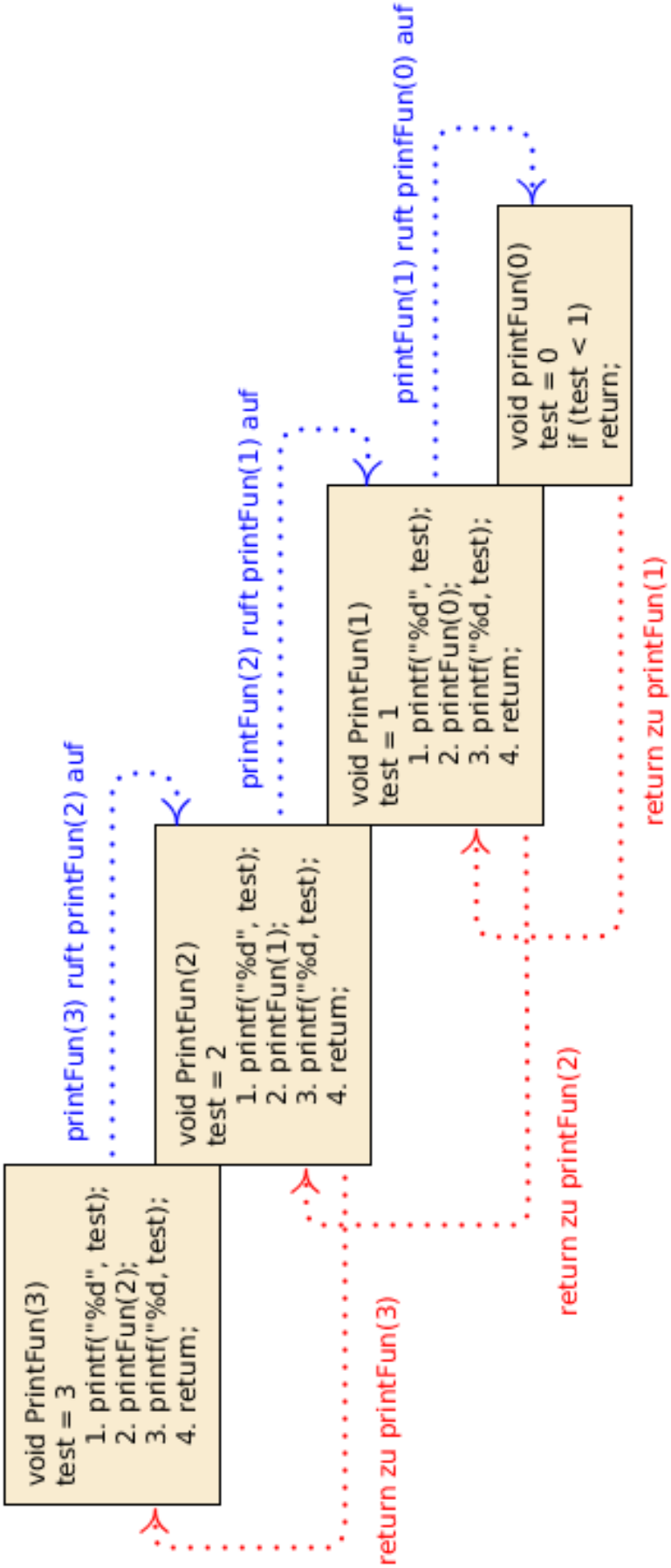
```
JOHN
|...KISSES MARY
|
|...HAS FLU

MARY
|
|...KISSES TIM
|
|...HAS FLU

TIM
|
|...KISSES JOAN
|
|...HAS FLU

JOAN
|
|...HAS FLU
```

3: Aufgabenstellung

Entscheide bei jedem der drei zuvor vorgestellten Programme, ob es *richtig* – verändert die Datenbank wie gewünscht – oder *falsch* – verändert die Datenbank nicht wie gewünscht – ist.

Lösung 1:	☐ richtig	☐ falsch
Lösung 2:	☐ richtig	☐ falsch
Lösung 3:	☐ richtig	☐ falsch

Begründe deine Entscheidung für jedes Programm einzeln in Stichpunkten auf einem extra Blatt.


void PrintFun(3)
test = 3
1. printf("%d", test);
2. printFun(2);
3. printf("%d, test);
4. return;
printFun(3) ruft printFun(2) auf
void PrintFun(2)
test = 2
1. printf("%d", test);
2. printFun(1);
3. printf("%d, test);
4. return;
printFun(2) ruft printFun(1) auf
void PrintFun(1)
test = 1
1. printf("%d", test);
2. printFun(0);
3. printf("%d, test);
4. return;
printFun(1) ruft prinfFun(0) auf
void printFun(0)
test = 0
if (test < 1)
return;
return zu printFun(3)
return zu printFun(2)
return zu printFun(1)

Literaturverzeichnis

[Abeck 2005] ABECK, Sebastian: *KURSBUCH INFORMATIK I - Formale Grundlagen der Informatik und Programmierkonzepte am Beispiel von Java*. Karlsruhe : Universitätsverlag Karlsruhe, 2005

[Abelson u. a. 2001] ABELSON, Harold ; SUSSMAN, Gerald J. ; SUSSMANN, Julie: *Struktur und Interpretation von Computerprogrammen - Eine Informatik-Einführung*. 4., durchges. Aufl. Wiesbaden : Springer Berlin Heidelberg, 2001 (Springer-Lehrbuch)

[Anthony u. a. 2003] ANTHONY, Robins ; ROUNTREE, Janet ; ROUNTREE, Nathan: Learning and Teaching Programming: A Review and Discussion. In: *Computer Science Education* 13 (2003), Nr. 2, S. 137–172

[Barzel 2011] BARZEL, Bärbel: *Mathematik unterrichten: planen, durchführen, reflektieren*. Berlin : Cornelsen Scriptor, 2011

[Bayman und Meyer 1983] BAYMAN, Piraye ; MEYER, Richard E.: A diagnosis of beginning programmers' misconceptions of BASIC programming statements. In: *Communication of the ACM* 26 (1983), September, Nr. 9, S. 677–679

[Bergin] BERGIN, Joseph: *Some Pattern Papers by Joseph Bergin & Others*. http://csis.pace.edu/~bergin/patterns/. – Abgerufen am 01.12.2018

[du Boulay u. a. 2009] BOULAY, Benedict du ; TIM, O'Shea ; MONK, John: The black box inside the glass box: presenting computer concepts to novices. In: SOLOWAY, Elliot (Hrsg.) ; SPOHRER, James C. (Hrsg.): *Studying the Novice Programmer*. New York, London : Psychology Press, 2009, S. 431–446

[Bruder und Collet 2011] BRUDER, Regina ; COLLET, Christina: *Problemlösen lernen im Mathematikunterricht*. Berlin : Cornelsen, 2011

[Church 1936] CHURCH, Alonzo: An unsolvable problem of elementary number theory. In: *American journal of mathematics* 58 (1936), Nr. 2, S. 345–363

[Dreyfus u. a. 1987] DREYFUS, Hubert L. ; DREYFUS, Stuart E. ; ZADEH, Lotfi A.: Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer. In: *IEEE Expert* 2 (1987), June, Nr. 2, S. 110–111. – ISSN 0885-9000

[Du Boulay 2009] DU BOULAY, Benedict: Some difficulties of Learning to Program. In: SOLOWAY, Elliot (Hrsg.) ; SPOHRER, James C. (Hrsg.): *Studying the Novice Programmer*. New York, London : Psychology Press, 2009, S. 283–300

[Eckerdal und Thuné 2005] ECKERDAL, Anna ; THUNÉ, Michael: Novice Java Programmers' Conceptions of "Object" and "Class", and Variation Theory. In: *SIGCSE Bull.* 37 (2005), Juni, Nr. 3, S. 89–93

[Eisenstadt 1983] EISENSTADT, Marc: A User-friendly Software Environment for the Novice Programmer. In: *Commun. ACM* 26 (1983), Dezember, Nr. 12, S. 1058–1064. – URL http://doi.acm.org/10.1145/358476.358500. – ISSN 0001-0782

[Ernst u. a. 2016] ERNST, Hartmut ; SCHMIDT, Jochen ; BENEKEN, Gerd: *Grundkurs Informatik - Grundlagen und Konzepte für die erfolgreiche IT-Praxis - Eine umfassende, praxisorientierte Einführung*. Berlin Heidelberg New York : Springer-Verlag, 2016

[Evans und Flanagan 2014] EVANS, Benjamin J. ; FLANAGAN, David: *Java in a Nutshell - A Desktop Quick Reference*. Sebastopol : Ö'Reilly Media, Inc.", 2014

[Fischer und Hofer 2011] FISCHER, Peter ; HOFER, Peter: *Lexikon der Informatik*. 15. Berlin, Heidelberg : Springer-Verlag, 2011

[Gailer 2018a] GAILER, Heinrich: *Java Tutorial interaktiv - Programmieren lernen mit Java*. http://www.gailer-net.de/tutorials/java/index.html. 2018. – Abgerufen am 01.12.2018

[Gailer 2018b] GAILER, Heinrich: *Java Tutorial interaktiv - Programmieren lernen mit Java, Datenausschuss*. http://www.gailer-net.de/tutorials/java/Notes/chap29/ch29_7.html. 2018. – Abgerufen am 01.12.2018

[Garner u. a. 2005] GARNER, Sandy ; HADEN, Patricia ; ROBINS, Anthony: My Program is Correct but It Doesn'T Run: A Preliminary Investigation of Novice Programmers' Problems. In: *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2005 (ACE '05), S. 173–180

[GeeksforGeeks 2018] GEEKSFORGEEKS: *Recursion*. https://www.geeksforgeeks.org/recursion/. 2018. – Abgerufen am 01.12.2018

[Götschi u. a. 2003] GÖTSCHI, Tina ; SANDERS, Ian ; GALPIN, Vashti: *Mental models of recursion*. ACM, 2003

[Gretsch und Holzäpfel 2016] GRETSCH, Petra ; HOLZÄPFEL, Lars: *Lernen mit Visualisierungen - Erkenntnisse aus der Forschung und deren Implikationen für die Fachdidaktik*. Münster : Waxmann Verlag, 2016

[Günther 2019] GÜNTHER, Manfred: *Pädagogisches Rollenspiel : Wissensbaustein und Leitfaden für die psychosoziale Praxis*. Wiesbaden : Springer Fachmedien, 2019 (essentials)

[Hattie und Wolkenschläger 2014] HATTIE, John ; WOLKENSCHLÄGER, Mareike: A conceptualization of feedback. In: DITTON, Hartmut (Hrsg.) ; MÜLLER, Andreas (Hrsg.): *Feedback und Rückmeldungen - theoretische Grundlagen, empirische Befunde, praktische Anwendungsfehlder*. Münster, New York : Waxmann Verlag, 2014, S. 135–150

[Heinrichs 2015] HEINRICHS, Hannah: *Diagnostische Kompetenz von Mathematik-Lehramtsstudierenden - Messung und Förderung*. Berlin Heidelberg New York : Springer-Verlag, 2015

[Heinrichs und Kaiser 2018] HEINRICHS, Hannah ; KAISER, Gabriele: Diagnostic Competence for Dealing with Students' Errors: Fostering Diagnostic Competence in Error Situations. In: LEUDERS, Timo (Hrsg.) ; PHILIPP, Kathleen (Hrsg.) ; LEUDERS, Juliane (Hrsg.): *Diagnostic Competence of Mathematics Teachers - Unpacking a Complex Construct in Teacher Education and Teacher Practice*. Berlin, Heidelberg : Springer, 2018, S. 79–94

[Hessische Lehrkräfteakademie 2015] HESSISCHE LEHRKRÄFTEAKADEMIE: *Begleitmaterial KCGO Informatik – Literaturhinweise*. https://kultusministerium.hessen.de/sites/default/

files/media/literaturhinweise_kcgo_informatik.pdf. 2015. – Abgerufen am 01.12.2018

[Heublein u. a. 2017] HEUBLEIN, Ulrich ; EBERT, Julia ; HUTZSCH, Christopher ; ISLEIB, Sören ; KÖNIG, Richard ; RICHTER, Johanna ; WOISCH, Andreas: Zwischen Studienerwartungen und Studienwirklichkeit - Ursachen des Studienabbruchs, beruflicher Verbleib der Studienabbrecherinnen und Studienabbrecher und Entwicklung der Studienabbruchquote an deutschen Hochschulen / Deutsches Zentrum für Hochschul- und Wissenschaftsforschung GmbH. Juni 2017. – Forschungsbericht

[Hill u. a. 2008] HILL, Heather C. ; BALL, Deborah L. ; SCHILLING, Stephen G.: Unpacking Pedagogical Content Knowledge: Conceptualizing and Measuring Teachers' Topic-Specific Knowledge of Students. In: *Journal for Research in Mathematics Education* 39 (2008), Nr. 4, S. 372–400

[HKM 2012] HKM: *Individuelle Förderung - individualisiertes Lernen*. https://kultusministerium.hessen.de/sites/default/files/media/hkm/individuelle_foerderung_lernen.pdf. 2012. – Abgerufen am 01.12.2018

[HKM 2016] HKM: *Kerncurriculum – Gymnasiale Oberstufe: Informatik*. https://kultusministerium.hessen.de/sites/default/files/media/kcgo-in.pdf. 2016. – Abgerufen am 01.12.2018

[Holland u. a. 1997] HOLLAND, Simon ; GRIFFITHS, Robert ; WOODMAN, Mark: Avoiding Object Misconceptions. In: *SIGCSE Bull.* 29 (1997), März, Nr. 1, S. 131–134. – ISSN 0097-8418

[Hölzl u. a. 2013] HÖLZL, Matthias ; RAED, Allaithy ; WIRSING, Martin: *Java kompakt - Eine Einführung in die Software-Entwicklung mit Java*. Berlin Heidelberg New York : Springer-Verlag, 2013

[Kahney 2009] KAHNEY, Hank: What Do Novice Programmers Know About Recursion? In: SOLOWAY, Elliot (Hrsg.) ; SPOHRER, James C. (Hrsg.): *Studying the Novice Programmer*. New York, London : Psychology Press, 2009, S. 209–228

[Kaplan 2012] KAPLAN, M.: Teaching Novice Programmers Programming Wisdom. (2012), 05

[Kjell und Gailer 2004] KJELL, Bradley ; GAILER, Heinrich: Enhancing usability by international students of a distance learning Web site. In: *2004 International Symposium on Technology and Society (IEEE Cat. No.04CH37548)*, June 2004, S. 104–106

[Klafki 1986] KLAFKI, Wolfgang: Das pädagogische Verhältnis und die Gruppenbeziehungen im Erziehungsprozeß. In: KLAFKI, Wolfgang (Hrsg.) ; RÜCKRIEM, Georg M. (Hrsg.) ; WOLF, Willi (Hrsg.) ; FREUDENSTEIN, Reinhold (Hrsg.) ; BECKMANN, Hans-Karl (Hrsg.) ; LINGELBACH, Karl-Christoph (Hrsg.) ; IBEN, Gerd (Hrsg.) ; DIEDERICH, Jürgen (Hrsg.): *Erziehungswissenschaft 1 - Eine Einführung*. Frankfurt am Main : Fischer Taschenbuch Verlag, 1986, S. 53–91

[Krasner u. a. 1988] KRASNER, Glenn E. ; POPE, Stephen T. u. a.: A description of the model-view-controller user interface paradigm in the smalltalk-80 system. In: *Journal of object oriented programming* 1 (1988), Nr. 3, S. 26–49

[Leuders u. a. 2018] LEUDERS, Timo ; DÖRFLER, Tobias ; LEUDERS, Juliane ; PHILIPP, Kathleen: Diagnostic Competence of Mathematics Teachers: Unpacking a Complex Construct. In: LEUDERS, Timo (Hrsg.) ; PHILIPP, Kathleen (Hrsg.) ; LEUDERS, Juliane (Hrsg.): *Diagnostic Competence of Mathematics Teachers - Unpacking a Complex Construct in Teacher Education and Teacher Practice*. Berlin, Heidelberg : Springer, 2018, S. 3–32

[Levy 2001] LEVY, Dalit: Insights and Conflicts in Discussing Recursion: A Case Study. December 2001 (2001), 08, S. 305–322

[Levy und Lapidot 2000] LEVY, Dalit ; LAPIDOT, Tami: Recursively Speaking: Analyzing Students' Discourse of Recursive Phenomena. In: *SIGCSE Bull.* 32 (2000), März, Nr. 1, S. 315–319

[Lewalter 1997] LEWALTER, Doris: *Lernen mit Bildern und Animationen*. Münster : Waxmann Verlag, 1997

[Magenheim und Schulte 2005] MAGENHEIM, Johannes ; SCHULTE, Carsten: Erwartungen und Wahlverhalten von Schülerinnen und Schülern gegenüber dem Schulfach Informatik – Ergebnisse einer Umfrage, Gesellschaft für Informatik e.V. (GI), 2005, S. 111–122

[Meyer 1996] MEYER, Bertrand: The many faces of inheritance: a taxonomy of taxonomy. 29 (1996), 06, S. 105 – 108

[Mulholland und Eisenstadt 1998] MULHOLLAND, Paul ; EISENSTADT, Marc: *Using software to teach computer programming: Past, present and future*. 1998

[Müller 2017] MÜLLER, Dorothee: Berufswahl Informatiklehrkraft, Gesellschaft für Informatik e.V. (GI), 2017, S. 127–136

[Painter 2006] PAINTER, Robert R.: *Software Plans: Multi-Dimensional Fine-Grained Separation of Concerns*. Williamsburg, The College of William & Mary in Virginia, Dissertation, 2006

[Paradies u. a. 2011] PARADIES, Liane ; LINSER, Hans J. ; GREVING, Johannes: *Diagnostizieren, Fordern und Fördern* -. 3. Berlin : Cornelsen, 2011

[Perkins u. a. 2009] PERKINS, David N. ; HANCOCK, Chris ; HOBBS, Renee ; MARTIN, Fay ; SIMMONS, Rebecca: Conditions of Learning in Novice Programmers. In: SOLOWAY, Elliot (Hrsg.) ; SPOHRER, James C. (Hrsg.): *Studying the Novice Programmer*. New York, London : Psychology Press, 2009, S. 261–280

[Peterßen 2009] PETERSSEN, Wilhelm H.: *Kleines Methoden-Lexikon*. 3. München : Oldenbourg, 2009

[Pillay 2010] PILLAY, Nelishia: Teaching Design Patterns, 2010

[Prediger 2010] PREDIGER, Susanne: How to develop mathematics-for-teaching and for understanding: the case of meanings of the equal sign. In: *Journal of Mathematics Teacher Education* 13 (2010), Feb, Nr. 1, S. 73–93

[Proulx 2000] PROULX, Viera K.: Programming Patterns and Design Patterns in the Introductory Computer Science Course. In: *SIGCSE Bull.* 32 (2000), März, Nr. 1, S. 80–84. – ISSN 0097-8418

[Qian und Lehman 2017] QIAN, Yizhou ; LEHMAN, James: Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. In: *ACM Trans. Comput. Educ.* 18 (2017), Oktober, Nr. 1, S. 1:1–1:24. – ISSN 1946-6226

[Reed 1998] REED, David: Incorporating Problem-solving Patterns in CS1. In: *SIGCSE Bull.* 30 (1998), März, Nr. 1, S. 6–9

[Robins u. a. 2001] ROBINS, Anthony ; ROUNTREE, Nathan ; ROUNTREE, Janet: My program is correct but it doesn't run: A review of novice programming and a study of an introduc- tory programming paper / Department of Computer Science, University of Otago. June 2001. – Forschungsbericht

[Schrader und Helmke 1987] SCHRADER, Friedrich-Wilhelm ; HELMKE, Andreas: Diagnostische Kompetenz von Lehrern: Komponenten und Wirkungen. In: *Empirische Pädagogik* 1 (1987), Nr. 1, S. 27–52

[Schubert und Schwill 2011] SCHUBERT, Sigrid ; SCHWILL, Andreas: *Didaktik der Informatik* -. 2. Berlin Heidelberg New York : Springer-Verlag, 2011

[Schumann und Müller 2013] SCHUMANN, Heidrun ; MÜLLER, Wolfgang: *Visualisierung - Grundlagen und allgemeine Methoden*. Berlin Heidelberg New York : Springer-Verlag, 2013

[Rito da Silva 1999] SILVA, António Manuel F. Rito da: *Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks*. Lissabon, Portugal, Universidade Técnica de Lisboa, Dissertation, 1999

[Simon 2013] SIMON, Beth: Soloway's Rainfall Problem Has Become Harder, IEEE Computer Society, March 2013, S. 130–135

[Soloway und Spohrer 2009] SOLOWAY, Elliot ; SPOHRER, James C.: *Studying the Novice Programmer* -. London : Psychology Press, 2009

[Sorva u. a. 2013] SORVA, Juha ; KARAVIRTA, Ville ; MALMI, Lauri: A Review of Generic Program Visualization Systems for Introductory Programming Education. In: *Trans. Comput. Educ.* 13 (2013), November, Nr. 4, S. 15:1–15:64

[Thomasson u. a. 2006] THOMASSON, Benjy ; RATCLIFFE, Mark ; THOMAS, Lynda: Identifying Novice Difficulties in Object Oriented Design. In: *SIGCSE Bull.* 38 (2006), Juni, Nr. 3, S. 28–32. – ISSN 0097-8418

[Thytos 2014] THYTOS: *Was ist die for-Schleife? | Programmieren lernen mit JavaScript #19*. urlhttps://www.youtube.com/watch?v=XVsrs3dSB5w veröffentlicht am 02.04.2014. 2014. – Abgerufen am 01.12.2018

[TU Darmstadt 2015] TU DARMSTADT: *Modulhandbuch - B.Sc./M.Sc. Informatik*. https://www.informatik.tu-darmstadt.de/media/informatik/fb20_studium/formulare_und_dokumente/ordnungen/ordnungen_bsc/pruefungsordnung_2015/Modulhandbuch.pdf. 2015. – Abgerufen am 01.12.2018

[Ulrich 2016] ULRICH, Immanuel: *Gute Lehre in der Hochschule - Praxistipps zur Planung und Gestaltung von Lehrveranstaltungen*. Berlin Heidelberg New York : Springer-Verlag, 2016

[Vlissides u. a. 1995] VLISSIDES, John ; HELM, Richard ; JOHNSON, Ralph ; GAMMA, Erich: Design patterns: Elements of reusable object-oriented software. In: *Reading: Addison-Wesley* 49 (1995), Nr. 120, S. 11

[Wagenknecht 2016] WAGENKNECHT, Christian: *Programmierparadigmen - Eine Einführung auf der Grundlage von Racket*. Berlin Heidelberg New York : Springer-Verlag, 2016

[Wahl 2013] WAHL, Diethelm: *Lernumgebungen erfolgreich gestalten - Vom trägen Wissen zum kompetenten Handeln*. 3. Bad Heilbrunn : Julius Klinkhardt, 2013

[Walker u. a. 2007] WALKER, James E. ; SHEA, Thomas M. ; BAUER, Anne M.: *Behavior management: A practical approach for educators*. 7. Upper Saddle River New York : Pearson Merrill Prentice Hall, 2007

[Weigend 2005] WEIGEND, Michael: *The Python Visual Sandbox: visualizing the invisible - explicating the implicit*. http://www.python-visual-sandbox.de/. 2005. – Abgerufen am 01.12.2018

[Weigend 2007] WEIGEND, Michael: *Intuitive Modelle der Informatik*. Potsdam : Universitätsverlag Potsdam, 2007

[Westermann u. a. 2012] WESTERMANN, Katharina ; RUMMEL, Nikol ; HOLZÄPFEL, Lars: Präkonzepte aufgreifen fördert den Verständniserwerb. In: *Beiträge zum Mathematikunterricht 2012 - Vorträge auf der 46. Tagung für Didaktik der Mathematik vom 05.03.2012 bis 09.03.2012 in Weingarten*, 2012, S. 937–940

[Wilson 2015] WILSON, Greg: *Teaching For Loops*. https://software-carpentry.org/blog/2015/12/teaching-for-loops.html. 2015. – Abgerufen am 01.12.2018

[Winslow 1996] WINSLOW, Leon E.: Programming Pedagogy - A Psychological Overview. In: *SIGCSE Bull.* 28 (1996), September, Nr. 3, S. 17–22. – ISSN 0097-8418

[Yoder und Schrag 1978] YODER, Cornelia M. ; SCHRAG, Marilyn L.: Nassi-Shneiderman charts an alternative to flowcharts for design. In: *ACM SIGMETRICS Performance Evaluation Review* 7 (1978), Nr. 3-4, S. 79–86

Eidesstattliche Erklärung

Hiermit versichere ich, Svana ESCHE, dass ich die Arbeit mit dem Titel „Erklären können – Fachdidaktische Schulung von Tutor*innen für Programmieranfänger*innen“ selbstständig verfasst, keine anderen als die angegebenen Hilfsmittel verwendet und sämtliche Stellen, die anderen Druckwerken oder digitalisierten Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem einzelnen Fall unter Angabe der Quelle kenntlich gemacht habe. Die Versicherung gilt auch für verwendete Zeichnungen, Skizzen, Notenbeispiele sowie bildliche und sonstige Darstellungen.

Unterschrift:

Datum: