# Webbasierte Ad-hoc-Programmieraufgaben zur Vermittlung von grundlegenden Konzepten der Programmierung in Vorlesungen

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von
Michael Wolfgang Ebert
aus Coburg

1. Gutachter: Prof. Dr. Bernhard Westfechtel
2. Gutachter: Prof. Dr. Wolfram Haupt
3. Gutachter: Prof. Dr. Martin Hennecke

Tag der Einreichung: 15. März 2018
Tag des Kolloquiums: 24. Oktober 2018

# Danksagung

Ein herzliches Dankeschön möchte ich an meine Betreuer Prof. Dr. Wolfram Haupt, Prof. Dr. Dieter Landes sowie Prof. Dr. Bernhard Westfechtel richten. Sie standen immer mit wertvollem Rat zur Verfügung und unterstützten mich stets durch Diskussionen und die Möglichkeit Lehre selbst zu leben. Weiterhin geht mein Dank insbesondere an Markus Ring und an meine Kolleginnen und Kollegen der Hochschule Coburg sowie des Projekts EVELIN.

Ein besonderer Dank gilt meiner Frau Evelyn, welche mich fortwährend bei allen Herausforderungen unterstützt hat.

**Abstract**

Digitalization has become a pervasive phenomenon in media, politics and business. Using cloud technology or Industrial Internet of the Things (IIoT), the shift to new ways of creating value add in industry and opening up new business cases has become essential (for businesses) to remain competitive. In order to address the demand for new skills and to apply computational concepts in a new context, educational institutions are challenged to teach the basics of programming and computer science. In particular, educators need to shift the teaching focus from highly specific programming languages, to a comprehensive understanding of fundamentals in computer science. However, it seems that students struggle to acquire these skills. Programming is not just an ability, it is a manifold process that requires students to cope with programming languages, problem solving, algorithms and limitations of computing. Lecture design needs to address all these elements in a situational and contextual way so that the students' learning process is successful. The objective of this dissertation, which is written within the research project EVELIN (Experimental Improvement of the Learning of Software Engineering), is to develop a didactical approach for integrating ad-hoc programming exercises for practical teaching of programming concepts. This approach is based on the cognitive apprenticeship, teaching and learning theories and active learning. The above approach has been adapted for implementation in programming lectures where one-to-one supervision is not possible. The intention of the concept is to impart canonical solutions for similar problems which are introduced and practiced in a problem- and practice-oriented manner. Additionally, those problems are successively increased in complexity and difficulty to show the application of the solutions on a more abstract level. First, the term programming and the related cognitive intricacies are discussed for designing the concept, and a method is devised for measuring and comparing the programming ability based on existing approaches. The efficient integration of the concept in lectures requires developing interactive teaching materials to enable students to deal with the new topics during lecture and at home. For this purpose, existing approaches and technologies must be adapted and refined for creating interactive learning content. Therefore, a web-based tool has been implemented for creating specific interactive learning materials and optimized for lecture hall use. This approach was used and evaluated in a course for introducing programming to electrical engineering students at the Coburg Applied University of Science. The subsequent evaluation of the concept, including exams results, indicates a correlation between student activities (ad-hoc exercises) and their ability to use programming concepts as canonical solutions.

**Zusammenfassung**

Die in der Öffentlichkeit und Politik proklamierte Digitalisierung zeigt den unter anderem in der Industrie aktuellen Trend zur Schöpfung von Mehrwerten durch Algorithmen und Software - insbesondere in der *Cloud* oder im Bereich des industriellen Internet der Dinge (IIoT). Gleichwohl sind die Grundlagen der Programmierung und die Informatik konstitutiv für eben diesen Trend und den damit assoziierten Technologien. Besonders in der Hochschullehre sind gefestigte Grundlagen wichtig, um die darauf aufbauenden Konzepte zu verstehen und anwenden zu können. Doch scheint das Erlernen dieser Fähigkeiten und Inhalte Studierende mit Problemen zu konfrontieren, die sich in den Noten und in mangelnder Programmierfähigkeit widerspiegeln. Programmieren ist keine reine Fertigkeit, sondern ein vielseitiger Prozess, in dessen Rahmen sich Studierende mit einer Programmiersprache, Problemlösungsstrategien, Algorithmisierung und den Grenzen der Berechenbarkeit von Computern auseinandersetzen müssen. Für die Konzeption einer Lehrveranstaltung müssen all diese Elemente situativ und kontextbezogen adressiert werden, um den Lernprozess von Studierenden erfolgreich begleiten zu können. Diese Dissertation, welche im Rahmen des Forschungsprojekts „EVELIN (Experimentelle Verbesserung der Lehre des Software Engineerings)“ erstellt wird, befasst sich mit der Erarbeitung eines didaktischen Konzepts für eine Integration von ad hoc Programmieraufgaben zur angewandten Vermittlung von Programmierkonzepten. Das Konzept lehnt sich an das *Cognitive Apprenticeship* und an lehr- und lerntheoretische Konzepte des *Aktiven Lernens* an, wurde jedoch sehr an Programmiervorlesungen adaptiert. Ziel des Konzepts ist die Vermittlung von Lösungswegen für ähnlich gelagerte Problemstellungen, welche problem- und praxisorientiert eingeführt und eingeübt und dabei sukzessive in Komplexität und Schwierigkeit gesteigert werden. Für diese Adaption wird zuerst der Begriff Programmieren und die damit verbundenen kognitiven Schwierigkeiten diskutiert, um anschließend ein Verfahren zur Erfassung und Bewertung der Programmierfähigkeit auf Basis vorhandener Ansätze zu entwickeln. Eine effiziente Integration des Konzepts in eine Lehrveranstaltung erfordert die Entwicklung interaktiver Lehr- und Lernmaterialien, welche Studierenden die Möglichkeit bieten, sich mit Inhalten in der Vorlesung und von zuhause aus zu beschäftigen. Dazu werden existierende Ansätze und Technologien auf das entwickelte Konzept angepasst und zu einem neuen Werkzeug zur Erstellung von interaktiven Inhalten weiterentwickelt. Im Rahmen einer Lehrveranstaltung zur Einführung in die Programmierung für Studierende der Elektrotechnik wurde das Konzept eingesetzt und evaluiert. Die anschließende Evaluation des Konzepts unter Einbezug von Klausuraufgaben deutet auf einen Zusammenhang zwischen den studentischen Aktivitäten (Bearbeitung der Ad-Hoc-Aufgaben) und der Fähigkeit der Anwendung von Programmierkonzepten als Lösungsbausteine hin.

# Inhaltsverzeichnis

**Abbildungsverzeichnis** v

**Tabellenverzeichnis** viii

**Quellcodeverzeichnis** xi

**I Didaktische Vorüberlegungen: Programmieren in der Lehre** 1

**1 Einleitung** 2
- 1.1 Relevanz der Informatik, Praxisbezüge und die Lehre . . . 2
  - 1.1.1 Praxisbezüge . . . 3
  - 1.1.2 Informatische Inhalte . . . 5
  - 1.1.3 Auswirkungen auf die Lehre . . . 7
- 1.2 Ziele und Lösungsansätze . . . 9
- 1.3 Aufbau der Arbeit . . . 10

**2 Problemstellung** 11
- 2.1 Ziele der Programmierausbildung in der Elektrotechnik . . . 11
- 2.2 Status quo der Lehre - Warum ist Programmieren schwer? . . . 17
- 2.3 Forschungsmethodische Vorgehensweise . . . 20
- 2.4 Forschungsfragen . . . 25
  - 2.4.1 Fragen . . . 25
  - 2.4.2 Hypothesen . . . 26

**3 Programmieren und Programmierfähigkeit** 28
- 3.1 Was bedeutet Programmieren? . . . 28
- 3.2 Kognitive Herausforderungen . . . 33
  - 3.2.1 Probleme und Aufgaben . . . 33
  - 3.2.2 Problemlösen . . . 35
  - 3.2.3 Computational Thinking . . . 39

3.2.4 Zwischenfazit - Worauf geachtet werden sollte . . . . . 40
3.3 Ist Programmierfähigkeit messbar? . . . . . 42
3.3.1 Programmierfähigkeit . . . . . 43
3.3.2 Vorhandene Ansätze . . . . . 44
3.3.3 Messung mittels Indikatoren . . . . . 49
3.4 Lehr- und Lerntheoretische Überlegungen . . . . . 56
3.4.1 Konstruktivistische Theorie . . . . . 56
3.4.2 Cognitive Apprenticeship . . . . . 59
3.4.3 Überforderungen: Cognitive Load Theory . . . . . 61
3.5 Schlussfolgerungen für die Lehre . . . . . 63

**II Hauptteil: Interaktive Vorlesungen 67**

**4 Aktives Lernen 68**
4.1 Charakteristika und Merkmale . . . . . 68
4.2 Strategien und Formen des Aktiven Lernens . . . . . 72
4.2.1 Formen . . . . . 72
4.2.2 Wirksamkeit . . . . . 78
4.3 Schlussfolgerungen . . . . . 79
4.4 Aktives Lernen in der Programmierausbildung . . . . . 80
4.4.1 Allgemeine Ansätze . . . . . 80
4.4.2 Kooperatives und kollaboratives Lernen . . . . . 85
4.4.3 Zusammenfassung . . . . . 90

**5 Integration aktiver Programmierübungen in Lehrveranstaltungen 93**
5.1 Voraussetzungen & Zielsetzungen . . . . . 93
5.1.1 Didaktische Synthese . . . . . 93
5.1.2 Lehrveranstaltungsübergreifende Ebene . . . . . 97
5.1.3 Ebene der Lehrveranstaltung . . . . . 98
5.1.4 Methodische Ebene . . . . . 102
5.2 Interaktive Aufgaben für Programmierkonzepte . . . . . 102
5.2.1 Problemorientierung und Problemkonfrontation . . . . . 103
5.2.2 Konzepteinführung . . . . . 104
5.2.3 Aufgaben innerhalb der Vorlesung . . . . . 105
5.2.4 Coaching und Feedback in den Übungen . . . . . 109
5.2.5 Anmerkungen . . . . . 109
5.2.6 Beispiel . . . . . 110
5.2.7 Fazit . . . . . 113
5.3 Auf dem Weg zur Interaktivität . . . . . 114
5.3.1 Interaktivität . . . . . 115

5.3.2 Effiziente Umsetzung . . . 116
5.3.3 Unterrichtssteuerung, Wiederholung und explorative Konzepte 117
5.3.4 Zusammenfassung . . . 118

**6 Plattform für interaktive Vorlesungen 120**
6.1 Vorhandene Ansätze . . . 120
6.1.1 Interaktive Materialien . . . 121
6.1.2 Code-Editoren . . . 133
6.1.3 Fazit zu den Ansätzen . . . 138
6.2 Ziele . . . 139
6.3 Technische Umsetzung . . . 140
6.3.1 Übersicht . . . 140
6.3.2 Sichere Ausführung von fremden Code . . . 141
6.3.3 Web-basierte Entwicklungsumgebung . . . 147
6.3.4 Interaktive Komponenten . . . 155
6.4 Interaktive Lernmaterialien . . . 161
6.4.1 Dokumente . . . 161
6.4.2 Präsentationen . . . 170
6.5 Indikatoren zur schnellen Einschätzung der Studierenden . . . 172
6.5.1 Analyse von Fehlern . . . 173
6.5.2 Fehler je Beispiel . . . 174
6.5.3 Studentische Partizipation . . . 177
6.5.4 Automatische Bewertungen . . . 178
6.6 Zusammenfassung . . . 178

**7 Evaluation 180**
7.1 Beschreibung der Durchführung . . . 180
7.1.1 Organisatorisches . . . 181
7.1.2 Inhalte und Ziele . . . 181
7.1.3 Verwendung der Plattform . . . 184
7.1.4 Realisierte Lehrhandlungen . . . 186
7.2 Untersuchung der studentischen Partizipation . . . 196
7.2.1 Beschreibung der Datenerfassung . . . 196
7.2.2 Ergebnisse . . . 197
7.2.3 Zusammenfassung & Diskussion . . . 210
7.3 Untersuchung der Aufgabenbewertung . . . 213
7.3.1 Datenerhebung . . . 214
7.3.2 Ergebnisse . . . 221
7.3.3 Diskussion . . . 226
7.4 Bewertung des Vorlesungskonzeptes (studentische Sicht) . . . 227
7.4.1 Beschreibung . . . 227

7.4.2 Ergebnisse . . . 228
7.4.3 Diskussion . . . 239
7.5 Diskussion der Hypothesen . . . 241
7.5.1 Einfluss auf die studentische Partizipation . . . 241
7.5.2 Einfluss auf die Programmierfähigkeit . . . 242
7.5.3 Auseinandersetzung mit Programmieren . . . 245
7.5.4 Einfache Programmierumgebung . . . 246
7.5.5 Indikatoren . . . 247
7.5.6 Fazit . . . 247

**8 Zusammenfassung 250**
8.1 Wissenschaftlicher Beitrag und Schlussfolgerungen . . . 250
8.2 Methodenbewertung . . . 253
8.3 Fazit und Ausblick . . . 255

**Literatur 257**

## Anhang 271

**A Beispiele für interaktive Lernmaterialien 271**
A.1 Interaktives Dokument . . . 271
A.2 Beispiel Dokumentenformat . . . 273
A.3 Informationen zur Plattform . . . 275
A.3.1 trycoding.io . . . 275
A.3.2 Repositories und Anleitungen . . . 275

**B Daten der Evaluation 278**
B.1 Zuordnung der Konzepte zu Übungsaufgaben . . . 278
B.2 Anzahl der Aktivitäten und Benutzer je Beispiel . . . 280
B.3 Anzahl der Aktivitäten je Aufgabe . . . 281
B.4 Häufigster Fehler je Aufgabe . . . 283
B.5 Fragebogen . . . 286

**Eigene Publikationen 289**

# Abbildungsverzeichnis


1 Fundamentale Ideen der Informatik - Algorithmisierung . . . . . 13
2 Fundamentale Ideen der Informatik - Strukturierte Zerlegung . . . . 13
3 Fundamentale Ideen der Informatik - Sprache . . . . . 14
4 Phasen des Programmiervorgangs . . . . . 33
5 Super Mario Bros. - Spielausschnitt Welt: 1-1 und falsche rechtsbündige Lösung . . . . . 37
6 Zuordnung der Phasen des Programmiervorganges zu den Merkmalen des Problemlösens und informatischen Denkens . . . . . 43
7 Beispiel für eine *Multiple-Choice*-Frage aus dem FCS1-Instrument . . 48
8 Bewertete Lösung mittels Indexbildung . . . . . 55
9 Stufenmodell der Formen des aktiven Lernens nach Wildt (entnommen aus [Wil11]) . . . . . 73
10 Grundprinzip des kooperativen Lernens nach Brüning und Saum . . 75
11 Prototypischer Ablauf des Problembasierten Lernens nach Zumbach . 77
12 Wechsel und Balance zwischen Instruktion und Konstruktion . . . . 80
13 Progressiver Lernprozess zur Dekonstruktion von Quelltext . . . . . 82
14 Code-Fragment-Frage in einem Quiz . . . . . 84
15 Beispielaufgabe aus dem Python Classroom Response System . . . . 89
16 Ebenen des didaktischen Handlungsspielraums . . . . . 96
17 Einordnung der Lehrveranstaltung in ein Elektrotechnikstudium . . . 97
18 Selbstständigkeit der Lernenden im Cognitive Apprenticeship . . . . 102
19 Konzept der interaktiven Aufgabenspirale . . . . . 104
20 Subgoal Labels - Beispiele nach Morrison et al. . . . . 107
21 Grafische Ausgabe des Turtle Beispiels . . . . . 111
22 Web-basierte Emulation des SenseHat-Moduls für den RaspberryPI . 113
23 Ausführung von Beispielen in einem interaktiven Textbuch . . . . . 121
24 Aufgabe mit Code-Magneten . . . . . 122
25 Interaktive Visualisierung in einem OpenDSA E-Book . . . . . 124
26 Programmieraufgabe in zyBooks . . . . . 125
27 Jupyter Notebook Beispiel mit Python Quelltexten . . . . . 126
28 Kommunikation in Jupyter Notebooks . . . . . 127

29 Beispielpräsentation mit Jupyter RiSE . . . . . 129
30 Erstellen einer Präsentation mit Jupyter RiSE . . . . . 130
31 Webbasierte Oberfläche von 5code . . . . . 134
32 CS50 IDE . . . . . 136
33 Codeboard.io Entwicklungsumgebung . . . . . 137
34 Aufbau der Plattform für interaktive Vorlesungen . . . . . 140
35 Aufbau der Sourcebox-Architektur einschließlich der Kommunikation der Module untereinander . . . . . 146
36 Web-basierter Editor – Quelltext (A), Konsolenausgabe (B), grafische Ausgabe (C) und Befehlsleiste (D). . . . . 149
37 Vereinfachtes Datenmodell für Beispiele bzw. Aufgaben . . . . . 150
38 Vereinfachtes Datenmodell für studentische Version der Beispiele und Aufgaben . . . . . 150
39 Menü der web-basierten Entwicklungsumgebung (aus Sicht des Besitzers) . . . . . 151
40 Terminal in der web-basierten Entwicklungsumgebung . . . . . 152
41 Anzeige von Fehlern und Warnungen im Editor . . . . . 153
42 Anzeige von Hilfetexten bei Fehlern innerhalb einer Entwicklungsumgebung . . . . . 153
43 Ansicht der Eingereichte Lösungen für den Lehrenden . . . . . 154
44 Vergrößerte Schrift bei einer Fenstergröße von $1024 \times 768$ Pixeln . . 155
45 Erstellung von Testfällen für die automatische Bewertung . . . . . 157
46 Ergebnisse der automatischen Bewertung in der Entwicklungsumgebung 157
47 Darstellung von *matplotlib* Diagrammen in der Entwicklungsumgebung 159
48 Beispiele für die web-basierte Simulation von Arduino-Aufbauten . . 160
49 3D-Simulation eines *RaspberryPis* mit aufgesetztem *Sense Hat* . . . . 160
50 Zusammenhang zwischen Kursen und Dokumenten . . . . . 163
51 Bearbeitungsmodus (links) und Leseansicht (rechts) von Textelementen 164
52 Ausführbare Programme innerhalb eines Dokuments . . . . . 166
53 Bearbeitungsmodus des Inhaltstyps *Code* mit Änderung der Programmiersprache . . . . . 167
54 Ungeladene (links) und geladene (rechts) Entwicklungsumgebung . . 167
55 Bearbeitungsmodus des Inhaltstyps *Aufgabe* . . . . . 168
56 *PythonTutor* eingebettet in ein Dokument . . . . . 169
57 Darstellung einer Folie in einer Präsentation . . . . . 172
58 Darstellung der häufigsten Fehler für ein bestimmtes Beispiel . . . . 175
59 Auswertungsmöglichkeit der studentischen Fehler in der Entwicklungsumgebung . . . . . 176
60 Darstellung der Ausführungen für ein Beispiel . . . . . 177
61 Mehrere *Blumen* mit unterschiedlicher Blätteranzahl . . . . . 189

62 Simulation des Auslesens der Temperatur in einer Endlosschleife (*SenseHat*) . . . 194
63 Aufgabe zur Umformung einer *for* Schleife in eine *while* Schleife . . . 195
64 Anzahl der Aktivitäten (oben) und bearbeiteten Aufgaben (unten) im Unterricht je Note . . . 200
65 Anzahl der Aktivitäten und bearbeiteten Aufgaben insgesamt je Note 201
66 Anzahl der bearbeiteten Aufgaben insgesamt je Gruppe . . . 202
67 Anzahl der Aktivitäten je Konzept für Notenstufe 1,0 (von 7 Studierenden) . . . 205
68 Anzahl der Aktivitäten je Konzept für Notenstufe 2,0 (von 11 Studierenden) . . . 206
69 Anzahl der Aktivitäten je Konzept für Notenstufe 3,0 (von 19 Studierenden) . . . 206
70 Anzahl der Aktivitäten je Konzept für Notenstufe 4,0 (von 19 Studierenden) . . . 207
71 Anzahl der Aktivitäten je Konzept für Notenstufe 5,0 (von 8 Studierenden) . . . 208
72 Vergleich der Prüfungsergebnisse mit dem neuen Bewertungsschema für Aufgabe 2 . . . 225
73 Vergleich der Prüfungsergebnisse mit dem neuen Bewertungsschema für Aufgabe 3 . . . 225

# Tabellenverzeichnis


1 Klassifizierung von Lerninhalten nach Allgemeingültigkeit . . . . . 15
2 Forschungsverlauf . . . . . 24
3 Beispiel zur Anwendung der Indexbildung . . . . . 54
4 Vollständige Anwendung der Indexbildung . . . . . 55
5 Unterschiede des kooperativen und kollaborativen Lernens . . . . . 74
6 Jupyter Notebook Inhaltselemente . . . . . 128
7 Vergleich der vorhandenen Ansätze zu interaktiven Lehrmedien . . . 132
8 Linux-Kernel Namespaces . . . . . 144
9 Linux-Kernel cgroups . . . . . 145
10 Darstellungsmöglichkeiten der Inhaltselemente in Präsentationen . . 171
11 Übersicht der Kapitel und Themen . . . . . 183
12 Aufgaben im Unterricht . . . . . 185
13 Finite Iteration: Aufgaben im Unterricht . . . . . 186
14 Übersicht der Teilnahme an der Prüfung . . . . . 197
15 Anzahl der Studierenden je Aufgabe . . . . . 199
16 Korrelationen der Aktivitäten der Studierenden . . . . . 203
17 p-Werte für die Korrelationskoeffizienten aus Tabelle 16 . . . . . 204
18 Verteilung der Aktivitäten je Notenstufe (gerundet) . . . . . 208
19 Die ersten zehn Aufgaben mit den meisten Aktivitäten . . . . . 209
20 Anwendung Bewertungsschema für Implementierungsaufgabe 1 . . . 215
21 Anwendung Bewertungsschema für Implementierungsaufgabe 2 . . . 217
22 Anwendung Bewertungsschema für Implementierungsaufgabe 3 . . . 219
23 Korrelationen der Ergebnisse der Aufgabenbewertung und der Aktivitäten für jedes Konzept . . . . . 222
24 p-Werte für die Korrelationskoeffizienten in Tabelle 23 . . . . . 223
25 Korrelationen der Ergebnisse der Aufgabenbewertung und der Aktivitäten für jedes Konzept . . . . . 224
26 Ergebnisse - Studiengang (Fragebogen) . . . . . 228
27 Ergebnisse - Zum Mitdenken und Durchdenken des Stoffes wurde angeregt (Fragebogen) . . . . . 229

28 Ergebnisse - In der Lehrveranstaltung habe ich mich aktiv beteiligt (Fragebogen) . . . 229
29 Ergebnisse - Beim Einbringen eigener Beiträge habe ich mich frei und äußerungsfähig gefühlt (Fragebogen) . . . 230
30 Ergebnisse - Ich habe die in der Vorlesung gestellten Aufgaben häufig bearbeitet. (Fragebogen) . . . 230
31 Ergebnisse - Es werden kommunikative Lehrformen, wie z. B. Diskussion, Aufgaben oder Quizzes eingesetzt (Fragebogen) . . . 231
32 Ergebnisse - Die Dozentin/Der Dozent fördert Fragen und aktive Mitarbeit (Fragebogen) . . . 231
33 Ergebnisse - Ich wurde zur Mitarbeit aufgefordert. (Fragebogen) . . . 231
34 Ergebnisse - Ich fühle mich in der Lage, kleine Programme zu schreiben. (Fragebogen) . . . 232
35 Ergebnisse - Es macht mir Spaß zu programmieren. (Fragebogen) . . . 232
36 Ergebnisse - Ich habe die Möglichkeit genutzt, die Aufgaben auch später online zu bearbeiten (Fragebogen) . . . 233
37 Ergebnisse - Ich habe die Übungen häufig von zuhause aus bearbeitet (Fragebogen) . . . 233
38 Ergebnisse - Ich habe die Übungen häufig an den Übungsterminen bearbeitet (Fragebogen) . . . 234
39 Ergebnisse - Die/Der Lehrende benutzte oft Beispiele, die zum Verständnis der Lehrinhalte beitrugen (Fragebogen) . . . 234
40 Ergebnisse - Die von der/dem Lehrenden ausgegebenen Materialien (z. B. Literatur, Skript, E-Learning-Inhalte, etc.) haben mir sehr geholfen, den Stoff zu erarbeiten. (Fragebogen) . . . 235
41 Ergebnisse - Ich bin mit der E-Learning Plattform (trycoding.io) gut zurechtgekommen. (Fragebogen) . . . 235
42 Ergebnisse - Es war leicht, Programme im Webbrowser zu erstellen und auszuführen. (Fragebogen) . . . 235
43 Ergebnisse - Die/Der Lehrende passte das Niveau der Lehrveranstaltung an den Wissensstand der Studierenden an (Fragebogen) . . . 236
44 Ergebnisse - Die Anforderungen der Lehrveranstaltung waren: (Fragebogen) . . . 236
45 Korrelationen - Es war leicht, Programme im Webbrowser zu erstellen und auszuführen (Fragebogen) . . . 237
46 Korrelationen - Es macht mir Spaß zu Programmieren (Fragebogen) . . . 238
47 Korrelationen - Zum Mitdenken und Durchdenken des Stoffes wurde angeregt. (Fragebogen) . . . 238
48 Korrelationen - Zum Mitdenken und Durchdenken des Stoffes wurde angeregt. (Fragebogen) . . . 239

49 Zuordnung der Aufgaben und des jeweiligen adressierten Programmierkonzepts . . . . . 279
50 Anzahl der Studierenden für alle Aufgaben im Unterricht . . . . . 280
51 Anzahl der Aktivitäten und Fehler je Aufgabe . . . . . 283
52 Häufigster Fehler je Aufgabe . . . . . 285
53 Häufigste Fehlerarten . . . . . 286

# Quellcodeverzeichnis


1 Beispiel für die Verwendung von Turtle . . . . . 110
2 Darstellung eines Polygons mittels Turtle . . . . . 111
3 Beispiel für reStructuredText und das Einbinden eines Code-Editors . 123
4 JSON-Format eines Dokuments . . . . . 162
5 JSON-Format einer Zelle in einem Dokument . . . . . 163
6 Funktion zum Zeichnen eines Quadrats mit beliebiger Länge . . . . . 187
7 Funktion zum Zeichnen eines Rechtecks . . . . . 188
8 Studentische Lösung zur bedingten Ausführung . . . . . 192
9 Mehrschrittige Entwicklung einer Lösung (rechts) zum Newton-Verfahren 194
10 Studentische Lösung zur bedingten Ausführung . . . . . 195
11 Eine mögliche Lösung zur Implementierungsaufgabe 1 . . . . . 214
12 Eine mögliche Lösung zur Implementierungsaufgabe 2 . . . . . 216
13 Eine mögliche Lösung zur Implementierungsaufgabe 3 . . . . . 220

# Teil I

# Didaktische Vorüberlegungen: Programmieren in der Lehre

# Kapitel 1

# Einleitung

Informatik und besonders Programmieren sind häufig Bestandteile vieler naturwissenschaftlicher Studiengänge und im Ingenieurbereich (vgl. [McC+01]). Die Frage dabei ist, ob Studierenden die intendierten Ziele solcher Veranstaltungen erreichen und wie Probleme identifiziert und somit die Lehre verbessert werden können. Zu Beginn werden die Relevanz der Informatik sowie mögliche Lehrveranstaltungsinhalte diskutiert und die grundsätzlichen Probleme dieser Fächer erläutert, da diese ausschlaggebend für die Entstehung dieser Arbeit waren.

## 1.1 Relevanz der Informatik, Praxisbezüge und die Lehre

Wozu und in welchem Umfang sollten Grundlagen der Informatik außerhalb der Informatik gelehrt werden? Im Vorgriff zur eigentlichen Arbeit, welche sich mit der Lehre des Programmierens (an Hochschulen[1]) beschäftigt, soll zunächst die Relevanz der Informatik in den Bereichen der Elektrotechnik, des Maschinenbaus und teilweise der Technischen Physik diskutiert werden. Denn die daraus resultieren Konsequenzen motivieren unter anderem die Gestaltung der Lehre. Im folgenden Kapitel werden zunächst die verschiedenen Sichtweisen auf den Praxisbezug und die damit verbundene Diskussion in der Lehre dargestellt.

[1] Die Ergebnisse und Erkenntnisse sind prinzipiell übertragbar, wurden jedoch im Kontext der Arbeit in Lehrveranstaltungen an Hochschulen angewandter Wissenschaften evaluiert.

### 1.1.1 Praxisbezüge

Im Gegensatz zur Kerninformatik sind der Umfang sowie Lehrziele bzw. Kompetenzen in ingenieurwissenschaftlichen Studiengängen anders gesetzt. Dabei stellt sich die Frage, wie sich diese Lehrziele und Kompetenzen ableiten und durch was sie beeinflusst werden. Die Anforderungen an einen Absolventen und damit die Lehre werden durch verschiedene Gruppen und den damit verbundenen Perspektiven beeinflusst. Zu diesen Gruppen zählen unter anderem die Studierenden selbst, die Lehrenden (in Bezug auf Freiheit der Forschung und Lehre) sowie die Industrie - wenn auch umstritten. Im Sinne des Bologna-Prozesses und der dort geforderten *Employability* (Beschäftigungsfähigkeit) [Wil15], dient das Studium, neben der Ausbildung des wissenschaftlichen Nachwuchses, als Vorbereitung für die Arbeitswelt. Nach einer gängigen Übersetzung des Employability-Begriffs

> [...] zielt Beschäftigungsfähigkeit auf die Fähigkeit ab, sich erforderliche Kompetenzen bei sich ändernden Bedingungen anzueignen, um Erwerbsfähigkeit zu erlangen beziehungsweise aufrechtzuerhalten [Wil15].

Hier muss zwischen Hochschulen (ehemals Fachhochschulen) und Universitäten, aufgrund des traditionell stärkeren Praxisbezuges der Hochschulen, differenziert werden. Denn diese Forderung nach Beschäftigungsfähigkeit und dem damit verbundenen Praxisbezug wird kontrovers diskutiert. Laut Schubarth wenden sich „viele Hochschulexpert(inn)en wie auch Dozent(inn)en und Studierende gegen eine Determinierung der Hochschulbildung durch den Arbeitsmarkt“ [vgl. Wil15]. Gleichzeitig wird von Hochschulexperten „die Notwendigkeit betont, den Zusammenhang von Studium und Arbeitsmarkt bewusst zu reflektieren“ [vgl. Wil15].

Der Begriff Praxisbezug kann dabei unterschiedlich interpretiert werden, dementsprechend muss dieser erst konkretisiert werden, um anschließend den Bogen auf die Relevanz der Informatik zu schlagen. Schubarth et. al definiert dabei den Begriff *Praxisbezug*, als „[...] die unmittelbare Anwendung oder Überprüfung der Gültigkeit einer Theorie [...]“ [vgl. WK14, S. 69]. Zu einer ähnlichen Definition kommt Tino Bargel in einer Analyse über den Stellenwert des Praxisbezuges aus Sicht der Studierenden im Studium und in der Lehre. Er unterscheidet zwischen drei Formen des Praxisbezuges:

> [...] (1) das Eingehen auf die Praxis in den Lehrveranstaltungen durch Beispiele und Konkretisierungen, (2) spezielle praktische Übungen und Seminare im Lehrangebot, vor allem auch zu Methodenfragen, und (3) die Vermittlung und Betreuung von Praktika in Betrieben und Einrichtungen außerhalb der Hochschulen. [Bar12]

Seitens der Studierenden wird allen drei Formen eine Wichtigkeit in ähnlicher Stärke zugesprochen, besonders Studierende der Fachhochschulen (Hochschulen für ange-

wandte Wissenschaften) schätzen den Praxisbezug als sehr wichtig ein. (vgl. [Bar12]) Bargel schlussfolgert daraus:

> Allein diese hohe Relevanz, die Studierende den verschiedenen Formen an Praxisbezügen im Studium zuschreiben, verlangt danach, sie bei der Gestaltung von Studium und Lehre ebenso wie Praktika außerhalb der Hochschule gleichermaßen ernst zu nehmen. [Bar12]

Diese Aussage kann nun in den Kontext der Relevanz der Informatik für andere Fachbereiche gestellt werden. Inwieweit müssen sich z. B. die Studierenden der Elektrotechnik mit der Informatik befassen und welche Auswirkungen hat dies auf die Lehre? Dazu beschäftigen wir uns exemplarisch mit dem Thema der Digitalisierung und *Industrie 4.0*. In den letzten Jahren hat die Digitalisierung einen immer stärkeren Einfluss auf die Industrie genommen, was somit Auswirkungen auf die Anforderungen an Absolventen hat. Die Digitalisierung wird dabei mit der vierten industriellen Revolution gleichgesetzt und wie folgenden vom Bundesministerium für Wirtschaft und Energie beschrieben:

> In der Industrie 4.0 verzahnt sich die Produktion mit modernster Informations- und Kommunikationstechnik. So können Produkte nach individuellen Kundenwünschen hergestellt werden: Sportschuhe mit maßgeschneiderter Sohle und in vom Kunden gewähltem Design oder ein passgenaues und individuell gestaltetes Möbelstück. Industrie 4.0 macht es möglich, Einzelstücke zum Preis von Massenware und das in höchster Qualität zu produzieren. Technische Grundlage hierfür sind intelligente, digital vernetzte Systeme und Produktionsprozesse. Industrie 4.0 bestimmt dabei die gesamte Lebensphase eines Produktes: Von der Idee über die Entwicklung, Fertigung, Nutzung und Wartung bis hin zum Recycling. [Bun]

Nun kann an dieser Stelle kritisch in Frage gestellt werden, inwieweit die Digitalisierung bereits in Unternehmen angekommen ist und dadurch die Beschäftigungsfähigkeit beeinflusst. Laut *Bitkom Research* hat die Digitalisierung, in einer Studie aus dem Jahr 2016, primär Anwendungsfälle im Maschinenbausektor, in der Herstellung von Elektronikerzeugnissen und dem Fahrzeugbau bzw. deren Zulieferern. Kleinere Unternehmen betreten das Feld bisher noch nicht und warten vorerst auf Erfahrungen anderer ab (vgl. [Bit16]). Dennoch ist eine Tendenz in Richtung Digitalisierung und der Teilbereich *Industrial Internet of the Things (IIOT)* zu erkennen. Neue Themenfelder bestimmen somit auch die Ausschreibungen am Arbeitsmarkt, wobei diese Änderungen nur bedingt Einfluss auf die Grundlagenvermittlung haben. Viel wichtiger ist die von Schubarth angesprochene Fähigkeit „sich erforderliche Kompetenzen bei sich verändernden Bedingungen anzueignen“ [Wil15]. Im Kontext der Elektrotechnik betrachtet, benötigen Studierende somit ein Grundverständnis der Informatik, um sich, neben den

aus sich der eigenen Disziplin ergebenen Themen, in verwandte Bereiche einarbeiten zu können. Besonders im Grundstudium und in den Einführungsveranstaltungen zu Programmieren und Informatik sollte der Fokus auf die abstrakten Konzepte und Vorgänge gelegt werden, jedoch ohne den Praxisbezug zu vernachlässigen.

### 1.1.2 Informatische Inhalte

Die Experimente dieser Arbeit fanden primär in den Studiengängen der Elektrotechnik und der Technischen Physik an der Hochschule Coburg statt. Besonders in diesen Studiengängen spielt eine Grundausbildung der Informatik eine wichtige Rolle. Die Gewichtung ist jedoch abhängig von der jeweiligen Hochschule bzw. Universität und den Studiengängen selbst und spiegelt sich teilweise im Umfang der Semesterwochenstunden (SWS) wider. In dieser Arbeit liegt der Fokus auf den Einführungsveranstaltungen zum Thema Programmieren bzw. Informatik. In der Elektrotechnik sind diese Themen meist Teil des Grundstudiums, auf welches viele nachfolgende Lehrveranstaltungen aufbauen. Es existiert jedoch kein einheitlicher Standard für die Inhalte. Die unterschiedlichen Perspektiven, Schwerpunkte und institutionellen Akzente ergeben sich durch das Curriculum und durch die von Professoren gestalteten Modulhandbücher. In einer Studie des Fachbereichstags Elektrotechnik und Informationstechnik (kurz FBTEI) wurden 400 Professorinnen und Professoren an (Fach)Hochschulen befragt, welche Grundlagen Bachelorabsolventen beherrschen sollten. Die einzelnen Bereiche wurden nach Kompetenzen aufgespalten und in die vier Stufen (Kennen, Verstehen, Anwenden und Umsetzen) eingeordnet. Die nachfolgende Aufzählung enthält die nach Relevanz sortierten Themen in der Informatik [Mic14]:

- Information und Codierung (Zahlen, Zeichen, Bit, Codes)
- Boolesche Funktionen und Algebra
- Höhere Programmiersprachen
- Betriebssysteme (Struktur, Funktion, Bedienung)
- Grundlagen Informatik (Automaten, Grammatik, Datenstrukturen)
- Kernelemente objektorientierter Programmierung
- Web-Anwendungen (Funktionsweise, Programmierung)
- Methoden SW-Engineering
- Algorithmen (Sort & Search, Berechenbarkeit, Komplexität)
- Internet (Protokolle, Domains, Dienste, HTML)

- Mobile Anwendungen

Ohne näher auf die Methoden und die Darstellung der Inhalte einzugehen, lassen sich jedoch aus der Liste einige Folgerungen und Kritiken ableiten. Erstens handelt es sich bei der Auflistung eher um Themen und nicht um Kompetenzen, wobei dies von der jeweiligen Definition abhängig ist. Jedoch wurden bis auf die grobe Beschreibung der einzelnen Stufen, keine Kompetenzdefinition angegeben. Zumindest lässt sich aus der Auflistung eine gewisse Relevanz der Informatik ableiten. Zweitens geben die veröffentlichten Ergebnisse weder Aufschluss über die einzelnen Themen im Bereich Programmieren, noch über Aufbau der Lehrveranstaltungen, in denen diese behandelt werden. Es werden zwar Automaten, Datenstrukturen sowie Boolesche Algebra angeführt, jedoch wird der Bereich Programmieren unter Programmiersprachen subsumiert. In Kapitel 3.1 wird die Tätigkeit Programmieren ausführlich diskutiert und in mehrere Phasen und Aktivitäten unterteilt und die damit verbundenen Herausforderungen analysiert. Hierbei wird im gleichen Zuge auf die Unterschiede zwischen Programmieren und Programmiersprachen eingegangen.

Im Bereich der Technischen Physik zeigt sich ein anderes Bild. Dort liegt der Fokus in der Informatik- bzw. Programmierausbildung auf der Auswertung und Numerik. Ähnlich ist es im Bereich des Maschinenbaus, in dem auch die Grundlagen des Programmierens und ggf. Software Engineering Inhalte vermittelt werden. Leider existiert in diesen Bereichen keine hochschulübergreifende Studie, weswegen hier nur auf eigene Erfahrungen zurückgegriffen werden kann. Dessen ungeachtet können die Tendenzen in Richtung Digitalisierung in der Maschinenbaubranche aus der Bitkom-Studie berücksichtigt werden. Wobei die Automatisierung und viele weitere informatische Themen im Maschinenbau inhärent sind. Ingenieure sind gezwungen sich mit dem Thema *Software Engineering* und Programmieren auseinanderzusetzen. In der Automobilbranche ist es de-facto Standard, dass eine Vielzahl an Systemen miteinander kommuniziert. Ein weiteres Beispiel ist die Entwicklung von Messwerkzeugen aller Art, welche enorme Mengen an Daten erzeugen und entsprechend ausgewertet müssen. Dies kann mit Standardsoftware wie z. B. MatLab oder LabView erfolgen, jedoch sind je nach Umfeld eigene Hilfsprogramme notwendig oder müssen ggf. angepasst werden und erfordern zumindest rudimentäre Kenntnisse in der Programmierung und deren unterstützenden Fähigkeiten.

Daneben ist die Zusammenarbeit und Kommunikation zwischen Informatikern, Maschinenbauern, Elektrotechnikern und weiteren Disziplinen essenziell. Denn dazu ist eine gemeinsame Sprache und ein Grundverständnis für die anderen Fachbereiche notwendig. Besonders das Wissen über informationstechnische Fachbegriffe und grundlegende Kenntnisse ist dabei wertvoll.

### 1.1.3 Auswirkungen auf die Lehre

Resümierend lässt sich feststellen, dass die Informatik eng mit der Elektrotechnik verbunden ist. Ohne dabei näher auf überfachliche Kompetenzen einzugehen, sind informatische Grundlagen ein elementarer Bestandteil der Disziplin. Aus der Forderung nach Employability im Rahmen des Bologna-Prozesses, lässt sich die Forderung nach einer besseren Verzahnung der Theorie und Programmierpraxis ableiten. Besonders für Hochschulen der angewandten Wissenschaften ist eine enge Verbindung zwischen der theoretischen Ausbildung und der Anwendung ein Alleinstellungsmerkmal. Jedoch sind Lehrveranstaltungen im Bereich Programmieren teilweise recht traditionell aufgebaut. Das heißt, dass diese aus zwei Komponenten, der Vorlesung und der Übung, bestehen. Somit ist bereits die Anwendung des Erlernten bzw. der Lerngegenstand in die Übung verlagert. Solche Lehr-/Lernarrangements können zweifelsfrei funktionieren und zu guten Resultaten führen, jedoch sind die Ergebnisse in den Einführungsveranstaltungen meist nicht zufriedenstellend. Schlechte Resultate in Prüfungen und eine mangelnde Programmierfähigkeit, ohne diese genauer zu definieren, wird sehr oft in der Forschungsliteratur konstatiert. Maßgeblich werden dabei die sogenannten McCracken Studien zitiert, welche die Programmierkenntnisse über mehrere Institute in unterschiedlichen Ländern überprüfen [McC+01; Utt+13]. Zusammengefasst, wurde in den Studien die Einschätzung der Lehrenden der Leistung der Studierenden gegenübergestellt. In beiden Studien wird die Leistung als ungenügend eingeschätzt, wobei sich in der zweiten Studie eine bessere Deckung zwischen der erwarteten und realen Leistung ergab. Doch nach wie vor sind die Ergebnisse nicht zufriedenstellend und damit die Programmierkenntnisse nur schlecht ausgeprägt. Zusätzlich wird in der Literatur das Phänomen der bimodalen Notenverteilung in Programmierveranstaltungen beschrieben [HE15]. Dabei tritt die zu erwartende Gaußglocke bei der Notenverteilung nicht auf, sondern die Noten häufen sich bei den sehr guten und schlechten Leistungen. Patitsas et al. haben in einer Studie belegt, dass dieses Phänomen meist auf einer Fehlinterpretation der Histogramme und Notenverteilungen beruht und in der Regel multimodale Verteilungen sind [Pat+16; siehe auch HE15]. Dies ist aber als eher methodische Kritik aufzufassen, denn die Leistungen sind nach wie vor mangelhaft. McGettrick et al. beschreiben in einer Arbeitsgruppe Programmieren als eine der größten Herausforderungen im Bereich der Informatik (eng. *Computer Sciences*; kurz CS) [McG04]. Insbesondere, wenn Informatik bzw. Programmieren nur ein Nebenfach ist, stellt das Erlernen einer Programmiersprache für viele Studierende eine große Aufgabe dar.

Die Anzahl der Publikationen über Programmieren in der Hochschulausbildung ist nach wie vor groß und wächst weiter. Der *goldene Pfad* für das Erlernen des Programmierens existiert nicht. Zusammenfassende wissenschaftliche Publikationen wie von Robins et al. [RRR03] sowie Pears et al. [Pea+07] zeigen eine Vielfalt an Lehrme-

thoden auf, die in unterschiedlicher Form eingesetzt werden. Für welche Methoden man sich entscheidet, die Auswahl des Lehrstoffs und die konkrete Umsetzung innerhalb einer Lehrveranstaltung, bleiben schlussendlich dem jeweiligen Lehrenden überlassen. Diese zu Entscheidungen müssen unter Berücksichtigung der teilweise geringeren Vorkenntnisse und Motivation getroffen werden. Denn die Studierenden im Bereich Maschinenbau oder Elektrotechnik haben sich bewusst gegen die reine Informatik entschieden haben. Die vorangegangene Diskussion zur Relevanz der Informatik und der Wichtigkeit der Praxisbezüge zeigt jedoch, dass Studierende sich gezwungenermaßen mit diesen Themen auseinandersetzen müssen. Hier stellt sich die Frage, wie die Studierenden zum Programmieren motiviert und Themen anwendungsbezogen gelehrt werden können, sodass diese ein grundlegendes Verständnis für die Tätigkeit mitbringen und ggf. eigene Programme entwickeln können. An dieser Stelle ist ein wohlüberlegtes Vorgehen, welches auf lehr- und lerntheoretischen Überlegungen fußt, notwendig, um vorhandene Probleme wissenschaftlich zu lösen. Dementsprechend ist eine Vorgehensweise zu wählen, welche die theorieorientierte Konzeption neuer Veranstaltungsformate für eine Integration des Praxisbezugs in Form von Übungsformen ermöglicht. Ebenso müssen diese Konzepte in Lehrexperimenten evaluiert werden, um die intendierten Ziele mit den wirklich erreichten zu vergleichen. Dafür müssen neue Konzepte in Lehrveranstaltungen umgesetzt und die benötigten Materialien und Werkzeuge entwickelt werden.

Unter dem Gesichtspunkt *Qualitätspakt Lehre*[2] erhält das Thema der zielgerichteten Konzeption von Lehr- und Lernarrangements weiteren Zuspruch. Der Qualitätspakt wurde im Rahmen der europäischen Studienreform beschlossen und soll deutsche Hochschulen bei der Reform durch zahlreiche Maßnahmen unterstützen. Diese Arbeit entstand im Rahmen des Projektes EVELIN (Experimentelle Verbesserung der Lehre im Software Engineering) - inklusive der Programmierausbildung - mit dem Fokus auf die Digitalisierung (der Lehre) und Kompetenzförderung der Programmierausbildung für Studierende der Elektrotechnik. Neben der besseren Personalausstattung ist das Ziel die Weiterentwicklung einer qualitativ hochwertigen Hochschullehre, welche die Digitalisierung der Lehre und die damit verbundene Integration elektronischer Lernmodule in das reguläre Curriculum inkludiert [3].

[2] Vgl. www.qualitaetspakt-lehre.de

[3] Vgl. Beschreibung des Qualitätspakt Lehre auf der offiziellen Homepage: http://www.qualitaetspakt-lehre.de

## 1.2 Ziele und Lösungsansätze

Wie können nun der Praxisbezug bzw. Übungen und Theorie besser miteinander kombiniert werden? Bevor diese Frage geklärt werden kann, müssen die Lehrziele bzw. zu fördernden Kompetenzen erarbeitet werden. Für Einführungsveranstaltungen stehen hierbei die grundlegende Funktionsweise eines PCs und die Programmierkonzepte im Vordergrund, ohne auf die unterschiedlichen Problemlösungsstrategien beim Programmieren einzugehen. Programmieren ist nicht nur das Erlernen einer formalen Sprache, sondern besteht aus mehreren Schritten und Prozessen, die alle in geeigneten Lehr/-Lernarrangements situativ und kontextbezogen adressiert werden müssen. In traditionellen Lehrveranstaltungen der Thematik Programmieren bzw. Programmierkonzepten werden meist Modellierungsaspekte und die theoretische Repräsentation der Konzepte bzw. Strukturen anhand einer Programmiersprache gezeigt. Beispiele und Inhalte orientieren sich dabei stark an den jeweiligen syntaktischen Konstrukten und nicht an den abstrakten Konzepten. Quelltexte und Beispiele sind in der Regel statisch, denn die Anwendung wird in die begleitenden Übungen verlagert. Eine Konsequenz daraus ist, dass die Modellierungsprozesse und die Implementierung getrennt voneinander gelehrt werden. Dies hat zur Folge, dass Studierende meist passive Rezipienten von Konzepten und statischen bzw. vorgeführten Beispielen sind. Der/die Studierende wird in einem solchen Rahmen nicht zum Agens und kann zunächst nur theoretisch die Algorithmen bzw. informatische Modelle erfassen. Da die Programmierfähigkeit und das Lösen von Problemen bzw. die Algorithmisierung Lehrziele im Ingenieurbereich sind, fehlt ein Konzept, diese Ziele situativ und kontextbezogen in einer Lehrveranstaltung umzusetzen. Besonders das Lernen durch praktische Anwendung hat auf das Erlernen einer Sprache, also auf das Programmieren, einen positiven Einfluss. Lehrinhalte und Aufgaben sollten Studierende dazu motivieren bzw. es ihnen ermöglichen, sich mit den einzelnen Schritten des Programmierens auseinanderzusetzen und somit bereits erste Fertigkeiten und Fähigkeiten dafür zu entwickeln.

Ziel einer solchen Lehrveranstaltung ist es, Studierenden einerseits die Konzepte abstrakt zu vermitteln, und andererseits die konkrete Ausgestaltung einer Umsetzung in einer Programmiersprache erarbeiten zu lassen. Durch ad hoc zu bearbeitende Übungsaufgaben im seminaristischen Unterricht kann die Umsetzung von Programmierkonzepten im Sinne aktiven Lernens gefördert werden. Dabei sind Studierende über das reine Zuhören hinaus im Lernprozess mit einbezogen. Dies geschieht durch Aktivitäten wie das Lesen, Schreiben, Diskutieren von Aufgaben und Beispielen, sowie deren Analyse, Synthese und Evaluation. Der Einsatz aktiver Lehr- und Lernmethoden sollte sich aus der inhaltlichen Planung der Lehreinheit heraus entwickeln, sinnvoll und erkennbar mit den zu vermittelnden Inhalten verzahnt sein und keineswegs als Selbstzweck betrieben werden. Entscheidend ist dabei das Ziel der Lehreinheit. Ziel der Arbeit ist es, Lehr-/Lernarrangements inklusive der dazu benötigen Werkzeuge

und Lernumgebungen zu erarbeiten und anschließend zu evaluieren. Diese sollen auf Basis des aktiven Lernens die Programmierfähigkeit und die Anwendung von abstrakten Programmierkonzepten innerhalb einer Vorlesung adressieren oder auch gar erst ermöglichen. Meist wird die Übung und damit die Anwendung der Konzepte (ganzheitliche Sicht) in die Übungsveranstaltungen verlagert. In einer Vorlesung kann ohne weitere Hilfsmittel der komplette Prozess des Programmierens nur bedingt abgebildet werden. Aber die sofortige Anwendung eines Konzepts bzw. Prinzips ist ein elementares Ziel und sollte möglichst gefördert werden. Zusammengefasst sollen die Vorlesung und die Übung stärker miteinander verschränkt werden.

## 1.3 Aufbau der Arbeit

Dieses Kapitel beschreibt den Aufbau der Arbeit und die Zusammenhänge der einzelnen Kapitel. Die in Kapitel 2.4 beschriebenen Forschungsfragen spiegeln sich hierbei direkt im Aufbau der Arbeit wider. In Kapitel 2 wird zuerst die Problemstellung dargelegt aus der anschließend die Forschungsfragen abgeleitet werden. Aus den Forschungsfragen und der in Kapitel 2.3 beschriebenen Forschungsmethodik sollen zuerst der Forschungsgegenstand, das Programmieren, definiert und ein Messinstrument zur Überprüfung der Programmierfähigkeit entwickelt werden. Darauffolgend werden in Kapitel 3.4 lehr- und lerntheoretische Überlegungen, welche für die Bewertung und Diskussion für Ansätze innerhalb des Aktiven Lernens notwendig sind, kritisch betrachtet.

Der zweite Teil der Arbeit beschäftigt sich mit der Entwicklung eines didaktischen Konzeptes, der Umsetzung eines Werkzeugs zur Unterstützung und der Evaluation des Ansatzes. In Kapitel 4 werden, unter Berücksichtigung der vorherigen theoretischen Überlegungen, die Merkmale des Aktiven Lernens und dessen verschiedene Formen diskutiert. Anschließend werden aus dem Stand der Forschung und bisherigen Ansätzen des Aktiven Lernens im Kontext Programmieren, mehrere Kernelemente für die Konzeptentwicklung identifiziert. Diese werden in ein didaktisches Gesamtkonzept (vgl. Kapitel 5) zur Steigerung der studentischen Partizipation, um die Verwendung und Anwendung von Konzepten der Programmierung zum informatischen Problemlösen zu fördern, eingebettet. Nachfolgend werden die in Kapitel 5 abgeleiteten Anforderungen an ein Werkzeug zur Umsetzung des Konzepts, technisch umgesetzt und beschrieben. Schlussendlich wird das Konzept in Kapitel 7 quantitativ unter Verwendung des in Kapitel 3.3.3 entwickelten Messinstruments evaluiert und diskutiert. In Kapitel 8 wird der wissenschaftliche Beitrag zusammengefasst und das methodische Vorgehen bewertet und ein Fazit gezogen.

# Kapitel 2

# Problemstellung

Die in der Einleitung angeführten Ziele der Ausbildung und die daraus folgenden Forderungen zu einer besseren Verzahnung der Theorie und Praxis bei der Programmierausbildung, soll in diesem Kapitel noch feiner ausgearbeitet werden. Zuerst werden die Ziele der Programmierausbildung anhand von existierenden informatikdidaktischen Ansätzen diskutiert. Danach werden die Probleme bei der Ausbildung konkretisiert und in Forschungsfragen überführt und daraus Hypothesen abgeleitet.

## 2.1 Ziele der Programmierausbildung in der Elektrotechnik

In der Einleitung wurden bereits der Stellenwert des Programmierens sowie die damit verbunden Auswirkungen für die Lehre angesprochen. Zur Planung und Verbesserung einer Lehrveranstaltung durch den Einsatz zielorientierter Methoden und Ansätze, müssen zuerst die Ziele der Lehrveranstaltung definiert werden. Wesentlich ist dafür die Einordnung in das Curriculum sowie ein grundlegendes Kompetenzprofil, besonders hinsichtlich des Bologna-Prozesses. Das Profil ist jedoch von individuellen, institutionellen Anforderungen und der jeweiligen Umgebung abhängig, weswegen nachfolgend die grundlegenden Inhalte diskutiert werden. Die Welt der Informatik schreitet immer weiter voran und durchdringt, wie in Kapitel 1.1.1 beschrieben, immer neue Felder und Branchen. Die Fortschritte in der Informatik können jedoch nicht in gleicher Geschwindigkeit in der Lehre umgesetzt werden. Deshalb stellt sich die Frage, welche Lerninhalte in eine zeitgemäße Einführungsveranstaltung zum Thema Programmieren gehören. Die Auswahl der Inhalte und der zu fördernden Kompetenzen darf nicht in unzusammenhängende Themen zerfallen. Lernende sollen rote

Fäden und Beziehungen zu anderen Bereichen erkennen und die Grundzüge dieses Faches erfassen. Nach Schwill sind die „grundlegenden Prinzipien, Denkweisen und Methoden ([die] fundamentalen Ideen) der Informatik“ [Sch93] zu vermitteln. Dabei soll sich der Unterricht in erster Linie an den Strukturen - den fundamentalen Ideen - der zugrundeliegenden Wissenschaft orientieren [Sch93], welche besonders für die Allgemeinbildung nützlich sind. Daneben existieren noch weitere Ansätze, welche über diese stärkere Orientierung an Konzepten hinausgehen. Zum Beispiel haben sich soziotechnische Ansätze entwickelt, die neben der Erstellung solcher Systeme, auch die Anwendung und die damit verbunden Auswirkungen auf die Umwelt betrachten [vgl. HMS99]. Jedoch wird an dieser Stelle auf die weitere Diskussion von stark objektorientierten Ansätzen und ganzheitlichen Informatiksystemen verzichtet, da der Fokus auf der Programmierausbildung liegt.

Das Konzept der fundamentalen Ideen zur Auswahl von Lerninhalten hat Bruner aus allgemeindidaktischen Überlegungen heraus entwickelt und wurde immer im Kontext des Problemlösens diskutiert [Bru72]. Daraus entwickelten sich vier Kriterien zur Identifizierung solcher Leitideen [Sch93]:

- *Horizontalkriterium*: umfassende Anwendbarkeit in vielen Bereichen
- *Vertikalkriterium*: in einem breiten Altersbereich vermittelbar
- *Sinnkriterium*: Verankerung im Alltagsdenken, lebensweltliche Bedeutung
- *Zeitkriterium*: Beobachtung der geschichtlichen Entwicklung und langfristiger Relevanz

Schwill hat Bruners didaktisches Prinzip für die Informatik durchgeführt und die identifizierten Ideen hierarchisch gruppiert. Dabei hat er drei sogenannte Masterideen bzw. Hauptideen identifiziert:

- Algorithmisierung
- strukturierte Zerlegung
- Sprache

Bewusst ist hier nicht von allen fundamentalen Ideen die Rede, sondern nur das die genannten Punkte dazugehören. Bei den Hauptideen muss weiterhin angemerkt werden, dass Schwill im Bereich der informatischen Ausbildung an Schulen forscht. Weswegen viele Elemente und Bereiche in den Ideen nicht auftauchen, die in Curricula von Studiengängen der Informatik verankert sind. Trotzdem bilden die Ideen das Programmieren und die Ziele einer Einführungsveranstaltung für Studierende ohne Vorkenntnisse sehr gut ab. Die Abbildungen 1, 2 und 3 zeigen Schwills deduzierte fundamentale Ideen inklusive ihrer Hierarchien in der Informatik.

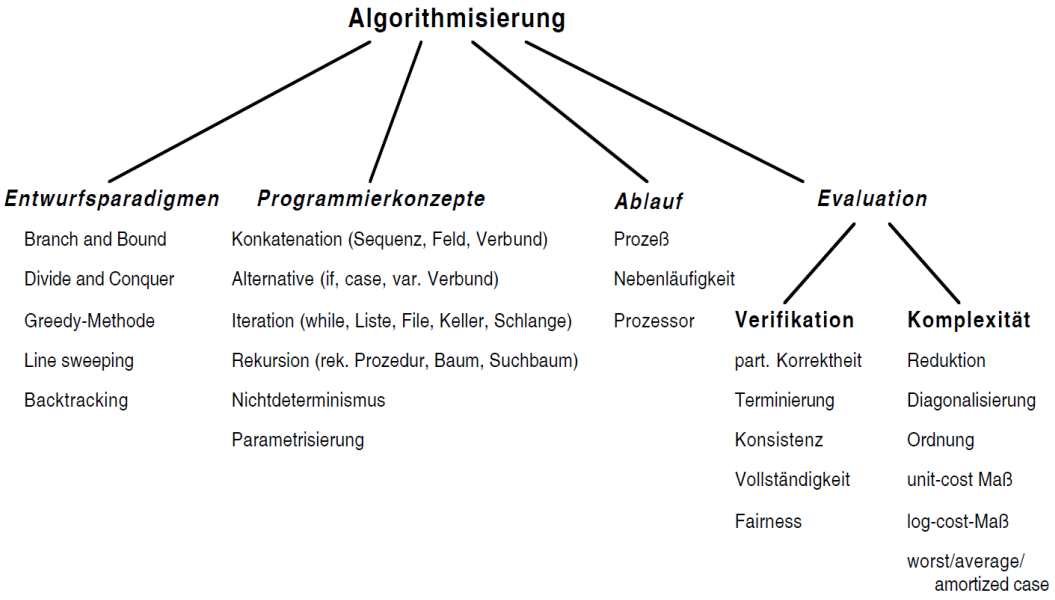



**Abbildung 1**: Fundamentale Ideen der Informatik - Algorithmisierung

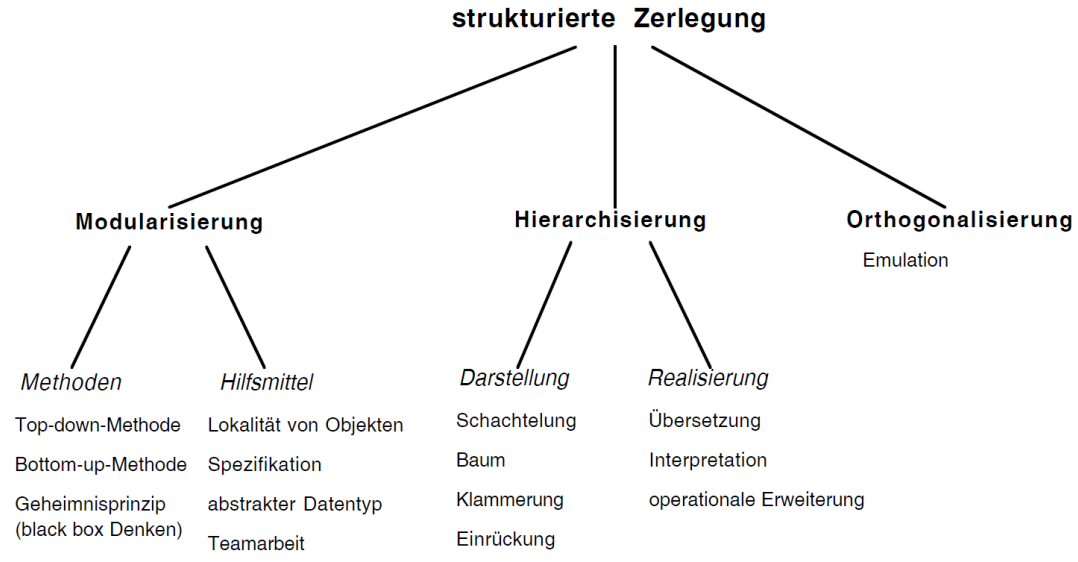



**Abbildung 2**: Fundamentale Ideen der Informatik - Strukturierte Zerlegung

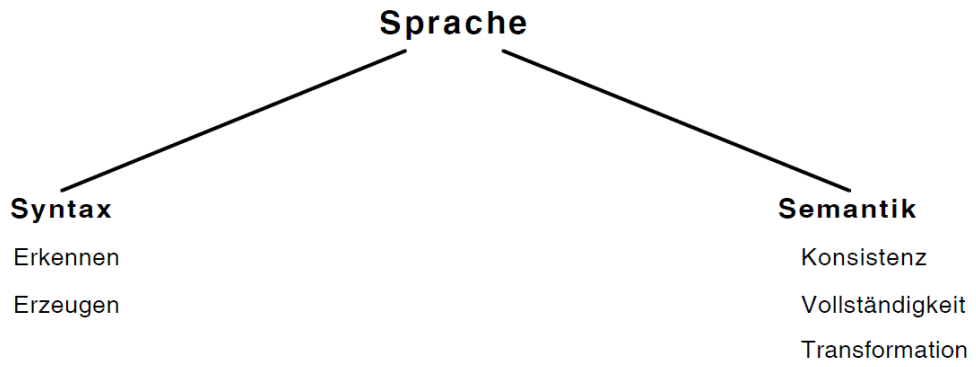



**Abbildung 3:** Fundamentale Ideen der Informatik - Sprache

Nach Schwill stellen die Problemlösungsstrategien, die Algorithmisierung und die Umsetzung fundamentale Ideen des Programmierens dar. Wie bereits oben angeklungen, bildet Schwill hier nur einen Teil der Informatik ab. In Kapitel 1.1.2 wurden die Freiheit der Lehre und die unterschiedlichen Einflüsse auf die Inhalte der Hochschullehre diskutiert. Viele Universitäten mit Lehrstühlen für Didaktik der Informatik fokussieren sich primär auf die Schul- und Lehrerausbildung und nicht auf die Erwachsenenbildung. Jedoch lassen sich die Forschungsergebnisse und Publikationen auf die Einführungsveranstaltungen zum Thema Programmieren übertragen, da hier besonders im Bereich Elektrotechnik nur wenig Vorwissen zu erwarten ist. Dies ist unter anderem dem Umstand geschuldet, dass viele Studierende nur grundlegende Informatikkenntnisse mitbringen. Informatik ist an bayrischen Schulen zwar seit 2004 für die Sekundarstufe I verpflichtend sind, aber je nach Schulform sind die Inhalte und der Umfang unterschiedlich stark ausgeprägt. Nachfolgend werden die unterschiedlichen fachdidaktischen Ansätze aus der Didaktik der Informatik betrachtet, um ein didaktisches Fundament für eine Vorlesung und die darin zu vermittelnden Inhalte zu schaffen.

Hubwieser stellt die Modellierung als inhaltlichen Kern für den Informatikunterricht (an Schulen) in den Vordergrund [Hub07]. Dazu ordnet er die fundamentalen Ideen von Schwill den Bereichen „[...] Verarbeitung, Darstellung und Verteilung von Informationen [...]“ [Hub07] zu. Bei ihm steht der möglichst breite Anwendungsbereich im Fokus, weswegen er die Inhalte, wie in Tabelle 1 dargestellt, in vier Klassen unterteilt [Hub07]:

Diese Klassifizierung hat nach Hubwieser direkte Auswirkungen auf die Auswahl der Inhalte.

Lerninhalte aus den ersten beiden Klassen sind für die Schulinformatik

**Tabelle 1**: Klassifizierung von Lerninhalten nach Allgemeingültigkeit

| Klasse | Beschreibung | Beispiele |
|---|---|---|
| 1 | Anwendung auch außerhalb des Bereiches der EDV möglich | Modellierungstechniken, Problemlösungsstrategien, |
| 2 | charakteristisch für alle elektronischen Informationssysteme | Komplexitätsklassen, Grenzen der Berechenbarkeit |
| 3 | Anwendung beschränkt sich auf eine Klasse von Informatiksystemen | Programmierparadigmen, spezielle Datenstrukturen Netzwerktopologien, Sortieralgorithmen |
| 4 | betrifft nur ein konkretes System | Syntax einer Programmiersprache, Menüstruktur eines Anwendersystems, Architektur eines Mikroprozessors |

> uneingeschränkt geeignet, Themen aus Klasse 3 nur, wenn eine sehr umfangreiche oder sehr wichtige Klasse von Subsystemen betroffen ist. Inhalte aus Klasse 4 sind lediglich soweit zu vermitteln, als sie für die Realisierung, Implementierung oder Simulation von Konzepten höherer Klassen unbedingt nötig sind [Hub07].

Das Spektrum der Inhalte, welche Hubwieser in der Tabelle beschreibt, soll über mehrere Jahrgangsstufen verteilt werden. Für eine Vorlesung über das Programmieren kann dementsprechend nur ein Ausschnitt behandelt werden, um auch den Anforderungen an nachfolgende Fächer gerecht zu werden. Somit muss einerseits sichergestellt werden, dass alle notwendigen Kompetenzen für nachfolgende Lehrveranstaltungen adressiert werden. Und andererseits sollten dabei die informatische Grundausbildung und die damit verbundenen übergeordneten Konzepte und Problemlösungsstrategien nicht vernachlässigt werden. Ohne auf weitere konkrete Inhalte einzugehen, die für jeden Studiengang spezifisch erhoben und in einem Kompetenzprofil dokumentiert werden sollten, ist zumindest das primäre Ziel der Lehrveranstaltung zu festzulegen. Dazu sollte der Begriff Programmieren definiert werden, da dieser bereits Modellierungsaspekte beinhaltet. Sigrid Schubert beschreibt in der Diskussion über Informatikunterricht als Programmierkurs Programmieren wie folgt:

> Informatikunterricht soll kein Programmierkurs sein. Warum eigentlich nicht? [...] Problemlösen (Modellieren und Strukturieren) unter Verwendung von Informatikprinzipien und -methoden gilt als erstrebenswert. Die Programmiersprache soll im Hintergrund (mittel zum Zweck) bleiben. Das aber ist Programmierung (nicht zu verwechseln mit Codierung).

> Das Grundmodell der heute erfolgreichen Informatikausbildung beruht auf der Entwicklung guter (strukturierter) Lösungspläne [...] [Sch91].

Hierbei wird bereits eine wichtige Differenzierung getroffen, Programmieren und die dazugehörigen Konzepte sind sprachunabhängig. Denkprozesse und Modellierungstätigkeiten finden vor der Codierung statt. Die Codierung ist Implementierung der Lösung in einer konkreten formalen Sprache. Das Resultat der Codierung ist ein Programm bzw. Software und damit die konkrete Anwendung. Nach Schubert erlaubt Software „[...] als vergegenständlichte Intelligenz [...] das Modellieren und Kombinieren von logischen Grundbausteinen (Sequenz, Zyklus, Alternative) mit ungewöhnlichen Freiheitsgraden“ [Sch91]. Der Duden der Informatik beschreibt Programmieren kompakter aber in ähnlicher Weise:

> Unter Programmieren versteht man zum einen den Vorgang der Programmerstellung und zum anderen das Teilgebiet der Informatik, das die Methoden und Denkweisen beim Entwickeln von Programmen umfasst. [Cla03]

Zusammengefasst sind folgende Elemente in einer Einführungsveranstaltung von Relevanz und stellen die primären Ziele der Programmierausbildung dar:

- Methoden und Denkweisen sowie die logischen Grundbausteine und deren Kombination und die daraus resultierende Lösungsvielfalt
- Die Fähigkeit Problemstellungen strukturiert aufzubereiten, zu zerlegen, zu algorithmisieren und anschließend in einer konkreten Sprache zu codieren

Die Ziele beinhalten die von Hubwieser in den Vordergrund gestellte Modellierung, da die Implementierung auf Basis eines vorher entwickelten Modells erfolgen sollte. Programmierung und Codierung nach dem *Trial & Error-Prinzip* sollte vermieden werden. Hubwieser fordert bei der didaktischen Umsetzung der Modellierungstechniken:

> Die Modellierungstechniken zunächst einzeln einzuführen, einzeln auf geeignete Probleme anzuwenden und die erzeugten Modelle möglichst sofort zu implementieren [Hub07].

Programmieren und die dazugehörige Implementierung kann somit auch als *Informatik im Kleinen* bezeichnet werden, da sie sich nur auf einen Teilaspekt der Modellierung bezieht. Software-Engineering und die Konzeption großer Softwaresysteme kann, wie Hubwieser richtig beschreibt, ohne die Implementierung auskommen und zielt deutlich stärker auf Modellierung ab. Sollen Studierende jedoch nach einem Semester bereits *Programmieren* können, muss die Codierung und Anwendung von Konzepten entsprechend adressiert werden. Leider lässt sich die letzte Forderung nur bedingt in Vorlesungssälen, ohne Computerzugang, umsetzen. Das Ziel dieser Arbeit ist aber, eine mögliche Antwort bzw. Lösung für das genannte Problem zu geben.

## 2.2 Status quo der Lehre - Warum ist Programmieren schwer?

Bei der Entwicklung und Anpassung neuer Konzepte und Ideen sollte auf eine Bestandsaufnahme nicht verzichtet werden. Der Forschungsgegenstand sowie die in Kapitel 1.2 beschriebenen Probleme und der geforderte Lösungsansatz haben sich aus den Beobachtungen innerhalb der Lehrveranstaltung *Programmieren 1* in der Elektrotechnik entwickelt. In der Literatur werden die Probleme beim Erlernen des Programmierens, die meist schlechten Prüfungsergebnisse, Anfängerfehler sowie hohe Abbruchquoten oft beschrieben [Wha+06]. Auch Forderungen, mehr Wert auf das Programmieren und nicht auf das Erlernen einer formalen Sprache zu legen, sind darunter zu finden. Die mit meistzitierten Publikationen sind die McCracken Studien[4] (siehe auch Kapitel 1.1.3), die internationale Erhebungen der Programmierfähigkeit der Studierenden den Einschätzungen der Dozenten gegenüberstellt [McC+01; Utt+13]. Zwar gibt es auch kritische Stimmen zur Methodik der Studie der ersten Studie, trotzdem lässt sich aus diesen Untersuchungen und anderen Publikationen eine gewisse Tendenz ableiten, dass Programmieren eine Herausforderung für die Studierenden darstellt. Um darauf zu reagieren, müssen zuerst die dahinter liegenden Probleme identifiziert werden.

Das vorherige Kapitel deutet bereits an, dass Programmieren keine leicht zu erlernende Fähigkeit ist. Für die Erarbeitung der Problemstellung wird mit einer kurzen Reflexion auf unser Verständnis davon, was beim Erlernens des Programmierens den Studierenden abverlangt wird, begonnen. Hierbei steht besonders der Bezug auf die Vorlesung als Veranstaltungsform im Fokus, d. h. viele der nachfolgend beschriebenen Probleme und die deren Synthese, treten hauptsächlich in den Vorlesungseinheiten der Lehrveranstaltung *Programmieren 1* auf.

**Anpassung der Ziele auf die Lehrveranstaltung** Das in Kapitel 2.1 beschriebene Ziel ist, dass Studierende Konzepte verstehen und anwenden können - und damit schlussendlich Programme *schreiben* (codieren). Unsere Lehrveranstaltungen nötigen die Studierenden teilweise dazu, dass sie durch das *Lesen von Beispielen die dahinter liegenden Konzepte verstehen und diese damit erlernen.* Dabei wird eine Lese- und Verständniskompetenz von Quelltexten seitens der Studierenden vorausgesetzt. Die eigentliche Anwendung, der Transfer der Konzepte und das damit verbundene Erlernen von Heuristiken wird in die zur Lehrveranstaltung zugehörige Übung verlagert. Einzelne Konzepte können dabei durch weitere Methoden und Illustrationen besser

[4] Die erste wurde unter Leitung von McCracken und die zweite wurde von Utting und McCracken durchgeführt.

dargestellt werden, dennoch ist nach wie vor die Anwendung und Übertragung auf andere Probleme der Kern. Zu ähnlichen Schlussfolgerungen kommen auch Lister et al. in ihrer internationalen Studie über Lese- und Tracing-Fähigkeiten[5] von Programmieranfängern [Lis+04]. In der Studie wird der gleiche Gegensatz zwischen den Erwartungen an die Studierenden und der gewählten Unterrichtsmethoden thematisiert.

**Inhaltliche Überforderung** Ein weiteres Problem beim Erlernen von Programmieren stellen die vielen unterschiedlichen Inhalte und Themen dar. Für die Anwendung der Konzepte in einer konkreten Sprache, müssen Lernende sich zusätzlich mit Editoren und Werkzeugketten auseinandersetzen, um die eigentlichen Inhalte zu üben. Somit stehen die Studierenden vor mehreren unterschiedlichen kognitiven Herausforderungen, welche eine Fokussierung auf einzelne Themen verhindern. Bei der Anwendung einer Schleife beispielsweise müssen sich Studierende gleichzeitig mit folgenden Themen beschäftigen:

- Verstehen und Zerlegen der Aufgabenstellung,
- das Programmierkonzept der Schleife bzw. der Iteration und dessen Übertragung auf eine neue Aufgabenstellung,
- die konkrete formale Syntax zur Umsetzung der Schleife,
- und einem Werkzeug zum Verfassen des Quelltextes und einer Werkzeugkette zur Übersetzung und Ausführung des Quelltextes.

Zusätzlich trägt die Geschwindigkeit der Vorlesung und die Einführung neuer Themen zur Überforderung bei. Viele weitere Konzepte bauen auf vorherigem Wissen auf, welches zu diesem Zeitpunkt meist nicht verfestigt ist und ein schwaches Fundament für neue Themen bildet.

**Themenspezifische Übungen und Aufgaben** Das Einüben der neuen Themen wird nun in Übungsveranstaltungen verlagert. Werden dabei grundsätzliche Verständnisprobleme beobachtet, kann auf diese zwar individuell reagiert werden, die Wiederholung bzw. Korrektur für alle ist aber erst in der nächsten Unterrichtseinheit möglich. Zusätzlich stehen dem Lehrenden während der Vorlesung nur wenige Rückkopplungskanäle zur Verfügung. Ohne geeignete Übungen während der Lehrveranstaltung haben Studierende nicht die Möglichkeit praxisnahe Erfahrungen zu sammeln und dabei auftretende Verständnisprobleme umgehend zu thematisieren. In der Vorlesung kann

[5] *Tracing* bedeutet in diesem Kontext die Fähigkeit den Ablauf eines Programmes nachvollziehen zu können. Darunter fällt auch die Interpretation der Ergebnisse und das Verstehen des Programmes.

der Modellierungsprozess und die Implementierung nur teilweise seitens der Studierenden eingeübt werden.

**Facheigene Probleme** Neben den bisher beschriebenen teilweise fachunabhängigen Problemen, wohnen dem Erlernen von Programmieren auch selbst Herausforderungen inne. Programmieren stellt eine gewisse Art des logischen und verknüpfenden Denkens dar, welche in Kapitel 3.1 detaillierter diskutiert wird. Dieses wird in der Literatur als *Computational Thinking* (zu dt. *Informatorisches Problemlösen*) (vgl. [Win06]) bezeichnet und beschreibt all die Aspekte, welche beim Erstellen von algorithmischen Lösungen erforderlich sind bzw. eine Rolle spielen. Denn zum Lösen von informatischen Aufgaben müssen Studierende sowohl theoretische Grundlagen wie auch Problemlösungsstrategien erlernen. Besonders letzteres erfordert das Wissen über Programmierkonzepte und deren Anwendung um einzelne Teilprobleme zu bewältigen und anschließend diese zu einer Gesamtlösung zu kombinieren. Die einzelnen Konzepte bzw. logischen Bausteine hängen meist voneinander ab oder bauen aufeinander auf, sodass die Studierenden oftmals auf nicht gefestigtem Wissen aufbauen müssen.

**Motivation** Das Erlernen von Programmieren ist für Informatiker und Nichtinformatiker eine schwierige Herausforderung, jedoch sind Studierende der Informatik im Vorteil in Bezug auf den durch die Motivation erzeugten Lernwillen. Dieser ist jedoch essentiell und deshalb das vordringlichste Ziel didaktischen Handelns. Hubwieser beschreibt Motivation „[...] als einen kurz andauernden Zustand des Angetriebenseins, unter Motivierung dagegen das aktive Bemühen um die Herstellung eines solchen Motivationszustands“ [Hub07].

Aus der Literatur und ausgehend von den bisherigen Prüfungsergebnissen lässt sich ableiten, dass Studierende Probleme beim Erlernen des Programmierens haben. Dieses Kapitel soll einige der grundlegenden Herausforderungen näher beleuchten.

**Das Kernproblem** Die bisher beschriebenen Probleme und die mangelnde Möglichkeit situativ Übungen und Aufgaben für die jeweiligen Unterrichtsinhalte durchzuführen, erfordern:

- Neue Inhalte und Konzepte gezielt und situativ mittels Aufgaben zu üben (Üben inkludiert dabei Tätigkeiten der Lernenden, die Besprechung der Ergebnisse und ggf. Richtigstellung seitens des Lehrenden),
- den Unterricht teilweise von der Leistung und den Fragen der Studierenden (auch indirekt) lenken zu lassen,
- Studierenden die Möglichkeit geben aktiv am Unterricht zu partizipieren

- und Anregungen zum aktiven Verarbeiten der Inhalte zu schaffen.

(Fach)Hochschulen haben seit ihrer Entstehung bereits das Konzept des seminaristischen Unterrichts als Alleinstellungsmerkmal gegenüber der Universitäten entwickelt und eingesetzt. In diesem Sinne sollen Studierenden durch aktivierende Methoden die Konzepte der Programmierung, deren Anwendung, die Kombination zu Algorithmen und schlussendlich deren Implementierung, vermittelt werden. Studierenden wird dadurch eine Heuristik an die Hand gegeben, die sie später zum Lösen ähnlicher Probleme verwenden können. Daraus ergibt sich folgende Fragestellung: *Wie können Studierende im Sinne eines seminaristischen Unterrichts, mit dem Ziel aktives und diskursives Arbeiten zu fördern bzw. zu ermöglichen, in einer Programmierveranstaltung eingebunden werden?*

## 2.3 Forschungsmethodische Vorgehensweise

Bevor die hier verfolgten Forschungsfragen formuliert werden, wird zunächst der gewählte Forschungsansatz beschrieben. Von dem Lehralltag und von der Planung der Lehrveranstaltungseinheiten unterscheidet sich die Entwicklung von Konzepten auf Basis wissenschaftlicher Theorien. Eine theoriegeleitete Entwicklung von Konzepten und Unterrichtsmaterialien und eine anschließende Evaluation hat eine deutlich stärkere Aussagekraft als ein reines lösungsorientiertes Vorgehen, denn die Entscheidungen bei der Auswahl der Methoden sind aus dem Stand der Wissenschaft heraus begründet und stellen nachprüfbare Ergebnisse (Intersubjektivität) sicher. Simon zeigt auf Basis seines entwickelten Klassifikationsschemas, dass viele Publikationen im Bereich der informatischen Ausbildung (im engl. *Computer Science Education*) oftmals nicht theoretisch fundierte Experimente und Praxisberichte sind und wie notwendig ein wissenschaftliches Vorgehen ist [Sim15].

Diese Studie zeigt warum ein theoretisch begründetes Vorgehen, bei dem die Theorien die wissenschaftlichen Zusammenhänge aufzeigen und Folgerungen für die Praxis erlauben, bei der Entwicklung neuer Ansätze immanent ist. Bei der Begründung und Konzeption von Methoden fließen einerseits lerntheoretische und andererseits fachdidaktische Überlegungen mit ein. Lerntheorien alleine können Gestaltungshinweise bei der Konzeption geben bzw. als Leitfaden bei der Entwicklung genutzt werden und das später ausgestaltete Konzept auf eine wissenschaftliche Basis setzen. Die Notwendigkeit für Interventionen in der Lehre sollte dabei auf Basis didaktischer bzw. fachdidaktischer Theorien und Vorgehen begründet werden. Bei der Auswahl von geeigneten Methoden bzw. der Entwicklung gibt es keine reinen schwarz-weiß Entscheidungen, sondern diese müssen aufgrund theoretischer Überlegungen und situativer Einflussfaktoren getroffen werden. Diese ersetzen jedoch nicht die spezifische Ausge-

staltung von den gewählten Methoden in das konkrete Szenario. Es existieren keine didaktischen *Allroundmethoden*, die ohne Anpassung auf jedes beliebige Thema anwendbar sind. Konzeptionelle Entwicklungen orientieren sich oftmals an mehreren theoretischen Ansätzen und kombinieren im Rahmen eines konsistenten Gesamtkonzepts. Tulodziecki und Herzig argumentieren deswegen für eine praxis- und theorieorientierte Entwicklung und Evaluation von Konzepten für pädagogisches Handeln [THB04]. Konzepte, die nach diesem Vorgehen entwickelt werden „[...] haben gegenüber herkömmlichen Unterricht den Vorteil, daß sie auf der Basis - mindestens bis zu einem gewissen Grad - bewährter lern- und lehrtheoretischer Annahmen entwickelt wurden“ [THB04]. Die Orientierung an bewährten Theorien garantiert jedoch nicht unbedingt entsprechende Lernerfolge. Sie erlauben aber ein begründetes Vorgehen, welches im Anschluss evaluiert werden muss. Der Evaluation solcher entwickelter Konzepte messen Tulodziecki und Herzig eine hohe Relevanz zu, da während der Konzeption Entscheidungen getroffen werden müssen, „[...] deren empirische Auswirkungen unter Umständen nur schwer vorhersehbar sind. Insofern ist in jedem Falle eine Erprobung der Unterrichtskonzepte sinnvoll. Wissenschaftsmethodisch gesprochen handelt es sich hierbei um eine Evaluation“ [THB04].

Zusammengefasst hat die theoretische Orientierung zwei Aufgaben: (1) Sie befördert die systematische Entwicklung und Evaluation von Konzepten auf Basis von Erkenntnissen nach dem Stand der Wissenschaft. (2) Weiterhin stellt die Forderung nach einer Evaluation sicher, dass das entwickelte Konzept auch die intendierten Ziele erreicht.

Eine fachdidaktische Fundierung der gewählten Methoden und an lerntheoretischen Ansätzen angelehnte Ausgestaltung, garantieren, dass die Auswahl der Methoden auf Basis der Lehrziele erfolgt und nicht in unbegründeten Experimenten endet. Bei der Formulierung von Forschungsfragen und Hypothesen, die auf lern- und lehrtheoretischen Annahmen basieren, müssen *Wenn-Dann-Aussagen* aus dem Kontext erschlossen werden, da diese meist in den Theorien nicht explizit angegeben sind [THB04]. Dies ist streng genommen auch nicht möglich, da die Überprüfung und eine präzise Formulierung nur im jeweiligen Kontext, indem die Theorien angewendet werden, möglich ist. Erst aus den konkreten Lernzielen können die zu untersuchenden Wechselwirkungen abgeleitet werden. Nach Tulodziecki und Herzig ergeben sich daraus folgende Schritte (vgl. [THB04, S.13]):

- Vermutungen zu verbesserungswürdigen Disposition auf Seiten der Lernenden,
- Entwickeln von Zielvorstellungen und deren Begründung,
- Entscheidung für einen geeigneten lern-lehrtheoretischen Ansatz auf Basis einer prüfenden Reflexion,
- Formulierung der Annahmen zu den Lernvoraussetzungen, der Zielvorstellun-

gen und der theoretischen Annahmen auf Basis des gewählten Ansatzes,

- Konkretisierung der anzustrebenden Lernaktivitäten und geeigneter Lehrhandlungen
- Entwickeln der notwendigen Lern- und Lehrmaterialien,
- Entwurf einer Handlungslinie für den Unterricht als Orientierung für die Lehrperson.

Dieser Entwicklungsprozess und die Entscheidungen in den einzelnen Schritten beeinflussen sich gegenseitig, weswegen im Laufe des Vorgangs unpassende oder unausgereifte Stellen durch Ansätze ausgetauscht oder stärker ausgearbeitet werden müssen. Für die empirische Evaluation des entwickelten Konzepts sollten nach Tulodziecki und Herzig folgende Fragestellungen berücksichtigt werden [THB04, S.16]:

1. Welche Voraussetzungen bringen die Lernenden mit?
2. Welche Lehrhandlungen werden realisiert?
3. Welche Lernaktivitäten sind auf Seiten der Lernenden zu beobachten?
4. Welche Nebenwirkungen sind festzustellen?
5. Welche Lernergebnisse werden erreicht?

Ausgehend von diesen Daten sollten folgende Gesichtspunkte bei der Beurteilung beachtet werden [THB04, S.16]:

1. Wurden die Lernvoraussetzungen im Rahmen des Konzepts angemessen eingeschätzt?
2. Konnten die Lehrhandlungen in der geplanten Weise durchgeführt werden? Erwies sich dies als sinnvoll?
3. Wurden die Lernenden in der angestrebten Weise aktiv? Zeigten sich unter Umständen erwünschte oder unerwünschte Nebenwirkungen?
4. Wie sind die erreichten Lernergebnisse im Aspekt der Zielvorstellungen zu beurteilen?

Für die Beantwortung der vierten Frage ist die Entwicklung eines Messinstruments zur Einschätzung notwendig. Wobei die Einschätzung bezüglich der Erreichung von Lernergebnissen auch auf Basis eines Wertes immer noch problematisch erscheint. Die gewählten Grenzen für das erfolgreiche Lernen müssen aus den empirischen Bedingungen und Lernzielen abgeleitet und argumentiert werden. Im Bereich des Programmierens ist hierbei eine gezielte Definition der zu untersuchenden Variablen zur

Überprüfung der Fähigkeiten der Lernenden wichtig. Eine zusätzliche summative Evaluation zur Bewertung des Konzeptes aus einer weiteren Perspektive scheint angebracht, um tiefere Einblicke zu erhalten. In Hinblick auf weiterführende Fragen, können unter anderem diese Evaluationsergebnisse unter den Gesichtspunkten (a) der Übertragbarkeit der Ergebnisse, (b) der Gültigkeit der dem Konzept zugrundeliegenden Voraussetzungs-Ziel-Mittel-Aussagen und (c) kritischen Auseinandersetzung des gewählten theoretischen Ansatzes diskutiert werden. Der dargelegte Entwicklungsprozess und Aspekte der Evaluation wurden bei der Bearbeitung des Forschungsthemas berücksichtigt.

**Tabelle 2**: Forschungsverlauf

| **Aktivitäten und Meilensteine** | **Erkenntnisse** |
|---|---|
| Interaktives Ebook-Kapitel | |
| Situativ eingebettete Darstellung objektorientierter Zusammenhänge für Studierende der Technischen Physik<br>Selbstständiges Erarbeiten neuer Inhalte im Basis eines Flipped-Classrooms-Ansatzes | Machbarkeitsstudie sowie Überprüfung der Annahme des Konzepts seitens der Studierenden |
| Interaktive Beispiele in Präsentationen | |
| Konzeption und technische Umsetzung von ausführbaren Beispielen zur Bearbeitung von Aufgaben während der Vorlesung | Aufgaben und Lernziele noch zu ungerichtet eingesetzt |
| | Umständliche Erstellung von Inhalten und deren Verbreitung - simplerer Ansatz notwendig |
| Konzept: Interaktive Aufgabenspirale | |
| Konzept zur Selektion und Integration gezielter Beispielaufgaben in der Vorlesung mit Rückkopplungsmechanismen. | Stärkerer Fokus auf die Konzepte und kein zu starkes Gewicht auf die Sprache |
| Erstellung einer Sandbox-Lösung zur sicheren Ausführung von Fremdcode im Browser | Interaktive Inhalte und greifbare Beispiele können einfach integriert werden |
| Umsetzung einer Plattform zur Erstellung von Skripten und Präsentationen mit Rückkopplungsmechanismen für Dozenten und Studierende | Deutlich einfachere Erstellung von Inhalten |
| | Eingehende Analyse von studentischen Fehlern |
| Abschlussdiskussion | |
| Evaluation der Lehrveranstaltung und abschließende Bearbeitung der Hypothesen | Sichtbarer Zusammenhang zwischen Anzahl der bearbeiten Beispiele bzw. Aktivitäten und der erreichten Prüfungsleistung bzw. Programmierfähigkeit |

Das Konzept und Anpassungen der theoretischen Ansätze wurden durch ein iteratives Vorgehen verfeinert. Ausgangspunkt war die Betreuung der Übungen der Vorlesung Programmieren 1 für Elektrotechniker und Automobiltechniker. Aufgrund der Nähe zu den Studierenden konnten erste Probleme bei der Bearbeitung der Übungsaufgaben beobachtet werden, welche später die Problemstellung dieser Arbeit beeinflusst haben. In Hinblick auf die *Programmieren 1* Veranstaltung für Studierende der Technischen Physik wurde zuerst ein interaktives eBook für die Einführung in die Objektorientierung anhand mathematischer Objekte nach Langtangen entwickelt [Lan10]. Basierend auf den Erkenntnissen von Brad Miller zu interaktiven eBooks im Bereich des Programmierens [MR12] sollte zunächst ein Kapitel so umgesetzt werden, dass die Studierenden dieses direkt in der Vorlesung und von zuhause aus bearbeiten konnten. Dieses erste Experiment [EH15a; EH15b] und die daraus gewonnenen Hinweise haben den Weg zur eigentlichen Konzeption und Literaturrecherche geebnet. Die Tabelle 2 zeigt dabei die wichtigsten Meilensteine und Phasen im Forschungsverlauf auf.

## 2.4 Forschungsfragen

Für die zielgerichtete Bearbeitung des Themas wurden zunächst forschungsleitende Fragen identifiziert, welche die Einflussfaktoren, Inhalte und angestrebten Ergebnisse bzw. Wirkungen der wissenschaftlichen Arbeit darstellen [Töp12, S. 155]. Anschließend werden aus den Fragen zu prüfende Hypothesen abgeleitet und diese näher beschrieben.

### 2.4.1 Fragen

Der in Tabelle 2 gezeigte Forschungsverlauf zeigt bereits, dass zuerst der Ansatz der interaktiven E-Books exemplarisch evaluiert wurde. Dieser Versuch hat jedoch noch konzeptionelle Schwächen in Bezug auf die Zielgerichtetheit gezeigt, weswegen ein stärker Fokus auf die lehr- und lerntheoretischen Ansätze gelegt wird. Neben den Leitfragen des forschungsmethodischen Vorgehens, stellen nachfolgende Fragen einen inhaltlichen Rahmen für die Bearbeitung dar. Die ersten beiden Fragen sind deskriptiver Natur und dienen zur Beschreibung relevanter Begriffe und empirisch nachvollziehbarer Sachverhalte. Die Fragen leiten sich aus den in Kapitel 2.2 identifizierten Problemen und der in Kapitel 1.2 aufgezeigten Lösung ab.

FF 1: Was ist Aktives Lernen?

FF 2: Welche Aktivitäten des aktiven Lernens existieren im Kontext des Erlernens von Programmieren?

FF 3: Wie können Schreib-/Lese- und Diskussionsaktivitäten in einer Vorlesung (Hochschulausbildung) integriert bzw. gefördert werden? (Welche Ansätze existieren im Kontext einer Präsenzveranstaltung?)

FF 4: Welche Instrumente werden zu Steuerung einer Lehrveranstaltung bei aktivierenden Aktivitäten benötigt?

FF 5: Welche (technischen) Voraussetzungen sind für die Integration in einer Präsenzlehrveranstaltung notwendig? (Zielt auf die Anforderungen an solch ein System ab)

Der postulierte Ansatz zur stärkeren Verschränkung von Übung und Vorlesung, orientiert sich an den Grundlagen des Aktiven Lernens. Die Bearbeitung der Forschungsfrage 1 soll das Konzept und die Ideen hinter dem Aktiven Lernen verdeutlichen. Daran schließt sich die Forschungsfrage 2 an, welche das Aktive Lernen im Kontext der Programmierausbildung stellt und bereits vorhandene Ansätze diskutiert. Für die Beantwortung der Forschungsfrage 3 soll ein Konzept für die Integration von Aktivitäten aus dem Aktiven Lernen erarbeitet werden. Dabei wird auch eine lernpsychologische Theorie und die darauf aufbauenden pädagogischen Methoden diskutiert, die sich für die Problemstellung besonders eigenen. Forschungsfrage 4 und 5 diskutieren die Anforderungen an ein Werkzeug, welches bei der Integration des zuvor beschriebenen Konzeptes benötigt wird. Hierbei werden auch mögliche Steuerungsinstrumente für die Vorlesung und Unterrichtsvorbereitung diskutiert.

### 2.4.2 Hypothesen

Aus den Forschungsfragen lassen sich Hypothesen ableiten, die nach der Evaluation des entwickelten Ansatzes überprüft werden sollen. Die stärkere Fokussierung auf Aktivitäten seitens der Studierenden beim Erlernen des Programmierens, indem Aufgaben in den seminaristischen Unterricht integriert werden, sollte einen Einfluss auf die Beteiligung der Studierenden haben. Studierende sollen sich im Unterricht mit den jeweiligen Inhalten auseinandersetzen und diese teilweise anwenden, weswegen folgende Hypothese daraus abgeleitet wird:
**Hypothese 1**. *Aktivitäten des Aktiven Lernens im Programmierunterricht erhöhen die studentische Partizipation im Unterricht.*

Darüber hinaus, wird vermutet, dass konkretes Einüben neuer Konzepte einen positiven Einfluss auf die Programmierfähigkeit hat. Studierende, die sich im Unterricht mehr mit den Konzepten auseinandersetzen und auch in den Übungen vertieft einüben, sollten tendenziell bessere Leistungen erzielen als Studierende, die sich nicht beteiligen:

**Hypothese 2**. *Aktives Lernen hat einen positiven Einfluss auf die Programmierfähigkeit der Studierenden.*

Ziel des hier zu untersuchenden Ansatzes ist es, die Auseinandersetzung mit Programmieren zu erhöhen, indem bereits im Unterricht Aufgaben integriert und diskutiert werden. Studierende sollen dadurch an das Thema herangeführt werden und sich auch über die Vorlesung hinaus mit Programmieren beschäftigen und Aufgaben bearbeiten.
**Hypothese 3**. *Aktives Lernen im Programmierunterricht erhöht die studentische Auseinandersetzung mit Programmieren.*

Die nächste Hypothese beschäftigt sich mit der Umsetzung bzw. Konkretisierung der Forschungsfrage Punkt 3, die sich mit der Integration von Aufgaben in den Unterricht beschäftigt. Besonders hinsichtlich der beschriebenen Probleme bei der Verwendung professioneller Entwicklungsumgebungen, sollen die Akzeptanz und der Einfluss einer leicht zugängliche Entwicklungsumgebung auf die Auseinandersetzung mit dem Programmieren untersucht werden. Daraus kann folgende Hypothese abgeleitet werden:
**Hypothese 4**. *Eine leicht zugängliche und nutzbare Programmierumgebung erleichtert/hat einen positiven Einfluss auf die Auseinandersetzung mit dem Programmieren.*

Schlussendlich wird vermutet, dass die Diskussion der Lösungen der Aufgaben und die daraus resultierenden Erkenntnisse über Fehler, die Steuerung der Lehrveranstaltung ermöglichen. Denn aufgrund dieser Rückmeldungen über die Aktivitäten der Studierenden kann der aktuelle Lernfortschritt besser eingeschätzt und entsprechend reagiert werden.
**Hypothese 5**. *Eine Analyse der Rückmeldungen bei der Bearbeitung von Aufgaben (im Unterricht) ermöglicht eine Steuerung einer Lehrveranstaltung.*

# Kapitel 3

# Programmieren und Programmierfähigkeit

Folgendes Kapitel beschreibt und definiert den Begriff Programmieren und dessen Bestandteile. In diesem Zuge wird der Begriff der Programmierfähigkeit diskutiert und eine Methode zu ihrer Überprüfung dargelegt. Die aus den Forschungsfragen 2, 3 und 5 sowie die daraus abgeleiteten Hypothesen über den Einfluss des Aktiven Lernens auf die Programmierfähigkeit (vgl. Hypothese 1) erfordern eine Diskussion und ein mögliches Schema zu deren Bewertung. Daraus ergeben sich Schlussfolgerungen, die bei der Konzeption einer Lehrveranstaltung für Programmieren berücksichtigt werden sollten und für die das in dieser Arbeit entwickelte Konzept beachtet wurden.

## 3.1 Was bedeutet Programmieren?

In Kapitel 2.1 wurden die Ziele einer informatischen Ausbildung unter dem Gesichtspunkt Programmieren anhand fachdidaktischer Ansätze diskutiert. Die theoriegeleitete Forschungsmethodik (vgl. Kapitel 2.3) erfordert fachdidaktische sowie lehr- und lerntheoretische Begründungen bei der Entwicklung eines Konzeptes und dessen Evaluation. Für diese theoretischen Überlegungen ist es erforderlich, die Zielvorstellungen und damit den Begriff Programmieren zu definieren und die eigenen Auffassungen zu begründen. Erst diese Definition und zugehörige Elemente erlauben eine zielgerichtete Diskussion der lehr- und lerntheoretischen sowie didaktisch-methodischen Ansätze.

In der Literatur existiert eine Vielzahl an Definitionen und Teilaspekten, die unter dem Oberbegriff Programmieren subsumiert werden können. In diesem Kapitel sollen

die unterschiedlichen Definitionen diskutiert und anschließend eine eigene Definition für diese Arbeit ausgearbeitet werden. Der Fokus liegt hier besonders auf den unterschiedlichen Teilprozessen, die beim Programmieren und somit auch bei seinem Erlernen eine Rolle spielen.

In Kapitel 1.2 wurden bereits verschiedene Ansichten über die Informatik und deren Teilgebiet des Programmierens diskutiert mit dem Ergebnis, dass eine stärkere Gewichtung der Modellierungsaspekte wichtig erscheint. Daraus ergibt sich, dass Programmieren ein Prozess bzw. ein Vorgang ist, der Methoden und Denkweisen zur Programmerstellung beinhaltet [Cla03]. Diese Denkweisen und Methoden funktionieren universell und setzen keine bestimmte formale Sprache zur eigentlichen Umsetzung voraus. Stattdessen werden die Vorgänge auf der übergeordneten Ebene beschrieben, welche zur eigentlichen Lösung führen. Die Implementierung in einer spezifischen Programmiersprache erfolgt erst am Ende des Prozesses. Schubert beschreibt dabei das Vorgehen beim Programmieren und die dabei entstehende Software, als „[...Modellieren und Kombinieren von logischen Grundbausteinen (Sequenz, Zyklus, Alternative) [...]“ [Sch91]. Diese Sichtweise beinhaltet, dass das zu lösende Problem bereits verstanden wurde und nun durch die Kombination höherwertiger logischer Bausteine gelöst werden kann. Für die Modellierung und die Verwendung der logischen Grundbausteine ist es aber erforderlich, dass das Problem verstanden und bereits in kleinere Teilprobleme unterteilt wurde. Erst auf dieser Ebene können diese Bausteine ausgewählt und angewendet werden. Casperen unterteilt Programmieren in zwei Teilprozesse:

> Programming is a creative process; when developing for a given problem, programmers are free to invent whatever structures suit their needs, though eventually these structures must be realized in a formal, executable language. It is wise not to worry too early about the final realization of data and processes but instead to invent suitable abstractions for the problem at hand then - as two separate concerns - solve the problem in terms of invented abstractions and realize these abstractions on the underlying machine and thereby extend the machine. This strategy can be applied repeatedly and at any level of abstraction [Cas07].

Die Differenzierung von der Lösungsidee und der eigentlichen Umsetzung ähnelt den Auffassungen von Hubwieser und Schubert, die von einem *Top-Down-Ansatz* spricht und das Problemlösen als iteratives Aufstellen von Lösungshypothesen beschreibt [Sch07a, S. 86ff]. Aber im Vergleich stellt Caspersen den iterativen Charakter des Vorgangs stärker in den Vordergrund und zeigt dessen Nichtlinearitität auf:

> [...] is a process of trial and error; it is not a strictly progressive process. If we were able to pick the right abstractions and get the code right, it

> would be a strictly progressive process, but we are not. The programming process can be characterized as an explorative activity of discovery and invention where the programmer investigates the problem and the underlying machine, makes hypothesis, and tries to verify these by writing and executing code. In doing so, the programmer creates abstractions, applies these, and realizes the abstractions on the underlying machine [Cas07].

Programmieren ist nur im Idealfall ein linearer Vorgang. Denn oftmals sind die gewählten logischen Bausteine oder deren Kombination nicht richtig und es muss auf der Ebene des Modells und somit auch die Implementierung korrigiert werden. Caspersen verwendet dabei den Begriff Abstraktionen, welcher mit den von Schubert beschriebenen logischen Bausteinen gleichzusetzen ist. Konzepte wie Zyklus oder Alternative sind abstrakte informatischen Prinzipien, die sprachunabhängig zu Lösungen zusammengesetzt werden können. Nach Caspersen werden beim Programmieren Hypothesen zur Lösung aufgestellt und anschließend implementiert und verifiziert.

> Ever so often in the process a hypothesis breaks and the programmer must alter or reject it. Either way, it affects the code that has been written, including abstractions that have been created, applied, and realized. Consequently, code must be modified or erased and new hypotheses must be made, from which new code is written and executed [Cas07].

Diese Darstellung seitens Caspersen ähnelt der Auffassung Bruners, dass neben dem Fachwissen (fundamentale Ideen) „[...] auch Techniken des Problemlösens, d.h. die Fähigkeiten, die Problemstellung zu analysieren, Hypothesen zu formulieren und zu prüfen“ [Ede94], dazugehören. Auch wenn Programmieren somit nicht immer linear verläuft und häufig nach dem Prinzip *Versuch & Irrtum* abläuft, muss jedoch auf die jeweilige Ebene geachtet werden. Versuch und Irrtum sollten auf Ebene der Modellierung stattfinden und kein wahlloses Anwenden von Syntaxelementen darstellen. Auch Caspersen unterscheidet hier zwischen syntaktischen Fehlern und fehlgeschlagenen Hypothesen, die anschließend überarbeitet werden müssen. Trotzdem sind diese Hypothesen und die von ihm beschriebenen Abstraktionen der Ausgangspunkt für die Implementierung, auch wenn beide Ebenen abwechselnd bzw. mehrfach durchlaufen werden. Auch die bereits in Kapitel 2.1 beschriebene Fokussierung der Modellierung seitens Hubwiesers ist dem Modell von Caspersen ähnlich da auch er die Implementierung der erzeugten Modelle fordert [Hub07]. Für die Durchführung der einzelnen Schritte können wiederum die fundamentalen Ideen (siehe Zweig Programmierkonzepte in Abbildung 1) von Schwill herangezogen werden. Die Algorithmisierung und die strukturierte Zerlegung helfen beim Auffinden der passenden Modelle bzw. Abstraktionen für den Lösungsalgorithmus, welcher abschließend in einer formalen Sprache implementiert und damit verifiziert wird.

Bei der nachfolgenden Definition des Programmiervorgangs wurde bewusst Wert auf die algorithmische Umsetzung gelegt. Das Modellieren von großen Informatiksystemen ist eher im Bereich des Software-Engineerings angesiedelt, welches eigene und teilweise ähnliche Methoden und Denkweisen erfordert. Zusammengefasst, kann aus den unterschiedlichen Auffassungen ein iterativer Prozess mit mehreren Schritten abgeleitet werden. Programmieren hat das Ziel, ein bestimmtes Problem unter zu Hilfenahme informatischer Systeme zu lösen. Der Vorgang zur Lösung des Problems beinhaltet mehrere Phasen, die mehrmals durchlaufen werden können. Die folgenden Phasen stehen immer im Kontext der Programmieranfänger. Denn eine Annahme für die Arbeit ist, dass Experten einzelne Schritte überspringen oder diese nicht explizit, sondern nur mental durchführen.

**Lesen & Verstehen** Bevor mit der Lösung begonnen werden kann, muss das geschilderte Problem zuerst verstanden werden. Im Kontext der Ausbildung ist hierbei das systematische Lesen und Erfassen der relevanten Stellen in einer Aufgabenstellung wichtig. Ohne das Problem zu verstehen, können keine geeigneten Modelle gebildet und später durch die Codierung evaluiert werden. Beim Verstehen müssen alle wichtigen Textstellen, ggf. durch Einholen weiterer Informationen, erfasst werden.

**Entwicklung einer Lösungsidee** Sobald das Problem verstanden wurde, erfolgt die systematische Zerlegung der Problemstellung in Teilprobleme. Diese wiederum lassen sich durch die Anwendung von Strategien und das Kombinieren von logischen Grundbausteinen lösen. Die Auswahl der Bausteine und deren Kombination basiert in der Regel auf Heurismen (erfahrungsbasiert), welche auf ein neues Problem angewendet werden [Tüc03]. Diese Auswahl der Bausteine wird durch das Erkennen von Analogien begünstigt, da diese das Aufstellen von Hypothesen und die Entwicklung einer Lösungsidee (wenn auch nur für Teile) erleichtern. Nach Tücke sind „[...] heuristische Lösungen ziemlich effektiv, aber auch fehleranfällig. Sie funktionieren nicht immer“ [Tüc03]. Deswegen sollten die in dieser Phase getroffenen Annahmen auch überprüft und ggf. angepasst werden. Ziel dieser Phase ist es, eine Idee für einen möglichen Lösungsalgorithmus zu erarbeiten, welcher bereits in logische und definierte Schritte unterteilt ist. Der Algorithmus bzw. Lösungsentwurf ist dabei von einer konkreten Sprache losgelöst und konzentriert sich primär auf die Abfolge von einzelnen Schritten. Die Entwicklung einer Lösungsidee unter Verwendung von logischen Bausteinen und Anwendung von Lösungsstrategien sehe ich hier gleichbedeutend mit dem der Modellierung.

**Umsetzung & Verifikation** Sobald ein Algorithmus zur Lösung des Problems erarbeitet wurde, kann dieser in einer formalen Sprache codiert werden. Ziel der Codie-

rung ist die Umsetzung der logischen Bausteine und des richtigen Programmablaufes zur Lösung des Problems. In dieser Phase können verschiedene Fehler entstehen, die auf unterschiedlichen Ebenen korrigiert werden müssen:

1. Syntaxfehler[6]: Können in aktueller Phase korrigiert und überprüft werden.
2. Laufzeitfehler[7]: Können auch in der aktuellen Phase behoben werden.
3. Semantische Fehler: Das Programm kann ausgeführt werden, jedoch ist der gewählte Algorithmus noch fehlerhaft. Zur Lösung muss zurück in die vorherige Phase gesprungen werden, um entweder die Kombination der logischen Bausteine oder deren Parametrisierung, wie z. B. Start oder Ende oder Bedingung einer Iteration.

Die Codierung des Algorithmus und die anschließende Programmausführung, ggf. unter Verwendung von Testwerten, dient dazu, dass die Lösungsidee auch überprüft werden kann - die *Verifikation* der Idee. Damit kann sichergestellt werden, dass die algorithmische (Teil-)Lösungsidee das (Teil-)Problem auch entsprechend der Anforderungen in der Problemstellung abdeckt. Dabei ist unter Algorithmus eine Verfahrensvorschrift zu verstehen, welche so genau beschrieben ist, dass sie von einem Automaten ausgeführt werden kann. Das Verständnis über die Funktionsweise eines Automaten, das Wissen über die logischen Bausteine sowie Wissen über die Darstellung von Informationen sind dabei grundsätzliche Voraussetzungen, um Probleme programmatisch mittels Algorithmen lösen zu können (Vgl. Ziele der Programmierausbildung in Kapitel 2.1). Die Ausführbarkeit eines Algorithmus wird später in Kapitel 3.2.3 näher erläutert. Eine ähnliche Beschreibung und Anpassung der Modellierung bzw. Lösungsidee wurde bereits von Howe bei Schülern beobachtet [How88]. Programmieren ist dabei kein linearer Prozess, sondern viel mehr ein iterativer, bei dem ein Plan zur Lösung verfolgt und überprüft wird [vgl. Vis90; Gre14, S.131]. Abbildung 4 zeigt die möglichen Pfade, wie der Vorgang durchlaufen werden kann und sollte. Eine fehlgeschlagene Verifikation kann mehrere Ursachen besitzen. Einerseits können Fehler bei der Umsetzung selbst auftreten, die nicht zur Änderung der Phase führen, sondern diese nur erneut anstoßen. Anderseits können auch eine Fehlvorstellung bzw. Fehler in der Lösungsidee dazu führen, dass diese überarbeitet werden muss. Schlussendlich kann eine fehlgeschlagene Verifikation auch Hinweise darauf geben, dass das Problem nicht richtig erfasst wurde. Es ist auch möglich diesen Prozess auf Teilprobleme anzuwenden, um so sich sukzessive einer Gesamtlösung zu nähern. Dies hat den Vorteil, dass bei der Umsetzung und Verifikation nur ein Teil betrachtet werden muss und sich so die Komplexität verringert. Eine sinnvolle Unterteilung ist abhängig vom jeweili-

[6] Syntaktische Fehler stellen fehlerhafte Stellen im Quelltext dar, welche zur Folge haben, dass das Programm nicht übersetzt bzw. interpretiert werden kann.

[7] Umfassen alle Fehler, die während der Ausführung eines Programmes auftreten und in der Regeln zu einem unerwartete Abbruch der Ausführung führen.

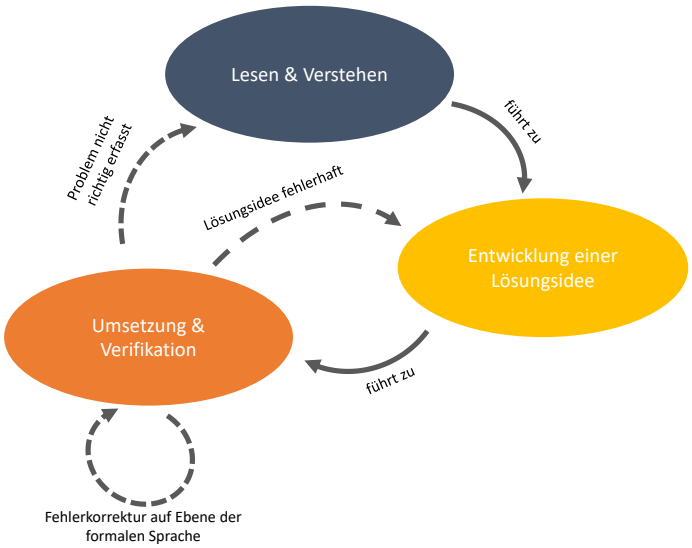



**Abbildung 4:** Phasen des Programmiervorgangs

gen Problem und kann nicht pauschal vorgenommen werden. Jedoch sollte die Anzahl der Iterationen und Lösungsschritte mit der Komplexität des Problems steigen, auch wenn hier Ausnahmen existieren.

## 3.2 Kognitive Herausforderungen

Die Schwierigkeit des Programmierens ist unter anderem der dazu benötigten Denkweise und anderen kognitiven Prozessen geschuldet. Programmieren ist eine Fähigkeit und kein reines Fachwissen. Für die Bearbeitung der beiden ersten Phasen müssen Lernende sich kognitive Fähigkeiten, wie *Problemlösen* (engl. problem-solving) und das in der Literatur häufig genannte *Computational Thinking* aneignen. Bevor diese beiden Themen näher diskutiert werden, ist zuerst der Begriff Problem zu definieren.

### 3.2.1 Probleme und Aufgaben

Laut Duden ist ein Problem eine „schwierige [ungelöste] Aufgabe, schwer zu beantwortende Frage, komplizierte Fragestellung“ [Dud13] oder eine „Schwierigkeit“ [Dud13]. Bei dieser Definition sind die Begriffe *Problem*, *Aufgabe* und *Frage* gleichgestellt. Im

Gegensatz dazu wird im Bereich der Lernpsychologie zwischen Problemen und Aufgaben unterschieden. Dörner beschreibt den Begriff Problem im Kontext des Zustandes einer Person:

> Ein Individuum steht einem Problem gegenüber, wenn es sich in einem inneren oder äußeren Zustand befindet, den es aus irgendwelchen Gründen nicht für wünschenswert hält, aber im Moment nicht über die Mittel verfügt, um den unerwünschten Zustand in den wünschenswerten Zustand zu überführen [Dör79].

Dörner erweitert in ein anderen Definition, unter Betrachtung des Zieles, die Beschreibung um mögliche Lösungsmittel:

> Von Problemen ist [...] die Rede, wenn die Mittel zum Erreichen eines Zieles unbekannt sind oder die bekannten Mittel auf neue Weise zu kombinieren sind, aber auch dann, wenn über das angestrebte Ziel keine klaren Vorstellungen existieren [Dör83].

Diese Beschreibung deckt sich mit Dunckers Auffassung, welche ebenso das zu erreichende Ziel in den Fokus stellt:

> Ein Problem entsteht z.B. dann, wenn ein Lebewesen ein Ziel hat und nicht weiß, wie es dieses Ziel erreichen soll. Wo immer der gegebene Zustand sich nicht durch bloßes Handeln (Ausführen selbstverständlicher Operationen in den erstrebten Zustand überführen lässt, wird das Denken auf den Plan gerufen. Ihm liegt es ob, ein vermittelndes Handeln allererst zu konzipieren) [Dun74].

Diese klassische Definition von Duncker wurde auch später in verschiedenen Varianten aufgefasst (vgl. [Kli76] und [Spa92]) und lässt sich in drei wesentliche Elemente unterteilen:

- Ein unerwünschter Anfangszustand,
- erwünschter Zielzustand
- und eine Hürde, welches die Transformation des Anfangszustandes in den Zielzustand zunächst verhindert.

Die Hürde ist vorerst bedingt durch die fehlenden Mittel oder deren noch unbekannte Kombination zum Lösen des Problems. Alle drei aufgezählten Elemente haben ihre eigenen Schwierigkeiten inne und machen das Problem zu dem, was es ist. Teilweise ist weder der Anfangszustand noch der erwünschte Zielzustand eindeutig, da die Beschreibung des Problems nicht alles erwähnt und situativ Informationen voraussetzt. Nicht alle Probleme sind lösbar bzw. werden in unterschiedliche Komplexitätsklassen

eingeordnet. Nicht-lösbare Probleme werden auch als *np-vollständig* bezeichnet, spielen jedoch im Kontext der Einführungsveranstaltung Programmieren - hinsichtlich unserer Ziele (vgl. Kapitel 1.2) - eine eher untergeordnete Rolle und werden dementsprechend nicht weiter berücksichtigt.

Die Definition aus dem Duden unterscheidet dahingegen nicht zwischen Aufgabe und Problem. Im Gegensatz dazu definiert Dörner den Begriff Aufgabe:

> Bei einer Aufgabe wird lediglich der Einsatz bekannter Mittel auf bekannte Weise zur Erreichung eines klar definierten Ziels gefordert [Dör83].

Weiterhin fasst er Aufgaben auf als „[...] geistige Anforderungen, für deren Bewältigung Methoden bekannt sind. [...] Aufgaben erfordern nur reproduktives Denken, beim Problemlösen muss etwas Neues geschaffen werden“ [Dör79]. Somit ist der Unterschied zwischen Problem und Aufgabe vom jeweiligen Vorwissen eines Individuums abhängig. Nach Duncker erzwingen Probleme neue Lösungswege zu erzeugen, durch neue Mittel oder deren noch unbekannte Kombination. Beispielsweise ist die Multiplikation von 7 mit 4 für die meisten kein Problem, sondern eine Aufgabe, da viele Personen die Rechenregeln zum Lösen bereits kennen und anwenden können. Das bedeutet, dass die gleiche Situation für eine Person ein Problem und für eine andere eine Aufgabe darstellen kann, je nachdem, ob die oben beschriebene Hürde aufgrund von Erfahrung oder Vorwissen fehlt oder wegen fehlender Erfahrung nicht direkt überwunden werden kann.

Im Kontext des Programmierens können daraus mehrere Folgerungen gezogen werden, die später für etwaige Problemlösungsstrategien von Relevanz sind. Probleme sind von Aufgaben durch das jeweilige Vorwissen der lösenden Person abhängig, denn ausgehend vom allgemeinen Problemlösen sind zur Lösung immer Mittel (Operatoren) und Wissen über das Problem notwendig. Probleme liegen auch dann vor, wenn die Mittel zwar bekannt sind, jedoch nicht deren Kombination. Ausgehend von algorithmisch lösbaren Problemen, stellt eben das sinnvolle und zielorientierte Kombinieren von Operatoren - logische Bausteine - die eigentliche Schwierigkeit dar. Mit genügend Vorwissen und Erfahrung können ähnliche Probleme gelöst werden, da die Operatoren und deren Kombination bekannt sind. Dabei bietet es sich an, Probleme zu zerlegen und Ähnlichkeiten zu identifizieren, um zu einer Lösung zu gelangen. Genau dieses zielgerichtete Vorgehen wird im Kontext der Programmierung im nächsten Kapitel näher diskutiert.

### 3.2.2 Problemlösen

Fasst man die oben angeführten Definitionen zum Thema Problem zusammen, so lässt sich Problemlösen als Suche nach der Lösung eines Problems, als nach dem Weg zum

Ziel beschreiben (vgl. [Fun03, S. 21]). Im Bereich des problemlösenden Denkens existieren mehrere Problemlösungsverfahren, welche auch als Heurismen bezeichnet werden. Edelmann [Ede94] teilt diese Verfahren in fünf Formen des problemlösenden Denkens ein:

- durch Versuch und Irrtum,
- durch Umstrukturieren,
- durch Anwendung von Strategien,
- durch Kreativität und
- durch Systemdenken.

Nachfolgend werden stellvertretend drei dieser Verfahren mit Bezug auf das Programmieren näher beschrieben. Diese Auswahl ist in der Art der Verfahren begründet, da diese einfache Lösungsverfahren darstellen mit denen Anfänger erste Schritte zur Problembewältigung durchführen können.

**Versuch und Irrtum** Das Problem wird nicht durch reines Nachdenken gelöst, sondern es wird solange herumprobiert, bis das Ziel und somit die Lösung des Problems erreicht werden. Dieser Heurismus wird meist bei unübersichtlichen Problemen angewendet, bei denen die Informationsfülle andere Lösungen verhindert. Es muss sich dabei jedoch nicht um blindes Ausprobieren handeln, vielmehr wird ansatzweise eine Strategie in Form von schrittweisem Überprüfen von Hypothesen angewendet. Diese Art Probleme zu lösen, kann durchaus verwendet werden, jedoch sollte hierbei verstärktes Gewicht auf die Bildung der Hypothesen gelegt werden.

**Umstrukturieren** Bei manchen Problemen, wie z. B. Textaufgaben beim Programmieren oder auch in der Mathematik befinden sich zahlreiche irrelevanten Informationen. Diese verwirrenden Informationen gilt es zu identifizieren und festzustellen, „[…] worin das Problem eigentlich besteht“ [Ede94]. Nach Edelmann besteht die Lösung „[…] in der Umstrukturierung (Umwandlung) der defekten Struktur in eine gute Struktur“ [Ede94]. Der Prozess der Umstrukturierung umfasst dabei vier Phasen. Nach Dunker (vgl. [Ede94, S. 334ff]) sind diese Phasen

- die Situationsanalyse (Was ist gesucht und was nicht),
- das allgemeine Lösungsprinzip (es wird ein Lösungsprinzip und nicht die endgültige Lösung gefunden),
- Entwicklung eines Suchmodells (Was ist gegeben und was ist brauchbar?) und

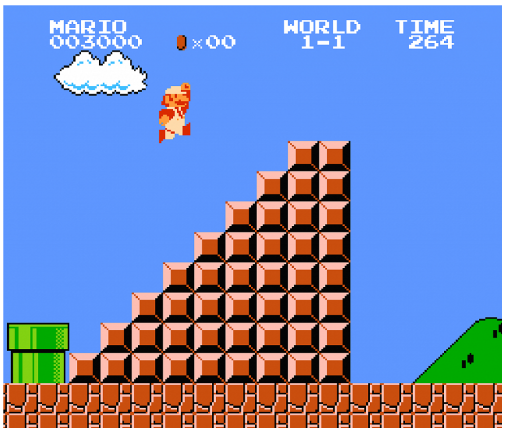
- die Mittelaktualisierung (Anwendung der brauchbaren Mittel führt zur spezifischen Lösung).

Nach Dunker ergeben sich daraus zwei wichtige Aspekte. Erstens geht der Entwicklung der Lösung die Entwicklung des Problems voraus, indem schrittweise die oben genannten Phasen durchlaufen werden. Zweitens können die allgemeinen Lösungsprinzipien auf eine ganze Klasse von Problemen durch Transfer angewandt werden.

Zusammengefasst kann das Problem ggf. leicht gelöst werden, indem der Blickwinkel, unter dem man das Problem betrachtet, geändert wird. Durch diesen Wechsel wird die Überwindung der anfänglichen Hürde deutlich leichter und ist oft beim informatischen Problemlösen der entscheidende Schritt. Besonders die allgemeinen Lösungsprinzipien, die im Kontext des imperativen Programmierens die logischen Bausteine bzw. Konzepte der Programmierung darstellen, sind für das Erlernen von Programmieren und damit für das Lösen von informatischen Problemen wichtig. Schauen wir uns dafür folgende Aufgabe an:

**Beispiel**

Am Ende des ersten Levels (1-1) von *Super Mario Bros.*, muss eine Pyramide erklommen werden, um die Fahnenstange zu erreichen. Überlegen Sie sich eine Lösung, welche die Pyramide (siehe Abbildung 5) in Abhängigkeit von der Höhe auf der Konsole ausgibt. Verwenden Sie das Zeichen # als einen einzelnen Block.




```
Höhe eingeben:6
##
####
######
########
##########
############
```

**Abbildung 5**: Super Mario Bros. - Spielausschnitt Welt: 1-1 und falsche rechtsbündige Lösung

Bei der Lösung mittels *Versuch und Irrtum* kann zuerst versucht werden, in einer Schleife erst zwei, anschließend drei und usw. Blöcke pro Zeile auszugeben. Jedoch, hat dies zur Folge, dass die Pyramide nicht rechts-, sondern linksorientiert dargestellt

wird. Sobald man sich jedoch die Ausgabe als Matrix vorstellt und auch nicht sichtbare Zeichen ausgegeben werden müssen, um die Blöcke nach *„rechts zu rücken“*, ist die Aufgabe durch die Berechnung der Anzahl der Blöcke bzw. Leerzeichen zu lösen. Ohne Berücksichtigung spezieller Sprachbesonderheiten kann die Aufgabe durch zwei ineinander verschachtelte Iterationen gelöst werden. Diese Kombination kommt häufig vor, wie z. B. beim Traversieren einer Matrix oder zur Lösung von mathematischen Ausdrücken in der Art $\sum \sum x + 2$.

**Anwendung von Strategien** Nach Edelmann bezieht sich Strategie auf Denkvorgänge, „[...] wenn das Denken in einer Abfolge von äußerlich sichtbaren Tätigkeiten und Entscheidungen beobachtet werden kann. Der Begriff der Strategie beinhaltet die Planung und Durchführung eines Gesamtkonzeptes, während der Begriff Taktik die Realisierung der einzelnen Schritte meint“ [Ede94]. Diese Methode des Problemlösens lässt sich gut auf den in Kapitel 3.1 beschriebenen Vorgang beim Programmieren zuordnen. Edelmann beschreibt dabei Strategie als „[...] eine heuristische Regel (Problemlöseverfahren), eine Suchanweisung, die die zu treffenden Entscheidungen in einem gewissen Rahmen festlegt“ [Ede94]. Ein Algorithmus ist dagegen „eine epistemische Regel (Wissen), die die Abfolge ganz bestimmter Handlungsschritte tatsächlich festlegt“ [Ede94]. Der Weg zur Lösung kann teilweise durch Versuch und Irrtum erreicht werden, aber mit wachsender Erfahrung und unter Anwendung der heuristischen Regeln wird die Lösung deutlich schneller erreicht. Bei der Anwendung dieses Verfahrens können unterschiedliche Einzelentscheidungen (Taktiken) im Rahmen eines strategischen Gesamtkonzeptes getroffen werden, welches eine flexible Lösungsfindung erlaubt.

**Programmieren und Problemlösen** Die beschriebenen Heurismen zur Findung einer Lösung, können nun das Programmieren, im Kontext des in Kapitel 3.1 beschriebenen Vorgangs, unterstützen. Für die Lösung mittels Versuch und Irrtum sind besonders die vorausgegangenen Überlegungen, die zu einem Versuch führen wichtig, d. h. die Lehre muss diesen Vorgängen auch entsprechend Gewicht verleihen. Ausgangs- und Zielzustände müssen beim Programmieren durch geeignete Repräsentationen dargestellt werden, um anschließend transformiert werden zu können. Die Repräsentationen sind dabei meist geeignete Datenstrukturen oder Datentypen, die einerseits epistemisches Wissen (Wissen einer Person in der aktuellen Situation) darstellen. Jedoch gleichermaßen ein Teilproblem beim Problemlösen darstellen, da eine falsche Darstellung Einfluss auf die Lösung haben kann. Informatisches Problemlösen ist meist vom Zerlegen eines Problems in kleinere lösbare Teilprobleme bzw. bewältigbare Aufgaben geprägt [FBT15]. Dieses Prinzip wird auch *Teile und Herrsche* (engl. *divide and conquer*) genannt, ohne auf die explizite Anwendung im effizienten Algo-

rithmenentwurf einzugehen. Eine ähnliche Vorgehensweise beschreibt das Problemlösungsverfahren *Mittel-Ziel-Analyse* aus dem Bereich der künstlichen Intelligenz, bei welchem der Zielzustand in Teilziele zerlegt wird, die anschließend mittels verfügbarer Operatoren erfüllt werden können [NS72]. Bei der Zerlegung eines Problems für die Erstellung einer algorithmischen Lösung, die anschließend codiert werde kann, müssen die einzelnen Schritte des Algorithmus ausführbar sein. Somit muss zum Lösen von informatischen Problemen bzw. zum Bewältigen von Aufgaben das Wissen über Ausführbarkeit, Grenzen eines Systems und die damit impliziten Annahmen aus der jeweiligen Domäne - hier die Informatik - bereits vorhanden sein. In der Lehre, besonders im angelsächsischen Bereich, wurde dieses Verständnis und die damit verbundene Denkweise von Jeanette Wing unter dem Begriff *Computational Thinking* geprägt [Win06].

### 3.2.3 Computational Thinking

*Computational Thinking* kann mit informatischen Denken übersetzt werden und ist maßgeblich von den Grenzen der Berechenbarkeit und Mächtigkeit von Berechnungen abhängig und baut auf den fundamentalen Konzepten der Informatik auf. Das bedeutet, dass wir z. B. bei Aussagen über die Schwierigkeit eines Problems, die Leistung des ausführenden Systems, die nutzbaren Ressourcen und mögliche Einschränkungen kennen und einschätzen müssen. Wing beschreibt dieses Denken wie folgt:

> Computational thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent [Jea11].

Der Kern dabei ist die Umsetzbarkeit der Lösung mittels eines informationsverarbeitenden Systems, die aber nicht zwingend in einer Programmiersprache erfolgen muss.

> My interpretation of the words problem and solution is broad. I mean not just mathematically well-defined problems whose solutions are completely analyzable, e.g., a proof, an algorithm, or a program, but also real-world problems whose solutions might be in the form of large, complex software systems. Thus, computational thinking overlaps with logical thinking and systems thinking. It includes algorithmic thinking and parallel thinking, which in turn engage other kinds of thought processes, such as compositional reasoning, pattern matching, procedural thinking, and recursive thinking. Computational thinking is used in the design and analysis of problems and their solutions, broadly interpreted [Jea11].

*Computational Thinking* umfasst mentale Werkzeuge wie Dekomposition, Abstraktion, algorithmischer Entwurf, Generalisierung und Evaluation[Win06; Jea11; Sel15]. Für Wing ist hier die Dekomposition - das Zerlegen eines Problems in kleinere lösbare Bestandteile - wichtig, sie betont dabei das Anwenden des Prinzips *Teile-und-Herrsche* auf beliebige Fachgebiete [Jea11]. Abstraktionsvermögen beschreibt die Fähigkeit relevante und irrelevante Stellen bei einem Problem zu identifizieren, um das Problem besser zu strukturieren und somit mit der Komplexität besser umgehen zu können [Win08]. Die Abstraktion beinhaltet dabei das Verbinden der einzelnen relevanten Stellen mit Blick auf Gesamtlösung. Dabei kann auch die Generalisierung helfen, die es erlaubt übertragbare allgemeine Lösungsprinzipien zu verstehen und anzuwenden. Für Janet Kolodner ist das Wiederverwenden von ähnlichen (Teil-)Lösungen aus verwandten Problem ein Bestandteil des informatischen Denkens [Nat11].

Im Vergleich mit den vorher beschriebenen Problemlöseverfahren, ist informatisches Denken durch die Verbindung dieser Verfahren und fundierter theoretischer Grundlagen der Informatik gezeichnet. Beim Programmieren ist folglich die Bildung von Abstraktionsebenen, die für Maschinen verständlich sind, für die Entwicklung einer Lösung wichtig. Dementsprechend müssen Studierende ein Verständnis für geschickte Abstraktionsebenen und deren Umsetzung entwickeln. Wobei in diesem Kontext das informatische Denken hier zwar Programmieren mit abdeckt, sich jedoch nicht darauf beschränkt. Denn es umfasst ebenso Informatik als Modellieren, wie von Hubwieser vertreten wird.

### 3.2.4 Zwischenfazit - Worauf geachtet werden sollte

Die Definition in Kapitel 3.1 beschreibt bereits eine grundlegende Vorgehensweise beim Programmieren, jedoch entspricht diese nicht immer derjenigen von Studierenden. Eine mögliche Vorgehensweise kombiniert epistemisches Wissen über die zugrundeliegende Funktionsweise einer Maschine mit Problemlösungsverfahren [Dav93]. Ohne solches theoretisches Wissen ist die Problemanalyse sowie das Verständnis der Hürde (in Bezug auf das Problemlösen) und das Ziel nur schwer zu greifen. Das kann dazu führen, dass die Bearbeitung von Programmieraufgaben häufig in Problemlösen durch *Versuch und Irrtum* ohne große Vorüberlegungen endet. Dieses ziellose Ausprobieren kann bei Studierenden schnell in Frustration übergehen und den gesamten Lernprozess behindern. Dementsprechend müssen Studierende von Beginn der Lehrveranstaltung an an ein systematisches Vorgehen herangeführt werden, welches auch den Umgang mit Fehlern beinhaltet. Programmieren und Algorithmen setzen ein streng logisches Denken und Entwickeln voraus, weswegen Anfänger immer wieder Fehler auf der Ebene des Algorithmenentwurfs sowie bei der Codierung machen werden. Jedoch sollte den Studierenden von Beginn an, auch bewusst sein, dass Fehler

auftreten werden und das diese auch behebbar sind. Eine mögliche Unterstützung für die Vermeidung von Fehlern und Heranführung an das Programmieren kann das Lehren von Lösungsschemata für wiederkehrende Probleme darstellen. Dadurch könnten Studierenden anfangs für Probleme ähnlicher Natur gewisse algorithmische Lösungen vermittelt werden, welche von den Studierenden zu einer Gesamtlösung kombiniert werden müssen. Zu einer ähnlichen Schlussfolgerung gelangen Eckerdal und Berglund in einer Studie über notwendige Fähigkeiten zum informatischen Problemlösen [ETB05]. Sie fordern das Lehren von sogenannten *Canonical Procedures*, welche Studierenden bei bestimmten Problemen mehr oder weniger automatisch als Lösung in den Sinn kommen.

> A canonical procedure is a procedure that is more or less automatically triggered by a given problem. This can happen either because the procedure is naturally suggested by the nature of the problem, or because prior training has firmly linked this kind of problem with this procedure. The availability of a canonical procedure enables students to obtain a solution without worrying too much about the mathematical properties of the concepts involved. [ETB05]

Diese sind aus meiner Sicht besonders auf kleinere Teilprobleme anwendbar, wie z.B. Iteration über Elemente zur Bildung der Summe, damit diese für Anfänger nicht zu stark abstrahiert sind und in unterschiedlichen Kontexten angewendet werden können.

Solche Lösungsbausteine für wiederkehrende Probleme werden jedoch Fehler nicht gänzlich verhindern. Programmieren ist ein sukzessiver Prozess und sollte in mehreren Iterationen ablaufen, sodass bereits erste identifizierte Teilprobleme korrekt gelöst und umgesetzt werden. Ein bei Studierenden auftretendes Phänomen ist die komplette Codierung eines möglichen Lösungsalgorithmus, ohne diesen ein einziges Mal zu evaluieren. Bei der ersten Durchführung der Überprüfung treten nun eine Vielzahl an Fehlern auf allen Stufen (vgl. Phase der Umsetzung in Kapitel 3.1) auf. Die logische Natur von Programmiersprachen hat zur Folge, dass anschließend weitere Fehler durch vorherige verursacht werden. Mit der Identifizierung des eigentlichen bzw. ersten Fehlers sind die Studierenden durch die teilweise unklaren und mehreren Fehlermeldungen überfordert, welches auch wieder für eine iterative Entwicklung bzw. iteratives Durchlaufen des Prozesses spricht. Alle der vorgestellten Problemlöseverfahren lassen sich durch Erfahrungen und epistemisches Wissen beschleunigen und unterstützen die Lösungsfindung. Ausgehend von einem imperativen Programmierparadigma sind somit die abstrakten Programmierkonzepte, welche zu Lösungen kombiniert werden können, zu lehren. Der sichere Umgang und das Verständnis dieser Programmierkonzepte haben einen positiven Einfluss auf die in Kapitel 3.2.2 vorgestellten Problemlöseverfahren und die damit verbundenen Schwierigkeiten. Denn die Teilprobleme,

welche aus der Dekomposition resultieren, können meist durch die Anwendung solcher Konzepte bzw. durch den Transfer ähnlicher Probleme gelöst werden. Daraus kann geschlossen werden, dass die systematische Anwendung der Konzepte und die Komposition durch das Prinzip *Teile und Herrsche* einen positiven Einfluss auf das erfolgreiche Lösen von Problemen bzw. Bewältigen von Aufgaben hat. Wenn wir bei der Differenzierung zwischen Problem und Aufgabe bleiben, ist somit das Identifizieren von Teilproblemen, welche für den Studierenden Aufgaben sind, erstrebenswert. Dazu müssen jedoch bei der Vermittlung der Konzepte der problemlösende Charakter und die Komposition adressiert werden, damit Studierende Heuristiken für die Lösungsfindung durch Transfer bzw. Analogien ausbilden. Für das Verständnis der Konzepte ist ein tieferes Wissen über den Ablauf und die Zustandsänderungen innerhalb eines ausführenden Systems notwendig, da dies direkten Einfluss auf die letzte Phase - der Codierung und Verifikation - im postulierten Programmierprozess haben. Dies deckt sich auch mit den Merkmalen des informatischen Denkens, welche ebenfalls theoretische Grundlagen als essenziell für das Lösen von Aufgaben sehen. Die Codierung ist jedoch wiederum abhängig von einer spezifischen Programmiersprache, welche trotz Fokus auf das Problemlösen und Modellieren bzw. Algorithmenentwurf, nicht vernachlässigt werden darf. Syntaxfehler sollten die Verifikation der Lösung bzw. Teillösung nicht behindern. Dies bringt den Schluss nahe eine Sprache auszuwählen, welche eine einfache Grammatik besitzt. Reine Schreibaufgaben sind ebenso nicht zielführend, wie Lopez et al. und Lister et al. bereits untersucht haben [Lis+04; Lop+08; Sim15]. Der Programmablauf inklusive der Zustandsänderungen und das Leseverständnis von Quelltexten, die primär zur Darstellung von Beispielen genutzt werden, sind essenzielle Bestandteile der Programmierausbildung [Sch07b].

Fasst man diese Folgerungen zusammen, sind alle Phasen des in Kapitel 3.1 vorgestellten Prozesses gleich stark zu gewichten, um eine fundierte Grundlagenvermittlung der Programmierung und die intendierten Lehrziele zu erreichen. Zusätzlich sollte die Verwendung von *Musterlösungen* für wiederkehrende bzw. ähnliche Probleme entsprechend vermittelt werden, um Studierenden einen gewissen *Fundus* an algorithmischen Lösungsbausteinen inklusive deren Umsetzung mitzugeben.

## 3.3 Ist Programmierfähigkeit messbar?

Ausgehend von der Beschreibung des Programmierens als iterativer Prozess und aus den weiteren Überlegungen, stellt sich die Frage wie man die Fähigkeit zu Programmieren nun bewerten oder messen kann.

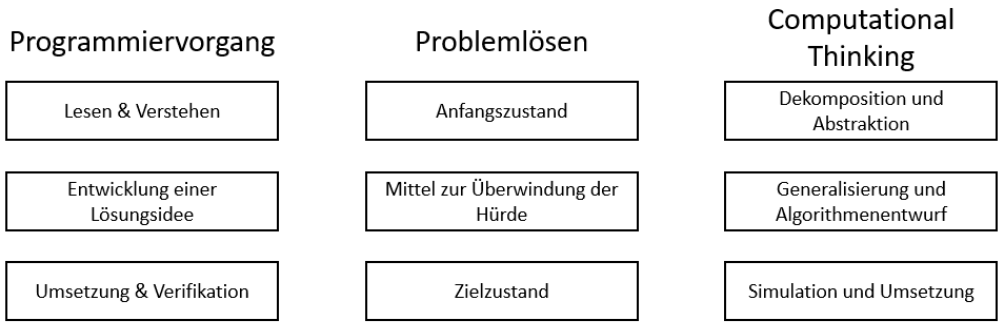



**Abbildung 6**: Zuordnung der Phasen des Programmiervorganges zu den Merkmalen des Problemlösens und informatischen Denkens

### 3.3.1 Programmierfähigkeit

Dazu müssen zuerst die wichtigen Merkmale der Programmierfähigkeit definiert werden, die für eine Operationalisierung eines theoretischen Konzeptes notwendig sind. Nach Bortz et al. legt die Operationalisierung

> [...] eines theoretischen Konzepts bzw. einer latenten Variable [...] fest, anhand welcher beobachtbaren Variablen (Indikatoren) die Ausprägung des theoretischen Konzepts bei den Untersuchungsobjekten festgestellt werden soll. Neben der Auswahl der Indikatoren gehört zur Operationalisierung auch die Festlegung des Messinstruments, mittels derer den Ausprägungen der einzelnen Indikatoren jeweils entsprechende numerische Werte zugeordnet und zu einem Gesamtmesswert für das Konstrukt verrechnet werden [BD16].

Dazu ist eine Nominaldefinition des theoretischen Konzeptes notwendig, welche durch die vorangegangene Diskussion bereits teilweise vorliegt und sich an in der Literatur vorhandenen Konzepten anlehnt. Der beschriebene Programmiervorgang (vgl. Kapitel 3.1) beinhaltet Aspekte aus dem problemlösenden Denken und grundlegendes theoretisches Wissen über die Maschine, die zur späteren Ausführung genutzt wird. Legt man die Merkmale des problemlösenden Denkens auf die einzelnen Phasen des postulierten Programmiervorgangs, ergibt sich die in Abbildung 6 gezeigte Zuordnung.

Für das Verstehen eines Problems ist die Identifizierung des Ausgangszustandes notwendig, die durch Dekomposition und Abstraktion - die Zerlegung des Problems und das Erkennen der relevanten Stellen - durchgeführt werden kann. Die Entwicklung einer Lösungsidee ist im Sinne des Problemlösens das Finden der Mittel zur Überwin-

dung der Hürde, welches die Transformation vom Anfangs- zum Zielzustand verhindert. Mögliche Lösungsprinzipien und Mittel können z. B. durch Generalisierung und programmatisches Problemlösen abgeleitet werden, wobei dazu das Verständnis über den Anfangszustand und die Dekomposition und Abstraktion unerlässlich ist. Zuletzt wird die Umsetzung und Verifikation durch die Codierung und Ausführung des erarbeiteten Algorithmus erreicht. Hierbei ist zu Überprüfen, ob die gewählten Mittel und kombinierten logischen Bausteine (vgl. Ausführung von Schubert in Kapitel 2.1) den Anfangs- in den Zielzustand transformieren. Diese Überprüfung impliziert eine schrittweise und sich an den Zielzustand annähernde Vorgehensweise, welche in Abhängigkeit der Komplexität des Problems ein immer stärkeres Gewicht erhält. Ähnlich ist beim informatischen Denken durch die Dekomposition und der geforderten Anwendung des Prinzips *Teile und Herrsche* der iterative Charakter des Vorgehens verankert.

In Bezug auf die dargelegten Zusammenhänge im vorherigen Absatz, ist Programmieren somit eine Handlungskompetenz, da diese neben dem Wissen auch Fähigkeiten beinhaltet, die nun mittels eines situativen Messinstruments erfasst werden sollen. Dies ist für die Beantwortung der Hypothese 1 über den Zusammenhang des Einsatzes von Aktiven Lernens auf die Programmierfähigkeit notwendig, um die später in Kapitel 7 gesammelten Daten unter anderem mit Ergebnissen aus der Klausur untersuchen zu können. Ausgehend vom problemlösenden Denken und den drei Merkmalen Ausgangslage, Mittel zum Erreichen und Zielzustand kann die Programmierfähigkeit beobachtet und bewertet werden. Bei der Untersuchung der Programmierfähigkeit unter Berücksichtigung unseres Programmierkonzepts können wir eine deduktive Konzeptspezifikation, die Nominaldefinition bzw. Explikation eines theoretischen Konzepts und latenter Merkmale, für die Entwicklung unseres Messinstruments durchführen (vgl. [BD16, S.224ff]). Bevor eine Nominaldefinition für die Programmierfähigkeit dargestellt wird, sollen zuerst bereits vorhandene Ansätze gewürdigt und diskutiert werden.

### 3.3.2 Vorhandene Ansätze

Ein bekannter Ansatz zur Feststellung der Programmierfähigkeit wurde im Rahmen einer Arbeitsgruppe der Konferenz *Innovation and Technology in Computer Science Education* unter Leitung von Michael McCracken entwickelt. Die multinationale Arbeitsgruppe hatte zum Ziel, die Fähigkeiten von Anfängern bei der Lösung eines spezifischen Problems zu erfassen und die Ergebnisse mit den Einschätzungen der Lehrenden zu vergleichen. Das von der Arbeitsgruppe postulierte Konzept basiert auch

auf problemlösendem Denken[8] und wurde in einem iterativen Prozess mit fünf Stufen geordnet (vgl. [McC+01]):

1. Abstraktion des Problems von der Beschreibung
2. Generierung von Teilproblemen (Dekomposition)
3. Transformation der Teilprobleme in Teillösungen
4. Komposition der Teillösungen in eine Gesamtlösung
5. Evaluation und erneutes Durchlaufen

Für die Durchführung der Studie wurde das Instrument, welches im Englischen als *Charette* bezeichnet wird, einer zeitlich begrenzten Entwurfsübung (aus dem Stegreif) gewählt.

> Charettes are short assignments, typically carried out during a fixed-length laboratory session that occurs on a regular basis. The closed nature of these sessions reduces the opportunity for plagiarism. Charettes provide coverage of the learning objectives framework, although in a manner that is more superficial and less cognitively complex than is possible with larger take-home assignments. The experience of completing a charette may not be as meaningful or generalizable as larger assignments. Charettes may be unfair to students who have test anxiety or troubles with time pressure [McC+01].

Dazu wurden drei Aufgaben entwickelt:

- Evaluation von Ausdrücken in Postfix-Notation
- Evaluation von Ausdrücken in Infix-Notation ohne Operatorrangfolge
- Evaluation von Ausdrücken mit Änderung der Auswertungsreihenfolge durch Klammern

Für die Bewertung wurde anschließend eine Indexbildung auf Basis von Punkten in mehreren Kategorien verwendet. Diese umfassen

- die Verifikation des Programms (Liefert das Programm die korrekten Lösungen für Testwerte zurück?),
- die Validierung (Wurde das Programm entsprechend den Anforderungen umgesetzt?),
- Ausführbarkeit (Kann das Programm ohne Fehler ausgeführt werden?)

[8] Die erste Studie wurde bereits im Jahr 2001 durchgeführt und somit vor der Verbreitung des Begriffes *Computational Thinking*.

- und den Programmierstil (Wurde der Programmierstil an der jeweiligen Institution eingehalten?).

Zusätzlich wurde noch ein Interpretationswert verwendet, welcher noch die Korrektheit des Lösungsansatzes mit einbezieht. Die Studie selbst wurde in verschiedenen Sprachen an unterschiedlichen Institutionen durchgeführt, wobei jedoch teilweise unterschiedliche Zeiten für die Bearbeitung gewährt wurden. Ergebnis der Studie ist, dass die Studierenden unabhängig vom Institut schlechte Leistungen zeigten (durchschnittlich 22,89 von 110 Punkten), da viele Studierende keine lauffähige Lösung produzieren konnten. Die Studie wurde unter anderem von McCartney et al. in mehreren Punkten kritisiert (vgl. [McC+13]):

- Zu hohe Schwierigkeit der Aufgaben für 90 Minuten Zeit und keinen Zugriff auf normalerweise verfügbare Hilfsmittel,
- Studierenden bekamen ca. 10 Minuten zum Einlesen und Verstehen der Postfix- und Infix-Notationen,
- Variation der Programmiersprachen,
- ein Institut verwendete die Aufgaben für die Benotung,
- unterschiedliches Vorwissen der Studierenden,
- und mehrere Institute gaben 90 und andere nur 60 Minuten zur Bearbeitung der Aufgaben zur Verfügung.

Um die Ergebnisse aus der Studie zu reproduzieren wurden dieselben Messinstrumente angewendet, jedoch die Rahmenbedingungen leicht abgeändert, um den Kritikpunkten gerecht zu werden. Dazu passte man einerseits die Aufgabenstellung an, stellte andererseits spezielle Klassen und Funktionen zum Einlesen zur Verfügung und gewährte Zugriff auf zusätzliches Material zum Nachschlagen. In diesem gewählten Setting konnten die schlechten Ergebnisse aus der McCracken-Studie nicht nachvollzogen werden, da die Studierenden im Vergleich besser abschnitten. Für eine bessere Aussagekraft wurde ein anderer Bewertungsansatz gewählt, der sich auf drei Aspekte konzentriert:

1. Struktur und Aufbau der Lösung,
2. Dekomposition der Aufgabe,
3. und Verwendung von Variablen und Datentypen.

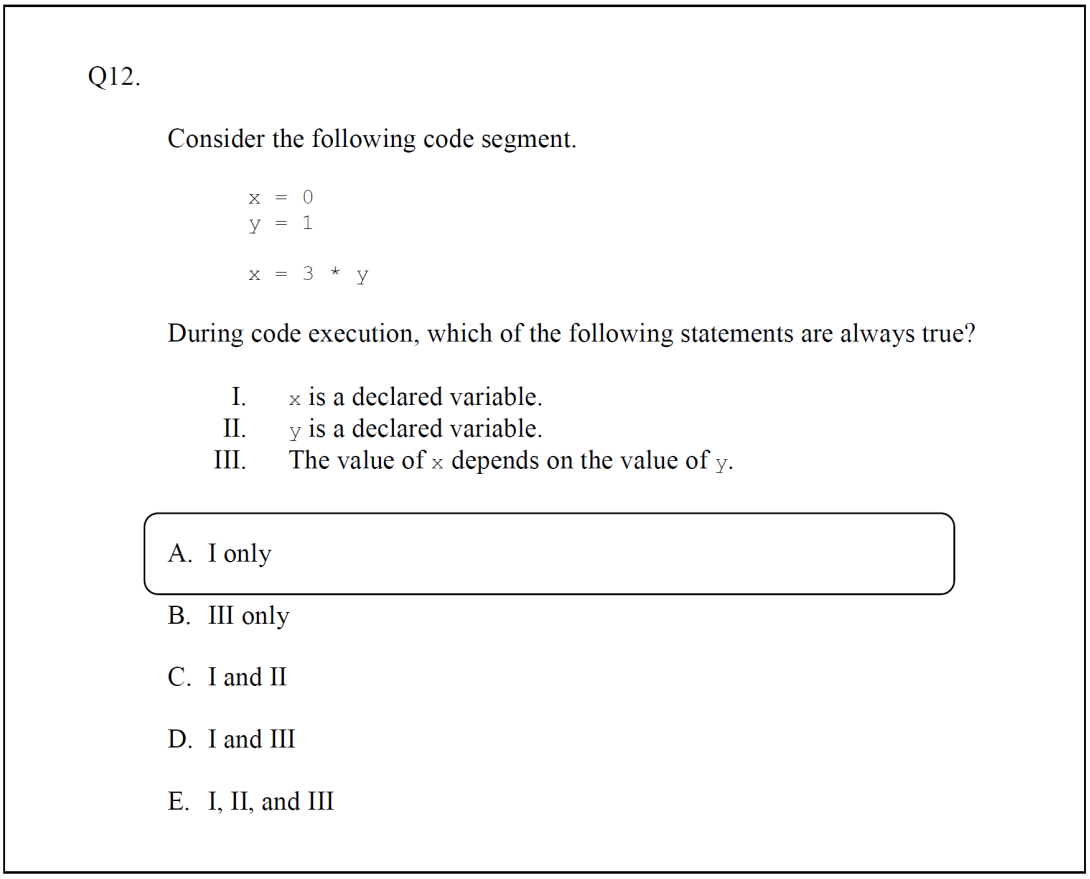
Die Reduzierung auf weniger Indikatoren bei der Bewertung und das Vernachlässigen der Bewertung von Programmierstilen ist ein wichtiger Schritt in Richtung eines besseren Instruments. Alle drei Aspekte sind auch mit der Darlegung des Program-

mierens im Kontext des Problemlösens konform wie bereits in Kapitel Kapitel 3.5 diskutiert. Wichtig bei solchen Studien sind das Vorwissen und die institutionellen Gegebenheiten, welche bei der McCracken-Studie nicht beachtet wurden. Ein anderer Kritikpunkt ist die Validität des gewählten Instruments in der McCracken-Studie, da hier zumindest keine weitere Ableitung und Begründung der gewählten Indikatoren im Sinne einer Operationalisierung stattgefunden hat. Aufgrund der Debatten um die Ergebnisse aus der ersten Studie wurde 2013 in einer neuen Arbeitsgruppe versucht neben den Programmieraufgaben, ein sprachunabhängiges Testinstrument zur Überprüfung des konzeptionellen Verständnisses zu verwenden [Utt+13]. Das Instrument namens *Fundamental CS1* (kurz *FCS1*) wurde zum Erfassen des Verständnisses von Programmierkonzepten unter Verwendung eines sprachunabhängigen Pseudocodes entwickelt [TG10; TG11], um Studien über mehrere Kurse, unabhängig von der gewählten Kurssprache, durchzuführen. Dazu wurden folgende Kernkonzepte für die Überprüfung ausgewählt:

- *Fundamentals* (Variablen, Zuweisungen, Mathematische Ausdrücke),
- logische Operatoren,
- Alternative Anweisung (`if/else`),
- N-malige Iteration (`for`) und Iteration mittels (`while`),
- Arrays,
- Funktionen und Parameterübergabe,
- sowie Rekursion und Objektorientierung.

Die Items des Instruments bestehen aus mehreren gewichteten *Multiple-Choice*-Fragen (siehe Abbildung 7), die zu einem Indexwert berechnet werden. Bei der Bewertung dieses Ansatzes gibt es meiner Meinung nach zwei Kritikpunkte. Das Instrument misst auf Basis von *Multiple-Choice*-Fragen das Verständnis einzelner Konzepte und nicht die Verwendung im Rahmen eines Prozesses zur Problemlösung von Programmieraufgaben. Daneben wurden die Fragen nicht veröffentlicht (außer innerhalb der Teilnehmer der zweiten Studie), da diese weiterhin für Prüfungen intendiert sind. Jedoch wurden in der Studie für die Lehrveranstaltung relevante Konzepte identifiziert, welche ich zur Bildung meines Messinstrumentes verwende. Neben dem FCS1, gibt es noch weitere Ansätze, wie von Decker [Dec07], die Items für die Sprache Java verwenden und einen starken Fokus auf die syntaktische Korrektheit und Verwendung der Sprachkonstrukte legen. Im angelsächsischen Raum existieren noch der *CS Advanced Placement Test* (Java)[9], der *Major Field Test for Computer Science* (Zielt auf das Ende eines Informatikstudiums und verwendet *Multiple-Choice*-Fragen) und der *GRE*

[9] Siehe auch https://apstudent.collegeboard.org/home



Q12.

Consider the following code segment.

```
x = 0
y = 1

x = 3 * y
```

During code execution, which of the following statements are always true?

I. `x` is a declared variable.
II. `y` is a declared variable.
III. The value of `x` depends on the value of `y`.

A. I only

B. III only

C. I and II

D. I and III

E. I, II, and III

**Abbildung 7**: Beispiel für eine *Multiple-Choice*-Frage aus dem FCS1-Instrument

*Computer Science Subject Test* [Edu17] zur Vorhersage des Erfolges für weiterführende Schulen. Schlussendlich gibt es noch den Ansatz von Lister et al.[Lis+04], welcher davon ausgeht, dass die Schwierigkeiten beim Programmieren bereits beim Leseverständnis liegen und darin den Ablauf des Programms auf mehreren Ebenen zu verstehen (im engl. *tracing skills*). Hier wurden auch wiederum *Multiple-Choice*-Fragen zur Messung verwendet, um festzustellen, welche Studierenden Probleme bei theoretischen Inhalten (epistemisches Wissen) und welche Schwierigkeiten beim Problemlösen haben:

> If a student can consistently demonstrate an understanding of existing code, but struggles to write similar programs, then it may be reasonable to conclude that the student lacks the skills for problem-solving. However, if a student cannot consistently demonstrate understanding of existing code, then such a student's difficulty is a lack of knowledge and skills that are a prerequisite for non-trivial, five-step, problem-solving [Lis+04].

Die Studie konnte zeigen, dass den Studierenden meist das theoretische Wissen gefehlt hat, um den von McCracken definierten fünfstufigen Problemlösungsprozess zu durchlaufen. Daneben existieren noch weitere Ansätze, wie z.B. von Parsons et al. [PWH15] mittels Aktivitätsdiagrammen. Studierende müssen zu einem vorgegebenen Beispielquelltext ein Aktivitätsdiagramm erstellen, was aus meiner Sicht jedoch eher mit dem Ansatz von Lister et al. vergleichbar ist, da hier primär das Leseverständnis und der Programmablauf überprüft werden und nicht das Problemlösen.

### 3.3.3 Messung mittels Indikatoren

#### Additive Indexbildung bei Programmierkonzepten

Das im Rahmen dieser Arbeit erarbeitete Konzept zur Förderung der Anwendung und Kombination logischer Bausteine erfordert einen neuen Ansatz. Die im vorherigen Kapitel 3.3.2 vorgestellten Instrumente zur Überprüfung der Programmierfähigkeit überprüfen alle unterschiedliche Konzeptspezifikationen bzw. versuchen ein übergreifendes Messinstrument zu entwickeln. Der Fokus dieser Arbeit und die spätere Evaluation liegen jedoch auf der Kombination und Verwendung von Programmierkonzepten in Programmieraufgaben. Weiterhin sollte das Instrument auf schriftliche Prüfungen anwendbar sein, um auch eine summative Evaluation durchführen zu können, da das entwickelte Konzept über die komplette Lehrveranstaltung und nicht nur für einzelne Lehr-/Lernarrangements angewendet wurde.

Aus dem Programmierprozess im Kontext des problemlösenden Denkens sollen nun Indikatoren zur Überprüfung des Einsatzes von Programmierkonzeptes abgeleitet wer-

den. Um auch in Zukunft Aussagen in weiteren Studien treffen zu können und ggf. ältere Prüfungen zu bewerten, können keine neuen Aufgabentypen, wie z. B. im Rahmen des FCS1 und von Lister et al., verwendet werden. Stattdessen sollen kurze Programmieraufgaben, welche zur Lösung die Verwendung mehrerer logischer Bausteine und das Durchlaufen des Programmierprozesses erfordern, als Messinstrument dienen. Nun leite ich die Indikatoren aus den Phasen des Programmierens ab, wobei ähnlich zu McCracken das Ziel eine Indexbildung zur Durchführung von Vergleichen ist. Ich folge hierbei der Definition von Bortz nach dem eine Messung „[…] in der quantitativen Sozialforschung eine Zuordnung von Zahlen zu Objekten oder Ereignissen [ist], sofern diese Zuordnung eine homomorphe (strukturerhaltende) Abbildung eines empirischen Relativs in ein numerisches Relativ ist." [BD16]

Bei der Prozessbeschreibung sind die Analyse und Auswahl eines geeigneten Mittels zum Erreichen des Zielzustandes enthalten, welche zur Operationalisierung verwendet werden. Das in Kapitel 5 vorgestellte Konzept versucht, durch die gezielte Einführung von Programmierkonzepten mittels Bausteinen, problemlösendes Denken beim Programmieren zu vermitteln. Bei der Betrachtung einer Lösung können aus der Verwendung und der Kombination der Bausteine, Rückschlüsse auf das Verständnis des Problems und die Dekomposition gezogen werden. Die Zerlegung in Teilprobleme, welche im besten Fall in Teillösungen resultiert, lässt sich anhand der verwendeten Programmierkonzepte und deren Parametrisierung ableiten. Das kann auch als Ziel-Plan-Überprüfung bezeichnet werden, da die verwendeten Konzepte zur Lösung den algorithmischen Entwurf (der Plan) widerspiegeln. Programmieren auf Papier erlaubt es Studierenden jedoch nicht die Verifikation durch Ausführung des Programms anzuwenden und somit den kompletten Prozess wie am PC zu durchlaufen Zum Lösen von Problemen und deren Umsetzung ist eine deutlich stärkere Gewichtung auf die ersten beiden Phasen im Prozess zu legen. Die Aufgaben an und für sich sollten im Umfang des zu produzierenden Quelltextes reduziert werden, sodass ein Problem seitens der Studierenden sofort überblickt und im Kopf *ausgeführt* werden kann. Dieser Unterschied muss bei der Messung berücksichtigt werden, indem z. B. syntaktische Fehler nicht in die Bewertung mit einfließen. Daneben ist die Bewertung eines Programmierstils ebenso vernachlässigbar, da besonders in einer ersten Lehrveranstaltung zum Thema Programmieren, die Zielsetzung nicht darin besteht, dass Studierende perfekte Lösungen produzieren, sondern vielmehr mit den Konzepten, dem Programmablauf und dem Problemlösen und damit verbundenen Prozessen zurechtkommen.

Zur Überprüfung wähle ich den Ansatz der Messung mittels Indexbildung, um dieses mehrdimensionale theoretische Konstrukt der Programmierfähigkeit zu erfassen und mehrere Studierende miteinander zu vergleichen und ggf. mit weiteren Informationen in Bezug setzen zu können. Um die Mittel zum Erreichen des Ziels im Sinne der Aufgabenstellung zu erfassen, wird ein nicht gewichteter additiver Index pro Auf-

gabe verwendet, welcher die Kombination und Anwendung einzelner Programmierkonzepte mittels dichotomer Einzelindikatoren erfasst. Darunter fallen die gewählten Konzepte, welche zu bestimmten relevanten Aufgabenstellen ausgewählt wurden, die Anwendung des Konzeptes durch z. B. gewählt Start- und Endwerte bei einer Iteration und die gewählte Reihenfolge bei der Komposition einer Gesamtlösung. Die McCracken-Studie hat zwar auch einen Interpretationswert erhoben, jedoch wurde dieser nur auf einer Ordinalskala (1-5) durch Entscheidung des jeweiligen Lehrenden gemessen, der die Nähe zu einer korrekten Prüfung widerspiegeln soll. Dadurch ist dieser Wert nur teilweise reliabel, da es sich um eine Einschätzung über die gesamte Lösung in Bezug auf die Aufgabenstellung handelt und somit probiert wird mittels eines Einzelindikators ein komplexes Konzept zu überprüfen. Im Gegensatz wurde des im Rahmen dieser Arbeit gewählten Ansatz eine additive Indexbildung durch multiple Indikatoren entwickelt.

**Konzeptauswahl** Programmieraufgaben können, ohne die effizienteste Lösung zu fordern, durch mehrere unterschiedliche Lösungsalgorithmen erfolgreich bearbeitet werden. Das bedeutet, dass z. B. die Kombination von einzelnen Bausteinen unterschiedlich sein kann, die auszuwählenden Bausteine bei einer geschickten Aufgabenstellung jedoch nicht. Kreative Lösungen dürfen und sollen nicht, aufgrund des Verlassens von vorgesehenen Wegen zur Lösung, schlechter bewertet werden. Das beeinflusst teilweise auch die Erstellung von Aufgaben, auf welche dieses Instrument später angewendet wird. Eine geschickte Erstellung von Problemstellungen erlaubt es jedoch Einfluss auf die mögliche Auswahl zu nehmen. Bei der Dekomposition müssen Studierende wichtige Stellen und anschließend deren Lösung erarbeiten. Daraus kann das die erste dichotome Indikator *gK* abgeleitet werden, welcher die Auswahl des richtigen Konzeptes in einer bestimmten Aufgabenstellung abbildet.

**Stelle der Anwendung eines Konstruktes** Der nächste Indikator basiert auf der Annahme, dass bei der Komposition die Reihenfolge der einzelnen Bausteine und somit Teillösungen im algorithmischen Entwurf eine Rolle spielen. Damit wird die Reihenfolge, welche im Idealfall einen korrekt transformierten Zielzustand erzeugt, bewertet. Die Korrektheit der Stelle *KS* (auch dichotom) kann nur im Kontext der restlichen Bausteine bewertet werden, jedoch werden dadurch auch mehrere Lösungsvarianten mit abgedeckt.

**Verwendung des Konstruktes** Schlussendlich, fließt noch die richtige Verwendung des Konzeptes im Abhängigkeit zur Aufgabenstellung ein. Dies kann entweder ein richtiger Boolescher Ausdruck für eine Bedingung darstellen, oder auch die Start-

und Endwerte für eine Iteration. Zusätzlich muss hierbei auf Nebeneffekte bei der Verwendung der Konzepte geachtet werden, sodass z. B. Zählvariablen außerhalb einer Schleife bereits korrekt initialisiert werden, da dies Rückschluss auf das Verständnis des Programmablaufes und die damit verbundenen Zustandsänderungen gibt. Dieser Faktor *VK*x* ist abhängig von der jeweiligen Aufgabe und dem zugehörigen Konzept zu gewichten, um auch komplexe Verwendungen von Konzepten abdecken zu können.

**Indexbildung** Für die Bewertung einer Aufgabe, können anschließend die notwendigen Bausteine, welche den Anfangszustand in den Zielzustand transformieren, einzeln bewertet und anschließend summiert werden. Daraus ergibt sich ein von der Aufgabe abhängiger Index für die Programmierfähigkeit $i_{PF}$:

$$i_{PF} = \sum_{konzepte} (Wert_{Konzeptauswahl} + Wert_{Stelle} + Wert_{Verwendung} \times x_{Parametergewichtung})$$

### Weitere Überlegungen

Bei der Auswahl einer oder mehrerer Aufgaben, auf die das Messinstrument angewendet werden soll, müssen mehrere Aspekte bedacht werden. Das Aufgabeniveau hängt von den gemachten Erfahrungen und den zur Verfügung stehenden Informationen ab, die während der Bewältigung bzw. Lösung zur Verfügung stehen. Erstens sollen die Aufgaben während der Klausur bearbeitet werden, was zumindest in einem gewissen Maße die Einzelleistung sicherstellt. Zweitens sollte Studierenden ausreichend Informationsmaterial zum Nachschlagen und Adaptieren, durch z. B. Generalisierung und Lösungsprinzipien, zur Verfügung stehen, da dies zumindest teilweise den realen Programmierprozess nahe kommt, da beim Programmieren am PC sofort in der Referenz oder im Vorlesungsmanuskript nachgeschlagen werden kann. Weiterhin müssen die Probleme mit den in der Lehrveranstaltung vorgestellten Konzepten zu lösen sein, um negative Effekte wie in der McCracken-Studie zu vermeiden. Denn in den in der Studie gestellten Aufgaben, wurde implizit die Funktionsweise von *Stacks* bzw. Kellerautomaten vorausgesetzt, welche nicht überall gleich behandelt wurden. Zusätzlich waren alle drei Aufgaben stark mathematisch orientiert. Daraus kann gefolgert werden, dass (a) die Aufgaben entsprechend dem Vorwissen der Studierenden ausgewählt werden müssen, (b) eine Varianz innerhalb der Themen der Kurzaufgaben den Transfer der Konzepte besser abdeckt und (c) die Aufgabenstellung die Verwendung der abzuprüfenden Konzepte und deren Kombination erfordert.

### Beispiel

Zur Verdeutlichung soll die Indexbildung auf ein einfaches Beispiel angewendet werden, um die einzelnen Elemente in einem spezifischen Kontext zu beschreiben:

**Beispiel - Indexbildung**

Entwerfen Sie ein Programm, welches zuerst eine ganze Zahl *n* einliest und anschließend das Produkt $\prod_{i=1}^{n} i$ bildet. Geben Sie das Ergebnis am Ende auf der Konsole aus.

Ausgehend von der Kategorisierung von Tew (vgl. FCS1 Kapitel 3.3.2) sind für die Lösung der Aufgabe mehrere Konzepte notwendig:

- Variablen: Zur Repräsentation und Speicherung der Eingabewerte und Ergebnisse
- Ein- und Ausgabe von Daten
- Iteration mit vorab bekannter Anzahl der Durchläufe

Die Aufgabenbeschreibung enthält bereits mehrere Schlüsselwörter, die bei der Problemanalyse erkannt werden müssen:

- „ganze Zahl“: Ist ein Indikator für den Datentyp und Einsatz von Variablen.
- „einliest“: Es muss eine Zahl eingegeben werden.
- „Produkt ... bildet“: Deutet auf eine Wiederholung/Schleife/Iteration hin.
- „Geben ... auf der Konsole aus“: Das Ergebnis muss ausgegeben werden.

Diese vier Stellen eigenen sich ebenfalls zur Zerlegung des Problems in mehrere Teilprobleme, die zwar zeitlich aufeinander angewiesen sind, jedoch trotzdem iterativ gelöst werden können. Bei der Verwendung des Instruments möchte ich mich an dieser Stelle auf die Verwendung der Schleife beschränken, da die Stelle des Konzeptes bereits durch die Aufgabenstellung vorgegeben ist.

Für die Umsetzung des Produkts muss zuerst ein geeignetes Programmierkonzept ausgewählt werden, welches zugleich den ersten Teil des Indexes darstellt. Das Produkt wird über alle Werte von $[1..n]$ gebildet, denen der Einsatz der Iteration nahe liegt. Genauer handelt es sich hierbei um eine Iteration mit vorab bekannter Anzahl an Durchläufen. Bei der Umsetzung in einer konkreten Sprache, in diesem Fall Python, kann nun zwischen einer `for` und einer `while` Schleife gewählt werden. Sobald die Anzahl der Durchläufe bekannt ist, eignet sich in diesem Fall der Einsatz der `for` Schleife, da diese exakt für solche Fälle konzipiert ist und bei der Parametrisierung (Verwendung

**Tabelle 3**: Beispiel zur Anwendung der Indexbildung

| Wert | Punkte | Beschreibung |
|---|---|---|
| Konzeptauswahl | 1 | Iteration mit bekannter Anzahl der Durchläufe: `for` |
| Stelle des Konzeptes | 1 | Die Produktbildung muss nach der Eingabe, aber vor der Ausgabe erfolgen. |
| Verwendung des Konstruktes | 1 | Richtige Auswahl der Start- und Stopwerte |
| | 1 | Initialisierung der Ergebnisvariable außerhalb der Schleife mit dem Wert *1* |
| Gesamt | 4 | |

des Konstruktes) einen Start- und einen Endwert erwartet. In Python durchläuft die `for` Schleife immer eine Folge an Elementen bzw. Werten, weswegen die `range(start, stop)` Funktion zum Erstellen der Folge genutzt werden muss. Dabei muss auf das exklusive Ende des Stop-Parameters geachtet werden, um die gewünschte Anzahl an Iterationen zu durchlaufen. Eine Besonderheit der Verwendung von Schleifen sind die Zustandsänderungen und Variablenzugriffe innerhalb des Schleifenrumpfes, weshalb die Initialisierung der Ergebnisvariable zwingend vor der Schleife erfolgen muss. Die Art der Initialisierung und Namensgebung der Variablen ist an dieser Stelle irrelevant. Es wird hier vielmehr das Verständnis von Schleifen auf den Programmzustand und Ablauf bewertet. Bei einer `while` Schleife müssen z. B. Zähl- oder Statusvariablen ebenfalls vor dem Durchlaufen der Schleife definiert bzw. initialisiert werden. Zusammenfassend können bei dieser Aufgabe der Index über 3 Variablen mit maximal 4 Punkten, wie in Tabelle 3 dargestellt, gebildet werden.

Eine mögliche korrekte Lösung kann anschließend wie in Abbildung 8 aus den einzelnen Teilproblemen und Teillösungen (1 bis 3) zusammengesetzt werden. In dieser Lösung sind keine Syntaxfehler enthalten, jedoch treten diese insbesondere in schriftlichen Prüfungen immer wieder auf. Um jedoch die mangelnde Unterstützung durch die Werkzeuge beim Lösen von Problemen bzw. Aufgaben in der Klausur auszugleichen, werden Syntaxfehler nicht bewertet. Ein fehlender Doppelpunkt oder eine nicht akkurate Einrückung fließt nicht in die Bewertung ein, jedoch sollten die Fehler keine logischen Auswirkungen in Bezug auf den Programmablauf haben. Darunter zählt unter anderem komplett fehlende Einrückung, die dann durch die variable *Stelle des Konzeptes* abgewertet wird, oder das Vertauschen von `false` und `False` (richtig) in Python. Die erste Schreibweise erstellt bzw. verwendet eine neue Variable namens

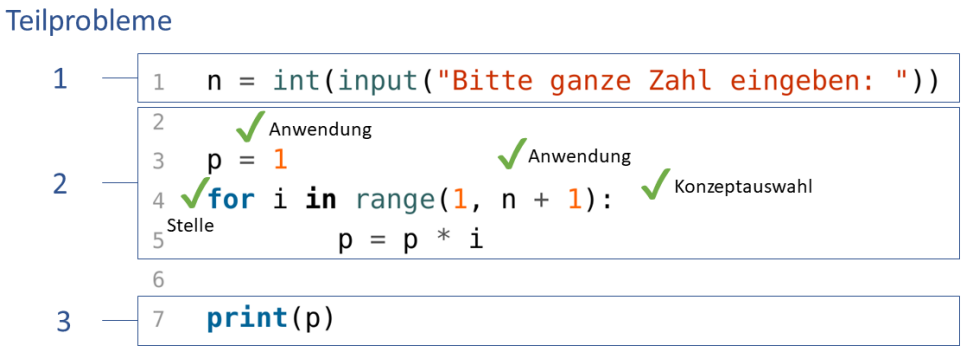



**Abbildung 8**: Bewertete Lösung mittels Indexbildung

**Tabelle 4**: Vollständige Anwendung der Indexbildung

| Konzept | Beschreibung | Punkte |
|---|---|---|
| Eingabe | Ganzzahl einlesen und Typkonvertierung durchführen | 3 |
| Produktbildung | Iteration mittels `for` | 4 |
| Ausgabe | Am Ende | 3 |
| Gesamt | | 10 |

`false` und nicht die Konstante für den Booleschen Wert *falsch*. Die in diesem Fall eher pedantische Bewertung hat dabei ihre Berechtigung, da Studierende sich über die Unterschiede der Datentypen, Konstanten und Variablen bewusst sein müssen. Ausgehend vom informatischen Problemlösen und der bisherigen Diskussion über das Programmieren, gehören die Unterschiede zu den theoretischen Grundlagen bzw. die Verwendung von Variablen selbst ist bereits ein fundamentales Konzept.

Wenden wir das Bewertungsschema ebenfalls auf die fundamentalen Konzepte wie die Ein- und Ausgabe an, so ergibt sich folgende Indexbildung in Tabelle 4. Die Anwendung auf fundamentale Konzepte erhöht die maximal erreichbare Punktzahl, lässt jedoch Freiraum, um kleine Fehler wie die Ausgabe im falschen Format oder die Verwendung eines falschen Typs bei der Eingabe zu berücksichtigen.

### Anmerkungen

Bei der beispielhaften Verwendung des Instruments wird bereits klar, dass dieses nicht ohne eindeutige und nachvollziehbare Regeln angewendet werden kann. Diese müs-

sen auf die jeweiligen Aufgaben angepasst (z. B. in Abhängigkeit vom Schwierigkeitsgrad) und vom Bewertenden eindeutig festgelegt werden. Der additive Index kann je nach Festlegung der Teilprobleme und dazugehörigen Konzepte an die Schwierigkeit der Aufgaben angepasst werden. Des Weiteren lässt sich zugleich nur die Verwendung bestimmter Konzepte bewerten, da der richtige Programmablauf durch die variable *Stelle des Konzepts* mit einfließt. Die funktionale Korrektheit steht bei diesem Bewertungsschema nicht direkt im Mittelpunkt, sondern die Auswahl, Verwendung und Kombination der Konzepte nach einer wählbaren Granularität, die abhängig von dem jeweiligen Vorwissen festgelegt werden sollte. Zusammengefasst ist das dargestellte Messinstrument eine Möglichkeit das informatische Problemlösen, durch die Kombination von logischen Bausteinen (Konzepten), quantitativ zu erfassen. Die Bewertung und Modellierung größerer Informatiksysteme wird nicht über dieses Instrument abgedeckt, wobei dies nicht intendiert ist. Dieses Instrument ist somit nur für diese spezielle Sichtweise nutzbar und eignet sich nicht für jeden Anwendungsfall, wo z. B. die Gewichtung der Bewertung auf der syntaktischen Korrektheit liegt. Ebenfalls ist ggf. bei der Bewertung von Klausuren die mögliche feingranulare Bepunktung nicht immer erwünscht, da unter Umständen bei der Bewertung ein Konzept, wie z. B. die Eingabe oder die Verwendung von Variablen, nur in Verbindung mit anderen bewertet wird.

## 3.4 Lehr- und Lerntheoretische Überlegungen

Nachdem meine Definition von Programmieren neben dem fachlichen Wissen auch die informatische Problemlösestrategien verbindet, möchte ich die für meine Arbeit zugrunde gelegte Lerntheorie bzw. Erkenntnistheorie vorstellen und diskutieren. Die Diskussion bewegt sich von den fachdidaktischen Überlegungen Programmieren zu den intendierten Lernprozessen hin. Ich verwende dazu die lernpsychologische Theorie des gemäßigten Konstruktivismus und leite daraus Leitideen für die Gestaltung und Konzeption der Lehrveranstaltung unter dem Gesichtspunkt der aktivierenden Lehre bzw. des aktiven Lernens ab.

### 3.4.1 Konstruktivistische Theorie

Der Konstruktivismus ist eine Weiterentwicklung des Kognitivismus, die sich deutlich vom Behaviorismus differenziert, weswegen ich zuerst den Kognitivismus kurz erläutere. Nach der kognitiven Lernforschung reicht die reine Reiz-Reaktions-Erklärung, wie sie im Behaviorismus (die äußeren Bedingungen des Lernens) [BS08; Ede94] angenommen wird, nicht aus. Stattdessen ist das Bindeglied zwischen dem Reiz und

der Reaktion von Interesse und somit der Untersuchungsgegenstand [Mie08]. Zusammengefasst kann man sagen, „[…] dass der Kognitivismus sich mit den Themen Wahrnehmung, Problemlösen durch Einsicht, Entscheidungsprozesse, Informationsverarbeitung und Verständnis auseinandersetzt“ [BS08]. Dabei wird kognitives Lernen als Informationsaufnahme und -verarbeitung aufgefasst, aus der sich nach Edelmann [Ede94] zwei Merkmale ableiten lassen: 1. Die jeweilige Person ist im Lernprozess aktiv beteiligt. 2. Das Resultat dieses Prozesses sind Strukturen und keine einfachen Verbindungen zwischen Reiz und Reaktion.

Basierend auf diesen Annahmen, wurde eine Reihe weiterer Ansätze (z. B. Ausubel, Neisser und Anderson) entwickelt. Dazu zählt auch der von Bruner geprägte Ansatz des *entdeckenden Lernens*, der davon ausgeht, „[…] dass es unmöglich ist, einem jungen Menschen alles das beizubringen, was er braucht, um im späteren Leben erfolgreich zu werden“ [BS08]. Daraus schlussfolgert er, dass es darauf ankommt die Fähigkeiten zu vermitteln, welche Menschen das selbstständige lösen von Problemen erlaubt [Bru72]. Aus Schwills Konzept der fundamentalen Ideen (vgl. Kapitel 2.1), leitet sich ab, dass zum Problemlösen in Domänen immer spezifisches Wissen dazugehört. Diese Kernideen stellen das notwendige Vorwissen zum Lösen der Probleme dar, da dieses das Überwinden der Hürde erleichtert (vgl. Kapitel 3.2.1). In diesem Bezug gleichen sich das *informatische Denken* und damit ebenfalls die in Kapitel 3.1 dargelegte Auffassung über das Programmieren, welches epistemisches Wissen mit Fähigkeiten zum Problemlösen verbindet und somit auch die kognitiven Prozesse bestärkt.

Eine mögliche Beschreibung der konstruktivistischen Sicht in der Lernforschung besagt Folgendes:

> Lernen auf Basis des Konstruktivismus bedeutet, dass Wissen durch eine interne und subjektive Konstruktion von Ideen und Konzepten entsteht [BS08].

Dementsprechend wird das Primat der Konstruktion anstelle der Instruktion gefordert [BS08; Hub07]. Jedoch gibt es nicht nur eine konstruktivistische Sicht, sondern eine Vielzahl an Konstruktivismen bzw. konstruktivistische Strömungen wie z. B. der radikale oder der soziale Konstruktivismus. In letzter Zeit gewinnt eine eher gemäßigte Auffassung des Konstruktivismus mehr Einfluss, welche die instruktionsbasierten Lehrmethoden nicht ablehnt [Tib11, S. 309]. Lernen kann im konstruktivistischen Sinne als Aufbau von Wissensnetzen bezeichnet werden, wobei Wissen nicht nur abrufbare Informationen darstellt, sondern auch einen subjektiven Erkenntnisprozess - also die Konstruktion des Subjekts [Sie05]. Dies bedeutet aber auch, dass konstruktives Lernen die Öffnung für Neues, Unbekanntes oder Irritierendes erfordert. Deswegen ist in der konstruktivistischen Pädagogik auch die Rede von *Perturbationen*, denn „[…] lernfähig ist, wer sich stören, irritieren, verunsichern lässt. Wer alles weiß (oder

zu wissen glaubt), ist nicht mehr lernfähig“ [Sie05]. Solche Perturbationen werden durch die Erfahrung bzw. Wahrnehmung von Unterschieden und Unterscheidungen erzeugt. Konstruktionen finden jedoch nur statt, wenn die Handlungen *viabel*, individuell und gesellschaftlich (soziale Komponente) nützlich bzw. relevant, sind [Tib11, vgl. hier vor allem S. 144f]. Handlungen sind mehr als reines *Tun mit der Hand*, sondern eine Wechselwirkung von kognitiven Prozessen und Aktivitäten gegenüber der Umwelt mit einem intendierten Ziel [Ede94, S.306f; BS08, S.208ff].

Bei dieser Diskussion stellt sich die Frage, was Konstruktion anstelle von Instruktion bei der Gestaltung von Lernprozessen bedeutet? Hubwieser [Hub07] sowie Brinker et al. [BS08] beschreiben die Prozessmerkmale wie folgt:

1. Zum Lernen ist die aktive Beteiligung des Lernenden notwendig.
2. Der Lernende übernimmt Steuerungs- und Kontrollprozesse, wobei diese situativ variieren können.
3. Lernen ist konstruktiv und ohne individuelle Erfahrungen und epistemisches Wissen finden keine kognitiven Prozesse statt.
4. Lernen findet in spezifischen Kontexten statt, weswegen jeder Lernprozess auch situativ ist.
5. Aus dem sozialen Konstruktivismus heraus ist Lernen immer ein sozialer Prozess und ein interaktives Geschehen.

Das hat zur Konsequenz, dass der Lehrende eher die Rolle eines Trainers hat, der eher im Hintergrund bleibt während die Lernenden - je nach konstruktivistischer Auffassung - Lerninhalte selbst wählen und konstruieren können. Dies kann jedoch in Beliebigkeit der Inhalte, die nicht im Sinne der Ausbildung sind, abdriften [BS08] bzw. im Sinne des radikalen Konstruktivismus „[...] die Abschaffung jedes institutionalisierten Unterrichts“ [Tib11] bedeuten. Tiberius folgert daraus, dass der Lehrende „[a]ls Korrektiv unangemessener Konstruktionen [...] nicht entbehrlich“ [Tib11] ist. Dies ist jedoch weder mit den in Kapitel 2.1 vorgestellten Zielen noch mit einer standardisierten bzw. vergleichbaren Hochschulausbildung im Sinne des Bolognaprozesses vereinbar oder überfordert die Lernenden aufgrund völliger Freiheit im Lernprozess. Somit ist eine reine konstruktivistische Ausrichtung der Lehre nicht möglich, jedoch können für bestimmte Situationen die Merkmale zum Entwurf von Lernumgebungen genutzt werden. Ebenso sind die Forderungen nach der aktiven Beteiligung bereits in reformpädagogischen Bewegungen (vgl. Konzept nach Kerscheinsteiner) [Tib11; BS08], wie z. B. *Lernen am Projekt* oder Wagenscheins *Epochenunterricht*, aufgekommen. Diese fordern ebenfalls, dass „[b]eim Lernen [...] der Lernende selbstständig tätig sein und mit authentischen Situationen konfrontiert werden“ [BS08] muss.

Zusammengefasst können Lehrende kognitives (geprägtes) Lernen durch aktives Handeln und Denken ermöglichen. Daraus kann, wie Brinker darstellt, der Grundsatz der Handlungsorientierung - eine konkrete Situation - und somit auch der handlungsorientierte Unterricht abgeleitet werden [BS08].

Betrachtet man diese Forderungen im Kontext der aktuellen Situation an Hochschulen mit über 100 Studierenden in einem Vorlesungssaal, die schweigend den Ausführungen des Lehrenden folgen (wenn überhaupt), dann ist eine Änderung dieser Situation notwendig. Dies deckt sich mit der in der Problemstellung (vgl. Kapitel 2) aufgestellten Forderung nach einem seminaristisch geprägten Unterricht, der interaktive Lernprozesse fördert. Dies kann auch durch die Kombination von instruktionellen und konstruktionellen (mit einer teilweisen und situativ passenden Selbststeuerung) Lehrmethoden geschehen, die in Abhängigkeit der jeweiligen Ziele gewählt werden können (vgl. hier [BHT07, S. 360f]). In diesen Überlegungen ist auch die Wahl des *Aktiven Lernens* als konzeptionelle Basis des in Kapitel 5 beschriebenen Konzepts und die dafür entwickelte Plattform (vgl. Kapitel 6), begründet. Jedoch, bevor ich näher auf das Aktive Lernen und Möglichkeiten für die Programmierausbildung eingehe, möchte ich noch einige wichtige Lehr-Lernmethoden, die auf den Ideen des kognitiven Lernens aufbauen und mein selbst entwickeltes Konzept prägen, vorstellen. An dieser Stelle sollte noch die von Straka [SM09] entwickelte lehr- und lernorientierte Didaktik erwähnt werden. Sein Konzept geht dabei von einem mehrdimensionalen Handlungsbegriff und stellt die Notwendigkeit der Interaktion und Kommunikation beim Lehren und Lernen, abgeleitet aus theoretischen Ansätzen, dar. Im Zuge der Diskussion wird einerseits argumentiert, dass handlungsorientierte Ansätze wie das *Cognitive Apprenticeship* einen stärkeren Praxisbezug in die Lehre bringen können. Gleichzeitig wird davor gewarnt, dass diese Methode zu einer zu starken Fokussierung auf den Lehrenden führen kann und nicht auf die Vermittlung der relevanten Inhalte. Im Rahmen dieser Arbeit wird ein Konzept, welches an das *Cognitive Apprenticeship* angelehnt ist, entwickelt.

### 3.4.2 Cognitive Apprenticeship

In den vorherigen Kapiteln (vgl. Kapitel 3.1 und Kapitel 3.2.2) wurde bereits dargelegt, dass Programmieren die Kombination von Fachwissen und Fähigkeiten beinhaltet. Besonders für das Problemlösen sind vorherige Erfahrungen und daraus gewonnene Erkenntnisse wichtig, da diese einen positiven Einfluss auf die nötigen Schritte zur Lösungsentwicklung haben. Aus dem kognitiv geprägten Lernen sind weitere Methoden zur Gestaltung des Unterrichts entwickelt worden. Dazu zählt die *situierte Erkenntnis*, bzw. die Annahme, „[…] dass bei der Konstruktion von Wissen stets die soziale Umgebung und der inhaltliche Kontext eine tragende Rolle spiel[en]“ [Hub07]. Der *narrative*

*Anker* ist eine weitere Methode, bei der das zu lernende Wissen in Geschichten, „[…] die in authentische und interessante Problemsituationen eingebettet sind“ [Hub07], aufgehängt wird. Beachtet man dies bei der Gestaltung des Unterrichts und auch im Gesamtkontext eines Studiums, können Lehrveranstaltungen ineinander übergehen und relevante Beispiele verwenden. Die in Kapitel 1.1.2 angesprochene *Employability* und daraus resultierenden Konsequenzen für die Lehre sind dabei ähnlich, denn auch hier wird die Gestaltung des Unterrichts in Hinblick auf die Berufsbildung gefordert. Dies trifft ebenfalls auf andere Methoden zu, die jedoch konstruktivistisch akzentuiert und fundiert werden. Siebert zeigt dabei, dass der thematische Bezug bei der Gestaltung von Lernumgebungen lernförderlich ist [Sie05, S. 108]. Tiberius warnt in diesem Zusammenhang vor Ablenkung des Lernenden durch eine zu starke Aufladung von Lernumgebungen, weswegen er eine Optimierung und keine Maximierung bei der kontextspezifischen Gestaltung fordert [Tib11, S.185]. Anders ausgedrückt können sehr realistische Lernumgebungen mit komplexen Problemen den Lernenden ebenso überfordern.

Eine Methode zur Reduzierung der Überforderung mit geleiteter Heranführung an komplexe (individuell und situativ abhängig) Problemstellungen ist das *Cognitive Apprenticeship* (zu dt. kognitive Handwerkslehre) [CBN88]. Siebert beschreibt diesen Ansatz als ein „[…] praxisbezogenes Lernen durch teilnehmende Beobachtung - in Anlehnung an die traditionelle Handwerkslehre“ [Sie05]. Der Lehrende gibt den Lernenden etwas modellhaft vor, um anschließend seine eigenen Erfahrungen unter Hilfestellung des Lehrenden zu machen. Aufgaben bzw. Problemstellungen werden dabei im Laufe komplexer gestaltet, bis zur eigenständigen Anwendung des Lernenden, und erfordern Kommunikation zwischen allen Teilnehmern. Nach Atkinson et al. ist „Cognitive Apprenticeship […] ein Oberbegriff für eine interaktive (zwischen Lernendem und Experten) Lernmethode, die die effektiven Bestandteile des traditionellen Meister-Lehrling Verhältnisses auf kognitive Lernziele anwendet“ [Atk+00]. In der traditionellen Handwerkslehre sind die Arbeitsschritte bei der Lösungserarbeitung durch die Handlungen ersichtlich, denn der Auszubildende kann die einzelnen Schritte bis zur Fertigstellung beobachten und einordnen. Bei stärker kognitiv geprägten Problemen müssen Überlegungen zur Lösungsfindung expliziert werden, da der Lehrling diese Prozesse nicht beobachten kann. Collins et al. unterteilen den Prozess der kognitiven Handwerkslehre in drei Gruppen mit insgesamt sechs Schritten ein [CBN88]: (1) Die ersten drei Schritte (*modelling*, *coaching*, *scaffolding*) stellen den Kern dar, der Studierenden das Erlernen (metakognitive Erkenntnisse) durch Beobachtung und ersten Erfahrungen durch angeleitetes Üben ermöglichen soll. (2) In Gruppe zwei (*articulation* und *reflection*) soll das eigene Handeln bewusst wahrgenommen und ihr Problemverständnis überprüft und ggf. angepasst werden. (3) Zuletzt soll die Exploration den Lernenden die selbstständige Erkundung von Problemen erlauben. Nachfolgend sind die einzelnen Lehrschritte noch einmal näher erläutert [BS08, S.195f]:

1. *Modelling* ist die Demonstration der Vorgehensweise des Lehrenden inklusive Begründung des eigenen Vorgehens, um die Schritte sichtbar zu machen.
2. *Coaching*: Die Lernenden versuchen unter Betreuung die zuvor beobachteten Handlungen selbst auszuführen, um sich mit den einzelnen Schritten vertraut zu machen.
3. Beim *Scaffolding* geben die Lehrenden den Lernenden nur noch so viel Unterstützung wie nötig und ziehen sich zurück (auch als *Fading* benannt).
4. *Articulation* beschreibt die Aufforderung an die Lernenden ihr Problemverständnis, ihr Wissen und die Vorgehensweise selbst zu formulieren und ggf. eine Anpassung vorzunehmen.
5. Der Schritt der *Reflection* soll den Vergleich der eigenen Vorgehensweise mit der des Lehrenden fördern, um das eigene Handeln bewusst wahrzunehmen.
6. Schlussendlich werden im Schritt der *Exploration* die Lernenden befähigt Problemstellungen eigenständig zu erkunden und zu bearbeiten.

Mit diesen Schritten soll das Erlernen von Lösungsstrategien erfolgen und nicht nur die reine Replikation vorgemachter Handlungen. Deswegen sind auch die allgemeinen, übertragbaren Fähigkeiten wichtiger, als spezielle Inhalte.

Ähnlich zum situierten Lernen hat die Gestaltung der Lernumgebung eine wichtige Rolle. Collins et al. unterscheiden dabei vier Dimensionen, die eine Lernumgebung beeinflussen [CBN88]: (1) Die inhaltliche Dimension beinhaltet fachspezifisches Wissen, heuristische und metakognitive Strategien zur Lösung des Problems und der eigenen Kontrolle und Lernstrategien des Lernenden. (2) In der Dimension der Methoden fasst Collins die sechs Schritte (vgl. oben) zusammen, die Lernenden die Möglichkeit geben, die Strategien (vom Meister) zu beobachten und anzuwenden. (3) Für Collins ist die Reihenfolge wichtig, was sich an dem Gesamtkonzept zu Einzelschritten und der Steigerung der Komplexität der Aufgaben zeigt. (4) Schließlich spielen soziologische Aspekte noch eine Rolle, da Lernende mit Lernenden und mit Lehrenden interagieren und kommunizieren sollen. Diese Dimensionen und die Beschreibung der Schritte sind allgemein gehalten und müssen an die jeweiligen Ziele und Inhalte angepasst werden, wie z. B. , dass der Lehrende bei kognitiven Prozessen mitteilt und die Schritte sichtbar ausführt.

### 3.4.3 Überforderungen: Cognitive Load Theory

Eine weitere kognitiv geprägte Theorie wurde von Sweller entwickelt und als *Cognitive Load Theory* bezeichnet und beschreibt die kognitive Belastung beim Lernen

[SvP98]. Unter Lernen wird dabei die Konstruktion von Schemata, die anschließend im Langzeitgedächtnis strukturiert und organisiert gespeichert werden, verstanden. Dies geschieht durch Elaboration, indem der Lernende neue Informationen mit Bedeutung versieht und mit vorhandenem Wissen verknüpft und somit im Sinne des Konstruktivismus konstruiert. Wichtig ist ebenfalls die Induktion, mittels welcher die Lernerfahrungen in abstraktere Schemata überführt werden, um neue Probleme und Aufgaben zu lösen (Anwendung von Analogien bzw. Generalisieren). Jedoch ist die Aufnahmefähigkeit des Gedächtnisses begrenzt, weswegen sich die verschiedenen kognitiven Belastungen auf das Lernen auswirken. Darunter fällt (1) der *intrinsic load* (intrinsische Belastung), welche vom Lernmaterial selbst ausgeht, denn je schwerer das Lernmaterial ist, umso höher ist die Belastung. Die Theorie geht davon aus, dass diese Art der Belastung auch nicht verändert werden kann, sondern immer an den Kontext und das Material, z. B. durch komplexe Verbindungen der zu erlernenden Elemente, gebunden ist. (2) Daneben wird noch von dem *germane load* (lernbezogenen Belastungen) ausgegangen. Diese sind direkt mit dem Lernen verbunden und beschreiben den Aufwand, welchen ein Lernender betreiben muss, um das Material zu verstehen. Besonders die Darstellung des Materials kann die Belastung, neben Aufmerksamkeit, Vorwissen und Motivation, beeinflussen. (3) Zuletzt gibt es noch die unnötigen Belastungen (*extraneous load*), die vom Lernen ablenken. Darunter fallen z. B. die Darstellung des Lehrmaterials sowie unnötige Erklärungen und Wiederholungen. Aus dieser Theorie können Hinweise zur Gestaltung von Lernumgebungen abgeleitet werden. Sweller nennt dabei bereits fünf verschiedene Effekte zur Gestaltung von Lernmaterialien [Swe05].

- *Worked Examples*: Durch ausgearbeitete Lösungswege fällt es den Lernenden einfacher Sachverhalte zu verstehen, als wenn diese vollständig selbst erarbeitet werden müssen, da hier das Arbeitsgedächtnis nicht zusätzlich belastet wird.

- Der *Split Attention Effect* besagt, dass die Lernleistung abnimmt, sobald die Informationen zu stark voneinander getrennt sind.

- *Modality Effect*: Dabei wird von zwei Kanälen (visuell und akustisch) zur Aufnahme von Informationen ausgegangen. Beide Kanäle sollen ergänzend verwendet werden, um die zu starke Belastung eines einzelnen Kanals zu vermeiden.

- *Redundanzen* sollten vermieden werden, sodass der Lernende sich nicht mit überflüssigen Quellen beschäftigen muss.

- Expertise Reversal Effect: Dieser Effekt beschreibt eine negative Auswirkung durch die gewählte Methode zur Vermittlung auf die Lernergebnisse der Studierenden, welche bereits Vorwissen haben.

Die *Cognitive Load Theory* steht jedoch auch in der Kritik, da motivationale und emotionale Prozesse nicht berücksichtigt werden und die kognitive Belastung bzw. die drei unterschiedlichen Belastungen nicht gemessen werden können [Rey09]. Jedoch wurden diese Effekte in Bezug auf multimediales Lernen bereits empirisch überprüft und bewiesen, weswegen durchaus die Gestaltungsempfehlungen bei der Konzeption von Lehrveranstaltungen und Methoden beachtet werden sollten [Rey09]. Schnotz et al. haben festgestellt, dass die intrinsische Belastung im Gegensatz zu Sweller veränderlich ist und von dem jeweiligen Vorwissen und der Situation abhängig ist. Dies ist bei der Gestaltung von Aufgaben und Materialien zu beachten, da zu schwere bzw. zu leichte Aufgaben, die nicht an das Vorwissen der Lernenden angepasst sind, einen negativen Einfluss auf den Lernerfolg haben [SK07, S.478f].

## 3.5 Schlussfolgerungen für die Lehre

Zum Abschluss dieses Kapitels möchte ich aus der dargelegten Auffassung von Programmieren und diskutierten lehr- und lerntheortischen Ansätze und Methoden einige Schlussfolgerungen bzw. Gestaltungshinweise für die Entwicklung des Konzeptes ziehen. Dazu soll kurz der Begriff des Unterrichts als Gegenstand der Didaktik definiert werden:

> Unterricht ist ein hochkomplexer Prozess mit zahlreichen, auch zyklischen Wechselwirkungen, bei dem Lehrende und Lernende unter gewissen gesellschaftlichen und materiellen Vorgaben und Rahmenbedingungen im Hinblick auf eine bestimmte Zielsetzung interagieren [Hub07].

Bei der Gestaltung sind aus meiner Sicht besonders die institutionellen Rahmenbedingungen eine relevante Einflussgröße, da diese die Auswahl von Methoden einschränkt bzw. beeinflusst. Ein Beispiel dafür ist eine begrenzte Anzahl an Räumen und die Raumausstattung, wie z. B. Schräghörsäle, welche Gruppenarbeit zumindest erschweren oder in denen keine Computerausstattung vorhanden ist. Diese Überlegungen sind jedoch nachgelagert, denn ausgehend von einem bildungstheoretischen Ansatz nach Klafki, sind zuerst die Lernziele zu klären, bevor eine geeignete Methode ausgewählt werden kann [Kla74]. Bereits der Aufbau dieser Arbeit spiegelt diese Ansicht wider, indem die Ziele der Lehrveranstaltung ausgehend von den Auswirkungen der *Industrie 4.0* und klassischen informatischen Ansätzen diskutiert wurden (vgl. Kapitel 1.1.2).

In Kapitel 3.1 wird Programmieren als iterativer Prozess zum Lösen von Problemen mittels Maschinen beschriebenen. Ziel ist es dabei algorithmische Lösungen zu entwerfen, die anschließend in einer formalen Sprache implementiert und überprüft wer-

den können. Die dargestellten Problemlösestrategien erfordern aber, neben dem Lösen durch Anwendung von Strategien, Versuch und Irrtum oder Analogien, theoretisches Wissen, um einerseits die Probleme zu verstehen und andererseits ausführbare Algorithmen zu entwerfen. Bei der Problemanalyse wird häufig die Zerlegung in Teilprobleme (vgl. informatisches Denken nach Wing in Kapitel 3.2.3) zur Reduzierung der Komplexität durchgeführt. Algorithmen sind dabei die geschickte Kombination einfacher logischer Bausteine, die im Sinne der imperativen Sprachen den Programmierkonzepten entsprechen (vgl. Kapitel 3.3.2 und Kapitel 3.3.3). Die Auswahl, Anwendung und Kombination dieser Bausteine umfasst somit einerseits Wissen über ein Konzept und auf der anderen Seite Fähigkeiten und Erfahrung diese auf neue (Teil-)Problemstellungen anzuwenden (Analogien). Aus diesen einzelnen Elementen des Programmierens lassen sich die Lernziele ableiten. Darunter fallen theoretische Grundlagen zur Vermittlung der Grenzen der Berechenbarkeit oder die Entwicklung einer algorithmischen Lösung für ein mathematisches Problem aus dem Bereich der Elektrotechnik. Einzelne Lernzeile sollten dabei auf ein übergeordnetes Ziel einer Lehrveranstaltung ausgerichtet sein.

Was bedeutet das konkret für die Inhalte und die Gestaltung einer Lehrveranstaltung? Ausgehend von den konstruktivistischen Ideen ist die Einordnung der Inhalte in einen größeren Zusammenhang dabei notwendig. Ausgangspunkt ist hierbei das Programmieren und der Algorithmenentwurf aus denen sich die theoretischen Inhalte ableiten. Überspitzt ausgedrückt, ist die Schreibweise einer `if`-Bedingung in *C*, weniger bedeutsam als das darunterliegende Konzept der alternativen Programmausführung und dessen Rolle bei der Entwicklung von Lösungen. Blömeke fasst die Notwendigkeit von übergeordneten Strukturen in Bezug auf Lernleistungen zusammen, denn „[…] Einführungen [sollen] die Verknüpfung von neuem und vorhandenem Wissen erleichtern, indem sie abstrakter als der zu vermittelnde Inhalt gehalten sind und in erster Linie Beziehungen zwischen den Inhalten aufzeigen […]. Mayer (1983) weist signifikant höhere Lernleistungen bei einem entsprechenden Vorgehen nach“ [BHT07]. Der allgemeinbildende Fokus darf nicht verloren gehen, auch wenn eine Lehrveranstaltung als Vorbereitung für weitere konsekutive Veranstaltungen dient. Im Umkehrschluss darf die Phase der Codierung als Verifikation des algorithmischen Entwurfs ebenso nicht vernachlässigt werden. Ziel der Lehrveranstaltung ist schließlich das Programmieren inklusive aller Teilprozesse. Die Programmierkonzepte helfen bei der Bewältigung der einzelnen Schritte des Programmiervorgangs. Variablen und die damit verbundenen Datentypen können z. B. ausgehend von der Repräsentation des Ausgangszustandes eines Problems eingeführt werden. Zusätzlich greifen viele Konzepte ineinander bzw. bauen aufeinander auf, wie z. B. Variablen und Schleifen oder auch Boolesche Ausdrücke und alternative Programmabläufe. Das hat zur Folge, dass häufig weitere Inhalte auf erst kürzlich gelerntes Wissen – ggf. nicht gefestigt und ausreichend geübt – aufgesetzt werden. Dadurch können Studierende leicht abgehängt werden und finden ggf.

keinen Anschluss mehr. Die Folgerung daraus ist, dass in der Lehre ggf. während der Vorlesung, aber auch nach der Übungsbearbeitung auf den aktuellen Wissensstand der Studierenden reagiert werden muss. Hier ist der Moment, in denen die konstruktivistischen Überlegungen einfließen können, denn diese fordern die aktive Beteiligung des Lernenden und eine situative Selbststeuerung und wie Tiberius (vgl. Kapitel 3.4.1) argumentiert, den Lehrenden als Korrektor falscher Konstruktionen. Programmieren beinhaltet kognitive Fähigkeiten und somit Handlungen, die in einer Vorlesung mit primären Einsatz von Frontalunterricht nicht durchgeführt werden können. Daraus resultiert auch die anfangs in der Problemstellung geforderte aktive Beteiligung der Studierenden im Sinne eines seminaristischen Unterrichts. Wie sollen Studierende Erfahrungen sammeln und Wissen konstruieren, wenn ihnen die Möglichkeit im Unterricht verwehrt bleibt? Hier sind Konzepte und Methoden zur Partizipation und gegenseitigen Interaktion notwendig, wie z. B. beim *Cognitive Apprenticeship*, welches eine Möglichkeit ist, Lernenden kognitive Fähigkeiten näher zu bringen und Studierende Erfahrungen in einer passenden Lernumgebung sammeln zu lassen. Dieses Konzept kann nicht einfach ohne Anpassungen an das Erlernen des Programmierens eingesetzt werden, sondern muss zu den Lernzielen passen.

Störende kognitive Belastungen treten auch beim Programmieren auf, sodass wie in der Diskussion um die Ziele (vgl. Kapitel 1.1.2) Hubwieser mehr Gewicht auf die Modellierung legt und die Codierung als eher notwendig betrachtet. Dadurch soll unter anderem der Aufwand, den das Erlernen und Einsetzen einer formalen Sprache verursacht, vermieden bzw. eine Priorisierung zu Gunsten des Modellierens durchgeführt werden. Die Phase des Codierens benötigt eine wie auch immer geartete Umgebung, in welcher der Algorithmus in einer formalen Sprache umgesetzt, ggf. kompiliert und anschließend ausgeführt werden kann. Studierende müssen sich dabei von Anfang an mit den ersten beiden Phasen sowie der Codierung mittels eines Editors oder Entwicklungsumgebung und der Werkzeugkette der jeweiligen Sprache auseinander setzen. Dies kann besonders am Anfang zur kognitiven Überlastung führen, da der in Umgebung mit neuen Werkzeugen zusätzlich noch geübt werden muss. Normale Entwicklungsumgebungen werden für Experten entwickelt und nicht für Programmieranfänger, auch wenn diese später im Rahmen der Befähigung für das Berufsleben ebenso damit umgehen können müssen. Bei der Wahl kann durchaus zu einer weniger ablenkende und belastende Umgebung, wie z. B. *BlueJ* oder *Scratch*, gegriffen werden. Auch bei der Gestaltung der Lehrmaterialien sollte auf eine geeignete Darstellung unter Berücksichtigung der beschriebenen Effekte aus der *Cognitive Load Theory* geachtet werden, da diese nachweislich einen positiven Einfluss auf den Lernerfolg haben.

Fasst man die bisherigen Punkte zusammen, so ist es erstrebenswert eine Lehrveranstaltung so zu gestalten, dass Studierende Erfahrungen sammeln und neue Fähigkeiten ausbilden können. Darunter fällt ebenfalls die teilweise Selbststeuerung durch die

Studierenden, wenn auch diese sich nur bedingt auf alle Studierenden umsetzen lässt. Tendenzen bezüglich der Geschwindigkeit und bei Verständnisproblemen seitens der Studierenden müssen bei der Planung und Durchführung einer Vorlesung berücksichtigt werden. Erfahrungen sammeln bedeutet beim Programmieren aber immer auch Codieren, um auch im Sinne des Problemlösens die eigenen Strategien anwenden und überprüfen zu können. Nur wie können Studierende nun *aktiv* in eine Lehrveranstaltung bzw. Vorlesung eingebunden werden? Wie müssen Lernumgebungen gestaltet sein, dass diese sich durch eigene Erfahrungen mit den Konzepten der Programmierung auseinandersetzen können, um schließlich eigene Heuristiken zu erkennen und zu verinnerlichen? Kann dabei handlungsorientierter Unterricht, abgeleitet aus den konstruktivistischen Überlegungen, hilfreich sein? Und wenn ja, wie kann solch eine Lösung aussehen? Genau diese Fragen möchte ich im nächsten Kapitel im Rahmen des entwickelten Konzeptes beantworten bzw. eine Möglichkeit auf Basis des Aktiven Lernens aufzeigen, das diese konstruktivistische und kognitive Sicht aufgreift und Methoden ableitet.

Zuletzt möchte ich noch den Gedanken von Schulte et al. über das Erlernen des Programmierens und Programmabläufen aufgreifen:

> Perhaps the final lesson for CS educators from this analysis of PC models is that we may need to be more patient with our students and acknowledge that some good things take time [Sch+10].

Das Aneignen des Programmierens ist und bleibt (unabhängig von Aufwand) ein langfristiger Lernprozess, in dem Erfahrungen gesammelt und reflektiert werden müssen. Dementsprechend müssen wir den Studierenden den Raum und die Möglichkeiten einräumen, Erfahrungen zu sammeln und diese anschließend zu reflektieren, ohne diese dabei zu überfordern [Bör07, vgl. S.13]. Reflektierende Phasen, in den die eigene Lösung und das Lösungsvorgehen (auf metakognitiver Ebene) kritisch hinterfragt werden, ist im Bereich der Software-Entwicklung nicht neu. Code-Reviews, Pair-Programming und ganze Software-Entwicklungsvorgehensmodelle, wie z. B. Scrum enthalten ganz bewusst reflektierende Elemente (Retrospektive in Scrum), um genau solche Überlegungen explizit anzustoßen. Solche reflektierenden Phasen sind aus konstruktivistischer Sicht und der Methode des *Cognitive Apprenticeship* erstrebenswert, denn reines Handeln der Studierenden ist nicht genug. Denn wie bereits diskutiert, sollten Fehlvorstellungen und falsche Annahmen der Studierenden rechtzeitig erkannt und entsprechend korrigiert werden. Die nächsten Kapitel nähern sich an einer mögliche Lösung für diese Überlegungen indem zuerst das *Aktive Lernen* diskutiert und aus vorhandenen theoretischen Ansätzen ein Konzept abgeleitet wird.

# Teil II

# Hauptteil: Interaktive Vorlesungen

# Kapitel 4

# Aktives Lernen

Der folgende Teil dieser Arbeit beschäftigt sich mit der Entwicklung eines Konzeptes und der benötigten Lern- und Lehrmaterialien bzw. Werkzeuge. Ausgangspunkt sind die in Kapitel 3.5 beschriebenen Schlussfolgerungen und Gestaltungshinweise, die bei der Entwicklung berücksichtigt wurden bzw. diese geleitet haben. Eine der Folgerungen ist eine Lernumgebung, welches die aktive Partizipation der Studierenden und teilweise selbstgesteuertes Lernen ermöglicht. Dazu, und auch bezugnehmend auf die Forschungsfragen, möchte ich die Methoden des *Aktiven Lernens* bzw. der *Aktivierenden Lehre* näher betrachten und eine mögliche Umsetzung (das Konzept) für die dargelegten Ziele vorstellen. Nach [Ten+17] ist im Bereich des *Aktiven Lernens* die Definition und Art der Verwendung dieses Ansatzes wichtig, da eine Vielfalt an Ansätzen und Interpretationen dieses Begriffes existieren.

## 4.1 Charakteristika und Merkmale

Nachfolgend soll der Begriff *Aktives Lernen* (engl. *active learning*) fassbar gemacht werden. Dabei stellt sich die primäre Frage: Was ist Aktives Lernen bzw. Aktivierende Lehre (unabhängig vom Programmierkontext)? Diese Fragen und die Forschungsfrage 1 soll in diesem Kapitel bearbeitet und begründet werden. Ausgehend von den konstruktivistischen und kognitivistischen Theorien ist die individuelle Konstruktion von Wissen durch eigene Erfahrungen lernförderlich. Im Kontrast dazu steht rezeptives Lernen durch instruktionelle Methoden, welche einen stärkeren Fokus auf den Lehrenden legen. Bei der Diskussion der Lehr- und Lerntheorien wurde jedoch bereits angemerkt, dass solche Methoden ihre Berechtigung haben und die Auswahl vom Ziel abhängig sein sollte.

Bonwell et al. beschreiben *Aktives Lernen* anhand mehrerer Merkmale, die mit dem Begriff verbunden werden. Dazu zählt unter anderem, dass

- Studierende über das reine Zuhören involviert sind,
- weniger Fokus auf das Transferieren von Information als auf das Entwickeln von Fähigkeiten gelegt wird,
- Studierende zu metakognitivem Denken (Analyse, Synthese und Evaluation)[10] angeregt werden,
- Studierende sich mit Aktivitäten beschäftigen (darunter fällt z. B. Lesen, Diskutieren und Schreiben)
- und ein stärkeres Gewicht auf studentische Exploration Werte gelegt wird [BE91].

Daraus leiten die Autoren eine Definition für *Aktives Lernen*, die wie folgt lautet: „[…] active learning [can] be defined as anything that 'involves students in doing things and thinking about the things they are doing'" [BE91]. Aus den Merkmalen können zwei unterschiedliche Arten der Aktivitäten abgeleitet werden: (1) beobachtbares Tun (Handeln) und (2) mentale Aktivitäten, die nicht direkt beobachtbar sind.

Um mich diesen beiden zu nähern und im Kontext der Lehr- und Lerntheorien zu diskutieren, beziehe ich mich auf Renkls Ausführungen über die Effektivität dieser Aktivitäten [Ren11, vgl. S.198]. Die erste Auffassung, das *aktive Tun*, beschreibt primär das aktive Problemlösen und den aktiven Diskurs (individuelles Handeln und soziales Interagieren) [GE14]. Zweitens wird das Aktive Lernen aus Sicht der aktiven Informationsverarbeitung, bzw. kognitiver Prozesse, angesehen. Beobachtbare Aktivitäten sind dabei nicht ausschlaggebend, stattdessen ist die aktive mentale Verarbeitung und Auseinandersetzung mit den Lerninhalten ausschlaggebend. Renkl erweitert letztere Sichtweise um eine differenzierte Beschreibung der mentalen Aktivitäten. „Danach führt weniger die mentale Aktivität an sich zu gelungenem Wissenserwerb, sondern die mentale Aktivität, die die zentralen Begriffe (z. B. Begriffe) und Prinzipien (z. B. mathematische Sätze) in einem Lernbereich fokussiert" [Ren11]. Die beiden Theorien haben gemein, dass Lernen einen aktiven Lernenden erfordert, der durch (selbstgesteuerte) Handlungen und Erfahrungen Wissen verarbeitet bzw. konstruiert.

Doch bleiben wir noch kurz bei den stärker beobachtbaren Aktivitäten, bei denen offene Lernaktivitäten für einen gelungenen Lernprozess notwendig sind. Renkl kritisiert dabei eine „[…] 1:1-Korrepondenz zwischen äußerlich sichtbaren Lernaktivitäten und dem, was internal, also im Kopf des Lernenden passiert" [Ren11]. Er versucht diese Aussage anhand drei Beispielen zu belegen in denen Lernende ähnliche Leistungen ohne eine aktive Beteiligung erreicht haben:

[10] Diese Elemente finden sich in Blooms Taxonomiestufen wieder [Fel+00; BE76, S.3].

> Die verbale Beteiligung der Schüler am Unterricht, die von den Autoren als aktive Partizipation beim Lernen angesehen wird, sagt nicht den Lernerfolg vorher. Vermeintlich aktive Schüler lernten also nicht mehr [Ren11].

Ich schließe mich dieser Kritik an, möchte jedoch weitere Punkte hinzufügen. Die Bezeichnung „offene Lernaktivitäten“ ist unscharf und ebenso wie das Aktive Lernen nicht eindeutig definiert. Lernförderliche Aktivitäten müssen nicht unbedingt auch erfolgreiches Lernen zur Folge haben. Zusätzlich wird hier das Vorwissen der jeweiligen Lernenden bei der Argumentation vernachlässigt, denn unter- bzw. überforderte Lernende zeigen unter Umständen nicht die antizipierten Lernaktivitäten, welche die von Renkls beschriebenen Beispiele abschwächen. Somit ist eine reine quantitative Messung der Aktivitäten nicht immer zielführend, ohne dabei qualitative Daten, die auch die Art der Aktivitäten beinhalten, zu berücksichtigen (siehe Ausführungen über die Evaluation in Kapitel 2.3). Ausgehend von Lernumgebungen, die eine (teilweise) Selbststeuerung des Lernenden ermöglichen, können die Lernaktivitäten, ggf. erst gar nicht erfasst werden.

Im Gegensatz zu den offenen Aktivitäten ist eine mentale inhaltsbezogene Aktivität, aus Sicht der aktiven Informationsverarbeitung, entscheidend. Diese lässt sich aus den kognitiven bzw. konstruktivistischen Ansätzen, bei denen Wissen nicht einfach vermittelt werden kann, sondern konstruiert bzw. verarbeitet werden muss, ableiten. Beide Perspektiven stehen nicht antagonistisch zueinander, sondern Aktivitäten des Tuns können sehr wohl mentale Aktivitäten inkludieren bzw. induzieren. Renkl beschreibt die aktive Informationsverarbeitung in Bezug auf das Ablegen von Wissen im Langzeitgedächtnis folgendermaßen:

> Darin hinterlässt einmal erworbenen Wissen eine überdauernde, wenn auch unter Umständen schwache Spur. Wissen kann dann gut wiedergefunden werden, wenn es mit zahlreichen anderen Wissenselemente assoziiert ist, da in diesem Fall viele „Zugriffsmöglichkeiten“ zu dem gesuchten Wissenselement bestehen [Ren11].

Konstruktivistisch betrachtet, erfordert das Lernen das Vernetzen von bisherigem Wissen mit neuen Informationen, um neues Wissen zu konstruieren. Bei dieser Auffassung kann jedoch, besonders bei realitätsnahen und situativ gestalteten Lernumgebungen, eine Überforderung durch die Komplexität und Informationsfülle und eine mangelnde Fokussierung verursacht werden. Hilbert et al. haben in einer Studie herausgefunden, dass beim Erlernen des mathematischen Beweisens anhand von komplexen Lösungsbeispielen, die Instruktion, deren Fokussierung einen Einfluss auf das Lernergebnis hat. Dabei wurden in zwei Kohorten unterschiedlich viele Anregungen zum aktiven Verarbeiten der Inhalte angeboten, mit der Ergebnis, dass sich zu vie-

le Anregungen negativ auf das Lernen auswirken können [Hil+08; Rei+08]. Für die Gestaltung und Konzeption von Lehr- und Lernmaterialien hat dies eine konkrete Schlussfolgerung: *Anregungen zum Aktiven Lernen sind mit Bedacht und mit einer Fokussierung auf Konzepte und Prinzipien einzusetzen* [RHS09; Ren11].

Rekurrierend auf die Forschungsfrage 1 lässt sich keine grundsätzliche Definition finden, sondern vielmehr Charakteristika und Empfehlungen. Fasst man diese Überlegungen zusammen, können mehrere Handlungsrichtlinien bei der Gestaltung abgeleitet werden. Reine Aktivitäten, die Lernenden zum Tun auffordern reichen nicht aus und reine beobachtbare Aktivität kann nicht mit Lernerfolg gleichgesetzt werden. Ein Beispiel dazu ist die Beantwortung durch Handmeldung von aktivierenden Fragen am Anfang einer Vorlesung. Nur einige Studierende werden sich melden, jedoch beschäftigt sich trotzdem ein Großteil (ausgehend von Beobachtungen) mit den Fragen. In anderen Studien wurde vielmehr gezeigt, dass es besser ist, „[...] mehrere Lösungsbeispiele zu studieren, statt sehr schnell (z. B. nach einem Beispiel) zum Bearbeiten von Aufgaben überzugehen“ [Ren11]. Passives Studieren von Lösungsbeispielen beim *anfänglichen* Erwerb, mit Betonung auf anfänglich, erzielen bessere Ergebnisse als ungerichtetes Bearbeiten von Aufgaben bzw. zu viele aktivierende Anregungen. Solche Lösungsbeispiele werden auch in der *Cognitive Load Theory* als *Worked Examples* zuträglich beschrieben, da die kognitive Belastung geringer ist, und somit das Erfassen der Sachverhalte leichter fällt (vgl. Kapitel 3.4.3). Bei diesen Lösungsbeispielen sind die mentalen Aktivitäten, wie z. B. Selbsterklärungen, von entscheidender Bedeutung und sollten bei der Gestaltung von Materialien und Umgebungen berücksichtigt werden [Ren11, S.200]. Im Umkehrschluss sollten im Laufe der Zeit, nach dem anfänglichen Erwerb, komplexere Aktivitäten betont werden. Dieses Prinzip der steigenden Komplexität und einer Reduzierung der Hilfestellungen, wie z. B. die Beispiellösungen, finden sich im Prinzip des *Cognitive Apprenticeship* wieder, welches durch das *Fading* die Kontrolle des Lehrenden und damit implizit auch der Beispiellösungen reduziert.

Aktives Lernen und aktivierende Lehrmethoden können den Lernprozess positiv beeinflussen, wobei nicht immer von einer Aktivierung der Studierenden ausgegangen werden darf. Ulrich fasst die zu erwartenden Effekte auf das Lernen durch die Aktivierung zusammen: „Es gibt nur einen gewissen Zusammenhang - aktive Personen lernen eher“ [Ulr16]. Daraus kann die Definition der *Aktivierenden Lehre* abgeleitet werden, welche als Aktivierung bzw. Motivierung durch die Gestaltung von Lernumgebungen, um aktives Lernen auf Seiten der Lernenden zu ermöglichen, aufgefasst werden kann. Doch aktivierende Methoden, wie z. B. intensives Betreuen, Anleiten und Zeit für die Auseinandersetzung mit Inhalten, haben ihren *Preis* - sie benötigen deutlich mehr Zeit als rezeptives Lernen, bei dem Lernende *„im Kinomodus“* Wissen aufnehmen. Weswegen eine wohlüberlegte inhaltliche Stoffreduktion höchstwahrscheinlich erforderlich ist. Allerdings haben aus konstruktivistischer Sicht die metakognitiven

Prozesse, die zur Ausbildung von Lernstrategien und Lösungsstrategien dienen, einen höheren Stellenwert als viel Fachwissen. Diese Prinzipien und Strategien tragen deswegen viel mehr zum lebenslangen Lernen und eines allgemeinbildenden Verständnisses über die Inhalte bei. Bewusst ist hier die Rede von einer wohlüberlegten Reduzierung der Inhalte, da auf der anderen Seite theoretische Grundlagen für das Problemlösen und Konstruieren von Wissen unabdingbar sind. Beim Programmieren stellen hier die Konzepte der Programmierung und Grenzen der Berechenbarkeit die Grundlagen dar.

## 4.2 Strategien und Formen des Aktiven Lernens

Die Charakteristika und Beschreibungen aus dem vorherigen Kapitel haben umsetzbare Formen bisher nicht näher thematisiert, weswegen ich in diesem Kapitel einen kurzen Überblick geben möchte.

### 4.2.1 Formen

Prince hat die Wirksamkeit des problembasierten, kollaborativen und des kooperativen Lernens im Ingenieurbereich untersucht [Pri04]. Wildt sieht als höchste Stufe des aktiven Lernens das forschende Lernen an und hat diese in einem Stufenmodell angeordnet (vgl. Abbildung 9) [Wil11]. Auf die erste Stufe des Erfahrungs- und des handlungsorientierten Lernen baut das entdeckende Lernen (nach Bruner vgl. Kapitel 3.4) auf. Charakteristisch für diese Form sind die Eigenverantwortlichkeit, Selbstorganisation und -steuerung beim Lernprozess. Das eigenverantwortliche und selbstorganisierte Vorgehen leidet dabei unter denselben Problemen wie die eher radikal konstruktivistischen Forderungen, denn eine Fokussierung auf relevante Hinhalte (Konzepte und Prinzipien) ist dabei nicht immer gegeben (vgl. dazu [Ren11, S.208]). In den nächsten Stufen wird das Lernen um die Elemente der Planung der Lernhandlungen und beim problemorientierten Lernen, um die Handlungspläne zur Bewältigung von Aufgaben und metakognitive Überlegungen des eigenen Vorgehens, erweitert. Wildt unterscheidet zwischen problem- und projektorientierten Lernen anhand der Vorgabe einer Aufgabenstellung [Wil11]. Zu letzterem gehört die selbstständige Entwicklung der Aufgabenstellung unter Berücksichtigung der Relevanz der zu erwartenden Ergebnisse. Im fallorientierten Lernen wird die Rolle des Kontextes, in welchem die Ergebnisse der Lernprozesse gestellt werden, betont. Schlussendlich ist nach Wildt das forschende Lernen die Hochform des *Aktiven Lernens*, welches den „[...] Akzent auf die theoretische und empirische Steuerung der Lernprozesse“ [Wil11] legt. Forschendes Lernen führt die beiden Hochschulbereiche der Lehre und der Forschung zusammen,

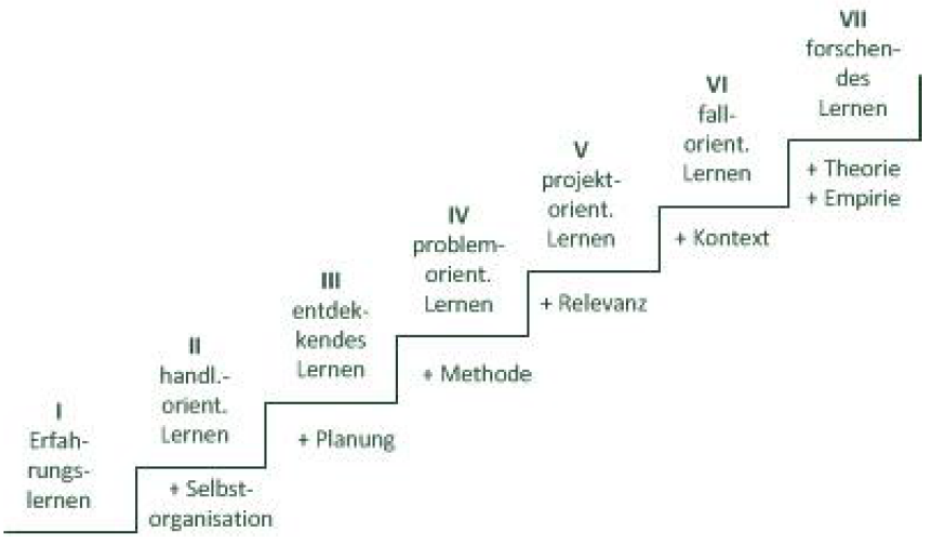



**Abbildung 9**: Stufenmodell der Formen des aktiven Lernens nach Wildt (entnommen aus [Wil11])

um „[…] Projekte Forschenden Lernens und Forschungsprojekte suis generis mit Kategorien des Forschungshandeln übereinstimmend […]“ [Wil11] zu beschreiben. Im Gegensatz zur reinen Forschung hat das forschende Lernen nicht (nur) das Ziel der wissenschaftlichen, sondern der individuellen Wissenserkenntnis [HHS13, S.12]. Auch die Bezugssysteme können verdreht werden, sodass die Forschung nicht die Inhalte der Lehre bestimmt. Stattdessen wird das forschende Lernen als Mechanismus für den Transfer beruflicher Praxis im Sinne der geforderten *Employability* (vgl. Kapitel 1.1.1) verwendet, indem der Feldbezug hergestellt wird [WK14]. Wildt fasst kooperatives und *Aktives Lernen*, im Bezug auf das forschende Lernen, als gegenseitig verstärkende Elemente und nicht als Formen des *Aktives Lernen* auf. Aktives Lernen steuert das Erfahrungslernen und das selbstorganisierte Handeln und das kooperative Lernen die Zusammenarbeit der Studierenden bei [Wil11].

Aus konstruktivistischer Sicht entsteht die Wissenskonstruktion aus der aktiven Auseinandersetzung, z. B. Problemlösen oder Diskurs, mit bzw. in authentischen Situationen. Nachfolgend möchte ich das problemorientierte, kooperative und kollaborative Lernen näher erläutern und daraus Schlussfolgerungen ableiten.

**Tabelle 5**: Unterschiede des kooperativen und kollaborativen Lernens

| | **Kooperation** | **Kollaboration** |
|---|---|---|
| Lernprozess | Wissen entsteht in einem (strukturierten) Prozess | Wissen wird (zusammen) konstruiert |
| Art der Zusammenarbeit | Individuelle Bearbeitung und Zusammenführung am Schluss | Permanente Zusammenarbeit in einer Gruppe |
| Augenmerk | Das entstehende Produkt im Prozess | Prozess der zum Produkt führt |

### Kollaboratives und Kooperatives Lernen

Beide Formen werden in der Literatur und im englischen (*peer learning*, *collaborative learning cooperative learning* oder *peer assistedlearning* [PR00; Kon14; WB16]) nicht immer voneinander abgegrenzt, weswegen die Beschreibung der Durchführung bei der Analysen von Studien relevant ist. Der Schlüssel bei dieser Form des Lernens ist die Art und Weise, in der die Lerninhalte von den Lernenden bearbeitet werden. Kollaboratives und kooperatives Lernen ist nicht mit Gruppenarbeit gleich zu setzen, sondern vielmehr die gezielte Verwendung von Gruppenarbeit in einem größeren Ablauf. Bei beiden Formen werden die Lernenden, zumindest zeitweise, in Kleingruppen aufgeteilt. Tabelle 5 zeigt dabei die Unterschiede der beiden Formen auf (in Anlehnung an [Kon14]). Kollaboratives Lernen legt den Fokus auf die Zusammenarbeit und die Interaktionen während des Lernprozesses. Individuen lernen im Diskurs und gegenseitigem Austausch mit dem Ziel neues Wissen und Fähigkeiten zu erwerben und eine ähnliche Konstruktion zu erarbeiten [Wah13].

Brüning und Saum haben drei Kernelemente (vgl. Abbildung 10) zu einem Grundprinzip des kooperativen Lernens zusammengefasst. Das Prinzip setzt als ersten Schritt eine individuelle Erarbeitung der Inhalte voraus. Lernende konstruieren (im konstruktivistischen Sinne) auf Basis ihrer Vorerfahrungen und ihres bisherigen Wissens neue Strukturen. Im Anschluss finden beim Austausch in der Gruppe weitere Prozesse statt, da Lernende ihre eigene Konstruktion mit anderen vergleichen und ggf. bei Widersprüchen erklären (sichtbar machen) und ggf. anpassen und überarbeiten müssen. Notwendigerweise müssen Lernende dabei ihre eigenen Ergebnisse für einen Vergleich oder die Formulierung ihrer Gedanken durchdenken. Zuletzt, werden die Ergebnisse im Plenum vorgestellt und erneut diskutiert, um ggf. *falsche* Konstruktionen zu berichtigen (vgl. dazu die konstruktivistischen Überlegungen in Kapitel 3.4.1) und auf Fragen eingehen zu können. Diese letzte Phase dient auch der Ergebnissicherung,

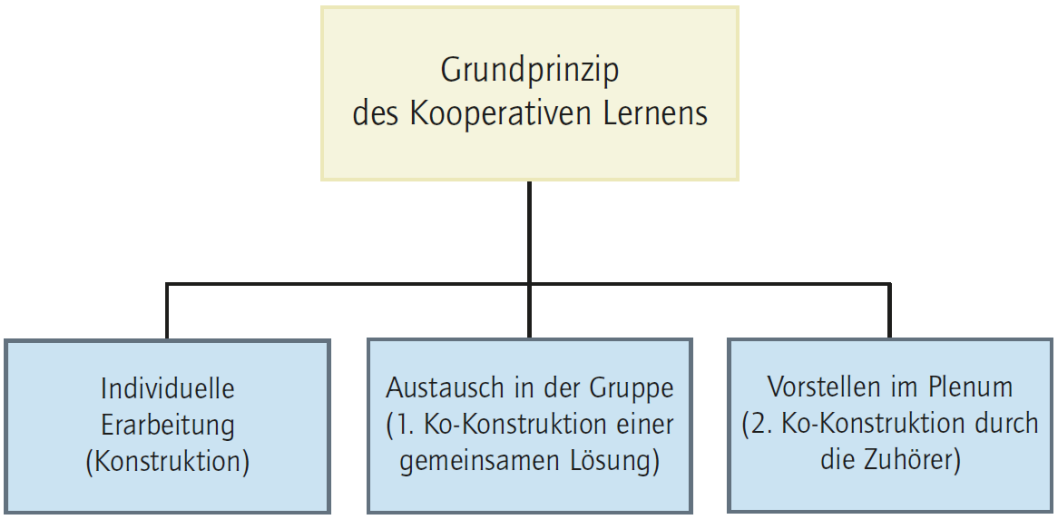



**Abbildung 10**: Grundprinzip des kooperativen Lernens nach Brüning und Saum

sodass die Beiträge und Erkenntnisse vom Lehrenden zusammengefasst und wiederholt in einen Gesamtzusammenhang eingeordnet werden.

Entscheidend ist beim kooperativen Lernen die diskursive Qualität der Lernerinteraktionen in der kooperativen Lernsituation für den individuellen Lernertrag [PR00]. Pauli und Reusser fassen die relevanten Merkmale wie folgt zusammen [PR00]:

- Das Gespräch soll sich auf den Lerngegenstand und den Austausch über ein gemeinsames Verständnis beziehen,
- alle Kooperationspartner müssen aktiv beteiligt sein und
- gegenseitig auf Beiträge und Fragen eingehen und dies ggf. diskutieren.

Diskussionen und die Bearbeitung der Aufgabenstellungen stellen in dieser Form den Bezug zum aktiven Lernen dar. Kooperatives und kollaboratives Lernen sind beides Formen, die den sozialen Prozess als Merkmal des Konstruktivismus gezielt einsetzen. Ziel dieser Interaktionen ist es den Lernerfolg und die Qualität zu steigern, wobei dies nicht immer eintreten muss. Dementsprechend ist eine gezielte Instruktion bzw. Entwicklung von Methoden zum Erreichen des Zieles vonnöten. Zusammengefasst kann die Distinktion zwischen kollaborativ und kooperativ an der Art der Arbeitsteilung erfolgen. Kollaborativ steht für das gemeinsame Bearbeiten von Problemen, während kooperativ das individuelle Bearbeiten von (Teil-)Problemen und deren Zusammenfügung beschreibt. Eine zwingende Aufteilung der Probleme ist nicht notwendig, denn Ziel sind die individuelle Konstruktion und der anschließende Austausch und Diskurs.

Beide Formen lassen sich nicht direkt in das Stufenmodell von Wildt (vgl. Abbildung 9) einordnen, sondern sind, je nach Umsetzung, Bestandteile der unterschiedlichen Stufen.

### Problemorientiertes Lernen

Die nächste Form, die zugleich eine Stufe im Wildts Modell darstellt, ist das problemorientierte (auch problembasierte) Lernen, welches *echte* (authentische), Problemstellungen zum Wissenserwerb und Ausbildung von Fähigkeiten in einer explorativen und selbstgesteuerten Umgebung verwendet.

> Studierende erarbeiten in problembasierten Lernumgebungen ihr gesamtes Wissen anhand von echten beruflichen Problemen, die didaktisch aufbereitet wurden [Wid09].

Problemorientiertes Lernen ist ebenso ein sehr breit verwendeter Begriff, weswegen eine Definition schwierig ist. Trotzdem gibt es einige gleiche Merkmale bei dieser Form, wie z. B. (1) eine authentische und fachbezogene Problemstellung (situiert), (2) eine vordefinierte und klare Aufgabenstellung (didaktisch aufbereitet), (3) oftmals eine Betreuung durch einen Lernbegleiter (Tutor oder Lehrender) und (4) eine meist induktive Lehrform, um problemspezifische Lösungen zu generalisieren. [Wil11; Fel+00; Wid09; Pri04]

Die Gewichtung der einzelnen Merkmale ist abhängig von den intendierten Lernzielen und der zur Verfügung stehenden Zeit. Das Basismodell nach Zumbach unterteilt diese Form in Problempräsentation, Problemdiskussion, individueller Lernphase und in eine Abschlussdiskussion (vgl. Abbildung 11) [Zum03]. Aus diesem Ablauf sind die Arten der Kommunikation, zwischen den Lernenden aber auch den Betreuenden und die zeitlichen Dimensionen (z. B. werden Teile zuhause bearbeitet) nicht ersichtlich. Doch aufgrund des diskursiven Charakters kann problemorientiertes Lernen auch als Form des kooperativen Lernens angesehen werden.

Als Nächstes möchte ich noch kurz auf einige spezifischere Umsetzungen des aktiven Lernens eingehen, bevor ich mich der Wirksamkeit der Formen beschäftige.

### Weitere Elemente und Methoden

Für Brinker et al. ist die Motivierung vor dem Einsatz aktivierender Methoden ein elementarer Bestandteil, um bei Lernenden das Interesse an den Lerninhalten zu wecken [BS08; WB16, S.22]. Den Lernenden muss der Bezug des Themas und die damit verfolgten Ziele bewusst sein. Ihnen sollte eine Antwort auf die Frage „Was bringt

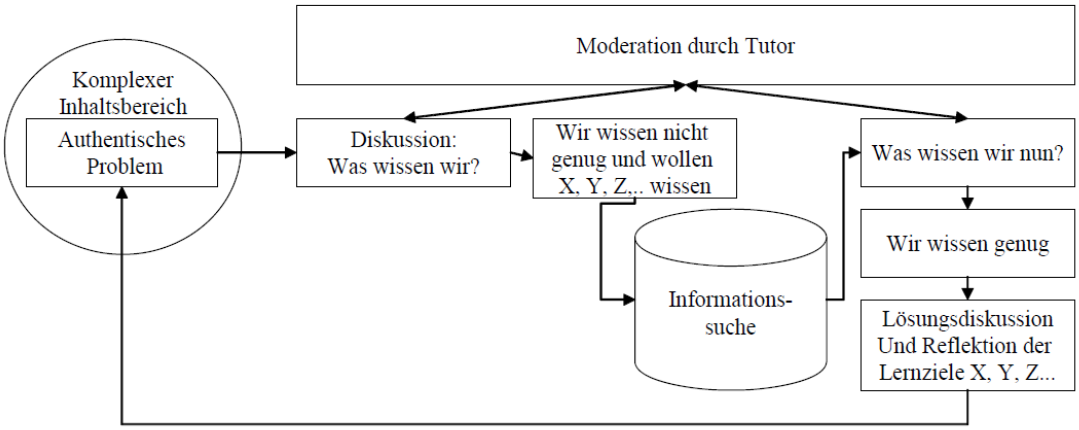



**Abbildung 11**: Prototypischer Ablauf des Problembasierten Lernens nach Zumbach

mir es mich mit diesem Thema bzw. Problem zu beschäftigen?“ gegeben werden. Aus konstruktivistischer Sicht ist das keine reine Selbststeuerung, wobei diese, wie bereits in Kapitel 3.4.1 diskutiert, nicht mit vorgegebenen Inhalten bzw. Curricula vereinbar ist. Jedoch ist „[o]hne den durch eine angemessene Motivierung erzeugten Lernwillen [...] jedes unterrichtliche Bemühen sinnlos“ [Hub07]. Erzeugte Motivation hängt dabei unter anderem von der Erwartungshaltung der Studierenden, dem Erfahrungsbereich der Lernenden und dem Lernprozess selbst ab [Hub07, S.16]. Ein hilfreiches Instrument bei der Gestaltung solcher motivationalen Elemente ist das ARCS-Modell von Keller und Kopp [KK87]. Das Akronym beschreibt gleichzeitig die vier Elemente des Modells:

- Aufmerksamkeit (*attention*): Emotionale und persönliche Informationen anbieten, Herausforderungen und interessante (relevante) Beispiele aufzeigen und Fragen stellen.
- Relevanz (*relevance*): Das Lernziel muss für den Lernenden als relevant erkennbar sein.
- Erfolgszuversicht (*confidence*): Gesetzte Ziele müssen erreichbar, jedoch auch herausfordernd sein.
- Zufriedenheit (*satisfaction*): Der Lernerfolg muss bereits während des Lernprozesses sichtbar sein und sollte den Erwartungen entsprechen.

Daraus können mehrere Gestaltungshinweise abgeleitet werden: Erstens sollte das Lernziel thematisch zum Studium passen und in eine klare Struktur eingebettet sein, welche die thematischen Zusammenhänge überblicken lässt. Zweitens ist für die Zufriedenheit eine Ergebnissicherung notwendig, sodass Lernfortschritte nachvollzieh-

bar dokumentiert sind und verdeutlicht werden. Unterrichtsleitende Fragen können dabei vom Lehrenden, aber auch von den Lernenden gestellt werden und sollten so berücksichtigt und beantwortet bzw. bearbeitet werden, um ein gewisses Maß an Selbststeuerung zu ermöglichen. Nach Brinker et al. können zum Beispiel Impulse, Fragen, Bilder, Gespräche oder Entscheidungsspiele als aktivierende Methoden in den Unterricht eingebettet werden [BS08, S.238ff]. Eine Besonderheit und mittlerweile weit adaptierte Methode ist der sogenannte *Flipped Classroom*-Ansatz, bei dem die Studierenden Inhalte zuhause erarbeiten und Fragen zu nicht klaren Inhalten vor der Vorlesung formulieren. Der Unterricht und der Lehrende wird durch diese bereits formulierten Fragen gesteuert und ist ein gutes Beispiel für eine lernenden-zentrierte Methode [JJ13].

### 4.2.2 Wirksamkeit

Die bisherige Diskussion ist von den Charakteristika des Aktiven Lernens zu einzelnen Formen übergegangen. Grundsätzlich wurde zwischen den zwei Perspektiven des aktiven Tuns und des aktiv informationsverarbeitenden Ansatzes differenziert. Abschließend möchte ich noch die Wirksamkeit näher betrachten, um die Wahl bei der Konzeptentwicklung besser begründen zu können. Mittlerweile gibt es mehrere Untersuchungen, die einen positiven Einfluss des aktiven Lernens auf das Verständnis und den Lernerfolg zeigen [Kon14; Spi07] und [Pri04]. Bei der Betrachtung der beiden Auffassungen aus Sicht des effektiven Lernens, stütze ich mich auf die Untersuchungen von Renkl, der dem informationsverarbeitenden Ansatzes eine höhere Effektivität zuspricht [Ren11]. Bei diesem Ansatz ist das reine „nicht unbedingt sichtbare Tun ausschlaggebend, vielmehr kommt es auf die aktive mentale Auseinandersetzung“ [Ren11] mit den Inhalten an. Aus konstruktivistischer Sicht, kann ein zu stark problemorientiertes Vorgehen die Lernenden überfordern, weswegen Renkl in diesem Zusammenhang den Vorteil (zumindest zu Beginn) von Lösungsbeispielen (auch *Worked Examples*) aufzeigt. Prince kommt zu einem ähnlichen Ergebnis bei seiner Metastudie über die Wirksamkeit der unterschiedlichen Formen des Aktiven Lernens. Das Verwenden von aktiven Elementen bzw. das Einbringen von Aktivitäten kann unter Umständen wirksam sein, jedoch kommt er zum gleichen Schluss wie Renkl, dass die Art der Aktivität eine Rolle spielt (vgl. informationsverarbeitender Ansatz) [Pri04, S.4]. Er zeigt durch die Analyse mehrerer Studien, dass die Einstellung der Studierenden gegenüber dem Thema einen (positiven) Einfluss auf den Lernerfolg hat. Dies deckt sich ebenfalls mit den Ausführungen über die Motivierung in Kapitel 4.2.1. Die unterschiedlichen Definitionen und Auffassungen des problemorientierten Lernens konnten keine signifikanten Verbesserungen gegenüber traditionellen passiven Lernens zeigen [Pri04, S.5ff]. Im Kontrast dazu konnten positive Auswirkungen des kooperativen und kollaborativen Lernens gezeigt werden [Pri04, S.4ff]. Sieht man pro-

blemorientiertes Lernen als Form des kooperativen Lernens an, so kann ebenfalls von einem positiven Effekt ausgegangen werden, jedoch ist dieser wieder von der jeweiligen Situation und Umsetzung abhängig. Zu beachten ist dabei, dass die Wirksamkeit einzelner spezifischer Anwendungen des problemorientierten Lernens nicht widerlegt ist, sondern dass diese immer im Kontext der Durchführung betrachtet werden muss. Dies deckt sich auch mit der hier gewählten forschungsmethodischen Vorgehensweise (vgl. Kapitel 2.3), welche bei der Evaluation die Erfassung der Einflussfaktoren fordert.

## 4.3 Schlussfolgerungen

Kapitel 4.1 und Kapitel 4.2 beantworten die Forschungsfrage 1, indem *Aktives Lernen* aus mehreren Perspektiven beschrieben und die Charakteristika dargelegt wurden. Weiterhin wurden in diesem Kontext weitere Formen wie das kooperative und kollaborative sowie weitere Ansätze beschrieben und deren Wirksamkeit diskutiert mit dem Ergebnis eines differenzierten, aber dennoch positiven Effekts auf den Lernerfolg. Aktives Lernen kann wirksam eingesetzt werden, jedoch erfordert dies eine Umstrukturierung der Lernumgebungen und Abläufe. Solche Methoden einzusetzen ist jedoch zeit- und betreuungsintensiv, weswegen sich z. B. forschendes Lernen nur schwierig in eine zweistündige Vorlesung integrieren lässt. Gleichermaßen ist die Vermittlung von Grundlagen mittels *Aktivem Lernen* sehr aufwändig. Die Diskussion der konstruktivistischen Theorien hat gezeigt, dass diese Grundlagen sich ebenfalls sehr gut mit instruktionellen Methoden vermitteln lassen. Letztendlich sind die äußeren Rahmenbedingungen bei der Auswahl von Methoden und Konzeption ebenso entscheidend wie lehr- und lerntheoretische Überlegungen.

Programmieren vereint theoretische Grundlagen mit problemlösendem Denken, welches auf Basis von Analogien und Heuristiken bei der Lösungsfindung von Problemen bzw. Bewältigung von Aufgaben unterstützt. Daraus kann eine Forderung nach einer eher zielorientierten Balance zwischen Instruktion und Konstruktion, durch Aktives Lernen, abgeleitet werden. Diese Unterschiede und die Beziehungen sind in Abbildung 12 dargestellt. Auf der instruktiven Seite werden die Aktivitäten vom Lehrenden durchgeführt, während im auf der konstruktiven Seite der Lernende im Vordergrund steht. Zur Ausbildung der problemlösenden Fähigkeiten können situierte und lernunterstützende Lernumgebungen konzipiert werden, die Studierenden die Auseinandersetzung mit Problemen ermöglichen und metakognitive und soziale Prozesse (kooperative oder kollaborativ) anregen. Dazu sind Methoden und Strategien notwendig, welche einerseits eher Aktivitäten der Lehrenden, aber ebenfalls die der Lernenden in einem stimmigen und zielorientierten Konzept kombinieren. Ein mögliches lehr- und lerntheoretisch begründetes Modell stellt der *Cognitive Apprenticeship* Ansatz dar, der


**Abbildung 12:** Wechsel und Balance zwischen Instruktion und Konstruktion

in den Phasen des *Modeling* den Fokus auf Lehrendenaktivitäten setzt, die nachfolgend (*Fading*) immer weiter nachlassen und auf die Seite des Lernenden übergehen. Ausgehend von Renkls und Hilberts Ergebnissen und den Ausführungen über die *Cognitive Load Theory* ist der Einsatz von Lösungsbeispielen (*Worked Examples*) für die Phase des *Modeling* zu empfehlen. Diese hier angedeutete Zusammenführung der Theorien wird in Kapitel 5 konzeptionell umgesetzt und näher beschrieben. Hauptaugenmerk bei dem Konzept stellen die Konzepte der Programmierung dar, die für das erfolgreiche informatische Problemlösen ausgewählt, angewendet und kombiniert werden müssen.

## 4.4 Aktives Lernen in der Programmierausbildung

Zuvor möchte ich noch auf vorhandene Ansätze des *Aktiven Lernens* im Bereich der Programmierausbildung, primär Studium und universitäre Ausbildung, eingehen. Bei den nachfolgenden Auszügen, konzentriere ich mich zuerst auf die allgemeinen Ansätze des *Aktiven Lernens* und gehe anschließend auf kooperatives, kollaboratives und problemorientiertes Lernen ein.

### 4.4.1 Allgemeine Ansätze

Programmieren ist die Verbindung von theoretischen Grundlagen und Anwenden einzelner Konzepte der Programmierung, wie z. B. Variablen und Schleifen, mit dem Ziel diese später zu Lösungen bzw. Algorithmen zu kombinieren. Hassinen und Mäyrä haben den Zusammenhang zwischen der Anzahl bearbeiteter Programmieraufgaben und der Prüfungsnote untersucht. In der Studie wird nicht direkt auf aktives Lernen verwiesen, jedoch stellt das gezielte Bearbeiten von Aufgaben und Zusatzaufgaben eine Art Handlungsorientierung dar. Die Auswertung der Studie hat mehrere interessante

Ergebnisse und Ansatzpunkte hervorgebracht:

- Die Anzahl der bearbeiteten Übungsaufgaben korreliert stark mit der Prüfungsnote. Überdurchschnittlich gute Studierende bearbeiteten neben Übungsaufgaben zusätzlich noch weitere, während Studierende, die lediglich das Minimum an Aufgaben bewältigten, schlechtere Noten erzielten. Ein weiteres Phänomen sind Studierende, die durch wenig Aufwand sehr gute Ergebnisse erzielen konnten. Studierende hingegen, die einen hohen Aufwand betrieben haben, konnten teilweise nur schlechte Prüfungsergebnisse erreichen. Dabei war die Anzahl der schlechten Studierenden trotz hohem Aufwand geringer als die der guten Studierenden mit wenig Aufwand. Eine mögliche Erklärung dafür ist das mögliche Vorwissen bei den guten Studierenden.
- Eine weitere interessante Beobachtung ist, dass durchschnittlich gute Studierende weniger Aufgaben bearbeitet haben als ihre Kommilitonen mit schlechteren Leistungen, ohne das eine Erklärung dafür gefunden wurde.
- Hervorragende Leistungen sind ohne intensives Üben nicht möglich mit Ausnahme der Studierenden mit Vorkenntnissen wie in den vorherigen Punkten beschrieben.

Diese Ergebnisse zeigen, dass eine höhere Aktivität seitens der Studierenden einen positiven Effekt auf den Lernerfolg und somit auf die Prüfungsleistungen hat. Jedoch ist die Art der Aufgaben und auf welche Konzepte sie sich beziehen nicht weiter geschildert und wurden nicht differenziert ausgewertet. Der Unterschied zwischen den durchschnittlichen und schlechten Studierenden kann möglicherweise durch mangelndes Feedback und Begleitung der Studierenden erklärt werden. Reines Üben ohne Rückkopplung und Betreuung kann, aber muss nicht zwingend erfolgreiches Lernen induzieren. Dies steht im Kontrast zu Renkls Extrapolierung der Beteiligung von Schülern auf alle Aktivitäten des Aktiven Lernens (vgl. Kapitel 4.1). Eine Analyse und Aufbereitung der auftretenden Fehler bei den Studierenden als Vorbereitung für Wiederholungseinheiten erscheint hier angebracht und könnte den positiven Effekt weiter stärken.

Esper et al. haben explorative Hausaufgaben anhand Aktiven Lernens entwickelt, was durch gezielte Aufgabenstellungen die Aktivitäten *Erforschen* (Erkunden), Vorhersagen, Erklärungen und Fragen für einzelne Elemente (z. B. `for`-Schleifen) anstoßen soll [ESC12]. Das Ziel sind situative Erfahrungen in denen Studierende sich Konzepte selbst erarbeiten und anschließend anwenden sollen was von den Autoren als *Cognitive Apprenticeship* ohne Tutoren bezeichnet wird. Diese Hausaufgaben sollten jeweils vor der nächsten Vorlesung bearbeitet werden und wurden mittels eines Quiz zu Beginn der Vorlesung abgefragt. Ein direkter Einfluss auf den Lernerfolg wurde nicht untersucht, jedoch scheinen die Studierenden besser auf nächste Vorlesungen vorbereitet

<table>
<tr><th>Timing</th><th>Skills</th><th>Activity</th><th colspan="3">Evidence Statements</th></tr>
<tr><td rowspan="3">Early</td><td rowspan="2">Read & Trace</td><td rowspan="2">Explore</td><td colspan="3">Students can read and trace well-written code. They can identify, label, or explain features, data, patterns, output, behaviors, etc. They can interact with a code visualizer or watch an instructional video. They can remix code by changing small sections of existing code and attempt to explain the consequences.</td></tr>
<tr><td colspan="3">Students perform categorization, matching, and choice activities to link terms and concepts with examples. Students compare code segments that accomplish the same task.</td></tr>
<tr><td>Debug</td><td>Explain</td><td colspan="3">Students who are shown the location of a bug can explain it (and/or fix it).</td></tr>
<tr><td>Middle</td><td>Read & Trace</td><td>Discern differences, dissonances</td><td colspan="3">Students can discern differences, e.g. determine if two code segments perform the same tasks. Students can resolve dissonances, e.g. solve Parsons problems (by re-ordering mis-ordered code).</td></tr>
<tr><td rowspan="4">Later</td><td rowspan="3">Write & Debug</td><td rowspan="3">Complete</td><td colspan="3">Students complete supplied code. Sample progression:</td></tr>
<tr><td rowspan="3"></td><td>Fill in the blank(s).</td><td>Fix a bug (bug location provided).</td></tr>
<tr><td colspan="2">Find and fix a bug (bug location not provided).</td></tr>
<tr><td colspan="2">Complete skeleton code (where minimal code provided).</td></tr>
<tr><td>Create</td><td colspan="3">Students write code from scratch.</td></tr>
</table>

**Abbildung 13**: Progressiver Lernprozess zur Dekonstruktion von Quelltext

zu sein. Den Studierenden waren teilweise die Ziele und wofür die Aufgaben bearbeitet werden mussten unklar, was für eine bessere Präzisierung der Aufgabenstellung oder Betreuung spricht. Eine gezielte Adressierung der Konzepte erscheint sinnvoll und dem Programmieren zuträglich und wurde von den Studierenden als bereichernd angesehen. Leider wurden noch keine Untersuchungen zur Auswirkung auf die Prüfungsleistungen oder auf die Programmierfähigkeit durchgeführt. Trotzdem scheint der Ansatz von zuhause einfach zu bearbeiteten Übungen und das Durcharbeiten von Konzepten als Vor- oder Nachbereitung vielversprechend. In Kontrast dazu steht das Konzept von Duffany, welcher direkt Übungen in Computerlabor durchführen lässt [Duf17]. Dies setzt aber entsprechende Räumlichkeiten und kleine Studierendenzahlen voraus.

Griffin beschreibt ausgehend von lerntheoretischen Überlegungen ein theoretisches Konzept zur Dekonstruktion von Quelltexten durch Lesen, das Nachverfolgen des Programmablaufs und *Debugging* [Gri16]. Die Grundlage stellen die *worked examples* (vgl. Punkt 3.4.3) und die Erkenntnisse von Lister über das Leseverständnis [Lis+04] dar, die um Elemente des *Cognitive Apprenticeship* erweitert werden. Abbildung 13 zeigt den prinzipiellen Ablauf und die einzelnen Aktivitäten. Ziel dieses Ablaufs ist das Ausbilden kognitiver Strukturen, um Ähnlichkeiten und Unterschiede in Quelltexten zu finden. In der ersten Phase (*Early* siehe Abbildung 13) steht das Lesen und Nachverfolgen im Vordergrund, um die Programme zu erkunden. Das Erstellen von Programmen erfolgt zwar erst in der letzten Phase, jedoch ist beim Erkunden von Programmen bzw. Programmfragmenten bereits das Verändern und Beobachten der Auswirkungen ein Bestandteil. Dieser Ansatz ist dabei besonders interessant, denn er fordert und ermöglicht Studierenden eine von Neugierde getriebene die Interaktion mit Programmen und Quelltexten, weswegen solche Möglichkeiten bei der Gestaltung von Lernumgebungen beachtet werden sollten. Einen ähnlichen Ansatz zur Förderung des Verständnisses beim Lesen von Quelltexten und späterem Erstellen wurde von Figas et al. theoretisch entwickelt [FBH15] und von unter anderem mir weiterentwickelt

und empirisch evaluiert [Fig+16].

Eine breit eingesetzte Methode ist der Einsatz von sogenannten *Classroom Response Systems*[11] (kurz CRS) mit denen Studierende während der Vorlesung Fragen beantworten können [Cal07]. Solche Systeme sind entweder in Kombination mit spezieller Hardware oder nur durch Software und Verwendung auf eigenen Mobilgeräten (z. B. Smartphone) umgesetzt. Judson et al. haben in einer Metastudie nachgewiesen, dass der Einsatz solcher Systeme tiefere Lernprozesse auslösen und somit ggf. metakognitive Prozesse anstoßen kann und dadurch die studentische Partizipation erhöht [Ea02; San10]. Kay et al. zeigen in einer weiteren Metastudie, dass die Verwendung von CRSs das Anpassen der Geschwindigkeit und Methoden (Instruktion) auf Basis der Antworten möglich ist [KL09]. Die Regulierung des Tempos und der Wiederholungsblöcke durch die Studierenden, können dadurch deutlich besser gesteuert werden. Beliebte Systeme unterstützen jedoch nur wenige Fragetypen wie z. B. Ja-Nein-Fragen, Freitextfragen oder Mehrfachauswahlfragen (*Multiple Choice Questions*). Letztere sind besonders im angelsächsischen Raum beliebt, was sich auch in den Instrumenten zur Erfassung der Programmierfähigkeit widerspiegelt (vgl. Kapitel 3.3.2). Robbins hat ein Software-Clicker-System entwickelt, welches eine Darstellung von Quelltexten in Fragen ermöglicht [Rob11; HA13]. Im Rahmen meiner Forschungstätigkeit konnte ich zwei weitere Fragetypen für das Programmieren entwickeln. Dabei wurde ein Quiz entwickelt, welches zusätzlich zu den bisherigen Fragetypen um einen Syntax-Fragetyp sowie einen Codefragment-Fragetyp für die Programmiersprache *C* erweitert wurde [EH15a]. Bei der Überprüfung der Fragen wird die konkrete Syntax auf Basis eines *C-Parsers* überprüft und die fehlerhafte Stelle, wie in Abbildung 14 zu sehen, markiert. In diesem konkreten Beispiel fehlt das Semikolon am Ende der Zeile und wird entsprechend in der Ausgabe angezeigt. Die Evaluierung des Quizzes hat sehr positive Ergebnisse erbracht, da Studierende durch die *Gamification*-Elemente [Kap12, vgl.], wie z. B. eine limitierte Anzahl von Leben, sich intensiv mit den Fragen und Syntaxelementen beschäftigt haben. Raab et al. haben eine Art interaktive Folien entwickelt, die Studierenden das Beantworten von Einfach- und Mehrfachantwortfragen, das Annotieren von fehlerhaften Stellen in Quelltext und Verschieben von Fragmenten im Rahmen eines Seminars über *Clean Code* ermöglicht [RFW12]. Eine von den Autoren genannte Verbesserungsmöglichkeit ist das Schreiben bzw. Verändern von Programmen, welches in dem System nicht möglich ist. Abschließend möchte ich noch auf die Untersuchung von Chamillard eingehen, die CRSs primär als aktivierendes Elemente in Hinsicht auf die studentische Partizipation analysiert hat mit dem Ergebnis, dass durchschnittlich 80% der Studierenden die Fragen beantwortet haben [CB00].

In den theoretischen Überlegungen und bei der Diskussion der Wirksamkeit ist mehr-

[11] Diese werden teilweise auch als *Clickers* bezeichnet.

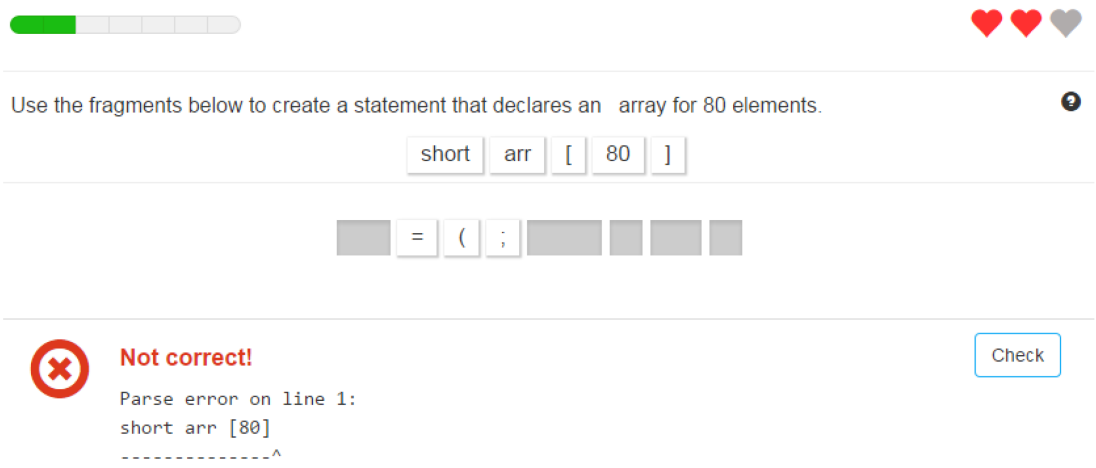



**Abbildung 14**: Code-Fragment-Frage in einem Quiz

mals auf das Konzept *Cognitive Apprenticeship* (kurz CA) als mögliche Struktur zur Einbettung von aktiven aber auch rezeptiven Elementen verwiesen worden. Astrachan und Reed haben dieses Konzept in Form eines *Outside-In-Teaching* [Mey03], bei dem ausgehend von komplexeren Beispielen Konzepte erklärt werden, umgesetzt [AR95]. Weitere Ansätze des *Cognitive Apprenticeship* wurden von Black theoretisch begründet sowie von Caspersen und Bennedsen anhand von Aufgaben zur Objektorientierung untersucht [Bla06; CB07]. Kölling und Barnes entwickelten *BlueJ* auf Basis des CA, um Lernenden die Möglichkeit zu bieten, auch komplexere Programme zu erkunden und den dazugehörigen Quelltext zu lesen [KB04]. Jin und Corbett haben einen kognitiven Tutor auf Basis des CA in Form von Arbeitsblättern entwickelt, der Studierenden bei der schrittweisen Erarbeitung der Lösungen von Programmieraufgaben unterstützt [JC11]. Studierende werden hierbei gezielt durch die Problemanalyse und die Zerlegung der Probleme und anschließenden Lösung geleitet, wobei in den einzelnen Schritten Aktivitäten der Lernenden vorausgesetzt werden. Bei der abschließenden Evaluation konnte eine positive Einstellung der Studierenden gegenüber dieses Tutors festgestellt werden. Der Fokus bei der Entwicklung lag jedoch auf der Aufgabenbearbeitung außerhalb der Vorlesung und das Konzept wurde nicht in Bezug auf den Einfluss auf die Programmierfähigkeit untersucht. Vihavainen et al. haben das Konzept *Cognitive Apprenticeship* in einer *radikalen* Weise als *Extreme Apprenticeship* weiterentwickelt und evaluiert [VPL11]. Das Konzept setzt dabei auf vier Leitsätze:

- *Learning by Doing*: Zum Erlernen von Programmieren, muss programmiert werden.
- *Keine Kompromisse*: Eine Fertigkeit wird solange geübt, wie es für einen Lernenden notwendig ist.

- *Kontinuierliches Feedback*: Der Prozess ist von einem gegenseitigen Feedback zwischen Lehrenden und Lernenden geprägt, um den Lernprozess zu überwachen und entsprechend reagieren zu können.
- *Vom Lehrling zum Meister*: Ziel ist es, dass Lernende zum Meister werden.

Das Konzept wurde in einem CS1-Kurs angewendet, bei dem der Vorlesungsanteil von fünf auf zwei Stunden reduziert wurde, um mehr Raum für Übungen zu schaffen. In der Vorlesung wurden hauptsächlich Konzepte und Beispiele erklärt und diskutiert. Die Evaluation hat eine Reduzierung der Durchfallquoten um teilweise 20% oder mehr gezeigt [VL13]. Neben den positiven Aussagen bezüglich der Motivation und des *Learning by Doing* Aspektes über den Kurs wurde vor allem die Verfügbarkeit eines Ansprechpartners in den Übungen hervorgehoben. Der Ansatz scheint zu funktionieren, jedoch erfordert die Umsetzung einen großen Ressourcenaufwand, da Lehrende und Tutoren pro Woche durchschnittlich acht Stunden zur Verfügung stehen müssen [DD12; Vih+13a]. Knobelsdorf et al. konnten vergleichbare Resultate durch den Einsatz von Tutorien in Form des CA nachweisen [KKB14]. Zusammengefasst existiert in der Literatur eine starke Evidenz für den Einsatz von Konzepten auf Basis des *Cognitive Apprenticeship*, auch wenn diese bisher meist in den Übungsstunden eingesetzt wurden. Eine systematische Verwendung des Ansatzes über alle Phasen existiert innerhalb von Vorlesungen bisher noch nicht. Für die Phasen des Vorführens von *Modeling* wird unter anderem *Live Coding* verwendet, bei dem der Lehrende vor den Studierenden programmiert und seine Gedanken mitteilt [Rub13; SS15]. Alternativ werden *Worked Examples* (vgl. Punkt 3.4.3) eingesetzt, die nachweislich einen positiven Effekt auf das Verständnis von Konzepten haben [MMG15; CB07; Rei+08; Atk+00]. Tirronen und Isomöttönen haben auf Basis der *Cognitive Load Theory* und im Speziellen der *Worked Examples* Übungsaufgaben für das selbstgesteuerte Lernen konzipiert [TI11]. Dabei haben sie drei Probleme bei der Umsetzung identifiziert: (1) Selbstgesteuerte Aktivitäten werden teilweise in Gruppen gehemmt, (2) individuelle Lernpfade sind konträr zu dem hohen Tempo des Kurses (3) und Studierende hatten Probleme den Inhalt bzw. den Nutzen der bereitgestellten Materialien zu erkennen. Trotzdem konnte eine geringere Durchfallquote und seitens der Studierenden eine verbesserte Lernsituation festgestellt werden [IT13].

### 4.4.2 Kooperatives und kollaboratives Lernen

Die bisher vorgestellten Ansätze können als allgemeines Aktives Lernen bezeichnet werden. Nachfolgend möchte ich noch Studien und Konzepte über die speziellen Formen des *Aktiven Lernens* diskutieren, welche sich die Zusammenarbeit zwischen Lernenden zu nutze machen. Denn auch in Vorlesungen können Synergien zwischen Lernenden als bewusste Methode zum aktiven Vermitteln der Inhalte eingesetzt werden.

O'Grady et al. haben in einer Metastudie den Einsatz von problemorientiertem Lernen in der Informatiklehre untersucht. Im Bereich des Programmierens werden Ansätze des problemorientierten Lernens bereits eingesetzt, jedoch wurden diese nur zu etwa zwei Dritteln evaluiert. Ein Großteil der Studien berichtet jedoch von positiven Erfahrungen, wobei hier weitere Evaluationen notwendig sind [OGr12]. Ambrosio und Costa haben in einem Algorithmen- und Programmierkurs neue Konzepte problemorientiert eingeführt, indem Gruppen jeweils eine Aufgabe zu einem Konzept in mehreren Schritten bearbeitet haben. Das Vorgehen sieht mehrere Treffen und individuelle Vorbereitungen der Lernenden vor. Studierende monierten den teils hohen Arbeitsaufwand, schätzten jedoch den positiven Einfluss der Gruppenarbeit. Trotzdem konnte keine signifikante Verbesserung der Noten beobachtet werden, aber im Gegensatz dazu eine Reduzierung der Durchfallquote von 45% auf 21% [AC10]. Der Ansatz erscheint vielversprechend, wenn auch arbeitsintensiv für die Lernenden und ist nur mit kleineren Kohorten durchführbar, denn bereits bei 100 Studierenden müssen 20 Gruppen, neben den normalen Übungen, zusätzlich betreut werden. Nuutila et al. kamen in einer Studie zur Skalierbarkeit des problemorientierten Lernens zu einem ähnlichen Schluss, dass der Ansatz in Veranstaltungen mit vielen Studierenden derzeit nur mit hohem Aufwand und Ressourcen umsetzbar ist [Nuu+08]. Dennoch ist eine Aktivierung der Studierenden durch Problemstellungen erfolgversprechend und sollte in ähnlicher Art und Weise erfolgen.

Gonzalez untersuchte kooperatives Lernen in *CS1* (Anfänger in *computer science*) und konsekutiven *CS2* Kursen mit insgesamt 147 Studierenden. Die Vorlesungen wurden in drei Blöcke unterteilt:

- 15 - 20 Minuten Gruppendiskussion zur Wiederholung,
- 20 - 30 Minuten Kurzvortrag mit neuen Themen,
- und 20 - 30 Minuten geplante Aktivität.

Der letzte Block der geplanten Aktivität beinhaltete unterschiedliche kooperative Aktivitäten, wie z. B. Programmieraufgaben im Labor am PC oder mit der Hand, Rollenspiele und veranschaulichende Analogien (im Original *manipulative*). Unter anderem wurden Plastikbecher zur Darstellung von Variablen und Speicherallozierung verwendet. Bei der Auswertung konnten positive Effekte der aktiven und kooperativen Elemente nachgewiesen werden, wenn auch der Effekt nur klein ist. Die Notenverteilung hat sich nur Geringfügig geändert, jedoch konnte ein größerer Zulauf beim konsekutiven Kurs festgestellt werden. Ähnliche Ergebnisse konnten auch von Beck et al. nachgewiesen werden [BC08]. In weiteren Untersuchungen, bei denen zwei Kurse unterschiedlich unterrichtet wurden, konnte ein positiver Effekt bei der Bearbeitung von Programmieraufgaben, jedoch nicht bei anderen Aufgabentypen, nachgewiesen werden [BC13]. Möglicherweise konnten mehr Studierende motiviert und begeistert

werden sich auch weiterhin mit Programmieren und informatischen Inhalten zu beschäftigen [Gon06]. Gonzalez kommt zum Schluss, dass der Ansatz gut ist, jedoch noch Verbesserungen bei der Durchführung durch zur Hilfenahme von Werkzeugen notwendig ist:

> To improve the active learning experience for all students, access to lab that fosters group work with adequate technology is also needed, plus better classroom management tools to control the increased amount of student feedback and other work and faciliate rapid feedback for students [Gon06].

Beck et al. thematisieren ebenfalls die Überwachung der Gruppenarbeit bei größeren Studierendenzahlen, da diese von einem Einzelnen nicht zu leisten ist. Ein Vorschlag zur Lösung dieser Problematik ist laut Beckt et al. eine Art Lerntagebuch, welches später ausgewertet wird, um problematische Gruppen zu identifizieren [BC13, S.18]. Diese Verbesserungsvorschläge wurden teilweise im Rahmen dieser Arbeit umgesetzt und werden in Kapitel 6 näher vorgestellt.

Ein interessanter Ansatz für aktive Aufgaben in der Vorlesung ist das von Jam Jenkins entwickelte *JavaWIDE* [Jen+10]. Dabei handelt es sich um einen web-basierten Editor für Java, der direkt ab der ersten Übungsstunde zum Schreiben von Programmen verwendet werden kann und nebenbei die Erstellung von Wikiseiten ermöglicht [Jen+12]. Aktives Lernen wird als Ausgangspunkt für die Entwicklung gegeben, ohne aber konkrete und evaluierte Beispiele zu nennen, allerdings erlaubt diese Plattform kollaboratives Codieren. Die technische Plattform ist zwar stark auf die Bearbeitung von Übungsaufgaben ausgelegt, jedoch könnte dies aufgrund der webbasierten Umsetzung auch sehr gut in die Vorlesung selbst verlagert werden. Einzig die Darstellung ist für das Anzeigen auf Leinwänden mittels *Beamer* nicht optimiert.

Neben den kooperativen und kollaborativen Ansätzen existieren noch Ansätze des forschenden Lernens. Clifton Kussmaul untersuchte in einer Studie das studentische Feedback über *Process Oriented Guided Inquiry Learning* (POGIL), welches als Form des problemorientierten Lernens angesehen werden kann [Kus12]. POGIL ist ebenfalls eine Art forschendes Lernen bei dem jedoch die Aktivitäten und der Prozess bereits vorgegeben sind. In der Studie wurden Datenstrukturen wie z. B. Stacks oder Queues von Studierenden in Gruppen bearbeitet. Die Aktivitäten beinhalten unter anderem auch Programmieraufgaben und das Entwerfen von Testfällen. Eine anschließend durchgeführte Befragung bei den Studierenden ergab ein eher neutrales Bild ohne starke positive oder negative Meinungen. Einzig die Gruppendiskussionen und der Ideenaustausch wurden als positiv bewertet. Der gewählte Ansatz wurde nicht auf fundamentale Konzepte der Programmierung, sondern auf komplexere Datenstrukturen angewendet, mit besonderer Wertlegung auf das aktive Programmieren. Eine Evaluation

des Ansatzes wurde (noch) nicht durchgeführt. Tao und Nandigam beschreiben einen fallbasierten (vgl. Abbildung 9) Ansatz des *Aktiven Lernens*, bei denen Studierende sich neue Konzepte anhand von speziell formulierten Fallstudien, wie z. B. Darstellung von Datumsangaben für die Erstellung einer Klasse aus objektorientierter Sicht, erarbeiten. Im Unterricht selbst müssen die Studierenden anschließend Programmieraufgaben unter der Verwendung mehrerer unterschiedlicher Klassenentwürfe durchführen. Die Studierenden nahmen die Übungen insgesamt als hilfreich auf, merkten jedoch an dass die Bearbeitung eher in den Computerräumen stattfinden sollte [TN16].

Ein weiterer Ansatz des kollaborativen Lernens beim Programmieren ist das sogenannte *Pair Programming*, bei dem in der Regel zwei Studierende nebeneinander an einer Aufgabenstellung arbeiten und zur Lösung miteinander kommunizieren und sich gegenseitig unterstützen. Diese Methode wird unter anderem in Vorgehensmodellen wie z. B. *Extreme Programming* propagiert. Estácio et al. haben in mehreren Studien den Einfluss von *Pair Programming* und einer Sonderform (*Coding Dojo Randori*) auf das Erlernen des Programmierens untersucht. Sie konnten eine Verbesserung beim Umgang mit der Syntax von Programmiersprachen und beim Algorithmenentwurf im Vergleich mit allein arbeitenden Studierenden quantitativ und qualitativ nachweisen [Est+15a; Est+15b]. In weiteren Studien konnten ebenfalls positive Effekte nachgewiesen werden [RPB17; Woo+13], wie beispielsweise dass Studierende in solchen Arrangements Aufgaben schneller lösen können [RW11; MHW03].

Zuletzt möchte ich auf einen weiteren Ansatz des Aktiven Lernens eingehen. Das von Eric Mazur geprägte Konzept *Peer Instruction* (PI), kombiniert CRSs und kooperatives Lernen [Maz97]. Hake konnte in einer Studie mit über 6000 Studierenden die Wirksamkeit von PI gegenüber traditionellen Methoden nachweisen [Hak98]. Porter et al. fassen den grundsätzlichen Ablauf wie folgt zusammen:

> PI is characterized by asking challenging, in-class conceptual questions of students. For each question, students individually respond, discuss the question in small groups, and respond again based on their new understanding. These questions should target common misconceptions and/or core course concepts [Por+16].

Für ein Gelingen müssen die Studierenden sich in der Regel auf die Vorlesungen vorbereiten, damit die Zeit optimal genutzt werden kann. Die Methode wurde von Porter et al. an sieben verschiedenen Universitäten evaluiert mit weitgehend gleichen Ergebnissen, die sich auch mit älteren Studien decken [PBS13; Sim+10]: (1) Der Großteil der Studierenden mag PI und würde diese Methode weiterempfehlen, (2) jedoch müssen die Intentionen bei der Verwendung der Methode klar kommuniziert werden, sodass sich die Lernenden nicht überwacht fühlen (3) und die Art der Benotung bei der Beantwortung der Fragen hat einen Einfluss auf die Teilnahme. Porter et al.

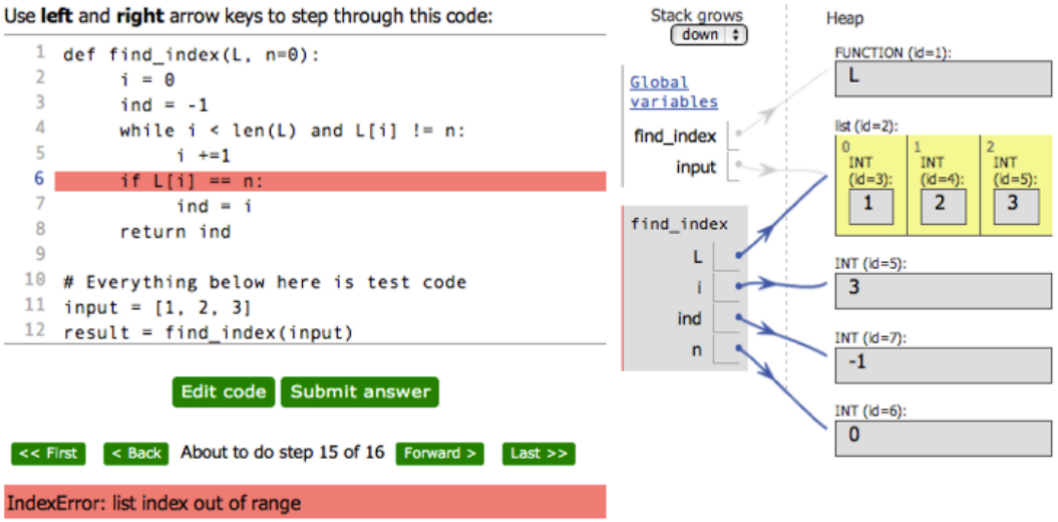



**Abbildung 15**: Beispielaufgabe aus dem Python Classroom Response System

konnten in einer früheren Studie zusätzlich eine geringere Abbrecherquote gegenüber traditionellen Kursen nachweisen [PBS13]. Ein im angel-sächsischen Raum häufig zu beobachtendes Phänomen ist der Fokus auf die Benotung (*Grading*) bei der Auswahl von Methoden, weswegen auch häufig *Multiple-Choice-Fragen* verwendet werden, um den Aufwand niedrig zu halten. Dies steht im Kontrast zur Forderung nach weniger verpflichtenden Veranstaltungen im Rahmen des Bologna-Prozesses, die eine höhere Eigenverantwortung der Studierenden erfordert. Studentische Partizipation kann und sollte auch anders geweckt und motiviert werden. Das soll keinesfalls den nachweislichen Einfluss auf den Lernerfolg anfechten, sondern ist eine zu treffende Entscheidung bei der didaktischen Konzeptentwicklung [Por+11; SPS13].

Zingaro et al. identifizierten den bereits bei den CRSs angemerkten Mangel von Aufgabentypen, die auf das Programmieren zugeschnitten sind und haben das *Python Classroom Response System* entwickelt [Zin+13]. Studierende können, anstatt von *Multiple-Choice-Questions*, Programme selbst schreiben und diese zur Überprüfung mittels *Unittests* einreichen. Aus fehlgeschlagenen Testfällen leitet der Lehrende anschließend problematische Inhalte und nicht verstandene Konzepte ab, um diese zu thematisieren und nochmals zu erklären. Die Beschreibung ist bezogen auf die Verwendung dieses Werkzeugs direkt in Vorlesungen unpräzise da kein tieferes Konzept bzw. der Ablauf nicht so beschrieben ist, um die Aussagen empirisch nachvollziehen zu können. Die Studierenden können die Fragen online bzw. als Vorbereitung zuhause direkt bearbeiten und bei Fehler den Ablauf schrittweise nachverfolgen (vgl. Abbildung 15). Dieser Ansatz wurde nur von Seiten der Lehrenden als positiv bewertet und als Bereicherung

empfunden. Besonders das Schreiben von Programmen wurde als Aktivität hervorgehoben und von Parreira et al. für die Sprache *C* umgesetzt [MPC15], jedoch auch nicht weiter evaluiert.

### 4.4.3 Zusammenfassung

Der Streifzug durch den Stand der Forschung zum Aktiven Lernen im Kontext Programmieren hat einige vielversprechende Konzepte und Methoden ergeben, deren Wirksamkeit anhand von Metastudien und literarischen Zusammenfassungen belegt wird.

Ein großer Teil der Ansätze adressiert die Bearbeitung der Übungsaufgaben von zuhause aus oder während der Übungen mittels *Coginitive Apprenticeship* und *Worked Examples*. Diese können gezielt eingesetzt werden, um selbstgesteuertes Lernen zu ermöglichen. Dabei zeigt sich, dass nur wenige evaluierte Ansätze zu Übungen innerhalb von Vorlesungen existieren, die gezielt die Konzepte *inklusive* der Implementierung auf Seiten der Studierenden im Fokus haben. Es werden zwar bereits Übungen zum Lesen und Schreiben verwendet, aber primär für das Verwenden der einzelnen Programmierkonzepte und nicht im Bezug auf das gezielte Problemlösen und Ausbilden von Heuristiken, das ich in Kapitel 3.5 als wichtig und notwendig begründet und diskutiert habe. Ein Teil der Studien ist eher eine Beschreibung von Experimenten ohne eine anschließende Evaluation (vgl. [OGr12]), wie bereits von Simon [Sim15] in einer Metastudie festgestellt wurde.

In den Studien von Hassinen Mäyrä, in den Ausführungen von Griffin und Figas et al. sowie von Vihavainen et al. wurden Programmier- und Schreibaktivitäten untersucht bzw. gefordert, da diese einen Einfluss auf das Verständnis und nachweislich auf die Noten haben. Gonzalez verwendet in ihrer Umsetzung des kooperativen Lernens ebenfalls Besuche im Computerlabor, bei denen Studierenden zusammen Aufgaben bearbeiten und sich dabei gegenseitig unterstützten. Unter anderem findet dieses Konzept bei kollaborativen und forschenden Ansätzen, wie z. B. *Pair Programming*, eine Anwendung. Gruppen- bzw. Teamarbeit scheint einen positiven Einfluss auf den Lernprozess zu haben, was auch aus sozial-konstruktivistischer Sicht diskutiert wird (vgl. Kapitel 3.4.1). Gonzalez und Beck et al. weisen auf den mit Gruppenarbeit verbundenen Aufwand und notwendige Betreuung hin, die durchaus ressourcenintensiv sein kann. Nuutila et al. kommen bei der Analyse von problemorientierten Ansätzen zu ähnlichen Schlüssen, wobei das Konzept zur Aktivierung der Studierenden beiträgt. Die Studien über problem- bzw. fallorientiertes Lernen und die vorherige Diskussion zeigen alle die Notwendigkeit und die Vorteile dieser Ansätze auf, da den Studierenden die Frage nach dem “Warum muss ich das machen?”beantwortet wird und im konstruktivistischen Sinne zur Erstellung situativer Lernumgebungen beiträgt. Daraus entwickelten

sich unter anderem Forderungen nach einer guten inhaltlichen und organisatorischen Struktur und adäquater Technik, die solche Aktivitäten entsprechend unterstützt.

*Cognitive Apprenticeship* wird ebenfalls in unterschiedlichen Formen primär als Konzept der Übungsbetreuung mit sehr guten Evaluationsergebnissen eingesetzt. Gemein ist allen Ansätzen, dass Studierende aktiv Beispiele und Aufgaben erkunden und bearbeiten, um sich mit den Konzepten auseinanderzusetzen. Dies wird entweder durch den *massiven* Einsatz von Tutoren oder von speziell erstellten Arbeitsblättern oder den sogenannten *Worked Examples* erreicht. Letztere sollen unter anderem das selbstgesteuerte Lernen unterstützten, indem die Studierenden sich eigenständig durch Lösungsbeispiele und vorgegebenen Schritten neue Konzepte erarbeiten. In der Vorlesung selbst wird das sogenannte *Live Coding* eingesetzt, welches den ersten Phasen des CA entspricht. Unabhängig von der konkreten Form des CA konnten gegenüber traditionellen Methoden bessere Ergebnisse und geringere Durchfallquoten erreicht werden.

Häufig werden sogenannte CRSs als aktivierende Methode eingesetzt, die nachweislich Denkprozess auf den oberen Stufen der Lernzieltaxonomie nach Bloom [BE76] auslösen können. In der Literatur wurde bereits der Mangel an spezifischen Aufgabentypen und der Darstellung von Quelltext festgestellt, weswegen in einer Studie zwei neue Fragetypen entwickelt und evaluiert wurden [EH15a]. Trotzdem kann den bisherigen Fragetypen mindestens eine gesteigerte Partizipation der Studierenden bescheinigt werden und somit eine stärkere Auseinandersetzung mit den Lerninhalten. Eine besondere Form solcher Fragen ist das Konzept der *Peer Instruction*, welche kooperative Elemente wie Diskussionen und Begründungen von Antworten hinzufügt und in mehreren Studien nachweislich zu einem besseren Verständnis beiträgt.

Zusammenfassend können mehrere Aktivitäten und Merkmale identifiziert werden, die in einem aktivierenden Konzept berücksichtigt bzw. nicht verhindert werden sollten.

- *Learning by Doing*: Fast alle Ansätze verwenden in der ersten oder in späteren Phasen das Programmieren und das Entwickeln von algorithmischen Lösungen. Dies ist zugleich eines der Merkmale des Aktiven Lernes (vgl. Kapitel 4.1), dass eine Beteiligung der Studierenden durch Lese- und Schreibaktivitäten fordert. Alle Ansätze haben gemein, dass die Vorlesung keine reine theoretische Inhaltsvermittlung mehr ist, sondern oftmals ein gegenseitig gesteuerter Prozess. Für die Steuerung eignen sich CRSs oder Ergebnisse aus abgegebenen und (automatisch) bewerteten Programmieraufgaben.
- *Pair-Programming*: Das gemeinsame bearbeiten von Aufgaben verringert die Anzahl an syntaktischen Fehlern und ermöglicht eine schnellere Lösung von Problemen.

- *Cognitive Apprenticeship*: Das Konzept wurde in mehreren Veranstaltungen und Formen bereits erfolgreich eingesetzt und eignet sich für die Vermittlung von Strategien zur Problemlösung.
- *Worked Examples*: Auf Basis der *Cognitive Load Theory* unterstützen diese nachweislich das Verständnis von Konzepten am Anfang, wobei im Laufe eine steigende Komplexität der Aufgaben und Probleme notwendig ist. Lister et al. kommen konnten ebenfalls einen zuträglichen Einfluss von Übungen zur Förderung der Lese- und Verständniskompetenz (Programmablauf nachvollziehen) von Programmen auf die späteren Programmieraktivitäten nachweisen.
- *Problemverständnis und Relevanz*: Studierende sollen die Notwendigkeit für die behandelten Inhalte erkennen und warum sie sich damit beschäftigen müssen.

Viele der benannten Ansätze beleuchten jeweils nur einzelne Konzepte und nicht deren Verbindung und Kombination in einem übergeordneten didaktischen Gesamtkonzept, das gleichermaßen selbstgesteuertes wie angeleitetes Lernen im Sinne des *Cognitive Apprenticeship* ermöglicht. Zurückblickend auf die Problemstellung und der Forderung nach mehr studentischer Partizipation im Sinne eines eher seminaristischen Unterrichts, erscheint *Aktives Lernen* ein vielversprechender Ansatz zu sein. Genau diese Verbindung der bisher alleinstehenden Konzepte möchte ich im nachfolgenden Kapitel 5 erarbeiten, indem bereits in der Vorlesung angeleitete und selbstgesteuerte Aktivitäten zur Vermittlung von Konzepten der Programmierung verwendet werden.

# Kapitel 5

# Integration aktiver Programmierübungen in Lehrveranstaltungen

In diesem Kapitel soll aus den bisherigen Vorüberlegungen ein didaktisches Gesamtkonzept erarbeitet werden, welches die Gestaltung einer aktivierenden Lernumgebung und eine Methode zum Einführen neuer Konzepte der Programmierung auf Basis der lehr- und lerntheoretischen Diskussionen berücksichtigt. Abschließend werden Voraussetzungen an eine technische Plattform zur Umsetzung dieses Konzeptes formuliert.

## 5.1 Voraussetzungen & Zielsetzungen

### 5.1.1 Didaktische Synthese

Bevor ich das Konzept erläutere, möchte ich die Problemstellung und die bisherigen lehr- und lerntheoretischen Diskussionen zusammenfassen. Die Motivation dieser Arbeit resultierte aus den in der Problemstellung beschriebenen Beobachtungen: Neue Konzepte und deren Anwendung erfordern nicht nur Wissen über das Konzept, sondern gleichzeitig das Verständnis über die damit verbunden Zustandsänderungen bei der Ausführung des Programms. Ebenso das Wissen über die sprachspezifische Syntax des Konzepts, ein Werkzeug zum Codieren des Konzepts und die Einordnung in das informatische Problemlösen. In einer traditionellen Vorlesung mit verstärktem Anteil an Frontalunterricht, können neue theoretische Inhalte nur schwer seitens der Stu-

dierenden erkundet und erfasst werden. Ihre Tätigkeiten beschränken sich auf das Nachverfolgen des Lehrenden und Lesen von Beispielen, ohne selbst den kompletten Programmierprozess von der Problemstellung bis hin zur Anwendung zu durchlaufen. Verständnisprobleme können während der Vorlesung ohne aktivierende und selbststeuerende Prozesse nur schwer erkannt und adressiert werden. In Kombination mit einer großen *Stofffülle* bleibt für situative Lernumgebungen, in denen Lernende enaktiv mit neuen Inhalten konfrontiert werden, keine Zeit. Stattdessen wird die Anwendung in die begleitende Übung verlagert, in denen Studierende sich zur Bearbeitung der Aufgaben zunächst wieder in die theoretischen Konzepte einarbeiten müssen. Durch diese Trennung wird automatisch ein deduktiver Lernprozess festgelegt, ohne weitere Steuerungsmöglichkeiten seitens der Lehrenden. Für einen erfolgreichen Lernprozess ist das Einordnen neuer Inhalte in eine zusammenhängende Struktur wichtig, um vorgegebene Lernziele zu studentischen Lernzielen zu transformieren. Aus konstruktivistischer und didaktischer Sicht sollten Lernumgebungen so gestaltet sein, dass den Lernenden die Relevanz bzw. die Viabilität verdeutlicht wird und einen Rahmen zur Konstruktion von Wissen ermöglicht (vgl. Kapitel 3.4.1). Darunter fallen auch motivationale Aspekte, wie z. B. der fachliche Zusammenhang zum Studium. Zwar wurde bisher ausgehend von der Industrie 4.0 und informatorisch-didaktischen Überlegungen eine stärker allgemeinbildende Ausbildung beim Programmieren gefordert, aber steht das nicht orthogonal zu fachspezifischen (situativ und authentisch) Problemstellungen aus denen induktiv neue Konzepte und Lösungsstrategien abgeleitet werden können. Genau dies ist eine der zu Beginn aufgestellten Forderungen nach der situativen und aktiven Vermittlung von neuen Inhalte und Konzepten und einer teilweisen Steuerung des Unterrichts durch die Studierenden selbst (vgl. Kapitel 2.2).

Vor den lehr- und lerntheoretischen Überlegungen, musste zuerst der Begriff Programmieren und Programmierfähigkeit definiert werden, denn diese bilden die Grundlage für die Lernziele (vgl. Kapitel 3.1). Aus den Überlegungen und Betrachtungen vorhandener Modelle konnten mehrere Hauptbestandteile synthetisiert werden: (1) Theoretische Grundlagen über die Ausführbarkeit und Maschinen, (2) Konzepte der Programmierung als Lösungsstrategien für informatische Probleme (3) und informatische Problemlösens durch einen iterativen Algorithmenentwurf inklusive deren Verifikation. Zusammengefasst ist Programmieren nicht nur das Erlernen einer Programmiersprache, sondern eine bestimmte Art zu Denken (vgl. *Computational Thinking* in Kapitel 3.2.3) um Probleme programmatisch zu lösen. Beim Problemlösen müssen mentale und enaktive Prozesse stattfinden, damit ausgehend vom Ausgangszustand durch eine algorithmische Transformation der Zielzustand erreicht wird. Programmieren ist eine *Fähigkeit*, die es gezielt zu fördern gilt. Die beschriebenen Problemlösungsstrategien in Kapitel 3.2.2 und der postulierte Programmiervorgang (vgl. Kapitel 3.1) legt eine Problemzerlegung und iterative Lösung dieser Teilprobleme durch Analogien und Heuristiken nahe. Schlussendlich müssen in einer Lehrveranstaltung zum Thema Pro-

grammieren die theoretischen Grundlagen vermittelt und die informatische Problemlösungsfähigkeit in einem gewissen Rahmen ausgebildet werden.

Aus der Forderung nach einer aktiven Partizipation der Studierenden innerhalb der Vorlesungen und der forschungsmethodischen Vorgehensweise (Kapitel 2.3) wurden konstruktivistische Theorien aus lehr- und lerntheoretischer Sicht untersucht und folgende wichtige Elemente identifiziert:

- Lernende müssen aktiv im Lernprozess beteiligt sein und übernehmen teilweise Steuerungsprozesse,
- Lernen findet durch situative Erfahrungen statt und basiert auf individuellem Vorwissen und
- Lernen ist immer ein sozialer Prozess und ein interaktives Geschehen.

Nach konstruktivistischer Auffassung sollen Erfahrungen in komplexen und authentischen Problemsituationen gesammelt werden. Doch diese können zur Überforderung führen und den Lernerfolg stören, weswegen daneben der *Cognitive Apprenticeship* Ansatz und die *Cognitive Load Theory* zur Gestaltung von Lernumgebungen näher betrachtet wurden.

Im Rahmen der Erfassung der Charakteristika des Aktiven Lernens und aus den vorhandenen Ansätzen im Bereich der Programmierausbildung konnten mehrere zentrale Punkte für die didaktische Konzeptentwicklung identifiziert werden: Das Durchlaufen des iterativen Programmierprozesses beinhaltet die Phase der Codierung, die zugleich der Verifikation des gewählten Lösungsansatzes dient, weswegen die Verschränkung der Vorlesung und Übung in der Problemstellung gefordert wurde. Studierende müssen enaktive Erfahrungen sammeln, sodass metakognitive Prozesse zur Ausbildung der geforderten Heuristiken bzw. Analogien angestoßen werden. Dies entspricht dem *Learning by Doing*, welches einen nachweislichen Effekt auf die Durchfallquote und Prüfungsleistungen hat. Aus diesen Aktivitäten kann mittels CRSs oder ähnlichen Methoden bereits eine Einschätzung über den aktuellen Lernprozess vorgenommen werden, wodurch eine teilweise Selbststeuerung durch die Studierenden erfolgen kann. Bei der Bearbeitung sollen und können Studierende gerne in Paaren oder Gruppen arbeiten, da hier ebenfalls eine positive Auswirkung auf die Leistung festgestellt werden konnte. Eine Möglichkeit dazu ist das *Pair Programming*, bei denen zwei Studierende ein Problem gemeinsam lösen.

Neben konstruktivistischen Ansätzen, haben instruktionelle Methoden nach wie vor ihre Berechtigung. Besonders sogenannte *Worked Examples* zeigen deutliche Vorteile am Anfang der Vermittlung von komplexen Zusammenhängen (vgl. Punkt 3.4.3), erfordern allerdings anschließend eine steigende Komplexität der Aufgaben und einen Übergang der Handlungen vom Lehrenden zu den Lernenden [CB07]. An dieser

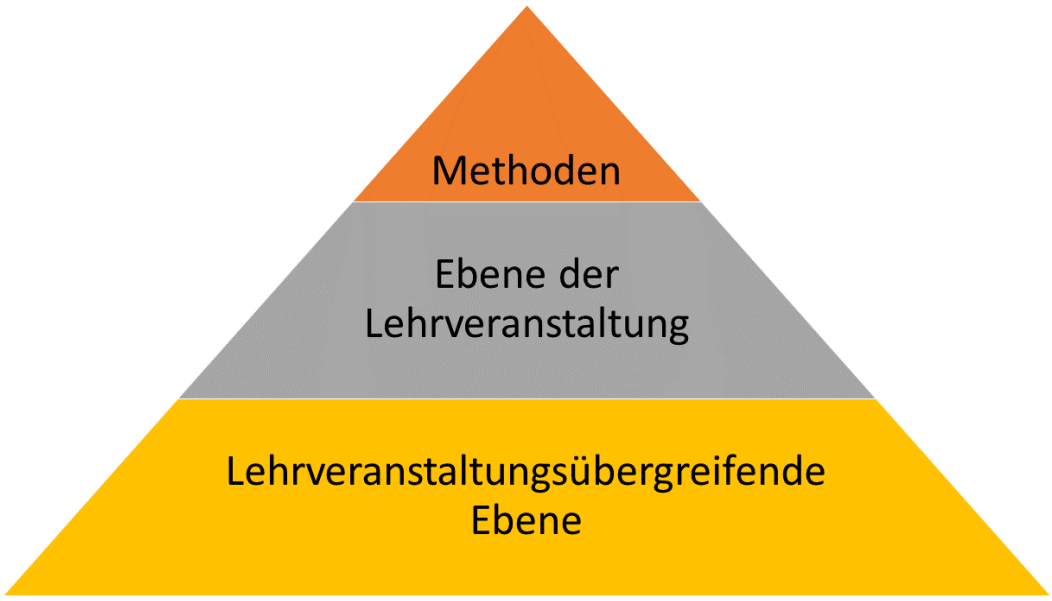



**Abbildung 16**: Ebenen des didaktischen Handlungsspielraums

Stelle tritt der Ansatz des *Cognitive Apprenticeship*, welcher diesen Übergang in mehreren Stufen organisiert. Bisherige Ansätze des CA wurden bisher in Tutorien und Übungsstunden evaluiert, jedoch noch nicht direkt in Vorlesungen. Genau dies ist das Ziel des nachfolgenden Konzeptes, welches interaktive Übungen zur Vermittlung von Programmierkonzepten in Vorlesungen ermöglichen soll. Metakognitive Prozesse erfordern das Diskutieren und Auseinandersetzen mit unterschiedlichen Problemlösungsstrategien, weswegen das Konzept zusätzlich auf die Kommunikation zwischen Studierenden und Lehrenden eingeht (vgl. Kapitel 4.1).

Für eine bessere Strukturierung dieses übergreifenden Konzeptes möchte ich die Voraus- und Zielsetzungen und dabei getroffenen didaktischen Entscheidungen auf drei unterschiedlichen Ebenen diskutieren (vgl. Abbildung 16). Die Ebenen sind dabei an das Modell von Wildt angelehnt [Joh03]:

1. Lehrveranstaltungsübergreifenden Ebene: Didaktische Entscheidungen für Module und Studiengänge

2. Lehrveranstaltungsebene: Entscheidungen für eine spezifische Veranstaltung

3. Methoden: Methodische Auswahl und Begründung bei der Gestaltung von Lernumgebungen

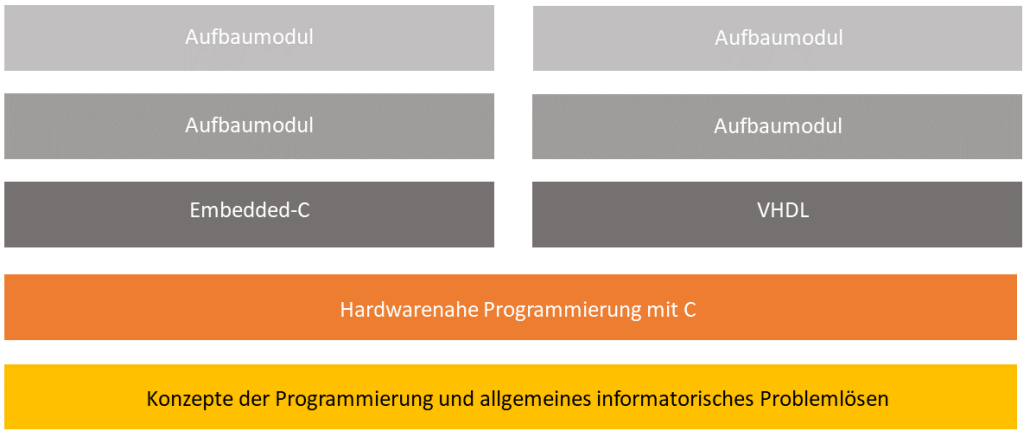



**Abbildung 17:** Einordnung der Lehrveranstaltung in ein Elektrotechnikstudium

### 5.1.2 Lehrveranstaltungsübergreifende Ebene

Ziel der Programmierausbildung für Studierende der Elektrotechnik ist in erster Linie die Ausbildung eines allgemeines Verständnis für die Konzepte der Programmierung, welche in konsekutiven Lehrveranstaltungen in Richtung Hardware vertieft werden. Das bedeutet aber nicht, dass das Programmieren vollkommen von dem Fachgebiet der Elektrotechnik losgelöst ist, sondern vielmehr als motivationaler Rahmen dient, in denen die Inhalte eingebettet sind. Die Ausrichtung eines Moduls bzw. einer Lehrveranstaltung sollte immer im Einklang mit den nachfolgenden Modulen erfolgen. Für eine Veranstaltung für Programmieranfänger wurde bewusst eine eher allgemeine informatische Programmierausbildung gewählt, die in anschließenden Fächern spezialisiert wird. Im Falle der Elektrotechnik thematisieren weiterführende Lehrveranstaltungen die hardwarenahe Programmierung in *C*, da diese Programmiersprache nach wie vor als Standard gilt. Die Abbildung 17 zeigt die Hierarchie der Module, welche gleichzeitig die Anforderungen bzw. Voraussetzungen der Lehrveranstaltungen definiert. Dieser Aufbau erfordert jedoch gleichzeitig eine Betonung des Algorithmenentwurfs und der Universalität der Programmierkonzepte. Ein Algorithmenentwurf sollte immer auf einer sprachunabhängigen Ebene erfolgen, auch wenn die Codierung eine spezifische Sprache erfordert. Studierenden sollte am Ende einer Lehrveranstaltung bewusst sein, dass diese Konzepte der Programmierung zur Lösung informatischer Problemen in fast allen Sprachen (im selben Paradigma) existieren und es hauptsächlich syntaktische Unterschiede gibt. Bei der Einführung der Programmiersprache *C* sollten die grundlegenden Konzepte bereits verinnerlicht sein, sodass Studierende sich mit Zugriffen auf Hardware und manueller Speicherallozierung auseinandersetzen können, ohne sich gleichzeitig mit dem Erlernen von Konzepten zu beschäftigen.

In der nächsten Stufe werden die Inhalte noch spezifischer und orientieren sich an speziellen Ausführungsumgebungen und deren Besonderheiten. Schlussendlich können darauf weitere Module, wie in der Abbildung 17 angedeutet, aufbauen.

### 5.1.3 Ebene der Lehrveranstaltung

Auf der darunterliegenden Ebene der Lehrveranstaltung können Veränderungen deutlich leichter umgesetzt werden, da bis auf die Abstimmung der Lernziele und Inhalte keine weiteren Einflüsse auf didaktische und methodische Entscheidungen mehr existieren. Des Weiteren müssen auf dieser Ebene noch weitere lehrveranstaltungsweite Entscheidungen getroffen werden.

Programmieren kann aus verschiedenen Sichtweisen gelehrt werden, die alle ihre Berechtigungen haben, weswegen eine Entscheidung getroffen werden muss. Ausgehend von den konsekutiven LVs wurde bewusst das imperative Paradigma ohne Objektorientierung gewählt, sodass Studierende sich vorerst kein neues Paradigma erlernen müssen. Stattdessen rückt in den folgenden LVs die Programmierung von Hardware und die damit verbundenen Besonderheiten in den Fokus. Der fachliche Kontext bleibt dabei nicht außen vor, sondern dient unter anderem als Grundlage für authentische Problemstellungen.

#### Motivationaler Rahmen

Bei der Konzeption der Lehrveranstaltung und Lernziele wurde ein motivationaler Rahmen, in welchem die einzelnen Inhalte eingeordnet werden, definiert. Die anfangs in Kapitel 1.1.1 erwähnte Industrie 4.0 ist eine von drei Säulen des motivationalen Rahmens. Software- und Hardwareentwicklung ist keine strikt getrennte Disziplin mehr, sondern oftmals ist ein Produkt die Symbiose aus beiden. Studierende müssen beide Seiten kennen, um adressatengerecht zu kommunizieren. Der Modulaufbau zeigt bereits die Relevanz der informatischen Inhalte innerhalb eines Elektrotechnikstudiums (siehe dazu ebenfalls Umfrage FBTEI in Kapitel 1.1.2). Die Anzahl vernetzter und digitaler Systeme steigt stetig an, ohne das dies auf den ersten Blick für Unbedarfte erkennbar ist. Bei der Wahl eines Studiums kann ein gewisses Interesse bereits vorgesetzt werden, jedoch der Überblick aktueller Entwicklungen nur bedingt, besonders in Branchen, die Studierende nicht unbedingt im Fokus haben. Ein Beispiel dafür ist die Überprüfung von Zugstrecken auf Schäden durch eine Verbindung von Sensoren und Algorithmen direkt während der Fahrt. Im Bereich der erneuerbaren Energien richten sich Windräder in Offshore-Parks durch teilweise selbstlernende Algorithmen gegenseitig aus, um einen maximalen Windertrag zu gewährleisten. An dieser Stelle verzich-

te ich auf weitere Beispiele. Für Lehrveranstaltungen, die für mehrere Studiengänge mit unterschiedlichen Schwerpunkten angeboten werden, muss hier eine wohlüberlegte Auswahl erfolgen.

Ebenso soll im Laufe der LV nicht auf die Anwendung von Algorithmen in *anfassbaren* und *interaktiven* Umgebungen verzichtet werden. Im Rahmen authentischer und situativer Problemstellungen eignen sich im Bereich der Elektrotechnik Plattformen wie z. B. *Arduino*, *RaspberryPI* oder *Lego Mindstorms*. Diese ermöglichen die Anwendung der Inhalte ohne zu tiefe digital-technische bzw. elektronische Vorkenntnisse vorauszusetzen. Die letzte Säule lässt sich als Programmieren als *Hilfsmittel zur Automatisierung von Prozessen* beschreiben. Hierbei soll Studierenden eine weitere Perspektive eröffnet werden, warum informatisches Problemlösen (Programmieren) später hilfreich sein kann. Konsequenz aus diesem Blickwinkel ist einerseits die Notwendigkeit zur Wahrnehmung des Problemlösens und des Algorithmenentwurfs als eigenständige Phasen, und andererseits das die Codierung mit einem *geeigneten* Werkzeug (Programmiersprache) erfolgen sollte. Für das Ein- oder Auslesen von Messdaten zur grafischen Auswertung, kann durchaus *C* verwendet werden, jedoch sind an dieser Stelle Sprachen, wie z. B. *Python*, deutlich besser geeignet. Zusammengefasst kann das informatische Problemlösen aus mehreren Perspektiven beleuchtet werden und ermöglicht die Gestaltung von situativen Lernumgebungen, die zielgruppengerecht neue Inhalte thematisieren. Studierende müssen, ausgehend von der konstruktivistischen Denkweise, die Notwendigkeit und Relevanz der zu behandelten Themen erkennen und verstehen wie diese in ein größeres Bezugssystem eingeordnet sind (vgl. Kapitel 4.4.3).

### Annahmen für die Lehrveranstaltung

Die Auswahl einer Programmiersprache und eines Paradigmas für Programmieranfänger wird in der Forschung häufig diskutiert. Eine Entscheidung muss auf lehrveranstaltungsübergreifender Ebene abgestimmt werden, um die Voraussetzungen für nachfolgende Module zu erfüllen. Im Rahmen der LV, in dem diese Arbeit entstanden ist, wurde die Sprache *Python* ausgewählt, da diese eine leichter verständliche Syntax hat und bei der Entwicklung weniger im Wege steht als im Vergleich zu *C* [SS13; Pea+07, S.207]. Ziel der Lehrveranstaltung ist das Verstehen und Anwenden von Konzepten der Programmierung zum Lösen informatischer Problemstellungen. Erst in nachfolgenden Kursen wird die Programmiersprache *C* eingeführt und verwendet und stellt eine Spezialisierung dar.

### Verschränkung von Vorlesung und Übung

Das in der Problemstellung genannte und geforderte Ziel ist die Verschränkung zwischen Übung und ehemals theoretischer Vorlesung im Sinne eines seminaristischen Unterrichts. Hubwieser versteht unter Übung „[…] das Gelernte durch Wiederholung zu festigen. Datenwissen wird dabei vom Kurzzeitgedächtnis in das Langzeitgedächtnis übertragen und dort in bestehende Strukturen eingebettet“ [Hub07]. Diese Festigung sollte somit zu einem Zeitpunkt stattfinden, an dem das Wissen noch im Kurzzeitgedächtnis vorhanden ist. Bei der Einführung neuer Konzepte und deren Anwendung sollen bereits während der Vorlesung apponierte Aufgaben (im Anschluss oder währenddessen) durchgeführt werden und nicht erst im Rahmen disponierter Übungen. Im Kontext des Programmierens und des dargestellten Programmiervorgangs inkludiert dies das Üben der Codierung als Schritt zur Verifikation der Lösung. Und Lösungen können anschließend im Sinne des Aktiven Lernens vorgestellt und diskutiert werden. Aus der Diskussion und den Ergebnissen wiederum lassen sich weitere Schritte für die Gestaltung des Unterrichts ableiten, die sofort umgesetzt werden können. Die Übungen ermöglichen den Lernenden eine Selbsteinschätzung und dem Lehrenden mittels geeigneter Kommunikationskanäle eine Einschätzung (zumindest grob) über den Lernfortschritt und welche Probleme dabei ggf. auftreten. Die Übungen müssen dabei an die Inhalte und jeweiligen Lernziele angepasst sein, wobei die Gestaltung anhand der identifizierten Merkmale des *Aktiven Lernens* erfolgen kann.

Gleichzeitig sind die Ergebnissicherung und Wiederholung in späteren Episoden, wie z. B. zuhause, ein elementarer Bestandteil bei der Umsetzung auf methodischer Ebene. Anwesenheitspflichten sind weitestgehend im Rahmen des Bologna-Prozess abgeschafft worden, um Studierenden zu ermöglichen, das Lernen nach ihren Bedürfnissen gestalten zu können. Umso wichtiger ist in Anbetracht dieser beiden Punkte die Gestaltung der Lernumgebungen bzw. Lernmaterialien. In der Vorlesung integrierte Aktivitäten, die *Aktives Lernen* fördern, sollten ebenfalls zuhause und in den disponierten Übungsstunden nachvollzogen werden können. Damit ist zugleich ein selbstgesteuertes Lernen möglich, wenn die Unterlagen die Schritte aus der Vorlesung detailliert beschreiben und die gesicherten Ergebnisse enthalten. Wichtig ist hierbei zu erwähnen, dass keine Versteifung auf reine Codieraktivitäten während der Vorlesung erfolgen soll, sondern die Methoden und die Art der Übungen sich nach den jeweiligen Themen richten.

### Seminaristischer Unterricht - Nur bedingt

Eine einheitliche Definition des *Seminaristischen Unterrichts* ist schwer zu finden, wenn auch diese Form oft in den Modulbeschreibungen auftaucht. Turner et al. beschreiben

diese Lehrform wie folgt:

> Seminaristischer Unterricht ist eine besondere Form der Vermittlung von Lehrinhalten; [...] Die Teilnehmerzahl ist auf 40 Studenten beschränkt. Dadurch wird ein unmittelbarer Kontakt zwischen Lehrenden und Lernenden ermöglicht, der es den Studenten erlaubt, Fragen zum Inhalt an die Lehrenden zu richten [...] [Tur11].

Die festgelegte Obergrenze auf 40 Studierende erscheint hierbei etwas willkürlich bzw. aus Klassenstärken an Schulen abgeleitet. Aus meiner Sicht sind die Fragen und Diskussionen der ausschlaggebende Punkt für das Modell. Denn auch hier muss die Form an die Rahmenbedingungen der Lehrveranstaltung angepasst werden, um zielgerichtet eingesetzt werden zu können. Dennoch sollten Fragen, Einwände und Rückmeldungen seitens der Studierenden, welche nicht zur aktuellen Methode passen, nicht ignoriert werden. Vielmehr müssen diese situativ beantwortet oder ggf. auf ein bilaterales Gespräch oder auf einen anderen Zeitpunkt zurückgestellt werden.

Sobald die Anzahl der Studierenden steigt, ist die direkte Kommunikation nur bedingt möglich, weswegen die Skalierbarkeit der Kommunikationskanäle und Aktivitäten innerhalb einer Vorlesung mit mehr als 40 Studierenden berücksichtigt werden sollten. Lösungen zu Programmieraufgaben werden in der Regel in textueller (sprachabhängig) Form niedergeschrieben und gelesen. Der Austausch von solchen Lösungen, um unterschiedliche Ansätze zu diskutieren, gestaltet sich dabei aufwendig. Werden die Aufgaben auf Papier gelöst, so können diese zwar mit speziellen Geräten direkt an die Wand projiziert werden. Eine Ausführung ist in diesem Falle erst nach *Abtippen* der Lösung möglich, um ggf. die Auswirkungen von Veränderungen zu beobachten und zu diskutieren.

Dieses Problem ist ebenfalls bei der Verwendung von Quelltextbeispielen relevant, denn bei Fragen können diese entweder theoretisch beantwortet werden oder alternativ muss in eine geeignete Entwicklungsumgebung gewechselt werden. Professionelle Entwicklungsplattformen zeichnen sich in der Regel nicht dadurch aus, dass diese eine optimierte Darstellung für Leinwände, mit Schaltflächen und Texten in ausreichender Größe, haben. In der gleichen Weise ist die Darstellung von Quelltexten in PowerPoint direkt bereits umständlich und größere Beispiele können entweder nur klein oder über mehrere Folien dargestellt werden[12]. Und schlussendlich müssen die Lösungen und Schritte im Sinne der Ergebnissicherung erfasst und zur Verfügung gestellt werden. Nachfolgend soll zuletzt die methodische Ebene erläutert und anschließend aus den Erkenntnissen das neue Konzept in Kapitel 5.2 abgeleitet werden.

[12] An dieser Stelle wird auf eine Diskussion geeigneter Beispiellängen verzichtet.

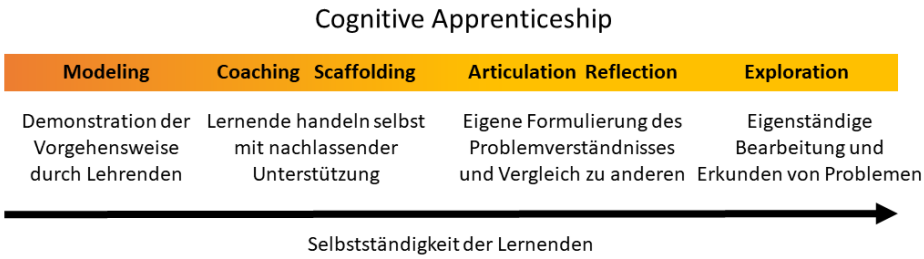



**Abbildung 18**: Selbstständigkeit der Lernenden im Cognitive Apprenticeship

### 5.1.4 Methodische Ebene

Die vorangegangene Diskussion der konstruktivistischen Ideen und der die geforderte Balance zwischen Instruktion und Konstruktion hat zum *Cognitive Apprenticeship* Ansatz geführt, der vorerst in allgemeine Schritte unterteilt ist. Die Abbildung 18 zeigt die einzelnen Phasen und zeigt den Übergang von Instruktion zur Konstruktion, durch die verstärkten selbstständigen Handlungen seitens der Lernenden, auf. Ausgehend vom Vorteil der *Worked Examples* bei der anfänglichen Auseinandersetzung, sollen neue Konzepte der Programmierung nach diesem Schema eingeführt werden. Auf einer Seite soll während der Vorlesung ebenfalls Aktives Lernen durch gezielte Übungen (Phasen *Coaching* und *Scaffolding*) integriert und Studierenden die Möglichkeit geben werden, sich enaktiv mit diesen auseinander zu setzen. Kapitel Kapitel 5.2 beschreibt die im Rahmen dieser Arbeit entwickelte Adaptierung und Umsetzung, sodass dies direkt in Vorlesungen angewendet werden kann. Das Konzept soll auf der anderen Seite weiteren Methoden des *Aktiven Lernens* nicht im Weg stehen, die sich für bestimmte Ziele und Inhalte besser eignen.

## 5.2 Interaktive Aufgaben für Programmierkonzepte

Für die Beantwortung der Forschungsfrage Punkt 3, welche einen Ansatz zur Integration von Schreib-,Lese- und Diskussionsaktivitäten in einer Vorlesung fordert, wurde ausgehend von den didaktischen Überlegungen ein Konzept entwickelt. Das Konzept verbindet dabei vorhandene Ansätze und erweitert diese, sodass ein Einsatz im Unterricht möglich ist. Im Gegensatz zu reinen Aufgaben fokussiert das Konzept ganz bewusst den Bereich der Programmierkonzepte, um diese problemorientiert (durch Perturbationen) und in einen übergeordneten Rahmen eingebettet zu vermitteln.

Das entwickelte Konzept, welches sich an *Cognitive Apprenticeship* Ansatz anlehnt, kann als *interaktive Aufgabenspirale für Programmierkonzepte* in Vorlesungen bezeich-

net werden. An dieser Stelle muss bewusst darauf hingewiesen werden, dass das im Rahmen dieser Arbeit entwickelte Konzept sich an das *Cognitive Apprenticeship* anlehnt, jedoch kein klassische Meister-Lehrlings-Verhältnis ermöglicht. Bei der Adaption des Konzepts wird bewusst von einem Lehrenden (der Meister) und einer Vielzahl von Lernenden (Lehrlingen) ausgegangen, weswegen die direkte Kommunikation nicht wie im ursprünglichen Konzept ermöglicht werden kann.

Studierenden soll dabei die Verwendung und Anwendung der einzelnen Konzepte vermittelt werden und Möglichkeiten zur Ausbildung von Heuristiken zur Lösung ähnlicher Probleme gegeben werden. Ausgehend von der Diskussion über das Programmieren in Kapitel 3.1 und den Problemlösungsstrategien sollen Studierende Konzepte als Baustein zur Lösung von (Teil-)Problemen präsentiert werden. Dabei ist die universale Verwendung der Bausteine bzw. Konzepte zu betonen und der Transfer auf ähnliche Problemstellungen mit fachlichem Bezug zu unterstützen. Dabei erfolgt die Einführung neuer Konzepte immer mit Einordnung in das Gesamtbild, welches der Algorithmenentwurf ist. Abbildung 19 zeigt den prinzipiellen Ablauf des Konzepts in vier Schritten, welche sich an den Phasen des *Cognitive Apprenticeship* orientieren und anschließend im Detail erklärt werden. Der erste Unterschied ist die Konfrontation der Studierenden mit einem bisher nicht lösbaren Problem, welches bei den Studierenden reflexive und problemlösende Denkprozesse anregen und damit zum Erlernen eines weiteren Konzepts motivieren soll. Im nächsten Schritt wird den Studierenden das neue Konzept, welches beim Lösen des Anfangsproblems hilft, vorgestellt. Nach der Vorstellung wird je nach Konzept eine interaktive Aufgabenspirale durchlaufen, bei der im Verlauf die Komplexität der Aufgaben steigt und die Unterstützung des Lehrenden abnimmt. Die Steuerung und Anzahl der Aufgaben erfolgt durch eine starke Kommunikation zwischen Lernenden und Lehrenden, die später näher erläutert wird. Schlussendlich werden die neuen Konzepte in unterschiedlichen Aufgaben, in den zur Vorlesung begleitenden Übungen, wiederholt und mit verstärkter Selbstständigkeit geübt. Aus dieser letzten Phase werden Erkenntnisse über den Lernerfolg der Studierenden gesammelt und als Vorbereitung auf die nächste Vorlesung verwendet. In den nächsten Kapiteln werden die einzelnen Phasen des in Abbildung 19 gezeigten Schritten ausführlich beschrieben.

### 5.2.1 Problemorientierung und Problemkonfrontation

Ausgehend von den Vorteilen des problem- bzw. fallorientieren Lernens, wird die Reihenfolge bei der Einführung umgedreht. Neue Konzepte werden ausgehend vom Problem eingeführt, welches zu einem neuen (abstrakten) Programmierkonzept führt. Diese Änderung spiegelt sich ebenfalls in den Inhalten wider, denn im Vordergrund stehen die Konzepte und nicht sprachspezifische Konstrukte, wie es in Lehrbüchern

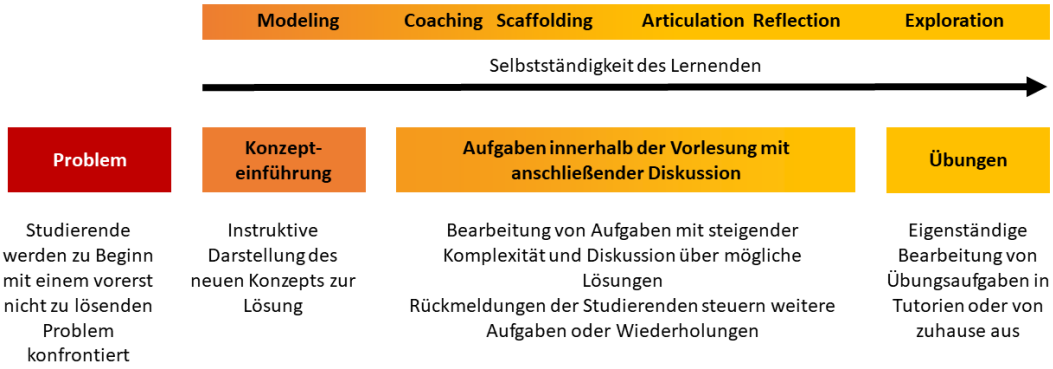



**Abbildung 19**: Konzept der interaktiven Aufgabenspirale

teilweise praktiziert wird [Dow15; The16]. Ziel ist das informatische Problemlösen, wozu in erster Linie ein Problem analysiert und verstanden werden muss. In Kapitel 3.2.1 wurde das Lösen eines Problems als Überwindung einer Hürde durch geeignete, aber noch unbekannte Mittel, beschrieben. Studierende sollen in diesem ersten Schritt mit einem bisher nicht lösbaren Problem konfrontiert werden und über mögliche Lösungen nachdenken. Diese Probleme müssen jedoch nicht immer gezwungenermaßen die Hürde darstellen, sondern können ebenso die Repräsentation des Ausgangszustandes thematisieren. Ein Beispiel dazu ist die Einführung von Variablen und Datentypen als Lösung zur Darstellung von Eingangswerten, die später verarbeitet werden. Studierenden sollen induktiv die Unterschiede zwischen einzelnen Datentypen und der Verwendung von Variablen aufgezeigt werden, um diese in ein allgemeines Konzept zu überführen. Dabei gilt es richtige und motivierende Problemstellungen für die Einführung zu gestalten, die das Problembewusstsein seitens der Studierenden fördern. In Kapitel 5.2.6 werden alle Schritte anhand eines Beispiels durchlaufen und aufgezeigt.

### 5.2.2 Konzepteinführung

Nach der Auseinandersetzung mit dem noch nicht lösbaren Problem, wird das neue Konzept instruktiv vom Lehrenden eingeführt. Bei der Einführung soll das Konzept im Zusammenhang mit bereits vorhandenen Themen vorgestellt und in ein größeres Bild eingeordnet werden. Konzepte sollen dabei von mehreren Seiten betrachtet werden. Darunter fällt der

1. Aufbau und die Zusammenhänge zu anderen Konzepten,
2. der Einfluss auf den Programmablauf,

3. und Zustandsänderungen im Programm, die durch das Konzept ausgelöst werden.

Die Betrachtung des Konzepts aus mehreren Perspektiven und das Aufgreifen vorheriger Konzepte ist eine Form des *Spiralprinzips* von Bruner [Bru72]. Danach sollen Lerngegenstände immer im Kontext des aktuellen Wissensstands vermittelt und später wieder aufgegriffen werden. Programmieren eignet sich dabei besonders, da die unterschiedlichen Konzepte für eine Vielzahl an darauf aufbauenden Algorithmen und im Zusammenhang mit anderen Konzepten erneut erläutert werden können.

Jeder dieser Punkte kann durch geeignete Darstellungen und Visualisierungen den Studierenden näher gebracht werden, ohne direkt auf Quelltexte zurückgreifen zu müssen. Konzepte sollen als abstrakte Lösungsbausteine wahrgenommen werden, die im Rahmen der Lehrveranstaltung in einer konkreten Sprache zum Lösen von Problemen angewendet und codiert werden. Für die Umsetzung sind die spezifischen Syntaxregeln und Besonderheiten zu erklären und aufzuzeigen. Im Sinne der *Worked Examples* sollen mehrere ausgearbeitete Beispiele vom Lehrenden vorgestellt werden. Die Vorstellung erfolgt während der Vorlesung durch *Live Coding*, bei dem der Lehrende beim Lösen eines Problems seine Gedanken laut artikuliert und das Vorgehen bei der Erarbeitung einer Lösung zeigt. Auf Fragen seitens der Studierenden soll und kann in dieser Phase direkt eingegangen oder auf einen fest definierten Zeitpunkt zurückgestellt werden Damit wird dieses Konzept ebenfalls dem seminaristischen Unterricht, zumindest bedingt, gerecht und fördert die sozialen Interaktionen und unterstreicht Adjektiv *interaktiv* im Namen des Konzeptes.

### 5.2.3 Aufgaben innerhalb der Vorlesung

Das vorgestellte Konzept wird umgehend im Anschluss durch Aufgaben, welche direkt in der Vorlesung von den Studierenden bearbeitet werden, aktiv angewendet und eingeübt. Dieser Schritt kann in mehrere Iterationen unterteilt sein, wobei die Anzahl durch das jeweilige Thema und durch die Studierenden selbst gesteuert werden sollen. In Bezug auf das *Cognitive Apprenticeship* werden Aktivitäten ähnlich zu den Phasen des *Coaching*, *Scaffolding*, *Articulation* und *Reflection* abgedeckt. Die Schwierigkeit der Aufgaben soll in den Iterationen steigen und schließlich zu einer oder mehreren Transferaufgaben führen. Gleichzeitig kann in diesen Iterationen kooperatives bzw. kollaboratives Lernen, z. B. durch *Pair Programming*, umgesetzt werden und die Aufgabenbearbeitung in Gruppen erfolgen. Der Austausch zwischen Studierenden oder das gemeinsame Bearbeiten und gegenseitige Erklären soll dabei nicht unterbunden werden, sondern verstärkt werden. Die Korrektur von *falschen* Konstruktionen erfolgt durch eine gemeinsame Besprechung von studentischen Lösungen, die gleichzeitig als Artikulation und Reflexion dient. Im Gegensatz zu kleineren Klassengrößen, können

dabei nicht alle Lösungen berücksichtigt werden, aber es kann zumindest eine Auswahl im Plenum vorgestellt und diskutiert und verglichen werden. Aufgaben sollen innerhalb einer passenden Umgebung bearbeitet und gelöst werden, wodurch eine gewisse Notwendigkeit für die Verwendung von Computern entsteht. Die Bearbeitung sollte mittels Laptops, Tablet-PCs mit Tastatur oder ähnlichen Geräten funktionieren, aber dabei Studierende ohne solche nicht ausschließen, was zudem wiederum den Punkt mit der adäquaten Lehr- und Lernumgebung aufgreift.

### Angeleitete Aufgabe

Bei der Vorstellung des Konzepts sollte dies bereits vom Lehrenden anhand eines Beispiels vom Problem bis zur Codierung gezeigt werden. Im Sinne des *Cognitive Apprenticeship* ist bei der ersten Aufgabe eine stärkere Unterstützung durch den Lehrenden notwendig, die in normalen Vorlesungsräumen und mit nur einem Dozierenden nur bedingt möglich ist. Dementsprechend muss die Aufgabe so gestaltet sein, dass diese sich durch das Übertragen der bisher gezeigten Beispiele lösen lässt. Zusätzlich können Hilfen nach bestimmter Zeit oder in Abhängigkeit von den Rückmeldungen der Studierenden gegeben werden. Eine andere Möglichkeit ist die Verwendung von Problemen, bei denen ein Teil der Lösung oder des Lösungsplans bereits vorgegeben ist. Morrison et al. haben dazu den Einfluss von sogenannten *Subgoal Labels* (Benannte Zwischenziele) auf den Lernerfolg im Kontext der *Worked Examples* untersucht [MMG15]. Abbildung 20 zeigt drei Beispiele in denen das erste ohne Zwischenzielbeschriftung, das zweite mit Beschriftungen und das dritte mit Platzhaltern versehen wurde. In der Studie wurde nachgewiesen, dass das Lernen anhand der angegebenen Zwischenziele im Vergleich zu den anderen Varianten förderlicher ist. Um Studierenden eine Erleichterung bzw. Unterstützung zu ermöglichen, können diese Zwischenziele bereits in der Aufgabe oder in unfertigen Lösungen mitgegeben werden. Studierende sollen sich in der ersten Phase mit den Zustandsänderungen und dem Programmablauf auseinandersetzen und erst in darauffolgenden Aufgaben mit dem Transfer beschäftigen.

Die Bearbeitung der Aufgaben kann, wie im vorherigen Kapitel angemerkt, in Paaren oder Gruppen erfolgen, sodass die positiven Effekte des kooperativen Lernens eine Möglichkeit zum Wirken erhalten. Anschließend soll die Aufgabe im Plenum anhand einer oder mehrerer studentischer Lösungen diskutiert werden. Damit werden mehrere Ziele verfolgt: Erstens soll eine zu starke Fixierung auf perfekte Musterlösungen vermieden werden und ein gewisser Lösungspluralismus etabliert werden, ohne stark auf den Programmierstil und die Effizienz einzugehen. Effiziente Algorithmen sind zwar erstrebenswert, aber nicht (unser) Ziel in Einführungsveranstaltungen (vgl. Diskussion der McCracken-Studie in Kapitel 3.3.2). Studierende sollen zu eigenen Lö-

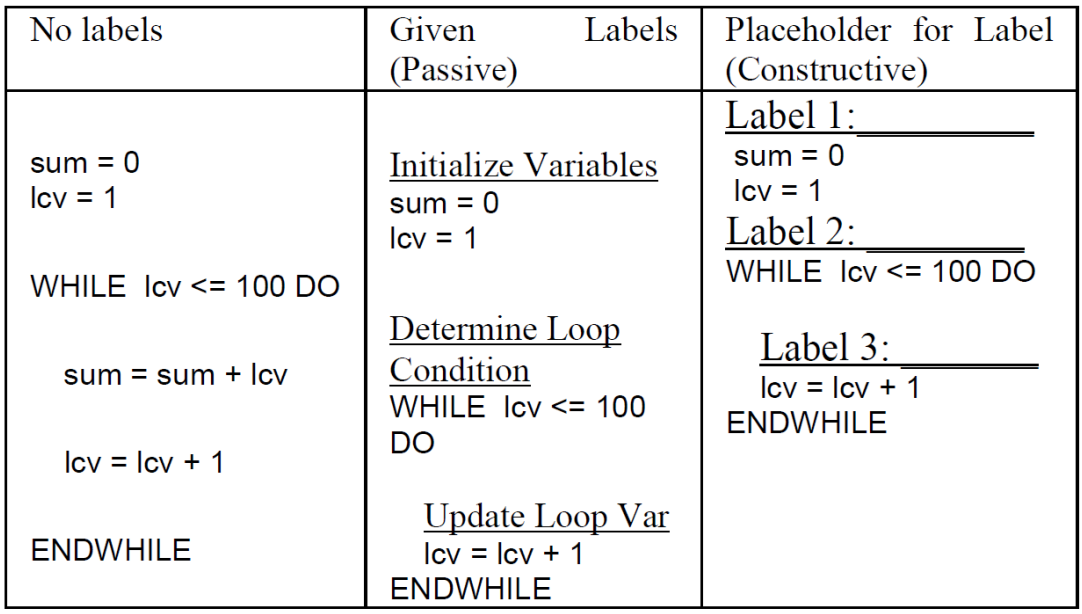



**Abbildung 20**: Subgoal Labels - Beispiele nach Morrison et al.

sungen kommen und nicht durch falsche Zielvorstellungen demotiviert werden, die nur sehr gute oder erfahrende Studierende erreichen können. Zweitens werden diese dadurch nicht gebremst und können ebenso bei der Besprechung hervorgehoben werden, um zumindest teilweise Probleme bei heterogenen Kohorten zu vermeiden [13]. Die Auswahl studentischer Lösungen soll dabei auf Freiwilligkeit beruhen und möglichst in digitaler Form vorliegen, sodass diese leicht mit dem Dozierenden geteilt und anschließend diskutiert werden kann. Studierende ohne entsprechender Ausrüstung sollten zur Lösung auf Papier ermutigt werden, sodass diese ebenfalls teilnehmen können. Doch zeigt sich ein Problem bei der Durchführung der Aufgaben auf Papier, da der letzte Schritt zur Verifikation (vgl. Kapitel 3.1) der Lösung nur bedingt durch die Studierenden selbst durchgeführt werden kann. Hinsichtlich der Lehre werden einerseits das Verständnis der Algorithmen und das informatische Problemlösen als Ziel verfolgt, jedoch sollten Studierende ebenfalls die konkrete Umsetzung mittels geeigneter Werkzeuge beherrschen. Entsprechend ist der Umfang der Aufgaben bzw. des zu produzierenden Quelltextes zu reduzieren, um diese Studierenden nicht zu benachteiligen. Ziel der Diskussion ist die Auseinandersetzung mit anderen Lösungsansätzen, welche den Vergleich mit der eigenen beinhaltet. Die Artikulation der eigenen Lösung und Gedanken ist dabei nur bedingt möglich, da nur eine begrenzte Auswahl

[13] Eine Diskussion über Probleme in starken heterogenen Studierendengruppe soll an dieser Stelle nicht weiter berücksichtigt werden, jedoch sind in späteren Kapiteln Anmerkungen zu möglichen Lösungen.

an Lösungen diskutiert werden kann, dennoch sollte explizit zu Anmerkungen und Nachfragen aufgefordert werden bzw. die Studierenden sollten ermutigt werden ihre Schwierigkeiten zu artikulieren.

Ausgehend von den studentischen Lösungen sowie Anmerkungen seitens des Lehrenden, soll im Anschluss auf Fragen und weitere Ansätze angemessen eingegangen werden. Änderungen und Auswirkungen sollen diskutiert und ausprobiert werden, sodass der Unterricht bereits eine Art explorative Lernumgebung darstellt, die zumindest teilweise den Interessen der Studierenden gerecht werden kann. Diese Rückmeldungen seitens der Studierenden müssen anschließend zur didaktischen Anpassung des Unterrichts verwendet werden, da ggf. Themen wiederholt oder bestimmte aufgetretene Fehler und Missverständnisse thematisiert werden müssen. Bei Bedarf kann eine weitere Übung durchgeführt werden, die entweder vorbereitet ist oder spontan integriert werden sollte, um das Konzept und die Probleme erneut zu adressieren. Im Gegensatz dazu kann bei keinen auftretenden Problemen sofort mit dem nächsten Schritt fortgefahren werden. Die im Laufe des Unterrichts entstandenen Lösungen sollen dabei als Ergebnis gesichert und zur Verfügung gestellt werden, sodass diese auch außerhalb der Vorlesung wiederholt werden können. Diese Ergebnissicherung muss entsprechend schnell durchführbar sein, um einen negativen Einfluss auf den Ablauf der Vorlesung zu vermeiden.

#### Transferaufgaben

Der nächste Schritt beschäftigt sich stärker mit der Ausbildung von Heuristiken. Dabei kann durch weitere instruktive Inhalte an ein weiteres Problem herangeführt werden, welches ebenfalls mit dem neu eingeführten Konzept lösbar ist, sich jedoch von den vorherigen Beispielen unterscheidet. Eine Unterscheidung kann durch einen anderen thematischen Rahmen, in den das Problem eingebettet ist, oder durch die Problemstellung selbst erfolgen. Hierbei sind vom Lehrenden die notwendige Unterstützung in Form von Teillösungen oder bei der Problemzerlegung in Abhängigkeit vom Kenntnisstand und den vorherigen Aufgaben einzuschätzen und anzubieten. Anschließend werden die Lösungen äquivalent zum vorherigen Schritt (vgl. Kapitel 5.2.3) verglichen und gemeinsam diskutiert, um daraus Entscheidungen für den weiteren Unterrichtsverlauf zu treffen. Ziel dieses Schrittes ist die Aktivierung metakognitiver Prozesse, um das Konzept als Lösungsstrategie für eine Vielzahl unterschiedlicher Probleme wahrnehmen zu lassen.

### 5.2.4 Coaching und Feedback in den Übungen

Der letzte Schritt im Modell findet außerhalb der Vorlesung statt und ist in den Übungen angesiedelt. Übungsaufgaben orientieren sich dabei an den in der Vorlesung thematisierten Konzepten und sollten sich ebenfalls in der Komplexität und Schwierigkeit steigern. Zu Beginn sollen Studierende das Konzept an einfachen Aufgaben durch analogischen Denken lösen, um in der Anwendung und Verwendung sicherer zu werden. Mit steigender Schwierigkeit müssen weitere Problemlösungsstrategien angewendet werden, um das Problem zu lösen. Dabei ist die Betreuung der Übung wichtig, um ebenfalls auf Fragen und Verständnisprobleme zu reagieren und individuelles Feedback zu geben.

Ausgehend von den Rückmeldungen und der Bearbeitung der Übungen durch die Studierenden, kann die nächste Vorlesung darauf ausgerichtet und angepasst werden. Lösungen der vorherigen Übungen können besprochen und aus den dabei beobachteten Problemen aktivierende Fragen (z. B. in Form eines Quizzes) generiert werden (vgl. CRSs in Kapitel 4.4.2). Durch diese Rückkopplung in die Vorlesung können Studierende bei ihrem aktuellen Wissensstand abgeholt werden und die Methoden darauf angepasst werden. Somit ist auch eine teilweise Selbststeuerung seitens der Studierenden möglich, wenn diese auch nicht jeden individuellen Studierenden repräsentiert.

### 5.2.5 Anmerkungen

Aus didaktischer und konstruktiver Sicht ist eine Darstellung der Ziele und Relevanz der Inhalte am Anfang der Vorlesung, sowie eine Zusammenfassung am Ende wichtig, um Studierenden bei der Strukturierung und Vernetzung der Inhalte mit bereits Vorhanden zu unterstützen. Die Durchführung solcher Übungen erfordert einerseits die Reduktion von Inhalten (vgl. Kapitel 4.1) und andererseits eine effiziente Umsetzung, sodass keine Zeit für Organisation verloren geht. Studierenden muss jedoch ausreichend Zeit zur Bearbeitung der Aufgaben eingeräumt werden, die bei Bedarf verlängert oder verkürzt werden kann. In Hinblick auf die Ziele der gesamten Lehrveranstaltung können diese zeitlichen Anpassungen nur in einem gewissen Rahmen durchgeführt werden, um alle essenziellen Ziele zu erreichen. Hierbei sollte bei der Planung der Lehr- und Lerneinheiten mit einem Puffer gearbeitet werden, welcher diese Unterschiede ausgleichen kann. Sobald auf Verständnisprobleme und Rückmeldungen nicht angemessen eingegangen werden kann, fällt der Unterricht wieder in eine eher instruktive Form zurück und relativiert die möglichen positiven Auswirkungen des Aktiven Lernens. Freie Pufferzeiten können zur Wiederholung, Klausurvorbeitung am Ende oder durch Impulse der Studierenden gefüllt werden. Letzteres steht besonders im Einklang mit den konstruktivistischen Forderungen nach einer

Selbststeuerung und der damit verbundenen Auswahl und Zielsetzung bei Inhalten.

Zusammengefasst ist die Vorlesung eine Interaktion zwischen Lehrenden und Lernenden und vom Wechsel von Instruktion und Konstruktion geprägt. Die Planung der nächsten Unterrichtseinheit ist mit einem höheren Aufwand verbunden, da diese abhängig von den studentischen Fortschritten und Rückmeldungen ist. Eine effiziente Gestaltung und Umsetzung der einzelnen Phasen ist de rigueur, dennoch benötigt die Gestaltung situativer Lernumgebungen Zeit. Trotzdem müssen den Diskussionen (*Articulation* und *Reflection*) ausreichend Zeit eingeräumt und die Ergebnissicherung durchgeführt werden.

### 5.2.6 Beispiel

Ein sehr gutes Beispiel, welches das Konzept in vielen Facetten abbildet ist die Iteration mit bekannter Anzahl an Durchläufen als Sonderfall der Iteration. Diese Unterscheidung zwischen finiter und infiniter Iteration, ist teils der Umsetzung mittels unterschiedlicher Sprachmittel geschuldet. Für die Beispiele und die schlussendliche Umsetzung werden nachfolgend Auszüge der Aufgaben in *Python* verwendet und auf *interaktive* und *authentische* Probleme etwas auf die in Kapitel 6 umgesetzte Plattform vorgriffen.

#### Problem

Ausgangspunkt für die Schleife ist das Zeichnen eines Polygons mit $n$ Seiten mittels *Turtle Graphics*, welches eine Bibliothek zum einfachen zeichnen von Linien und Formen ist. Der Name ist eigentlich eine simple Metapher für die Funktionsweise der Bibliothek. Ausgangspunkt ist eine Schildkröte, die Befehle wie z. B. *vorwärts* oder *drehe dich nach links* versteht und dabei eine Linie hinter sich herzieht. Quellcode 1 zeigt die Verwendung der Bibliothek, welche nach dem Ausführen die Ausgabe in Abbildung 21 erzeugt. Der Vorteil von *Turtle* ist eben diese grafische Ausgabe, da Studierende ihre entworfenen Lösungen visuell nachverfolgen und überprüfen können. Zum Beginn

```
1 import turtle # import des turtle-Moduls
2 wn = turtle.Screen() # Grafische Ausgabe aktivieren
3
4 alex = turtle.Turtle() # Zeichenstift erzeugen
5 alex.shape("turtle") # Stiftform auf Schildkröte setzen
6 alex.forward(150) # wir bewegen uns 150 Pixel vorwärts
7 alex.left(90)   # wir biegen um 90° links ab
8 alex.forward(75)
```

**Quellcode 1**: Beispiel für die Verwendung von Turtle

**Abbildung 21:** Grafische Ausgabe des Turtle Beispiels

der Unterrichtseinheit wird zuerst *Turtle* erklärt und einige Beispiele zur Verwendung gezeigt, welche unter anderem das Zeichnen eines sechsseitigen Polygons beinhaltet. Das Beispiel ist in Quellcode 2 abgebildet und zeigt einen Ausschnitt aus der Lösung, die sechsmal die zwei Anweisungen zum Zeichnen einer Linie und dem Ausrichten des Stiftes beinhaltet. Die Studierenden werden anschließend aufgefordert das Beispiel so zu ändern, dass ein 20-seitiges Polygon gezeichnet wird. Studierende können

```
1  import turtle
2
3  polygon = turtle.Turtle()
4
5  num_sides = 6
6  side_length = 70
7  angle = 360.0 / num_sides
8
9  # 1 Seite
10 polygon.forward(side_length)
11 polygon.right(angle)
12
13 ...
```

**Quellcode 2:** Darstellung eines Polygons mittels Turtle

dabei entweder die Lösung durch Fleiß produzieren, indem sie die beiden Zeilen noch 14-mal untereinander kopieren oder über eine geschicktere Möglichkeit nachdenken.

### Konzepteinführung

Die Schwierigkeiten bzw. das *Kopieren* von immer wieder den gleichen Anweisungen ist der Ausgangspunkt für die Konzepteinführung. Bei der Einführung werden

folgende Aspekte beleuchtet:

- Rückführung auf den Algorithmenentwurf, indem ein neues Element in das Repertoire der logischen Bausteine hinzugefügt wird.
- Aufbau und Funktionweise der Schleife mittels Beispielen, Flussdiagrammen und Visualisierungen.
- *Worked Examples*, welche die Verwendung von Schleifen zur Umsetzung von Summen und Produkten in mathematischen Formel zeigt.
- Verwendung der `range` Funktion für das wählen von Start- und Endwerten[14]

Schlussendlich wird die Lösung des ersten Problems (das 20-seitige Polygon) im Form des *Live Coding* durchgeführt und erste Fragen werden beantwortet werden.

### Angeleitetes Üben

Im nächsten Schritt wechselt die eher von Instruktion geprägte Einführung der Konzepte zum Aktiven Lernen über. Die Studierenden sollen in einer Aufgabe unter Verwendung der `for` Schleife ein Haus, bestehend aus einer viereckigen Front und einem gleichschenkligem Dreieck für das Dach zeichnen. Für die Lösung sind bereits die *Subgoal Labels* und Teile der Schleife vorgegeben, sodass die Studierenden sich primär mit den Anzahlen der Iterationen auseinandersetzen können. Weiterhin müssen die Studierenden beim Zeichnen des Dreiecks über die Winkel nachdenken, da in diesem Fall nicht die Innen-, sondern die Außenwinkel zum Zeichnen verwendet werden müssen. Die Lösungen werden anschließend im Plenum diskutiert und aufkommende Fragen angemessen beantwortet. Weitere Aufgaben können in Abhängigkeit der Rückmeldungen in den Unterricht integriert werden. Alternativ ist eine Erweiterung der vorhandenen Aufgabe möglich, um das Üben zu wiederholen.

### Transferaufgaben

Vor der Durchführung einer Transferaufgabe wird die Verwendung der Schleife zur Lösung doppelter Summenzeichen gezeigt und wie eine Punktematrix durch verschachtelte Schleifen erfolgen kann. Gleichzeitig ist es ein Ziel der Lehrveranstaltung situative und authentische Problemstellungen zu präsentieren, weswegen im nächsten Schritt die Anwendung der Schleifen unter Verwendung von Hardware erfolgt. Dies ist in Vorlesungen mit über 100 Studierenden nur bedingt möglich. Die folgenden

[14] Schrittweiten werden in einem späteren Kontext eingeführt und dienen so zur Wiederholung des Konzepts aus einer anderen Perspektive.

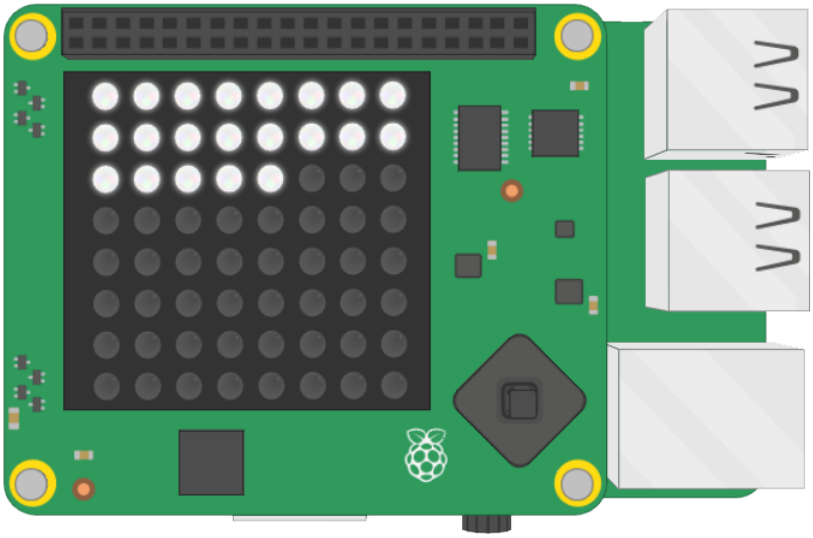


**Abbildung 22:** Web-basierte Emulation des SenseHat-Moduls für den RaspberryPI

Aufgaben verwenden deshalb einen web-basierten Emulator für das *SenseHat* Modul der RaspberryPI Foundation, welches die Programmierung einer LED-Matrix ermöglicht.[15] Abbildung 22 zeigt die Darstellung des Emulators im Browser, auf dem gerade ein Lauflicht abläuft. Die Transferaufgabe kann dabei in zwei Stufen erfolgen: Erstens ein Lauflicht, welches von rechts links in der ersten Zeile der LED-Matrix entlang läuft und zweitens eine komplexere Variante, bei der unter Verwendung der verschachtelten Schleifen zeilenweise die LED-Matrix durchlaufen wird. An dieser Stelle sind weitere Variationen der Aufgabenstellung möglich und erlauben einen gewissen Raum zum Anpassen der Schwierigkeit und vorgegebenen Lösungsstrukturen.

Die Diskussionen und der Vergleich der Lösungen erfolgt dabei wie in Kapitel 5.2.3 beschrieben. In den begleitenden Übungen können diese Aufgaben noch erweitert werden, indem eine Vorwärts- und Rückwärtsbewegung des Lauflichtes gefordert wird. In späteren Unterrichtseinheiten kann die Iteration im Sinne des Spiralprinzips erneut z. B. beim Traversieren von Listen oder Zeichenketten gezeigt werden.

### 5.2.7 Fazit

Das vorgestellte Konzept stellt einen abstrakten Ablaufplan für die Durchführung von Programmieraufgaben zur Konzepteinführung dar und beantwortet somit Forschungsfrage Punkt 3, indem das Konzept Schreib-, Lese- und Diskussionsaktivitäten

[15] Für weitere Informationen über die Verwendung der Hardware in der Lehre siehe [McC17].

in Vorlesungen integriert. Dabei werden nicht nur Aufgaben eingebettet, um den Unterricht aktiver zu gestalten, sondern es wird zielgerichtet die Einführung neuer Programmierkonzepte fokussiert. Betont wird dabei die problemorientierte Einführung und die verdrehte Reihenfolge, sodass neue Inhalte in induktiver Form vorgestellt und auf weitere Bereiche angewendet werden können. Im Vergleich zu den in Kapitel 4.2.1 vorgestellten Ansätzen, stellt die Problemorientierung nicht das Problemlösen in den Vordergrund, sondern dient der Einführung, die an Problemen ausgerichtet ist. Bei der Konzeptvorstellung und den Erläuterungen können wiederum passende Methoden zur Vermittlung gewählt werden. Wichtig ist dabei die Einordnung in größere Zusammenhänge, die ausgelösten Zustandsänderungen sowie der Programmablauf, der durch das Konzept verändert wird. Ebenfalls ist die Anzahl der Übungen und deren Durchführung (z. B. in Gruppen) frei wählbar. Wichtig sind die anschließenden Diskussionsphasen in denen die Studierenden ihre Ergebnisse mit anderen vergleichen und ggf. vorstellen müssen. Dementsprechend muss diesen Phasen die benötige Zeit eingeräumt werden und auf die Fragen und Rückmeldungen angemessen eingegangen werden. Die Anzahl der zu diskutierenden Lösungen hängt dabei von der Studierendenzahl und den Rückmeldungen ab und muss dabei vom Lehrenden während des Unterrichts entschieden werden.

Zusammengefasst zeigt das vorgestellte Konzept der *Interaktiven Aufgabenspirale* eine mögliche Integration des *Cognitive Apprenticeship* in Vorlesungen. Das Adjektiv *interaktiv* bezieht sich einerseits dabei auf die Kommunikation zwischen Lernenden untereinander und mit Lehrenden. Auf der anderen Seite deutet es auf den aktiven Part der Studierenden durch die Bearbeitung der Aufgaben an, sodass diese die Gelegenheit erhalten, sich enaktiv mit dem Konzept auseinanderzusetzen und erste Heuristiken auf metakognitiver Ebene auszubilden. Ziel ist es ein *Rezept* zum Lösen ähnlicher Probleme zu präsentieren und dieses mehrmals anzuwenden. Die Transferaufgaben ermöglichen eine Erweiterung der ersten Beispiele, um den abstrakten Charakter der Programmierkonzepte hervorzuheben. Weiterhin wird durch diese Interaktivität eine Atmosphäre vergleichbar zum seminaristischen Unterricht geschaffen, in denen Studierende frühzeitig Rückmeldung über Probleme und nicht verstandene Inhalte geben sollen und dürfen. Ohne diese Rückmeldungen kann der Lehrenden den Unterricht nicht anpassen, um die Studierenden am aktuellen Lernfortschritt abzuholen.

## 5.3 Auf dem Weg zur Interaktivität

Das in den vorherigen Kapiteln beschriebene Konzept und Beispiele erfordert zwei unterschiedliche Arten von Interaktivität, die bei der Gestaltung und Auswahl von Lernmaterialien und unterstützenden Werkzeugen notwendig sind. Nachfolgend wer-

den einige Überlegungen zur Umsetzung des vorher beschriebenen Konzepts in Bezug auf die Interaktivität, die Effizienz und auf die Wiederholung angestellt. Diese führen schlussendlich zur Entwicklung eines Systems, welches einerseits den Lehrende unterstützt und den Studierenden die Durchführung von Programmieraufgaben ermöglicht.

### 5.3.1 Interaktivität

Das erarbeitete Konzept basiert auf der Mitarbeit der Studierenden während der Vorlesung, um aktivierende Prozesse zu initiieren und somit das Lernen zu fördern. Aus der Beschreibung lassen sich einige Schlüsselelemente für die Umsetzung extrahieren, die für das Gelingen notwendig sind. Die Grundlage des Konzeptes bildet das *Aktive Lernen*, welches dementsprechend einen Lerner-zentrierten Unterricht mit Aktivitäten seitens der Studierenden vorsieht. Jedoch müssen die Aktivitäten zielgerichtet in den Unterricht eingebettet werden und von Diskussionen und reflektierenden Phasen begleitet werden, um metakognitive Prozesse zu ermöglichen. Dabei sind zwei Perspektiven bei der Umsetzung des Konzeptes wichtig, erstens die der Studierenden und zweitens die der Lehrenden. Ausgehend vom *Cognitive Apprenticeship* Ansatz und der zusätzlichen expliziten Phase der problemorientierten Einführung der Konzepte changieren instruktionelle und konstruktive Unterrichtsaktivitäten. Bedingt durch diesen Wechsel entsteht eine Interaktivität zwischen den Akteuren, welche eine Möglichkeit bietet den Unterricht aneinander auszurichten. Demonstrative Handlungen wie das *Live Coding* und die explizite Artikulation der kognitiven Prozesse seitens der Lehrenden, geben den Lernenden Einblicke in das informatische Problemlösen und mögliche Lösungsstrategien. Auf der anderen Seite können sich Studierende bei der Bearbeitung von Aufgaben mit Kommilitonen auseinandersetzen und die positiven Effekte des kooperativen bzw. kollaborativen Lernen nutzen. Schlussendlich führt die Diskussion möglicher Lösungsansätze der Studierenden und dem Lehrenden zu reflektierenden und diskursiven Aktivitäten. Anhand dieser kann der Unterricht angepasst werden, um besser auf den aktuellen Lernfortschritt und Fehlkonzeptionen zu reagieren. Zugleich soll das Konzept kein zu festes Korsett darstellen, sondern als Rahmen dienen, der bei Bedarf modifiziert werden kann. Denn die Betonung des Austausches zwischen Lernenden und Lehrenden sollte nicht auf einzelne Schritte reduziert werden, sondern als *modus operandi* in der Vorlesung wahrgenommen werden. Auf Fragen und Rückmeldungen sollte immer angemessen eingegangen werden, um Studierenden die Möglichkeit zur Steuerung der Vorlesung zu geben. Dies funktioniert bei Berücksichtigung des Curriculums nur in einem begrenzten Rahmen, sollte aber die Ansätze des Aktiven Lernens nicht konterkarieren. Aktivitäten des A*Aktiven Lernens* erfordern Zeit, welche den Lernenden entsprechend eingeräumt werden muss, um nicht bei zu engen Vorgaben negative Auswirkungen auf die Motivation auszulösen. Dabei kann

die Umsetzung durch eine geeignete Wahl der unterstützenden Werkzeuge effizient gestaltet werden.

### 5.3.2 Effiziente Umsetzung

Die Durchführung der einzelnen Phasen sollte so effizient wie möglich und von den Studierenden nachvollziehbar erfolgen. Entsprechend lassen sich Anforderungen an mögliche Werkzeuge und Materialien ableiten, um den Unterricht zu gestalten. Größere Studierendenzahlen (50+) verhindern die Verwendung eines Computerlabors zur Einbettung der Aktivitäten, da die Räume keine ausreichende Kapazität vorweisen. Dementsprechend wird bei den folgenden Überlegungen ein Seminarraum bzw. Vorlesungssaal inklusive eines *Beamers*, jedoch ohne weitere Ausstattung, angenommen. Aus Sicht des Lehrenden ist die Darstellung von Beispielen, die Möglichkeit zur Demonstration des Vorgehens sowie die Diskussion studentischer Lösungen für die Umsetzung des Konzepts entscheidend. Einerseits bieten gängige Präsentationswerkzeuge wie *PowerPoint* nur begrenzt die Möglichkeit zur Darstellung von Quelltexten und deren Ausführung. Auf der anderen Seite sind professionelle Entwicklungsumgebungen in der Regel nicht auf die Darstellung auf Projektoren und Leinwänden optimiert, sodass entweder der Text oder die Schaltflächen zu klein bzw. zu groß sind. Die Verwendung einer Entwicklungsumgebung ist für das Verständnis der Konzepte und deren Anwendung zweitrangig. Zwar sollte der komplette Programmierprozess inklusive der Verifikation durch die Implementierung durchlaufen werden, jedoch kann dies mittels einer geeigneten Umgebung erfolgen. Diese sollte eine für *Beamer* optimierte Darstellung zur Verwendung innerhalb der Vorlesung unterstützen. In Kapitel 2.2 wurde bereits die Überforderung bei der Bearbeitung von Programmieraufgaben durch eine Vielzahl neuer Konzepte angesprochen. Zusätzlich müssen die Lernenden sich zu Beginn mit einer Entwicklungsumgebung und der dazugehörigen Werkzeugkette auseinandersetzen, um diese bedienen zu können. Der Fokus sollte aber nicht auf der Bedienung einer Anwendung, sondern auf den einzelnen Phasen des Programmierprozesses liegen, weswegen bei der Auswahl der Entwicklungswerkzeuge auf eine gute Bedienbarkeit geachtet werden sollte.

Auch bezogen auf die Interaktionen während des Unterrichts müssen entsprechende Werkzeuge und Prozesse gewählt werden, um einen effizienten Ablauf zu ermöglichen. Studentische Lösungen werden im Rahmen des Konzeptes im Plenum und für andere Studierende sichtbar diskutiert, weswegen ein Mechanismus zur Abgaben von Lösungen notwendig ist. Die Abgabe sollte in digitaler Form erfolgen, um den Aufwand gering zu halten, der beim Abtippen von schriftlichen Lösungen oder Tafelanschriften entstehen könnte. Gleichzeitig sollte der Lehrende die abgegebenen Lösungen in wenigen Schritten für die Diskussion für alle Lernenden verfügbar machen.

Der Aufwand zur Abgabe und bis zur Verwendung durch den Lehrenden sollte dabei so leicht und unkompliziert sein, sodass keine technischen oder organisatorischen Probleme die Durchführung behindern. Die Bearbeitung von Aufgaben inklusive der Implementierung sollte dabei in einer geeigneten Umgebung stattfinden, die ohne Vorkonfiguration oder Einrichtung zu verwenden ist. Im Angesicht des aktuellen Trends zu mobilen Geräten wie *Tablets* (z. B. iPad), sollte eine mögliche Umgebung diese unterstützen. Zusätzlich müssen Studierenden die Aufgabenstellungen und die bereits teilweise vor ausgefüllten Lösungen (vgl. Kapitel 5.2.3) zur Verfügung stehen, sodass diese keine Zeit mit unnötigem Abtippen verbringen.

Für die problemorientierte Einführung neuer Konzepte, sollte die Verwendung anschaulicher Beispiele, die ggf. fachbezogene Inhalte thematisieren, möglich sein. Hierbei ist eine simple Verwendung und Konfiguration für die Studierenden erstrebenswert, um die Anwendung der Konzepte zu ermöglichen. Dies stellt eine weitere Art der Interaktivität dar, denn die Studierenden sollen innerhalb des Programmierprozesses den eigenen Lösungsentwurf durch die Implementierung verifizieren. Bei einem iterativen Vorgehen ist die Verwendung von geeigneten Ein- und Ausgabewerten für die Überprüfung der einzelnen Schritte notwendig. Ebenso ist die Verwendung von Beispielen, die eine grafische Ausgabe oder eine Simulation bzw. Emulation von Hardware benötigen interaktiv, da der Studierende sich aktiv mit den Beispielen und seiner Lösung auseinandersetzen kann. Diese müssen jedoch ebenfalls effizient und ohne große Vorbereitung benutzbar sein, um den zeitlichen Aufwand für solche Beispiele in Grenzen zu halten.

Für die Durchführung der angeleiteten Übungen und der Transferaufgaben ist eine effiziente Umsetzung wünschenswert, um die Zeitverluste durch organisatorische Aufgaben zu reduzieren. Gleichzeitig erfordert das Konzept einen Mechanismus zum Einreichen von Lösungen und einer geeigneten Darstellung dieser in Vorlesungssälen.

### 5.3.3 Unterrichtssteuerung, Wiederholung und explorative Konzepte

In den lehr- und lerntheoretischen Überlegungen wurden die Steuerung der Lernprozesse und damit die der Vorlesung und des Unterrichts durch die Lernenden gefordert. Die Ergebnisse der studentischen Aktivitäten müssen durch den Lehrenden interpretiert und anschließend ggf. an den Unterricht angepasst werden. Hierbei stellt sich die Frage, ob bei der Einschätzung der Studierenden neben den Lösungen noch weitere Indikatoren hilfreich sind und wie diese erfasst werden können? Der letzte Schritt im beschriebenen Konzept (vgl. Kapitel 5.2) thematisiert die Aufgaben und Anwendung des Konzeptes in komplexeren Aufgaben in den Übungsveranstaltungen. Auftretende

Fragen und Missverständnisse bei der Bearbeitung der Aufgaben können zur Vorbereitung der nächsten Vorlesung verwendet werden. Jedoch ist dazu eine passende Erfassung und Auswertung notwendig, um entsprechend zu reagieren und den Unterricht anzupassen.

Bereits in der Beschreibung des Programmiervorgangs wurde in der Definition von Capsersen der Begriff *explorativ* verwendet, um den iterativen Charakter des Programmierens zu beschreiben. Ebenso haben Esper et al. explorative Hausaufgaben entwickelt, die bei der Entwicklung des Unterrichtskonzepts schlussendlich mit eingeflossen sind. Studierende sollen Konzepte nicht nur mental nachvollziehen, sondern ebenfalls enaktiv durcharbeiten und erkunden, um Heuristiken auszubilden. Dementsprechend muss die Verwendung und das Ausprobieren eines neuen Ansatzes leicht möglich sein. Auch die Beantwortung von Fragen und ein interessengeleitetes Erkunden von Inhalten ist wünschenswert und sollte nicht durch eine komplizierte Installation von Entwicklungswerkzeugen erschwert werden. Lernmaterialien und Lernumgebungen sollen Studierende ermutigen sich mit den Inhalten zu beschäftigen und diese zu erkunden.

Einen weiteren Aspekt stellt die Ergebnissicherung und Wiederholung der Aktivitäten in der Vorlesung dar. Denn Lernende sollen sich auch nach der Vorlesung mit den Inhalten und ihren eigenen Lösungen bzw. anderen Ansätzen beschäftigen. Dabei stellt sich die Frage, wie Aufgaben und Lösungen zur Verfügung gestellt werden können? Übungen und die Wiederholung dienen als Festigung des Gelernten und sollten somit nicht nur in der Vorlesung, sondern ebenso in der Übung und von den Studierenden ohne Beaufsichtigung durchgeführt werden können. Für das Programmieren ist somit eine Entwicklungsumgebung, die leicht zu installieren ist und die drei gängigen Betriebssystemen unterstützt, wichtig.

### 5.3.4 Zusammenfassung

Die bisherigen Überlegungen können in mehrere Punkten zusammengefasst werden und dienen als Leitlinie für die Auswahl und ggf. Entwicklung geeigneter Werkzeuge zur Umsetzung des Konzepts. Die im Konzept und in der Problemstellung bereits geforderte aktive Partizipation der Studierenden soll durch geeignete Lernmaterialien während der Vorlesung und an anderen beliebigen Orten möglich sein. Programmierkonzepte und der komplette Programmierprozess muss von den Studierenden durchlaufen werden und auch später nachvollziehbar sein. Das bedeutet, dass Lernergebnisse und studentische Lösungen gesichert werden müssen, um ggf. zuhause nochmal nachvollzogen werden zu können. Eine problemorientierte Einführung und interaktive bzw. anschauliche Beispiele sind dabei von großer Bedeutung. Denn dadurch können Studierende einerseits durch einen möglichen Fachbezug motiviert werden. An-

dererseits ermöglichen interaktive Elemente, wie z. B. *Turtle Graphics* oder Simulationen, eine grafische Überprüfung des Algorithmenentwurfes. Studierende sollen dazu angeregt werden sich mit Inhalten aktiv auseinanderzusetzen und teilweise selbstgesteuert zu lernen. Das Üben findet nicht nur innerhalb der Vorlesung und den Übungsveranstaltungen statt, sondern ist im jeweiligen Habitat der Studierenden selbst möglich. Somit sind leicht zu verwendende und zugängliche Materialien erstrebenswert, um Studierenden einen einfachen Einstieg zum Thema zu ermöglichen. Besonders am Anfang liegt der Fokus beim Programmieren nicht auf der Bedienung einer Anwendung und Werkzeugkette, stattdessen auf den dahinter liegenden Konzepten. Dies soll jedoch nicht die Codierung außen vorlassen, sondern es müssen Umgebungen konzipiert werden, die eine einfache Bearbeitung von solchen Aufgaben in Vorlesungen und außerhalb ermöglicht. Unweigerlich führt Programmieren und das Erlernen einer Programmiersprache immer zu einer Auswahl und Entscheidung für Entwicklungsumgebungen und Werkzeugen, die dabei notwendig sind. Hinsichtlich der geforderten effizienten und interaktiven Umsetzung des Konzepts stellen sich mehrere Fragen:

1. Mit welcher Software bzw. welchen Werkzeugen sollen die Aufgaben innerhalb der Vorlesung von den Studierenden bearbeitet werden?
2. Wie lassen sich studentische Lösungen in der Vorlesung ohne große Zeitverluste diskutieren und ggf. zur Verfügung stellen?
3. Wie können Aufgaben und Angaben in der Vorlesung den Studierenden leicht zugänglich gemacht werden?
4. Mittels welcher Software kann *Live Coding* innerhalb der Vorlesung durchgeführt werden?
5. Wie können Vorlesungsmaterialien inklusive der im Unterricht durchgeführten Aufgaben Studierenden leicht zugänglich gemacht werden?
6. Wie können die Materialien gestaltet werden, sodass Konzepte explorativ ausprobiert und wiederholt werden können?
7. Wie kann der Lehrende durch Indikatoren den Unterricht besser anpassen bzw. planen?

Im Rahmen dieser Arbeit wurde eine interaktive Plattform zur Einbettung solcher Aufgaben entwickelt, die gleichzeitig Antworten auf die gestellten Fragen gibt. Diese wurde zugleich für die Durchführung und Evaluation des Konzeptes benötigt, sodass eine Bewertung des Ansatzes im Sinne des forschungsmethodischen Vorgehens möglich ist.

# Kapitel 6

# Plattform für interaktive Vorlesungen

Ausgehend von den Überlegungen in Kapitel 5.3 wurde eine Plattform zur effizienten Integration von Programmieraufgaben innerhalb von Vorlesungen entwickelt, da vorhandene Werkzeuge nur teilweise die Anforderungen und Überlegungen abdecken. Dementsprechend werden zuerst vorhandene Ansätze und Werkzeuge diskutiert und anschließend die im Rahmen dieser Arbeit entwickelte Plattform zur Integration von Aktivitäten des *Aktiven Lernens* vorgestellt. Diese ermöglicht gleichzeitig Studierenden das Erkunden von Konzepten außerhalb der Vorlesung, ohne die Notwendigkeit zur Installation von Werkzeugketten und Entwicklungsumgebungen gezwungen zu sein. Zugleich ist die Plattform nicht auf statische Inhalte beschränkt, sondern ermöglicht das Einbinden beliebiger Inhalte und die Erweiterung um neue Aktivitäten.

## 6.1 Vorhandene Ansätze

In Kapitel 2.3 wurde bereits auf den Einfluss der *Interactive eBooks* (interaktive E-Books) eingegangen, die den Impuls für die Entwicklung des Konzepts und die Umsetzung mittels der nachfolgend beschriebenen Plattform gegeben haben, um interaktive Lernmaterialien zu gestalten. Diese sind jedoch primär für die Verwendung außerhalb von Präsenzveranstaltungen ausgelegt, weswegen weitere Ansätze für eine Eignung innerhalb der Vorlesung analysiert wurden. Zuletzt werden noch einige Werkzeuge zur einfachen Bearbeitung von Programmieraufgaben vorgestellt, welche maßgeblich in die entwickelte Plattform eingeflossen sind.

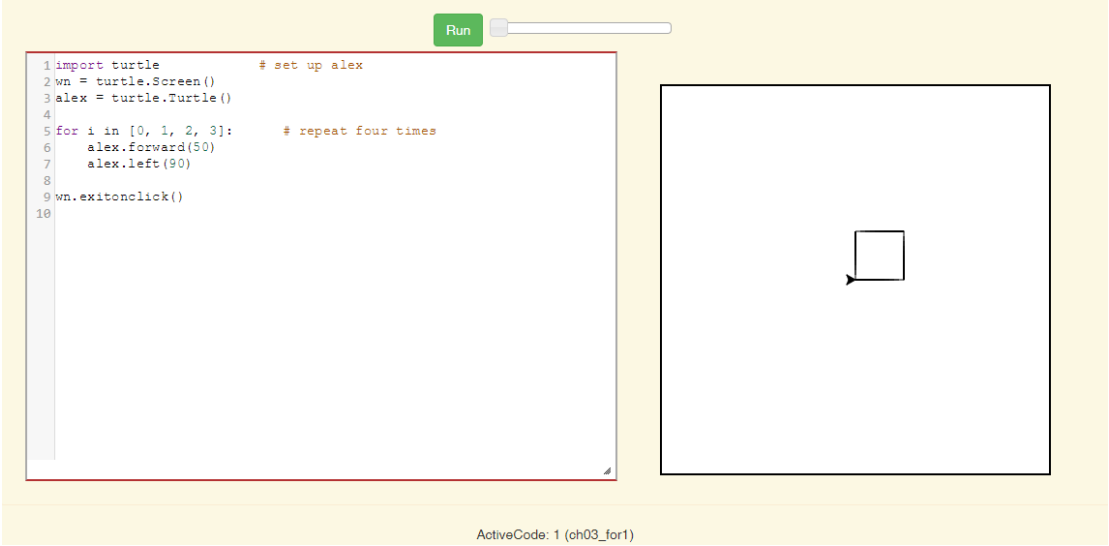



**Abbildung 23**: Ausführung von Beispielen in einem interaktiven Textbuch

### 6.1.1 Interaktive Materialien

#### Interaktive Textbücher

Korhonen et al. haben in einer Arbeitsgruppe Anforderungen und Strategien zum Erstellen von *Open Source Interactive Computer Science eBooks* erarbeitet [Kor+13]. Anlass dazu waren neue Anbieter für sogenannte *MOOCs* (*Massive Open Online Courses*) wie z. B. *Khan Academy*[16], die interaktive Onlinekurse für Themen wie Informatik oder Mathematik anbieten [MD14]. Ziel solcher Angebote ist die Bereitstellung von Inhalten über das Internet als eine Art Fernunterricht inklusive einer (teil)automatisierten Lernerfolgskontrolle durch automatische Bewertungen von Aufgaben. Die Arbeitsgruppe hat dabei vier Kernelemente solcher E-Books identifiziert, welche in den Fließtext integriert werden sollen:

1. Interaktive Visualisierungen,
2. automatisch bewertete Quizze und regelmäßiges Feedback für den Lernenden,
3. interaktive Lernaktivitäten für Konzepte inklusive der Bearbeitung und das Ausführen von Codesegmenten
4. sowie eine Anpassung der Inhalte und Lernmodule durch den Lehrenden.

Alle diese Punkte stellen ungeordnete Aktivitäten und Elemente dar, die allerdings im

[16] Siehe www.khanacademy.org

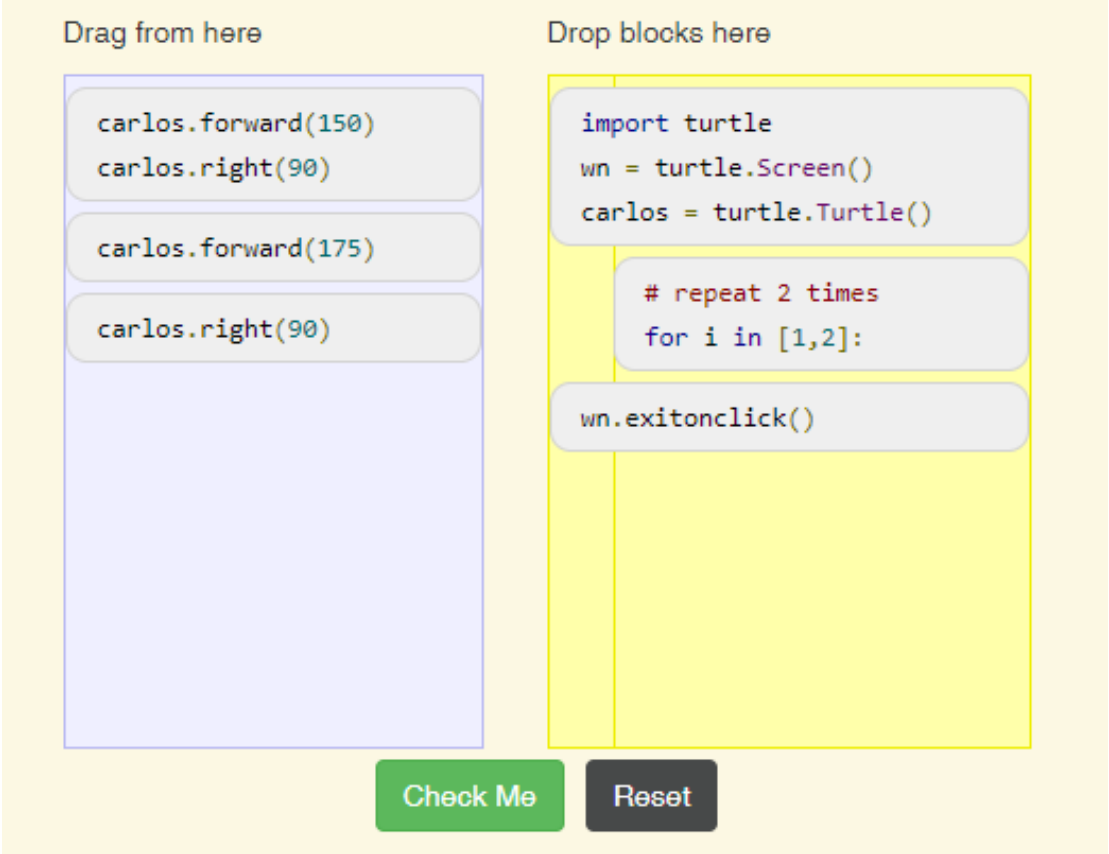



**Abbildung 24:** Aufgabe mit Code-Magneten

Rahmen der Arbeitsgruppe nicht zu einem didaktischen Gesamtkonzept weiterentwickelt wurden. Vielmehr wurden nur Hinweise auf die Nutzung von *Peer Instruction* oder die Verwendung von solche E-Books als Ersatz von Textbüchern gegeben. Diese Elemente sind stark an Bradley Millers Arbeit zur Entwicklung interaktiver Textbücher orientiert und beschreiben hauptsächlich die technischen Aspekte [MR12]. Miller entwickelte die *Runestone* Plattform zum Erstellen dieser Textbücher, welche die Ausführung von Python direkt im Browser ermöglichen (vgl. Abbildung 23). Die Ausführung des Codes und die Darstellung der grafischen Ausgaben basiert auf *skulpt*[17]. *skulpt* ist ein von Scott Graham entwickelter JavaScript-Interpreter für Python, der vollständig im Browser Python-Quelltexte ausführt. Dieser Interpreter unterstützt primär Python 2 und Teile der Python 3 Syntax, weswegen nur wenige Standardbibliotheken verwendet werden können. Auf Basis dieses Interpreters wurden noch weitere interaktive Elemente wie z. B. Codemagneten entwickelt, bei denen zur Lösung zufällig geordnete Codezeilen in die richtige Reihenfolge gebracht werden müssen. (vgl. Abbil-

[17] Siehe www.skulpt.org

dung 24). Im angelsächsischen Raum werden diese auch als *Parson's programming puzzels* bezeichnet [IK10; PH06]. Weiterhin ist die Verwendung von klassischen *Multiple Choice* Fragen möglich, die im Fließtext eingebettet sind. Benutzer können sich anmelden und ihre jeweiligen Lösungen speichern, um diese später wieder abzurufen. Die von Brad Miller entwickelte Plattform basiert dabei auf dem Dokumentationswerkzeug *Sphinx*, welches aus in Ordner strukturierten Kapiteln das E-Book erstellt wurde. Inhalte werden in einer speziellen Auszeichnungssprache namens *reStructuredText* verfasst, welche das Verwenden der interaktiven Elemente vereinfacht. In Quellcode 3

```
1  ====
2  Beispiel in reStructuredText
3  ====
4
5  Die nachfolgenden Zeilen werden in einen Code-Editor umgewandelt:
6
7  .. activecode:: codeid
8     :language: python
9
10    print("hello world!")
```

**Quellcode 3**: Beispiel für reStructuredText und das Einbinden eines Code-Editors

zeigt ein Beispiel in der Auszeichnungssprache, welches anschließend durch *Sphinx* in Formate wie z. B. *HTML* oder *PDF* transformiert wird. Die Zeile `.. activecode:: codeid` erzeugt den in Abbildung 23 gezeigten Editor, der direkt in Webseiten verwendet werden kann. Ein Nachteil dieser technischen Lösung ist die notwendige Transformation vom Ausgangsdokument in das jeweilige Zielformat, welche bei jeder Änderung erneut durchgeführt werden muss. Sollen Ergebnisse aus der Vorlesung direkt im Textbuch zur Verfügung gestellt werden, muss dafür ebenfalls der beschriebene Prozess durchlaufen werden. Die Verwendung einer Auszeichnungssprache ist zwar angenehmer als die Verwendung von *HTML*, jedoch ist die Transformation bei einer iterativen Entwicklung der Unterrichtsmaterialien hinderlich. Zusätzlich wird derzeit keine optimierte Darstellung zur Verwendung in Vorlesungssälen unterstützt. Sphinx ermöglicht zwar das Erstellen von Präsentationen, jedoch müssen die Inhalte dazu entsprechend angepasst werden. Deutlich einfacher wäre eine Möglichkeit zu Pflege eines einzelnen Dokumentes, welches in der Leseansicht umfangreiche und in der Präsentation nur ausgewählte Inhalte anzeigt. In Bezug auf die Interaktivität zwischen Lehrenden und Lernenden gibt es bisher nur Unterstützung durch ggf. die Einbindung von anderen Diensten wie *Perseus*[18] (Werkzeug zum Erstellen und Anzeigen von interaktiven Fragen und Aufgaben). Diese Bibliotheken und Werkzeuge sind allerdings nur für die Verwendung in *MOOCs* konzipiert und nicht für die Kommunikation direkt im Unterricht. Jedoch bietet *Runestone* die Möglichkeit eine Kommentarfunktion

[18] Siehe https://github.com/Khan/perseus

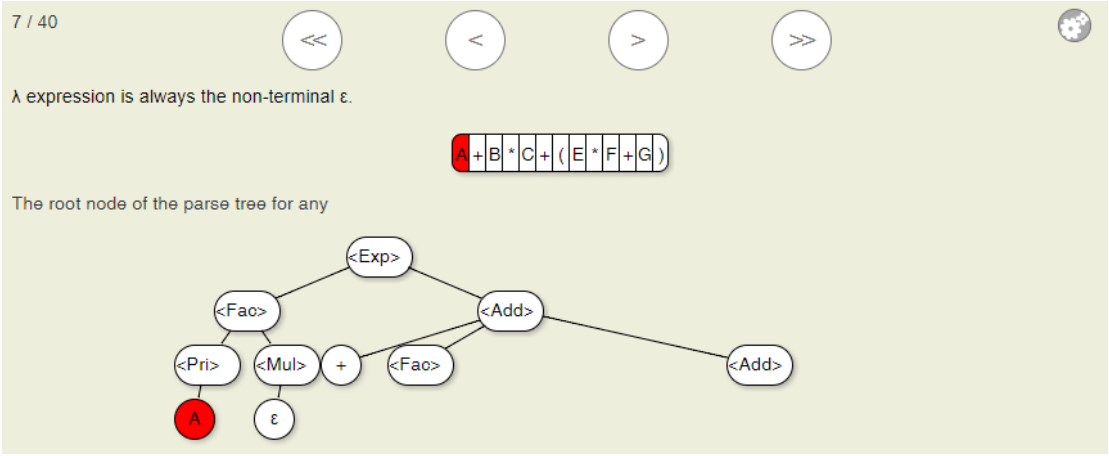



**Abbildung 25**: Interaktive Visualisierung in einem OpenDSA E-Book

neben den Programmieraufgaben einzublenden, sodass Studierende sich gegenseitig helfen können.

Zusammengefasst ermöglicht *Runestone* das Erstellen interaktiver Inhalte für Programmiervorlesungen, unterliegt dabei aber einigen Einschränkungen. Denn die Programmiersprache ist auf Python beschränkt und Änderungen an den Inhalten und Texten erfordern eine erneute Kompilierung in das gewünschte Zielformat. In mehreren Studien wurde der positive Einfluss auf den Lernerfolg durch die Verwendung interaktiver E-Books und insbesondere durch die Verwendung von *Worked Examples* und eingebetteten Aufgaben nachgewiesen [Eri+15; EGM15]. Färnqvist et al. sowie Miller und Ranum konnten ebenfalls eine Verbesserung der Durchfallquoten und der Noten nach der Einführung solcher E-Books nachweisen [MR12; Fär+16].

Neben der *Runestone* Plattform existieren noch weitere, die einen ähnlichen Funktionsumfang bieten. *OpenDSA* ist eine Plattform zum Erstellen von interaktiven E-Books, die ebenfalls auf *Sphinx* basiert, jedoch andere interaktive Elemente, wie z. B. die Möglichkeit zur Visualisierung von Algorithmen[19], bietet (siehe Beispiel in Abbildung 25). Weiterhin wird das Generieren von Präsentationen unterstützt, wobei immer der komplette Inhalt angezeigt wird. Aus dem Ausgangsdokument können keine unterschiedlichen Varianten, z. B. Präsentation und Skript, generiert werden, sondern beide benötigen eigene Quellen. Wie bereits bei der *Runestone* Plattform angemerkt, muss hier ebenfalls bei jeder Änderung das komplette Textbuch bzw. die Präsentation nochmals kompiliert werden.

*zyBooks* ist ein kommerzieller Anbieter für interaktive Textbücher, die ebenfalls eingebettete Programmieraufgaben unterstützen. Abbildung 26 zeigt eine Programmier-

[19] Diese Visualisierungen müssen mittels einer Bibliothek in JavaScript erstellt werden.

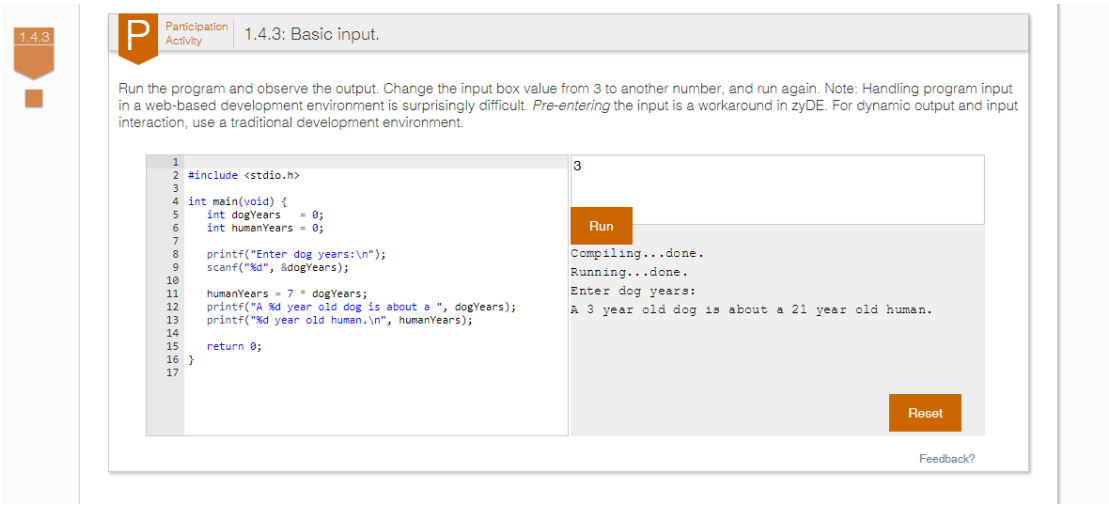



**Abbildung 26**: Programmieraufgabe in zyBooks

aufgabe in *zyBooks*, in welcher eine Eingabe eingelesen wird. Die Eingaben müssen jedoch bereits vor der Ausführung angegeben werden, da während der Ausführung nicht mit dem Programm interagiert werden kann. Der Nachteil dieser Plattform ist die starke Fokussierung auf den Ersatz traditioneller Textbücher, weswegen keine eigenen Texte verfasst werden können, sondern nur die zur Verfügung gestellten Inhalte genutzt werden können. Aus diesem Grund wird dieser Anbieter bei weiteren Diskussionen und Vergleichen nicht weiter berücksichtigt.

Bei allen Beispielen kommen immer wiederkehrende interaktive Elemente vor. Brusilovsky et al. bezeichnen diese als *Smart Learning Content* im Bereich der Informatik und haben dazu eine umfassende Recherche von frei verfügbaren interaktiven Elementen durchgeführt und ein Protokoll zum Integrieren dieser postuliert. Denn ein Großteil sind eigenständige Systeme, die den *Learning Tools Interoperability* (kurz LTI) Standard nicht unterstützen und somit nicht in Moodle oder anderen Plattformen einsetzbar sind. Eine konkrete Implementierung wurde von Brusilovsky et al. nicht zur Verfügung gestellt, weswegen Sirkiä und Haaranen darauf einen eigenen Server zum Einbinden dieser interaktiven Elemente in bereits vorhandene Plattformen entworfen haben [SH15]. All diese angebotenen Elemente lassen sich wiederum nur schwer für die Darstellung mit *Beamern* optimieren bzw. müssen immer in Verbindung mit einer Plattform zum Erstellen der Inhalte genutzt werden.

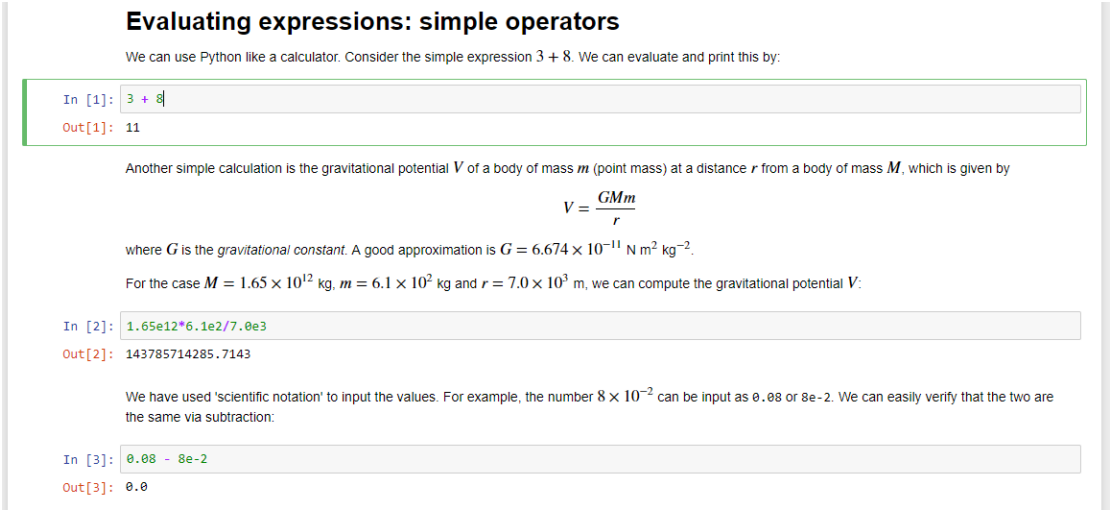



**Abbildung 27**: Jupyter Notebook Beispiel mit Python Quelltexten

### Jupyter Notebooks

Ein etwas anderer Ansatz, der mittlerweile großen Zuspruch und Aufmerksamkeit in der Lehre gewonnen, sind die sogenannten *Jupyter Notebooks*. Diese stellen eine interaktive web-basierte Umgebung zum Kombinieren von ausführbaren Code, Texten, mathematischen Formeln, Diagrammen und weiteren Medien zur Verfügung [Ros15][S.2ff]. Ursprünglich wurden diese im Jahre 2011 als interaktives Notizbuch für Wissenschaftler konzipiert und in Verbindung mit *IPython* (Interactive Python) entwickelt [PG07]. Doch innerhalb kürzester Zeit wurde das System zu einer allgemeinen Plattform für eine Vielzahl an Programmiersprachen, wie z. B. R, Julia oder Lua, entwickelt. In Abbildung 27 ist ein Beispiel für ein solches Notebook zu sehen, welches Text und ausführbaren Quelltext (Python 3) enthält. Notebooks sind dabei eine Aneinanderreihung von mehreren Inhaltselementen, die auch als Zellen bezeichnet werden. Im Gegensatz zu den bisher vorgestellten Lösungen in Kapitel 6.1.1 können die Inhalte direkt im Browser, ohne einen Zwischenschritt zur Erzeugung des Zielformates, bearbeitet werden. Während die anderen Ansätze jeweils ganze Bücher mit mehreren Kapiteln erfassen, sind Notebooks immer einzelne Dokumente, die als Datei abgespeichert werden. *Jupyter Notebook* ist die Kombination aus einem Web-Server, welcher Programme ausführen kann, und einem web-basierten Klienten zur Darstellung und Bearbeitung der Inhalte. Somit muss vor der Verwendung immer zuerst die Software lokal installiert oder ein extern betriebener Server genutzt werden. Notebooks können mittels *nbviewer*[20] in einer Leseansicht statisch angezeigt werden, jedoch ohne die Möglichkeit das Dokument zu bearbeiten oder damit zu interagieren. Notebooks kom-

[20] Siehe nbviewer.jupyter.org für weitere Informationen

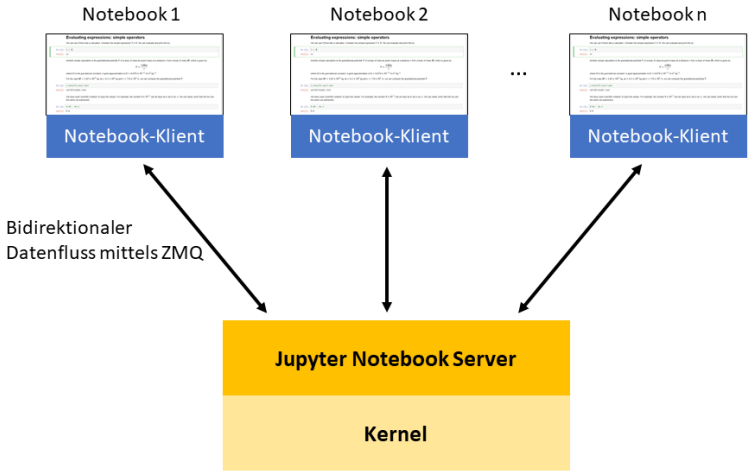



**Abbildung 28**: Kommunikation in Jupyter Notebooks

munizieren dabei über ein Protokoll (*ZMQ* [Akg13]) mit dem Computer bzw. Server, um Code auszuführen und Ein- und Ausgaben zu übermitteln. Ein Notebook-Server startet zum Ausführen einen sogenannten *Kernel*, der Programme in einer bestimmten Sprache kompilieren und ausführen kann und anschließend Nachrichten empfängt und versendet. Der Kernel stellt dabei das Backend für die Notebooks dar und ermöglicht die interaktiven Elemente, Variablen und Datenzugriffe. Das Gegenstück ist die Notebook-Oberfläche, welche die Dokumente darstellt und das Interagieren mit diesen ermöglicht. In Abbildung 28 ist zu sehen, dass ein Server mehrere Notebooks gleichzeitig ausführen kann. Jedoch verwendet der Notebook-Server keine Benutzerverwaltung, sondern ist für nur einen einzigen Benutzer ausgelegt. Für dieses Problem wurde *Jupyter Hub* entwickelt, welches eine umfangreiche Benutzerverwaltung ermöglicht und jedem Benutzer eine eigene Instanz eines Notebook-Servers zur Verfügung stellt.

Die Inhaltselemente sind in dem Format *nbformat* festgelegt und können von beliebigen Notebook-Darstellern anzeigt werden. Tabelle 6 enthält eine Aufzählung der vier möglichen Inhaltselemente, die in Notebooks verwendet werden können. Jeder Inhaltstyp kann zusätzlich mit weiteren Metadaten versehen werden, welche die Darstellung oder das Verhalten verändern können. Die Stärke von den Notebooks kommt besonders bei Visualisierungen zu tragen, da z. B. Diagramme direkt im Notebook dynamisch angezeigt werden können. Weiterhin wird im Vergleich zu *Runestone* der komplette Sprachumfang unterstützt und nicht nur eine Teilmenge.

**Tabelle 6:** Jupyter Notebook Inhaltselemente

| Element | Beschreibung |
|---|---|
| **Markdown** | Formatierter Text mittels Markdown |
| **Code** | Quelltext |
| **Code-Output** | Gespeicherte Ausgabe einer Code-Ausführung |
| **Raw** | Von Metadaten abhängige Darstellung von Inhalten, die nicht angezeigt werden sollten |

Bei näherer Betrachtung zeigen sich mehrere Nachteile für Programmieranfänger. Erstens findet die Ausführung von Code in einem Interpreter statt, der die jeweiligen Ergebnisse zwischenspeichert, da dieser nicht neu gestartet wird. Werden Code-Elemente durcheinander ausgeführt, kann dies in nicht gewollten Nebeneffekten resultieren. Im Bereich der wissenschaftlichen Auswertung von Daten ist dieses Verhalten jedoch von Vorteil, da bei Bedarf nur einzelne Berechnungen nach Anpassungen durchgeführt werden müssen, da die Ergebnisse noch zwischengespeichert sind. Auf der anderen Seite kann dies Programmieranfänger verwirren, da ihnen der Einfluss und die Zwischenspeicherung der Ergebnisse innerhalb des Interpreters nicht bewusst sind. Zweitens können Code-Beispiele nicht mehrere Dateien umspannen und die Bearbeitung von Quelltexten ist auf Quelltextfragmente ausgelegt. Hilfreiche Unterstützungen wie Fehlermarkierungen und andere Funktionen, wie sie in Entwicklungsumgebungen angeboten werden, sind nicht vorhanden. Schlussendlich ist die Verbreitung der Notebooks ein schwieriges Thema, da diese entweder von den Studierenden heruntergeladen und auf dem eigenen Computer in einer Jupyter Notebook Installation verwendet werden müssen. Oder andererseits muss ein *Jupyter Hub*, ein Web-Server mit Benutzerverwaltung, betrieben werden, auf dem die Studierenden die Notebooks bearbeiten können. Dazu wird für jeden Benutzer ein eigener Server bzw. Anwendungs-Container gestartet und verwaltet. Allerdings können bisher keine Sicherheitsregeln bei der Ausführung von fremden Code direkt in der Konfiguration vorgenommen werden, sondern müssen in Abhängigkeit der jeweiligen Infrastruktur implementiert werden. Dieses Modell bedingt unter anderem, dass entsprechende Ressourcen für eine Vielzahl an Studierenden zur Verfügung gestellt werden. Mittlerweile existieren Bemühungen von *O'Reilly Media* Jupyter Kernel auch in beliebigen Webseiten und Dokumenten einzupflegen, um ein kommerzielles Angebot an interaktiven Textbüchern zu ermöglichen[21]. Durch den Aufbau müssen Studierende immer auf einer Kopie des vom Lehrenden bereitgestellten Notebooks arbeiten, weswegen Änderungen und Aktualisierungen immer mit dem erneuten Kopieren verbunden sind. Dies

[21] *O'Reilly* stellt dazu eine freie Bibliothek namens *Thebe* unter https://github.com/oreillymedia/thebe zur Verfügung

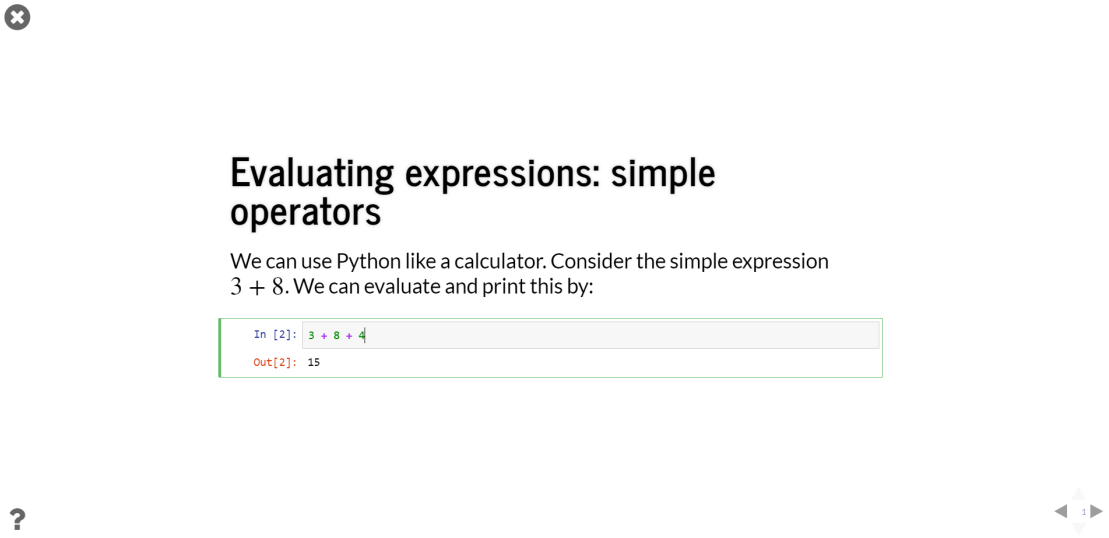



**Abbildung 29**: Beispielpräsentation mit Jupyter RiSE

macht das Verteilen von neuen Versionen mühselig bzw. verursacht Inkonsistenzen in den Skripten der Studierenden.

Im Bereich der Lehre wurde eine Erweiterung zum Generieren von Präsentationen aus den Notebooks erstellt. *RiSE*[22] ermöglicht das Annotieren der Inhaltselemente, um die Darstellung auf Präsentationsfolien festzulegen. In Abbildung 29 ist eine Folie zu sehen, die mittels *RiSE* generiert wurde. Im Vergleich zu traditionellen Präsentationsanwendungen ist die Präsentation ebenfalls web-basiert und ermöglicht das direkte Ausführen und Bearbeiten der interaktiven Elemente innerhalb der Folien. Dies erlaubt dem Lehrenden im Unterricht *Live Coding* durchzuführen bzw. auf Fragen der Studierenden zu antworten, ohne die Präsentation verlassen zu müssen. Beim Erstellen der Präsentation können die Inhaltselemente unterschiedlich auf der Folie dargestellt werden (vgl. Abbildung 30). Ein Element kann durch den *Slide Type* als Folie, Element auf der Folie oder als Fragment, welches erst durch Betätigen einer Taste erscheint, in den Metadaten markiert werden. Weiterhin können bestimmte Elemente aus der Präsentation ausgeschlossen werden bzw. als Präsentationsnotizen eingeblendet werden. Dadurch kann aus einem Skript eine angepasste Präsentation erstellt werden, ohne mehrere Dokumente pflegen zu müssen.

Zusammengefasst bieten *Jupyter Notebooks* eine gute Möglichkeit Inhalte interaktiv und erkundbar zu gestalten, dabei ist aber eine Installation bzw. Infrastruktur zur Bereitstellung der Inhalte notwendig. Der Fokus liegt stärker auf wissenschaftlichen Visualisierungen und weniger auf interaktiven und spielerischen Inhalten wie z. B.

[22] Siehe https://github.com/damianavila/RISE

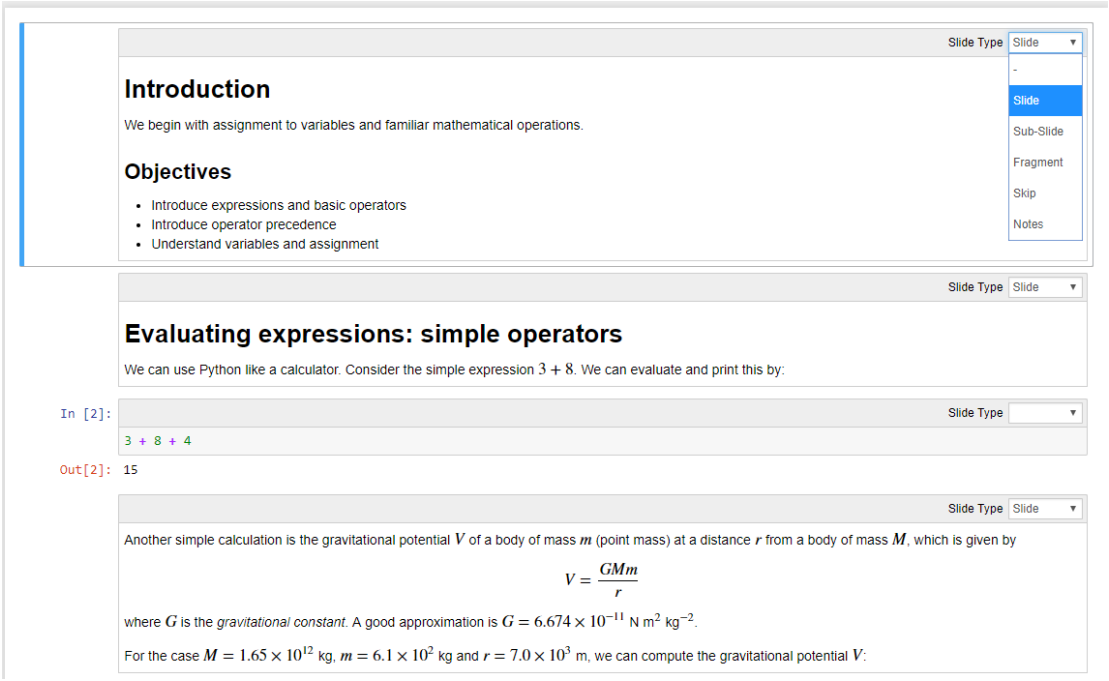



**Abbildung 30**: Erstellen einer Präsentation mit Jupyter RiSE

*Turtle Graphics*, welches nicht direkt unterstützt wird. Dies ist auch der Unterstützung und den Angeboten vieler Firmen, wie z. B. Microsoft (Azure Notebooks[23].) oder Kaggle (Wettwerbe und Kurse über Datenauswertung[24]) zu sehen, die eigene Online-Plattformen für wissenschaftliche Auswertungen (und teilweise für die Lehre) auf Basis des *Jupyter Notebooks* entwickeln.

### Fazit

Die Betrachtung der vorhandenen Ansätze zeigt, dass interaktive Lehrmaterialien bereits entwickelt und genutzt werden. Aus Sicht des Konzeptes und den daraus resultierenden Anforderungen zur effizienten Umsetzung sind jedoch beide Ansätze nur bedingt geeignet. Während die interaktiven Textbücher und die Plattform *Runestone* eine interessante Umgebung zum Erstellen und Verbreiten von interaktiven Materialien bieten, sind jedoch keine Mechanismen zur Verwendung innerhalb von Vorlesungen oder das Einreichen von Lösungen zum Besprechen im Unterricht vorhanden. Darüber hinaus muss bei Änderungen jeweils das komplette Textbuch neu generiert werden, sodass kleinere Änderungen nur aufwändig umzusetzen sind. Diese Vorge-

[23] Siehe https://notebooks.azure.com/
[24] Vgl. https://www.kaggle.com/.

hensweise ist der starken Orientierung an traditionellen Textbüchern, wie sie oft im angelsächsischen Raum verwendet werden, geschuldet. Somit ist Flexibilität und das Anpassen der nächsten Vorlesung auf Basis des studentischen Lernfortschritts nur eingeschränkt möglich.

Im Gegensatz dazu überzeugen *Jupyter Notebooks* mit direkter Bearbeitung der Dokumente im Browser, ohne einen Kompilierungsschritt durchlaufen zu müssen. Zusätzlich unterstützt das System eine Vielzahl an Programmiersprachen, auch wenn diese meist auf in einem Interpretermodus ausgeführt werden, wodurch Anfänger verwirrt werden können. Beide Ansätze ermöglichen das Verbinden von Beschreibungen und Codebeispielen und sind dementsprechend sehr gut für das Konzipieren der *Worked Examples* und interaktiver und teilweise selbstgesteuerter Lernumgebungen geeignet. *Jupyter Notebooks* unterstützen jedoch nur wenige interaktive Elemente im Vergleich zu *Runestone* bzw. diese müssen extern eingebunden werden. Um eine bessere Entscheidung zu treffen werden die Eigenschaften und Möglichkeiten der interaktiven E-Books und der *Jupyter Notebooks* in Bezug auf die Anforderungen zur Umsetzung des didaktischen Konzepts aus Kapitel 5 gegenübergestellt.

**Tabelle 7**: Vergleich der vorhandenen Ansätze zu interaktiven Lehrmedien

| | **Interaktive E-Books** | **Jupyter Notebooks** |
|---|---|---|
| **Programmieraufgaben** | Aufgaben können direkt im Text eingebettet werden. Eine automatische Bewertungen der Aufgaben ist möglich. | Dokumente können jederzeit bearbeitet und der Code ausgeführt werden. In Verbindung mit Texten können Aufgaben direkt eingebettet werden. Die automatische Bewertung ist durch Erweiterungen möglich. |
| **Sprachunterstützung** | nur eingeschränkte Python 2 bzw. 3 Unterstützung | Vielzahl an Sprachen möglich, jedoch jeweils in einem Interpretermodus |
| **Visualisierungen** | Unterstützung von *Turtle Graphics* und Visualisierung von Algorithmen möglich. | Grafische Ausgabe von Diagrammen und Tabellen möglich. |
| **Unterstützung in Vorlesungen (Präsentationen)** | Nicht vorgesehen bzw. nur durch doppelte Datenhaltung möglich. | Interaktive Präsentationen können direkt aus den Dokumenten heraus generiert werden. |
| **Einfache Diskussion der Lösungen im Unterricht** | Nicht möglich. | Nicht möglich |
| **Verteilung der Materialien** | Textbücher sind über eine Webseite erreichbar. | Weitergabe der Dokumente bzw. durch Kopieren des bereitgestellten Dokuments. |
| **Aktivitäten** | Viele unterschiedliche Aktivitäten wie z. B. Quizze oder Codemagneten möglich. | Ggf. durch Erweiterungen oder Eigenentwicklungen möglich. |
| **Auswertungen** | Auswertung der Quizze und anderen Aktivitäten möglich. | Dezentraler Ansatz erschwert die Auswertung, da kein Endpunkt zum Erfassen der Daten existiert. |
| **Entwicklungsumgebung** | Einfacher Editor ohne weitere Unterstützung | Einfacher Editor mit einfacher Autovervollständigung, jedoch keine Anzeige von Fehlern im Quelltext. |
| **Unterstützung mehrerer Dateien in Code-Beispielen** | Nicht möglich | Nicht möglich |
| **Simulationen (Hardware)** | Nicht möglich | Nicht möglich |

Leider unterstützt keiner der beiden Ansätze eine einfache und effiziente Diskussion von Aufgaben innerhalb von Vorlesungen. Im Falle der interaktiven E-Books kann dies durch eigene Erweiterungen umgesetzt werden, da die E-Books zentral gespeichert werden. Bei *Jupyter Notebooks* lässt sich ein solcher Ansatz schwieriger umsetzen, da Notebooks nicht zentral, sondern pro Benutzer gespeichert werden. Dies ist nur durch Verwendung des *Jupyter Hub* und eigener Entwicklung möglich, wobei es nicht das Problem der Verteilung der Dateien bzw. Daten löst. Weiterhin wird nicht die Fülle an Aktivitäten wie bei den E-Books unterstützt, sondern müsste erst umgesetzt werden. Simulationen von Hardware sind ebenfalls nicht möglich bzw. es existieren keine vorhandenen web-basierten Lösungen dazu. Weder die E-Books noch *Jupyter Notebooks* erlauben das Erstellen von Beispielen mit mehreren Dateien oder das Anzeigen von weiteren Ressourcen, die in Programmen verarbeitet werden sollen. Zwar können bei E-Books Textdokumente als Dateien verfügbar gemacht werden, jedoch können Studierende diese nicht anschauen oder bearbeiten. Die Notebooks können zwar vom Dateisystem aus Daten lesen, aber diese müssen beim Verteilen der Dokumente mitgeliefert werden. Beide bieten außerdem keine Unterstützungen innerhalb des Editors, wie z. B. das Markieren von Fehlern direkt im Code oder eine Konsole für Ein- und Ausgaben. Es sind zwar Eingaben möglich, jedoch werden diese über ein einfaches Textfeld realisiert, welches nicht die eigentliche Funktionsweise widerspiegelt. Ein weiterer wichtiger Aspekt, der in Kapitel 5.3 identifiziert wurde, ist die Anpassung des Unterrichts auf Basis des studentischen Fortschritts bzw. der Arbeitsergebnisse. *Runestone* stellt Lehrenden Statistiken für die Beantwortung von Quizzen und über die automatisch bewerteten Aufgaben zur Verfügung. Darüber hinaus ist kein Zugriff auf häufige Fehler oder andere Informationen zur Einschätzung möglich. Bei Notebooks existiert bisher keine Möglichkeit, diese Verwendung und auftretenden Fehler zu analysieren oder zu erfassen.

Wünschenswert ist ein hybrider Ansatz, welcher die Vorteile der interaktiven E-Books und der einfachen Erstellung und Bearbeitung kombiniert. Besonders die Erstellung der Präsentationen inklusive interaktiver Elemente der *Jupyter Notebooks* ermöglicht die effiziente Umsetzung von *Live Coding* und die Beantwortung von Fragen, ohne zwischen Präsentation und Entwicklungsumgebung wechseln zu müssen. Zugleich können Anmerkungen und Erweiterungen direkt im Dokument eingefügt werden und somit die Ergebnissicherung nach der Durchführung von Aufgaben gewährleistet werden.

### 6.1.2 Code-Editoren

Im Falle eines kombinierten Ansatzes oder bei der Verwendung einer Lernplattform wie *Moodle* wäre eine unkomplizierte Programmierumgebung wünschenswert. Diese

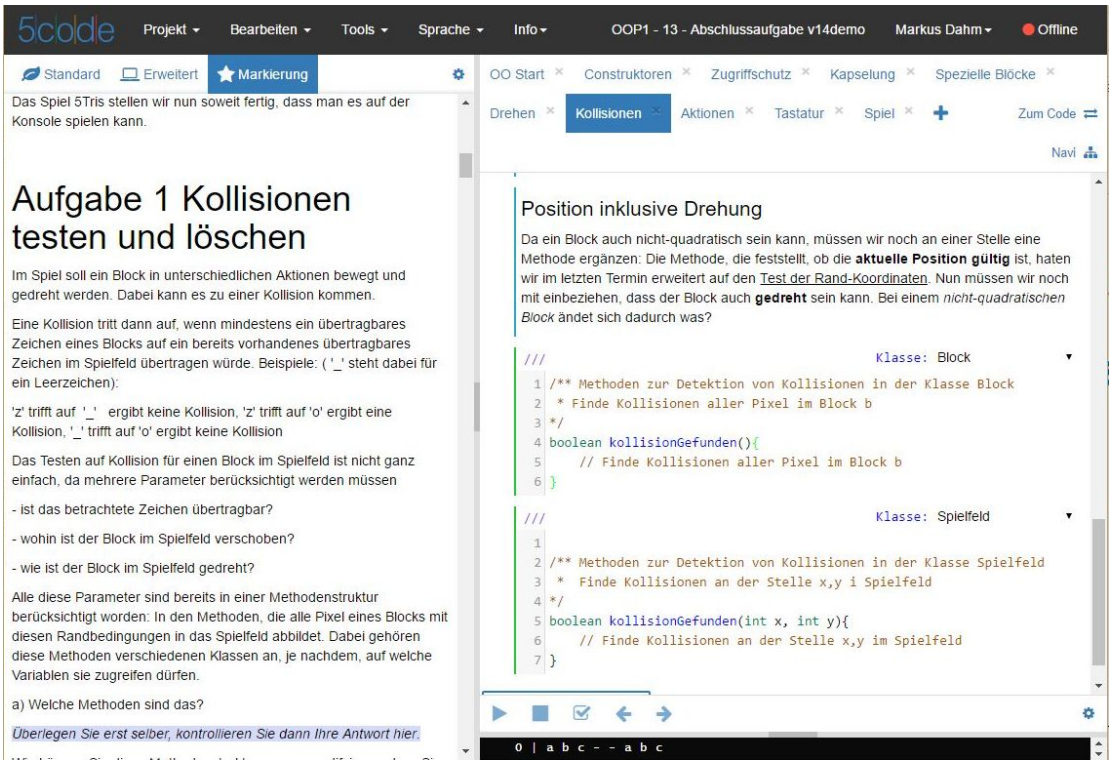



**Abbildung 31:** Webbasierte Oberfläche von 5code

Zielsetzung wurde bereits in mehreren Ansätzen versucht umzusetzen, indem eigene Umgebungen für die Lehre entwickelt wurden. Nachfolgend werden diese Ansätze beschrieben und deren mögliche Verwendung zur Umsetzung des Konzepts untersucht.

Dahm et al. haben eine integrierte Entwicklungsumgebung speziell für Programmieranfänger entwickelt, die das Problemlösen bereits während der Entwicklung unterstützt [Mar15; DBH16]. Im Vordergrund stehen dabei die Schritte *Lesen* > *Verstehen* > *Überlegen* > *Aufschreiben* > *Codieren*, welche den Studierenden als eine Lösungsstrategie vermittelt und durch die Entwicklungsumgebung unterstützt werden. Die Besonderheit dieser Umgebung ist die Verbindung von Aufgabenstellung und Editor wie in Abbildung 31 abgebildet. Bahm et al. argumentieren mittels der *Cognitive Load Theory*, dass zu viele Kontextwechsel zwischen Angabenblättern und professionellen Entwicklungsumgebungen Lernende zu stark belasten. Dementsprechend soll die entwickelte Umgebung diese beiden Elemente zusammenführen und direkt unterstützen. Sie kritisieren darüber hinaus Angebote von *MOOCs*, die zwar Inhalte und deren Anwendung alternierend einsetzen, jedoch ohne das Problemlösen zu thematisieren. Wobei diese Kritik sich eher auf die inhaltliche Seite bezieht und nicht auf die technische Umsetzung, weswegen eine Erweiterung vorhandener Systeme ebenfalls möglich gewesen

wäre. Besonders hinsichtlich dem Wechsel zwischen Skript und der neuen Entwicklungsumgebung, die es ja beim Nachschlagen und der Suche nach Beispielen nach wie vor existieren. In der zweiten Version wurde ein Ansatz ähnlich zu *Jupyter Notebooks* umgesetzt, der Lösungen aus mehreren Text- und Codebausteinen erstellen lässt. Alle Codebausteine werden anschließend in eine Datei überführt und können anschließend ausgeführt werden. Die Umgebung wurde zum reinen Entwickeln von Java-Programmen konzipiert und unterstützt keine weiteren interaktiven Elemente bzw. Visualisierungen. In der Evaluation des Ansatzes wurde sichtbar, dass ungefähr 30% der Studierenden die Möglichkeit zum Markieren von Textstellen und 38% der Studierenden zum Kommentieren genutzt haben. Ein Großteil würde zwar die Verwendung weiterempfehlen, doch fühlten sich einige Studierenden durch technische Einschränkungen gegängelt. Daneben stellt sich die Frage, inwiefern das Kommentieren und Markieren direkt im Aufgabentext beim Lösen hilft. Denn ein gezielter Algorithmenentwurf auf Papier, z. B. mittels Diagrammen oder Pseudocode, ermöglicht eine starke Trennung der Überlegungen in Bezug auf das informatische Problemlösen und nachfolgender Codierung zur Verifikation. Weiterhin muss bei Aufgaben im Umfang und bei den adressierten Lernzielen unterschieden werden. Denn bei kleineren Aufgaben, die nur eine geringe Komplexität aufweisen bzw. ein bestimmtes Konzept einüben, sind nur wenige Notizen notwendig. Zusammengefasst fokussiert dieser Ansatz ebenfalls das Problemlösen und ein schrittweises Vorgehen beim Lösen informatischer Probleme. Die erstellte Oberfläche lässt sich jedoch nur bedingt in Vorlesungen nutzen, da diese nicht für die Verwendung in Vorlesungssälen optimiert ist. Aktuell wird nur die Programmiersprache Java unterstützt, welche in einer nicht näher spezifizierten sicheren Umgebung ausgeführt wird. Das System wurde bisher nicht veröffentlicht, sondern kann nur an der Hochschule Düsseldorf verwendet werden.

*Pythy* ist eine weitere web-basierte Entwicklungsumgebung, die ebenfalls Aufgabenstellungen und einen Editor in einer Umgebung zur Python-Programmierung zur Verfügung stellt [ETA14]. Diese kann ebenfalls direkt in der Vorlesung zum *Live Coding* verwendet werden, was nach Edwards et al. [ETA14]besonders von den Studierenden positiv angenommen wurde. Das System unterstützt auch das automatisierte Bewerten von abgegebenen Lösungen der Studierenden. Ziel dieses Ansatzes ist es, Studierenden eine einfache Programmierumgebung bereitzustellen, ohne die Notwendigkeit zur Installation von Werkzeugketten oder Entwicklungsumgebungen. Aufgrund technischer Probleme und einer anderen Fokussierung wird das Projekt nicht weiterentwickelt. Ebenso existiert keine Unterstützung für Visualisierungen wie z. B. *Turtle Graphics*.

Im professionellen Umfeld haben sich in den letzten Jahren komplette web-basierte Entwicklungsumgebungen etabliert. Eine der bekanntesten ist *Cloud9*[25], die komplet-

[25] Siehe https://c9.io/.

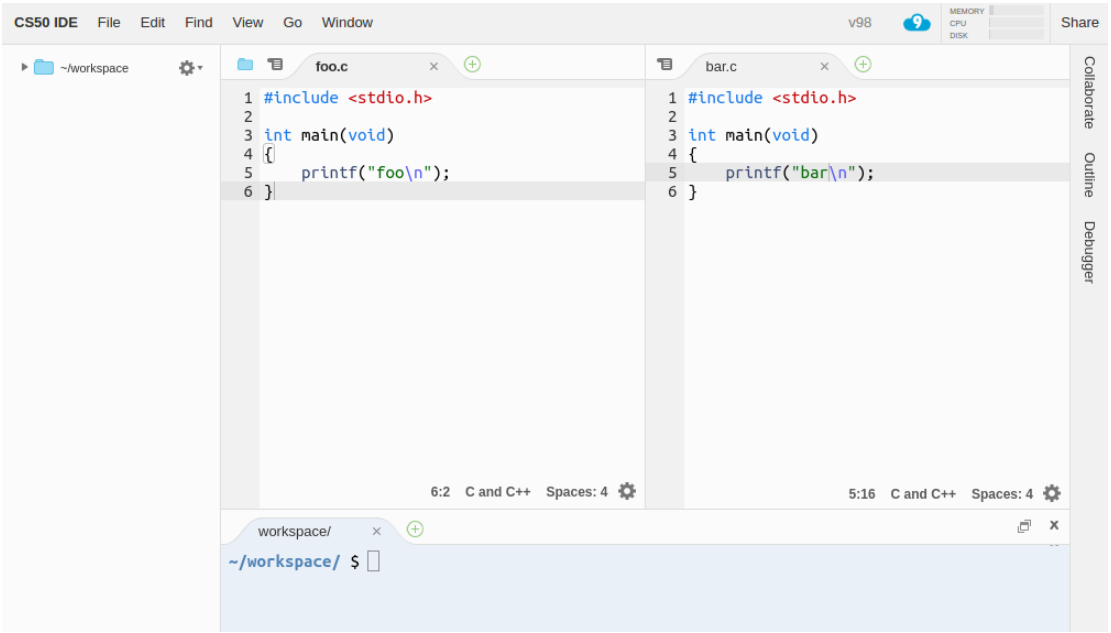



**Abbildung 32**: CS50 IDE

te Arbeitsumgebungen zur Verfügung stellt. Die Entwicklung findet dabei nicht lokal auf dem eigenen Computer, sondern auf einem Server über eine web-basierte Oberfläche statt. Vorteil ist der Zugriff auf ein komplettes eigenständiges Betriebssystem, über welches der Benutzer die vollständige Kontrolle besitzt. Diese Idee wurde von David J. Malan aufgegriffen und eine spezielle Variante für einen Programmieranfängerkurs entwickelt (siehe Abbildung 32) [MOA17]. Die Entwicklungsumgebung bietet dabei Zugriff auf Konsolen, intelligente Sprachunterstützung, einen Dateibaum sowie Zugriff auf einen Debugger. Ziel dieser Umstellung war es Studierenden eine einfache Möglichkeit zum Programmieren zur Verfügung zu stellen, sodass diese nicht mehr virtuelle Maschinen oder Entwicklungswerkzeuge installieren müssen [Mal13]. Nachteil dieser Lösung ist die Geschwindigkeit, in der die Entwicklungsumgebung gestartet wird, da im Hintergrund jeweils ein komplettes Betriebssystem pro Benutzer gestartet werden muss. Jedoch stellt *Cloud9* schneller startende Umgebungen gegen Bezahlung zur Verfügung. Interaktivitäten sind auf die Ein- und Ausgabe sowie das Debuggen limitiert, da es sich hierbei um eine eher professionelle Umgebung handelt, deren Oberfläche auf das Notwendigste reduziert ist.

Einen ähnlichen Ansatz verfolgt *Eclipse Che*[26], die ebenfalls auf eine komplette web-basierte Entwicklungsumgebung auf Basis von *Docker*, einem System zur Betriebssystemvirtualisierung auf Basis von Containern, setzen. Ein Container ist dabei eine

[26] Vgl. https://www.eclipse.org/che/

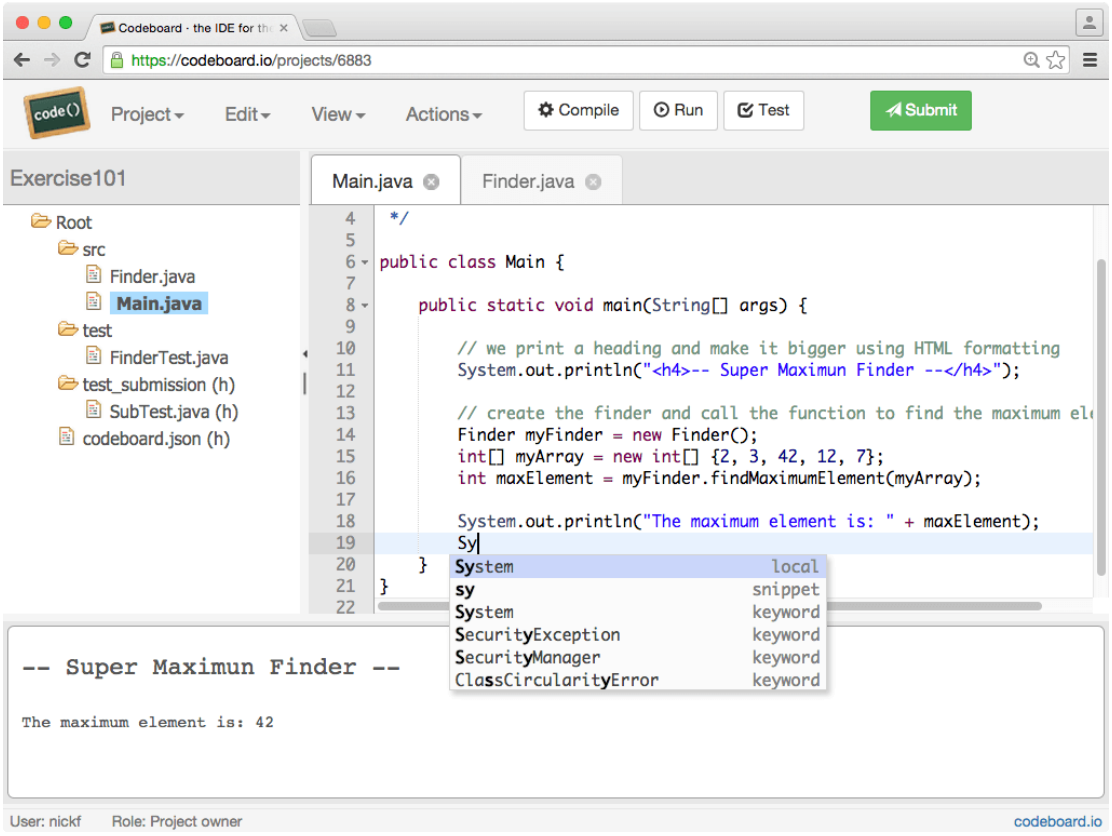



**Abbildung 33**: Codeboard.io Entwicklungsumgebung

isolierte Umgebung, die es erlaubt mehrere unterschiedliche Systeme auf einem Rechner gleichzeitig zu verwenden. Dieser Ansatz leidet unter den gleichen Problemen wie *Cloud9*, da sich die Umgebung primär an professionelle Entwickler richtet und nicht an Programmieranfänger. Dementsprechend ist die Oberfläche ähnlich komplex wie lokal installierte IDEs und bieten keine Möglichkeiten zur Darstellung von Visualisierungen.

An der ETH-Zürich wurde eine weitere web-basierte Entwicklungsumgebung für den Einsatz in der Lehre konzipiert. *Codeboard.io*[27] ermöglicht das Schreiben und Ausführen von Programmen und unterstützt das in Kapitel 6.1.1 angesprochene Einbinden in bereits vorhandene Lernplattformen, wie z. B. *Moodle* oder *MOOC*-Plattformen wie *coursera* oder *edx*. Primäre Anwendung findet diese Umgebung in den *MOOC*-Angeboten der ETH-Zürich und für Programmieraufgaben innerhalb von *Moodle* [PEM14]. Die Umgebung ist nicht zum Präsentieren von Beispielen und *Live Coding* im Unterricht optimiert, jedoch ist die Plattform frei verfügbar und kann an die eigenen Be-

[27] Siehe https://codeboard.io.

dürfnisse angepasst werden.

*Virtual Programming Labs* stellt eine Moodle Erweiterung zur Verfügung mit der sich Programmieraufgaben in Moodle einbetten lassen [RRH12]. Vordergründiges Ziel dieser Erweiterung ist die Erleichterung der Organisation von studentischen Einreichungen und deren automatischen Bewertung. Dazu wurde unter anderem ein Plagiatsprüfer integriert, der kopierte Lösungen erkennt und meldet. Zur Absicherung der Server, welche die studentischen Programme ausführen, wird auf `chroot` zurückgegriffen, welches jedoch leicht zu umgehen ist und nicht dafür für solche Szenarien gedacht ist [Jos13; Joh15, S.16f]. Zusätzlich wurde eine simple Umgebung zur webbasierten Programmierung implementiert, die eine Vielzahl an Sprachen unterstützt. Diese ist aber ebenso wenig für die Verwendung innerhalb von Vorlesungen gedacht wie auch nicht für die Darstellung mittels eines *Beamers* optimiert. Einzig die Möglichkeit ganze Bildschirminhalte mittels *VNC* (Remote Access Software) zu übertragen stellt ein Novum dar, wodurch ebenfalls Programme samt Benutzeroberfläche gestartet bzw. erstellt werden können. Dabei stellt sich die Frage, inwieweit diese Funktion wirklich für Anfängerveranstaltungen verwendet werden kann oder ob diese nicht eine zusätzliche Komplexität erzeugt. *Jupyter Notebooks* und andere Ansätze versuchen eben diese zu reduzieren, indem interaktive Elemente direkt integriert werden und nicht eine komplette professionelle Umgebung verwenden.

### 6.1.3 Fazit zu den Ansätzen

Betrachtet man die vorgestellten Lösungen im Kontext des in Kapitel 5 vorgestellten Konzepts, bieten diese alle keine zufriedenstellenden Funktionen für den Einsatz innerhalb von Vorlesungen zur Verfügung. Tabelle 7 zeigt die Merkmale und Probleme der interaktiven E-Books und der *Jupyter Notebooks*, die entweder gute Möglichkeiten für Aufgaben und Aktivitäten bieten oder andererseits das Erstellen von interaktiven Präsentationen ermöglichen. Es existieren bisher auch Ansätze für das Umsetzen von einzelnen Programmieraufgaben innerhalb von Vorlesungen, jedoch ist ein Großteil der angebotenen Systeme nicht für den Einsatz in Vorlesungen geeignet bzw. wird nicht weiter entwickelt. Komplette web-basierte Entwicklungsumgebungen ermöglichen zwar das Programmieren im Browser und somit ohne weitere Installation von Werkzeugen, aber diese richten sich hauptsächlich an professionelle Entwickler. Zudem lösen diese nicht das Problem Aufgabenstellungen und weitere Angaben einfach mit Studierenden zu teilen und diese ggf. zu aktualisieren. Die meisten verfügbaren Lösungen sind für den Einsatz in Übungsveranstaltungen bzw. für *MOOCs* konzipiert und entsprechend nicht für den Einsatz im Unterricht optimiert. Einzig *Jupyter Notebook* ermöglicht die Kombination von Inhalten und Aufgaben, allerdings macht der datei-basierte Ansatz der Dokumente ein stetiges Aktualisieren aufwendig und kom-

pliziert. Keine der Lösungen ermöglicht das einfache Einreichen der Lösungen, um diese zu diskutieren oder anschließend für alle Lernenden verfügbar zu machen. Einzig *Moodle* als Lernplattform besitzt entsprechende Funktionen, um Aufgaben abzugeben und Inhalte bereitzustellen. Doch *Moodle* ist auf die Bereitstellung von Lerninhalten und Kommunikation in Foren ausgelegt und nicht zur Erstellung interaktiver Programmieraufgaben und Inhalten. Erst Umgebungen wie *Codeboard.io* bieten eine solche Integration an.

Die vorhandenen Ansätze ermöglichen einerseits den Einsatz interaktiver Lernmaterialien oder sind stärker auf die Bearbeitung von Aufgaben im Browser ausgelegt. Beide Seiten sind nur teilweise durch ein didaktisches Konzept, wie z. B. *5Code*, unterlegt, sondern werden zum Codieren verwendet, ohne eine Betonung auf Konzepte oder das informatische Problemlösen. Ziel dieser Arbeit ist jedoch die Umsetzung des in Kapitel 5 entwickelten Konzepts und eine effiziente Umsetzung der Aufgaben. Dies kann keines der genannten Systeme in dem angedachten Umfang bieten, weswegen im Rahmen dieser Arbeit eine neue Plattform entwickelt wurde.

## 6.2 Ziele

Ziel dieser neuen Entwicklung ist interaktive Aufgaben direkt in der Vorlesung zu ermöglichen, zu präsentieren, bearbeiten zu lassen und Lehrende sowie Lernende bestmöglich zu unterstützen. Dabei sollen die Vorteile der interaktiven E-Books und der *Jupyter Notebooks* kombiniert werden, um interaktive Vorlesungsmaterialien und Präsentationen zu erstellen und diese direkt im Unterricht einzusetzen. Lehrende sollen bei der Umsetzung des in Kapitel 5 beschriebenen Konzepts unterstützt werden. Dies betrifft die problembasierte Einführung der Konzepte und Aufgaben bei denen Studierende sofort mitarbeiten können. Ausgaben und Probleme sollen, wenn möglich, durch Visualisierungen unterstützt werden. Im Schritt der *Konzepteinführung* wird *Live Coding* verwendet, bei dem eine optimierte Darstellung in ausreichender Größe essenziell ist. Die daran anschließenden Aufgaben sollen durch die Studierenden bearbeitet und diskutiert werden. Hierfür ist ein Mechanismus zum Einreichen von Lösungen und eine Möglichkeit zur Darstellung für alle Lernenden hilfreich. Programmieren sollte dabei nicht auf das reine Codieren bzw. Problemlösen reduziert werden, da ebenfalls theoretische Grundlagen wichtige Inhalte sind und diese gleichermaßen vermittelt werden müssen. Dementsprechend sollte der Lehrende in der Lage sein, beliebige Aktivitäten und interaktive Elemente in Lernmaterialien einzubetten und diese in einer Präsentation zu nutzen.

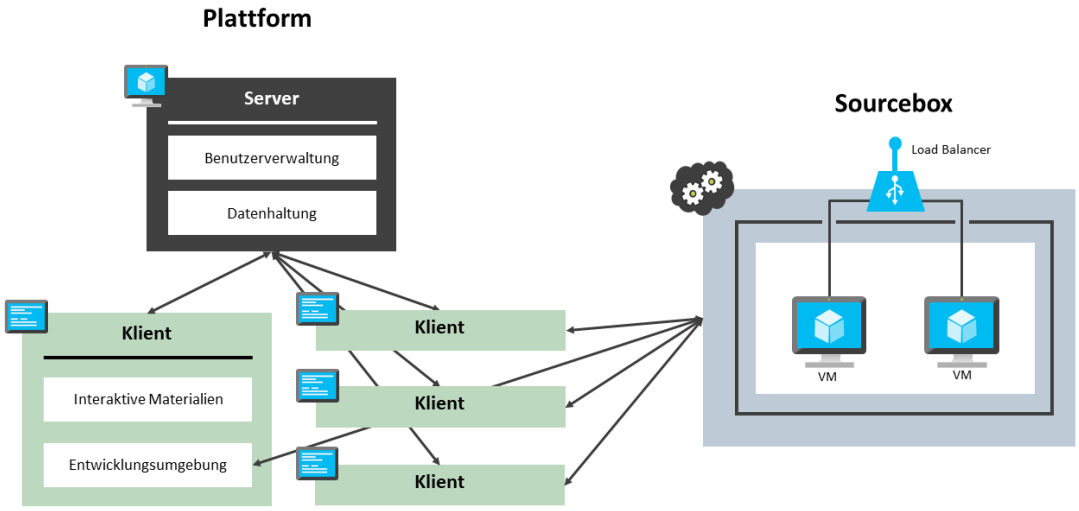



**Abbildung 34**: Aufbau der Plattform für interaktive Vorlesungen

## 6.3 Technische Umsetzung

In diesem Kapitel wird die technische Umsetzung der entwickelten Plattform näher beschrieben. Dabei werden die Techniken zur sicheren Ausführung von fremden Code und Mechanismen zum Einbetten interaktiver Komponenten näher beschrieben. Zuletzt werden einige Indikatoren zum Einschätzen der studentischen Partizipation präsentiert, die durch eine Datenerfassung innerhalb der Plattform möglich sind.

### 6.3.1 Übersicht

Zuerst soll ein grober Überblick auf die Plattform für interaktive Vorlesungen gegeben werden, die aus vier Hauptbestandteilen besteht:

- Eine Umgebung zur sicheren Ausführung von Programmen (bzw. Zugriff auf ein vollwertiges Betriebssystem),
- ein Server zur Verwaltung von Daten und Benutzern,
- eine web-basierte Entwicklungsumgebung
- und interaktive Lernmaterialien, die direkt bearbeitet werden können.

Abbildung 34 zeigt den Aufbau und die Verbindungen zwischen den einzelnen Komponenten, die anschließend näher beschrieben werden. Prinzipiell stellen die isolierten Ausführungsumgebungen (Sandboxes) ein eigenständiges System namens *Sourcebox* dar, welches zur weiteren Absicherung hinter einem Lastverteiler (engl. *Load Balancer*) in einem virtuellen Netzwerk betrieben werden sollte. Dadurch kann man erstens auf höhere Lasten reagieren, indem weitere Instanzen des Sandbox-Servers erstellt werden und zweitens können diese nicht auf weitere Systeme zugreifen. Einzig die Klienten kommunizieren mittels einer Entwicklungsumgebung über eine bidirektionale Verbindung mit den jeweiligen Sandbox-Umgebungen. Alle Daten, die Benutzerverwaltung sowie Berechtigungen werden von einem eigenständigen Server verwaltet, der nicht mit den Sandbox-Servern kommuniziert. Dieser liefert die web-basierten interaktiven Lernmaterialien (und Präsentationen) und die Entwicklungsumgebung zum Programmieren und Ausführen von Programmen an den Klienten aus.

### 6.3.2 Sichere Ausführung von fremden Code

#### Ansätze zur sicheren Ausführung

Alle vorgestellten web-basierten Plattformen und Systeme sind in der Lage fremden Code bzw. Programme auszuführen. Im Bereich der IT-Sicherheit stellen Fehler, die das Ausführen von beliebigem Code auf einem (entfernten) System ermöglichen (engl. *Arbitrary Remote Code Execution*), eine der fatalsten Sicherheitslücken dar. Durch solche Schwachstellen können Angreifer vollständig ein System übernehmen und missbrauchen. Aus diesem Grund ist die sichere Ausführung von fremden Code, in mehreren Aspekten ein wichtiges Thema bei solchen Lernplattformen. Denn nicht nur Angreifer, sondern bereits Studierende können durch Fehler ggf. die Ressourcen eines Systems aufbrauchen oder dieses gar zum Absturz bringen.

Die interaktiven E-Books auf Basis von *Runstone* und andere bereits vorgestellte Ansätze umgehen dieses Problem, indem sie die Ausführung auf den jeweiligen Klienten verlagern. Der bereits erwähnte JavaScript-Interpreter *skulpt* stellt zum Beispiel eine solche Umgebung zur Verfügung, die eine Untermenge der Programmiersprache Python im Browser ausführen kann. Weiterhin können durch den browser-basierten Ansatz interaktive Visualisierungen, wie z. B. *Turtle Graphics* umgesetzt werden. Einen Nachteil stellen die Geschwindigkeit und die mangelnde Unterstützung für eine Vielzahl an Bibliotheken dar, mit denen z. B. statische Auswertungen und Diagramme erstellt werden können. Für andere Sprachen existieren bisher kaum vergleichbare Ansätze, die einen ähnlichen Funktionsumfang bieten. Einzig der von Derrell Lipman implementierte web-basierte *C*-Parser und Compiler, bieten eine ähnliche Umgebung [Lip14]. Dieses wurde seit ungefähr 2 Jahren nicht weiterentwickelt und ist eng mit

einer web-basierten Umgebung verbunden, sodass dieses System nur schwer eigenständig zu verwenden ist[28]. In Hinsicht auf die Sicherheit sind diese Systeme perfekt für Programmieranfänger, jedoch ist die Pflege solcher Ansätze schwierig und meist bieten diese nur eine eingeschränkte Sprachunterstützung.

Ein anderer Ansatz stellt die Ausführung von fremden Code in einer sogenannten *Sandbox* dar, die eine sichere und isolierte Umgebung für Programme bereitstellt. Dabei werden im Wesentlichen zwei Ziele verfolgt: Die Limitation von Ressourcen und die Isolierung von Ressourcen und Systemen. Im Bereich der Lehre existieren hier mehrere Ansätze, die primär für automatisierte Überprüfungen von studentischen Lösungen dienen. Diese werden in der Literatur als sogenannte *Autograder* bezeichnet. Eines der bekanntesten Vertreter dieser Systeme ist das an der Universität Duisburg-Essen entwickelte *Jack* [SG09]. Vorderstes Ziel war zum Zeitpunkt der Entwicklung die Einsparung von Personal bzw. Reduzierung des Personalaufwands, da Programmieraufgaben nun automatisch überprüft und Feedback generiert werden konnte. Mittlerweile wird das System auch für elektronische Prüfungen verwendet. Ein ähnliches System namens *Praktomat* wird an der Universität Siegen entwickelt und soll die Bewertung und Rückmeldungen in Praktika erleichtern [BHS16]. An der Universität Helsinki wurde *Test My Code* entwickelt, welches ebenfalls die automatisierte Bewertung von Aufgaben ermöglicht und zugleich eine Einbindung in die Entwicklungsumgebung *NetBeans* liefert, sodass die Abgabe in dieser möglich ist [Pär+13; Vih+13b]. Weiterhin existieren noch eine Vielzahl weiterer Systeme, die hier nicht weiter aufgezählt werden, da diese alle einen ähnlichen Funktionsumfang bieten. All diese Systeme haben gemein, dass Studierende keinen Einfluss auf die Ein- und Ausgaben haben, sondern die Programme, ohne weitere Kommunikation zum Klienten ausgeführt werden. Mittels funktionaler Tests und statischer Überprüfungen werden anschließend zu den Lösungen Anmerkungen generiert. Somit eignen sich diese nicht für den Einsatz direkt in Vorlesungen, da die Lösungen im Gegensatz zu den interaktiven E-Books oder *Jupyter Notebooks* in einer normalen Entwicklungsumgebung umgesetzt und anschließend eingereicht werden müssen. Alle zu überprüfenden Programme werden von einem System in eine Warteschlange eingereiht und ggf. nicht sofort ausgeführt, sodass keine Interaktionen im Sinne von Ein- und Ausgaben durch den Benutzer möglich sind.

Plattformen wie *Codeboard.io* oder die vorgestellten web-basierten Editoren nutzen einen Ansatz auf Basis von Containern. Darunter fällt auch der *Jupyter Hub*, der für jeden Benutzer einen eigenen Notebook-Server startet bzw. einen Container zur Verfügung stellt. Gleichzeitig ermöglichen diese Ansätze die Ein- und Ausgabe durch den Benutzer während das Programm ausgeführt wird, jedoch keine Interaktionen und Visualisierungen wie beispielsweise mit *skulpt*.

[28] Siehe https://github.com/derrell/LearnCS für mehr Informationen.

### Sourcebox - Kurzlebige Sandbox-Umgebungen

Aufgrund dieser Ausgangslage wurden einige Sandboxing-Ansätze in Bezug auf die Sicherheit und Interaktivität evaluiert. Gleichzeitig sollen diese direkt in Lernmaterialien und Präsentationen eingebunden werden, sodass eine schnelle und unkomplizierte Ausführung von Programmen möglich ist. Prinzipiell eignen sich virtuelle Maschinen, wie z. B. VirtualBox, Xen oder VMware, um isolierte Umgebungen zu realisieren. Diese abstrahieren die Hardware eines Systems und erlauben dadurch die Ausführung von Gastbetriebssystemen, die vom eigentlichen (Host-) Betriebssystem getrennt sind. Ein Nachteil solcher Lösungen ist die Bereitstellung eines kompletten Betriebssystems, da dies meist mit einem großen Aufwand verbunden ist und somit einen nicht unwesentlichen Anteil der Systemressourcen benötigt. Möchte ein Benutzer ein Programm innerhalb einer Präsentation oder in einem Skript ausführen, müsste zuerst dieses System zur Verfügung gestellt werden, welches unter Umständen mehrere Minuten (je nach Infrastruktur und verfügbaren Ressourcen) benötigt. Im Gegensatz dazu können browser-basierte Lösungen innerhalb von Sekunden bereits Programme ausführen, weswegen eine ähnliche Geschwindigkeit in Verbindung mit einem kompletten Sprachumfang wünschenswert wäre. Ziel ist es dabei mit wenig Ressourcen vielen Anwendern die Möglichkeit zur Ausführung von Programmen bzw. eine Umgebung zur Verfügung zu stellen, die eine Vielzahl an Programmiersprachen sowie Interaktivität unterstützt.

Im Rahmen einer Masterarbeit wurde eine neue Sandbox-Lösung namens *Sourcebox* auf Basis von *LXC* (Linux Containern) entwickelt, die kurzlebige und schnell startende Sandbox-Umgebungen zum Ziel hatte [Joh15]. Diese implementiert die von mir für das Konzept vorgesehene sichere Ausführung von kleinen Programmen in schnell startenden Sandbox-Umgebungen. Während der Umsetzung konnte ebenfalls der von mir angedachte Mechanismus für Interaktionen zwischen dem Benutzer und einem laufenden Programm umgesetzt werden. Die Interaktionen selbst wurden anschließend von mir auf Basis dieser Technik umgesetzt und in das interaktive Vorlesungsmanuskript integriert. Bisherige Ansätze zur Absicherung, wie z. B. *ptrace* oder *Seccomp*, konnten nachweislich ausgeschlossen werden, da entweder Sicherheitslücken existieren oder diese die Programmausführung unter starken Einschränkungen ermöglichen. Container teilen sich im Gegensatz zu virtuellen Maschinen den Kernel mit dem Hostsystem, welches den Vorteil hat, dass Container so gut wie keine zusätzlichen Ressourcen verwenden. In einem Container ausgeführte Programme sind dabei lediglich weitere Prozesse des Hostsystems. Somit können Programme nicht nur getrennt von anderen ausgeführt werden, sondern ermöglichen zugleich die Bereitstellung eines kompletten Linux-Systems in einer isolierten Umgebung.

Dies löst jedoch nur einen Teil der Anforderungen zur sicheren Ausführung von Pro-

**Tabelle 8**: Linux-Kernel Namespaces

| Namespace | Beschreibung |
|---|---|
| Mount | Isoliert Mount-Punkte im Dateisysteme, wodurch pro Container eine unterschiedliche Sicht auf die Dateisystem-Hierarchie haben. |
| Network | Isoliert den kompletten Netzwerk-Stack eines Systems bzw. ermöglicht die Steuerung des Zugriffs. |
| UTS | Isoliert den Hostnamen und den NIS Domainnamen des Betriebssystems, sodass Änderungen keine Auswirkungen auf das Host-System haben. |
| IPC | Isoliert die Kommunikation zwischen Prozessen und Threads (Inter-Process Communication). |
| PID | Isoliert bzw. kontrolliert die Sichtbarkeit von Prozessen zwischen mehreren Containern und dem Host-System. |
| User | Isoliert die Benutzer bzw. deren Rechte von anderen Containern. |

grammen, da nach wie vor durch den Zugriff auf das Netzwerk, Festplatten oder übermäßige Nutzung der CPU das System zum Absturz gebracht bzw. missbraucht werden kann. Diese können nun auch durch *LXC* unterbunden werden, da zu dem Zeitpunkt der Erstellung der Masterarbeit und Evaluation neue vom Linux-Kernel bereitgestellten Funktionen zur Ressourcen-Isolation durch *Namespaces* und der Ressourcen-Limitation durch *Control Groups* (kurz *cgroups*) verfügbar wurden. Mittlerweile nutzen auch andere Projekte wie z. B. *Codeboard.io* bzw. *Jupyter Hub* diese Funktionen zur besseren Isolierung und Limitation von Ressourcen, was jedoch zum Zeitpunkt der Konzeption nicht der Fall gewesen ist. Nachfolgend werden die für die Sandbox verwendeten Funktionen zur Isolierung und Limitation von Ressourcen kurz beschrieben.

**Namespaces** *Namespaces* (Namenräume) wurden explizit für die Isolation von Containern entwickelt. Ein Namensraum ermöglicht das Verstecken einer globalen Systemressource hinter einer Abstraktion und lässt, die dem Namensraum angehörigen Prozesse in dem Glauben, dass sie die alleinigen Nutzer der jeweiligen Ressource sind [Joh15, S.15ff]. Änderungen einer isolierten Ressource wirken sich ausschließlich auf den jeweiligen Namensraum aus und sind nur von den Prozessen im selben Namensraum sichtbar. Derzeit werden sechs verschiedene Namensräume im Linux-Kernel zur Isolation von Systemressourcen unterstützt, die in Tabelle 8 beschrieben sind. Durch diese Mechanismen können die Systeme bereits sehr gut voneinander isoliert werden, dennoch können innerhalb von Containern Ressourcen noch unbegrenzt verwendet werden.

**Tabelle 9**: Linux-Kernel cgroups

| cgroup | Beschreibung |
|---|---|
| CPU | Mittels des *CFS Bandwidth Controllers* kann die absolute Rechenzeit einer Prozessgruppe innerhalb einer gewissen Zeitperiode festgelegt werden. Somit können Containern z. B. 10% der Systemressourcen zur Verfügung gestellt werden. |
| Memory | Durch den *Memory-Controller* kann der maximal belegbare Arbeitsspeicher festgelegt werden. Zusätzlich kann die Swap-Partition für ausgelagerte Speicherseiten begrenzt werden. |
| PID | Der *PID-Controller* ermöglicht die Limitation von Prozessen und Threads in einem Container, sodass keine *fork*-Bomben, durch rekursive Erstellung neuer Prozesse, das System zum Absturz bringen. |

**Limitation von Speicherressourcen** Damit mehrere Benutzer gleichzeitig auf einem System Programme ausführen können, sollen einzelne Benutzer nicht die vollständigen Systemressourcen aufbrauchen. Dies ist durch die sogenannten *cgroups* möglich, welche Prozesse in eine hierarchische Baumstruktur einordnen und dadurch eine Ressourcenlimitation durch *cgroup*-Subsysteme ermöglichen [Joh15, S.22ff]. In Tabelle 9 werden die drei wichtigen *cgroups* bzw. *Controller* zum Limitieren der Systemressourcen vorgestellt.

**Effiziente Festplattenlimitation** Durch den Einsatz von *cgroups* lassen sich bereits fast alle relevanten Systemressourcen limitieren, sodass mehreren Benutzern ein Container zum Ausführen von Programmen zur Verfügung gestellt werden kann. Einzig die Limitation von physischen Speicher (Festplatte) wird nicht abgedeckt, weswegen ein spezielles Dateisystem für die Sandbox-Umgebungen verwendet wird. *Btrfs*[29] (B-Tree File System) ist ein junges Dateisystem, welches Funktionen zur Limitation des Festplattenspeichers und eine Reduzierung der Ressourcen ermöglicht. Ziel ist es jedem Benutzer eine eigene isolierte Umgebung zur Ausführung von Programmen bereitzustellen. Container verwenden zwar die Ressourcen des Hostsystems, stellen jedoch in jedem Container ein komplettes Betriebssystem zur Verfügung, wodurch bei mehreren Benutzern viel physischer Speicher benötigt wird. *Btrfs* verwendet einen sogenannte Copy-On-Write Ansatz, um Dateien zu klonen. Ein Dateiklon ist eine Kopie einer Datei, die sich den Speicherplatz mit dem Original teilt, weswegen bis auf Metadaten keine zusätzlichen Daten angelegt werden. Beim Bearbeiten des Originals oder der Kopie werden nur die jeweiligen Änderungen geschrieben, wodurch nur je-

[29] Vgl. https://btrfs.wiki.kernel.org

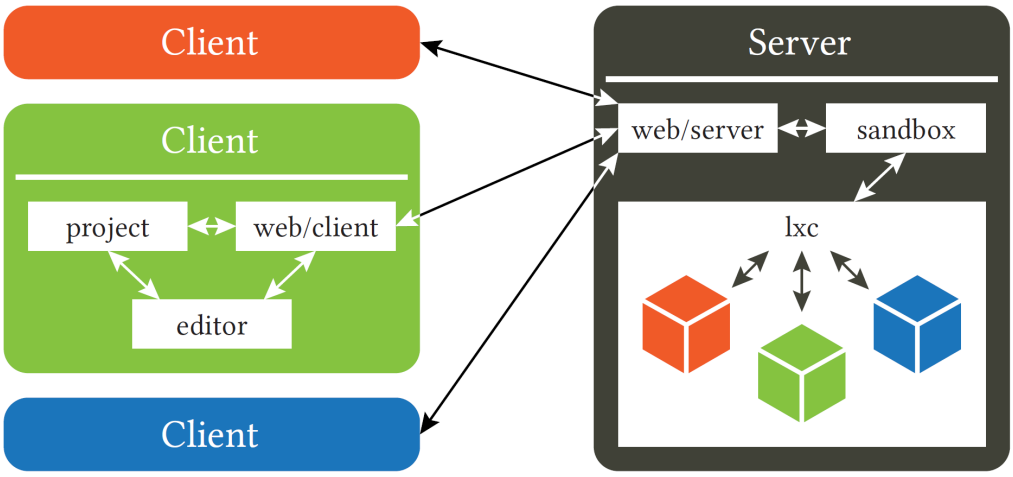



**Abbildung 35:** Aufbau der Sourcebox-Architektur einschließlich der Kommunikation der Module untereinander

weils die Differenz zwischen den beiden Dateien gespeichert wird. Diese effiziente Speicherung ist ebenfalls auf Ebene von sogenannten *Subvolumes*, einer eigenständige eingebundenen Dateisystemhierarchie, möglich. In Kombination mit *Subvolume*-Abbildungen (*Snapshots*) kann ein kompletter Dateisystembaum in Sekundenbruchteilen kopiert werden, ohne zusätzlichen Speicherplatz zu belegen [Joh15, S.37].

*LXC* unterstützt bei der Erstellung von Containern die Verwendung von *Subvolumes* und *Snaphots*. Ausgehend von einem vorkonfigurierten Container, können für jeden Benutzer eigene Container mittels *Snapshots* in kürzester Zeit erstellt werden, ohne weiteren Speicherplatz zu verbrauchen. Weiterhin können für jedes *Subvolume* Quotas definiert werden, welche den Speicherplatz, den ein *Subvolume* belegen darf, einschränken. Dadurch können auf einem einzigen Server, abhängig von den Ressourcen, eine Vielzahl an Containern bereitgestellt werden.

### Sourcebox-Architektur

Um diese Sandbox-Umgebungen nutzbar zu machen und den Benutzern Ein- und Ausgaben, sowie weitere Visualisierungen wie *Turtle Graphics* oder Diagramme zu ermöglichen, wurde in Rahmen der Masterarbeit ein Web-Service entwickelt, der Benutzer automatisch solche Sandbox-Umgebungen zur Verfügung stellen kann. Abbildung 35 stellt den Aufbau der Sourcebox-Architektur dar. In der Masterarbeit wurde ein Web-Service (schwarzer Kasten) entwickelt, der für Benutzer Sandbox-Umgebungen zur Verfügung stellt [Joh15]. Gleichzeitig wurde ein *Remote Stream*-Protokoll umgesetzt,

welches die voll-duplex Kommunikation über beliebige *Streams*, wie z. B. `stdin` (Eingabe) oder `stdout` (Ausgabe), mittels *Web Sockets* an einen web-basierten Klienten realisiert. Der Klient kann somit Programme ausführen und während der Ausführung Ein- und Ausgaben tätigen. Zusätzlich können noch weitere *Streams* erzeugt werden, um weitere interaktive Elemente (siehe Kapitel 6.3.4) zu implementieren. Auf die Beschreibung der konkreten Implementierung wird in dieser Arbeit verzichtet, da diese nur in Teilen von mir erfolgte und ansonsten in Rahmen der Masterarbeit von Johannes Henninger umgesetzt wurde.

Zusammengefasst wurde ein eigenständiges und äußerst effizientes Sandboxing-Verfahren entwickelt, welches Benutzern nicht nur die Ausführung von einzelnen Programmen ermöglicht, sondern jedem ein komplettes isoliertes Betriebssystem zur Verfügung stellt. Somit können vom Benutzer beliebige Prozesse gestartet und mit diesen über *Streams* kommuniziert werden. Dieses kann je nach Anforderungen der Lehrveranstaltung konfiguriert und notwendige Pakete und Programme können installiert werden. Den Vorteil gegenüber anderen Lösungen stellt die sichere Isolierung und Limitation der Ressourcen dar, sowie das Erstellen neuer Umgebungen in weniger als einer Sekunde. Im Vergleich zu den rein web-basierten Lösungen wie *Runestone*, ist kein Unterschied in der Geschwindigkeit, bis ein Programm ausgeführt wird, sichtbar. Jedoch können in dem neuen Ansatz jede Bibliothek und sogar Kommandozeilenfenster verwendet werden.

### 6.3.3 Web-basierte Entwicklungsumgebung

Das eigentliche Ziel war jedoch nicht die Schaffung einer neuen Sandbox-Umgebung, sondern die Unterstützung von Programmieraufgaben direkt im Unterricht. Die beschriebene Sandbox-Umgebung stellt zuerst die technologische Grundlage für die Umsetzung der interaktiven Inhalte bzw. der Programmieraufgaben dar. Aus diesem Grund wurde ein web-basierte Umgebung konzipiert, die die in Kapitel 6.3.2 beschriebene Sandbox-Umgebung zur Ausführung von Programmen verwendet. Die Entwicklungsumgebung ist dabei ein Teil einer größeren Plattform, um web-basierte Lernmaterialien und Präsentation zur Verfügung zu stellen. In Abbildung 35 kommunizieren mehrere Klienten mit dem *Sourcebox-Server*, die jedoch nicht von diesem ausgeliefert werden. Dazu wurde eine Webanwendung entwickelt, welche die Entwicklungsumgebung ausliefert und die Datenhaltung sowie eine Benutzerverwaltung implementiert. Somit stellt der *Sourcebox-Server* nur einen *Microservice* dar, der je nach Auslastung und Infrastruktur skaliert werden kann.

Zugleich vereint die Entwicklungsumgebung Funktionen zur Verwendung im Unterricht und innerhalb interaktiver Lernmaterialien. Im Folgenden werden diese Funktionen und Besonderheiten in Bezug auf die Verwendung näher beschrieben.

### Aufbau

Im Gegensatz zu *Jupyter Notebooks* und *Runestone* ähnelt die entwickelte Umgebung dem Aufbau traditioneller Entwicklungsumgebungen (vgl. Abbildung 36). Trotzdem wurde diese sehr einfach gehalten, um den Fokus auf das Programmieren zu lenken, was bereits durch das Fehlen eines Dateibaums zum Navigieren zwischen Dateien deutlich wird. Prinzipiell ist der Editor in drei Hauptbestandteile gegliedert:

- Die obere Leiste dient zur Navigation zwischen einzelnen Reitern und beinhaltet ein Menü,
- welche in der Mitte angezeigt werden können (auch einzeln)
- und schlussendlich unten in einer Informationsleiste enden.

In der Mitte der Umgebung können mehrere verschiedene Inhaltselemente angezeigt werden. Statt eines Dateibaums werden alle Dateien direkt in der oberen Leiste als Reiter angezeigt. Dies dient dazu, den Fokus auf die Inhalte der Datei und nicht auf eine Projektverwaltung oder Ähnliches zu lenken. Zudem besteht ein Großteil der Beispiele meist aus nur einer oder wenigen Dateien, weswegen diese immer angezeigt werden können. Dateiinhalte werden in Abhängigkeit ihres Typs automatisch farblich hervorgehoben (*Syntax-Highlighting*). Neben Dateien können noch weitere beliebige Inhalte angezeigt werden. Beispielsweise wird in Abbildung 36 in der Mitte (B) eine Konsole dargestellt, in der die Eingaben vom Benutzer eingelesen und die Ausgaben ausgegeben werden. Rechts daneben ist eine grafische Ausgabe (C) zu sehen, die in diesem Fall *Turtle Graphics* zeigt. Darüber (D) ist das Menü angebracht, welches Befehle, wie z. B. zum Ausführen/Stoppen von Programmen, beinhaltet. Die Anzeige von Inhaltselementen wird über die Reiter gesteuert, sodass entweder ein einzelnes Element den kompletten Platz in der Mitte einnimmt oder sich diesen mit weiteren teilt. Somit können der Quelltext und die Konsole zur Programmausführung nebeneinander dargestellt werden.

### Datenmodell

Die Dateien und die zu verwendende Programmiersprache zur Ausführung werden durch ein Datenmodell definiert, welches anschließend von der Entwicklungsumgebung zur Darstellung und Interaktion verwendet wird. Eines der Ziele ist es den Studierenden die Möglichkeit zur Bearbeitung von Aufgaben, das Abspeichern dieser und das Einreichen von Lösungen zu bieten. Für die Umsetzung wurde ein Datenmodell zur Speicherung einer Aufgabe bzw. eines Projektes konzipiert, welches zugleich die Speicherung von studentischen Lösungen unterstützt. Abbildung 37 zeigt den vereinfachten Aufbau des Modells an. Jedes Projekt bzw. Beispiel wird von einem *Besitzer*

erstellt und kann eine beliebige Anzahl an Quelltextdateien umfassen. Die *Sprache*, die zu verwendende *Ausführungsumgebung*, ein *Name* sowie eine *Hauptdatei*, welche zur Ausführung verwendet werden soll, wird in den *Metadaten* definiert. Mittels der *Ausführumgebung* kann zwischen der neuen Sandbox-Umgebung und dem JavaScript-Interpreter für Python (*skulpt*) umgeschaltet werden, sodass die jeweiligen Vorteile zur Visualisierung genutzt werden können. Darüber hinaus können an ein Beispiel weitere Daten, die über *Typ*, *Metadata* (vereinfacht) *Inhalt* verfügen, angehängt (vgl. Entität *Anhang*) werden. Eine Verwendung dieser Möglichkeit wird in den nachfolgenden Kapiteln noch näher beschrieben.

Der *Besitzer* erstellt ein Beispiel und gibt mindestens eine Datei zum Bearbeiten sowie ggf. Quelltext vor und legt die Sprache fest. Studierende können nun auf dieses Beispiel zugreifen, Ausführen und bei Bedarf Änderungen vornehmen. Diese können anschließend gespeichert werden, wodurch für den Studierenden eine eigene *Quelltext*-Relation erstellt wird (vgl. Abbildung 38). Anschließend bekommen Studierende immer wieder diese Version angezeigt, sobald das Beispiel aufgerufen wird. Dennoch ist es möglich über Parameter andere Versionen, wie z. B. die Ursprüngliche oder eines Kommilitonen, in der Umgebung zu laden. Diese können jedoch von dem jeweiligen Studierenden nicht überspeichert werden, sondern es kann nur lesend auf diese zugegriffen werden. Studierende können in ihrer eigenen Version ebenfalls weitere Dateien hinzufügen, jedoch nicht die Programmiersprache oder die Umgebung, in der das Beispiel ausgeführt wird, ändern. Auf Basis dieses Datenmodells konnten einige notwendige Funktionen für den Einsatz innerhalb von Vorlesungen und für interak-

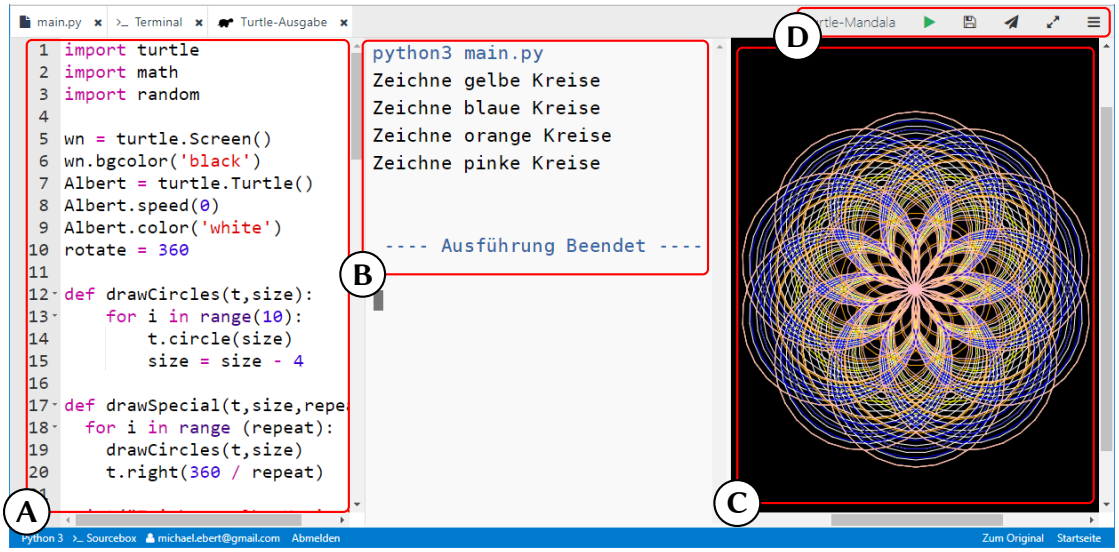



**Abbildung 36:** Web-basierter Editor – Quelltext (A), Konsolenausgabe (B), grafische Ausgabe (C) und Befehlsleiste (D).

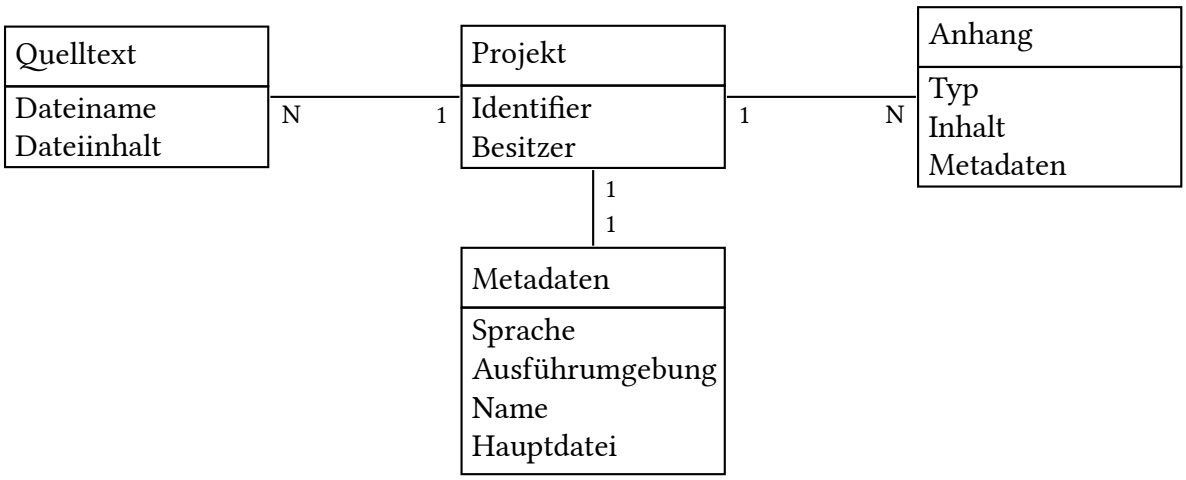



**Abbildung 37**: Vereinfachtes Datenmodell für Beispiele bzw. Aufgaben

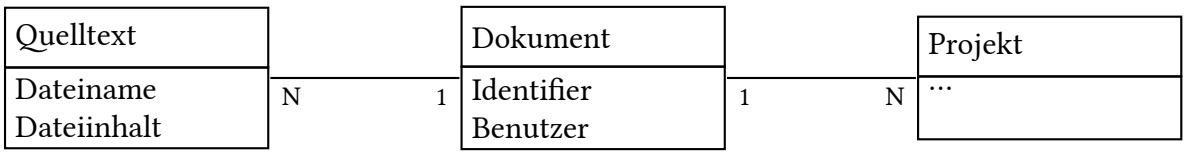



**Abbildung 38**: Vereinfachtes Datenmodell für studentische Version der Beispiele und Aufgaben

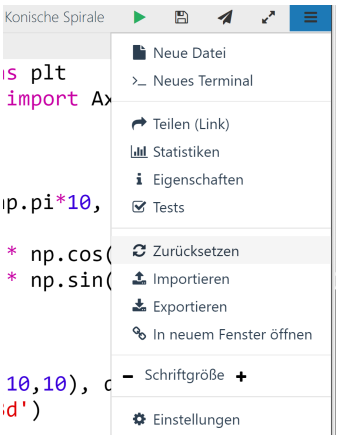



**Abbildung 39**: Menü der web-basierten Entwicklungsumgebung (aus Sicht des Besitzers)

tive Lernmaterialien umgesetzt werden, die im nächsten Abschnitt näher beschrieben werden.

### Befehle und Funktionen

Im folgenden Abschnitt werden die wichtigsten Funktionen für die Verwendung im Unterricht und zur Bearbeitung der Aufgaben beschrieben. Grundlegend können alle Beispiele über die Schaltfläche ▶ ausgeführt bzw. mit ■ gestoppt werden. Bei der Ausführung werden alle Ausgaben in einem Inhaltselemente (vgl. Abbildung 36 (B)) ausgegeben bzw. die Eingaben eingegeben. Die Umgebung verwendet abhängig von der gewählten Sprache (und Version) die richtige Werkzeugkette und schreibt alle Daten in die Sandbox-Umgebung, kompiliert diese ggf. und führt diese anschließend aus.

**Speichern & Zurücksetzen** Über das Symbol 🖫 kann der Besitzer des Beispiels dieses speichern bzw. aktualisieren. Studierende können ebenfalls darüber ihre eigene Version abspeichern, welche Sie bei erneutem Aufruf (von überall) wieder geladen wird. Zusätzlich kann eine veränderte Version auf die ursprüngliche über die Schaltfläche 🔄 *Zurücksetzen* (vgl. Abbildung 39) zurückgesetzt werden. Diese setzt zunächst nur die Daten innerhalb der Umgebung zurück, erst durch das Abspeichern werden diese Daten persistent abgelegt. Studierende haben zudem die Möglichkeit ihre eigene Version mit der ursprünglichen zu vergleichen, indem sie auf die Schaltfläche *Zum Original* in der *Informationsleiste* (siehe unten rechts in Abbildung 36) klicken. Dadurch öffnet sich die ursprüngliche Version in einem neuen Fenster in einem Lesemodus.

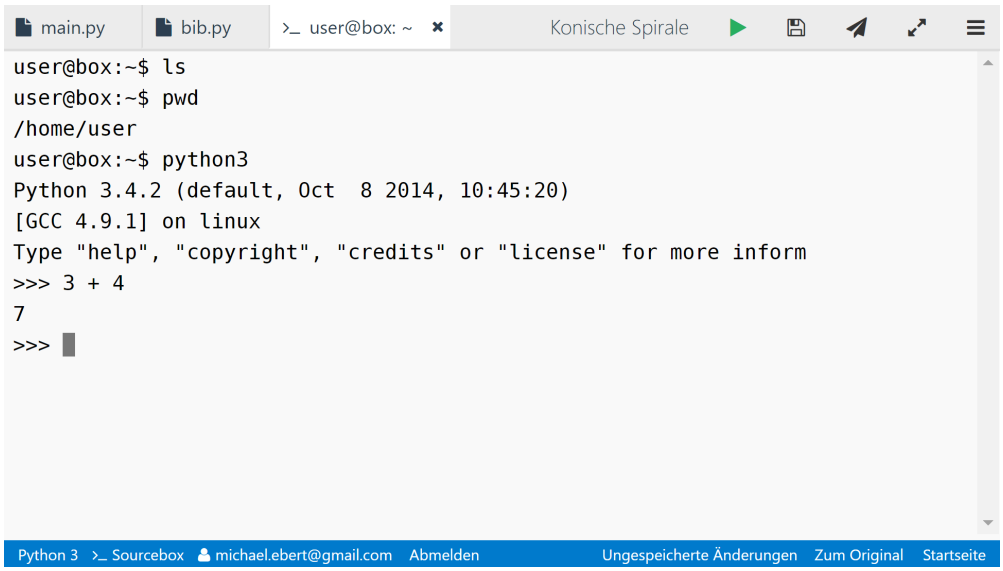



**Abbildung 40**: Terminal in der web-basierten Entwicklungsumgebung

**Import & Export** Lokale Dateien können über die Schaltfläche *Importieren* bzw. über *Drag & Drop* in die Umgebung importiert werden. Somit können alte Lösungen oder bereits lokal erstellte Dateien, beispielsweise vom Lehrenden, direkt verwendet werden. Weiterhin ist es möglich alle Dateien aus der Umgebung zu exportieren, sodass diese lokal weiterverwendet werden können, um z. B. diese in eine traditionellen Entwicklungsumgebung zu laden.

**Dateien** Die Umgebung stellt zusätzlich Funktionen zum Erstellen, Umbenennen und Löschen von Dateien zur Verfügung. Im Gegensatz zu Lösungen wie *Runestone* und *Jupyter Notebooks* können Studierende mit mehreren Dateien, beispielsweise beim Klassenentwurf, oder zum Lesen oder Schreiben von Textdateien, üben bzw. diese ausprobieren. In den meisten Fällen reichen zwar einzelne Dateien, trotzdem ist eine breite Nutzung der Umgebung und umfangreichere Aufgaben intendiert.

**Terminal** Ein weiterer Unterschied zu den bisherigen Ansätzen ist die Möglichkeit zur Nutzung eines vollständigen *Terminals*, um den Umgang mit der Konsole oder z. B. Interpreter zu verwenden. Dies ist den Vorzügen der Sandbox-Umgebung zu verdanken, da diese nicht nur Programme isoliert ausführt, sondern jedem Benutzer ein eigenes Betriebssystem zur Verfügung steht und somit beliebige Prozesse gestartet und die *Streams* mit dem Klienten kommunizieren können. Somit können zu Beginn der Fokus auf die Programmierung gelegt werden und später die Befehle zur Kompilierung und Ausführung zuerst durch die Verwendung des Terminals vermittelt werden.

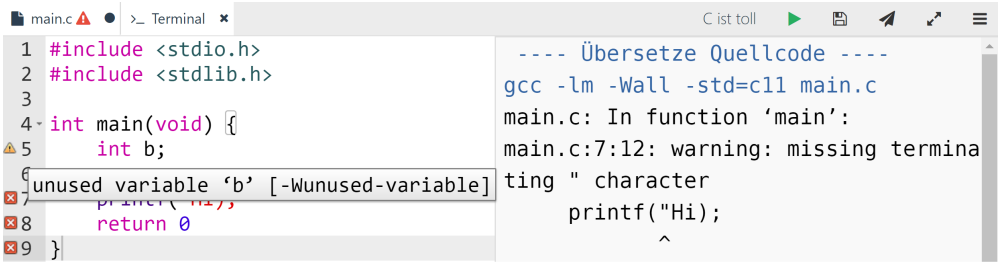



**Abbildung 41**: Anzeige von Fehlern und Warnungen im Editor

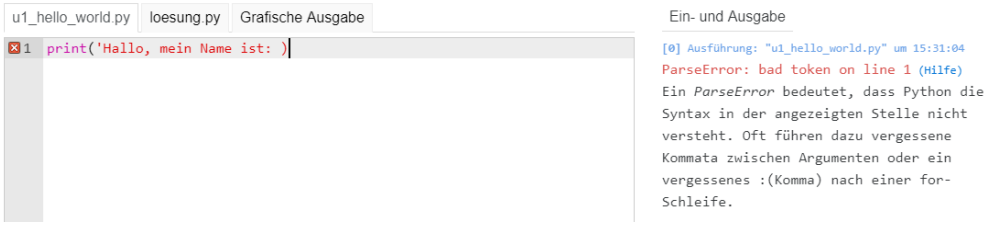



**Abbildung 42**: Anzeige von Hilfetexten bei Fehlern innerhalb einer Entwicklungsumgebung

**Editor** Der Editor zur Bearbeitung der Quelltexte unterstützt im Gegensatz zu den interaktiven E-Books und *Jupyter Notebooks* einige Funktionen, die aus traditionellen Umgebungen bekannt sind. Für die Darstellung des Quelltextes kann ein *Farbschema* ausgewählt werden, um ggf. eine bessere Lesbarkeit auf Bildschirmen mit niedriger Helligkeit zu ermöglichen. Weiterhin können unsichtbare Zeichen, wie z. B. Leerzeichen oder Einrückungen, dargestellt werden. Zusätzlich ist eine sprachabhängige Autovervollständigung integriert, sodass Befehle bei Bedarf vervollständigt werden können.

Im Kontrast zu den anderen Lösungen werden auftretende Fehler beim Kompiliervorgang oder während der Ausführung (bei Interpretern) automatisch erkannt und in den entsprechenden Dateien und Zeilen markiert (vgl. Abbildung 41). Die Erkennung kann für jede Programmiersprache konfiguriert werden, sodass die Anzeige je nach Anforderungen gesteuert werden kann. Darüber hinaus können zu einem späteren Zeitpunkt weitere Unterstützungen für die Studierenden implementiert werden, sodass beispielsweise Fehler automatisch mit Hilfetexten verknüpft oder Tippfehler bei der Verwendung von Variablen sofort im Editor angezeigt werden können. Eine solche Verknüpfung wurde in einem Vortest bereits implementiert und getestet (vgl. Abbildung 42) [MM16]. Weitere Funktionen und Hilfestellungen konnten im Rahmen dieser Arbeit nicht umgesetzt und evaluiert werden, jedoch bietet die Plattform eine Vielzahl an Erweiterungsmöglichkeiten, um diese umzusetzen.


**Abbildung 43:** Ansicht der Eingereichte Lösungen für den Lehrenden

**Einreichen von Lösungen** Um Diskussionen möglicher Lösungen im Unterricht zu unterstützen wurde eine Funktion zum Einreichen von Lösungen integriert, welches die organisatorischen Abläufe effizient umsetzt. Dazu muss der Lehrende zuerst die Abgabe von Lösungen aktivieren, indem er im Menü die Schaltfläche *Statistiken* (vgl. Abbildung 39) anklickt, wodurch sich eine neue Oberfläche zur Steuerung der Funktion öffnet. Studierende können nun über die Schaltfläche in der Befehlsleiste (vgl. Abbildung 36 (D)) das Einreichen einer Lösung initiieren. Anschließend öffnet sich ein Dialog, welcher die Eingabe einer optionalen Nachricht zur Beschreibung der Lösung ermöglicht. Die eingereichten Lösungen werden anschließend sofort an den Lehrenden weitergeleitet und angezeigt. Abbildung 43 zeigt, wie die studentischen Einreichungen innerhalb der Entwicklungsumgebung dargestellt werden. Studierende können ihre Einreichungen jedoch auch überarbeiten und diese erneut abgeben (Revisionen), wodurch die bisherige Lösung ersetzt wird.

Der Lehrende kann nun mittels eines einzigen Klicks auf die Schaltfläche *Anschauen* die jeweiligen Lösungen in einem neuen Fenster öffnen und somit im Unterricht verwenden. Einreichungen werden sofort in der Entwicklungsumgebung angezeigt und aktualisieren sich selbstständig. Dieser Mechanismus ermöglicht einen effizienten Austausch der Lösungen im Unterricht, ohne die Entwicklungsumgebung zu verlassen oder eine andere Plattform wie beispielsweise *Moodle* zu verwenden.

### Optimierte Darstellung

Eine weitere Anforderung ist die optimierte Darstellung der Inhalte mittels eines *Beamers*, sodass der Quelltext und die anderen Inhalte in einem größeren Hörsaal noch lesbar sind. Dazu wurden im Menü zwei Schaltflächen zur Änderung der Schriftgröße integriert, die gleichzeitig den Quelltext, Terminal und Programmausgaben vergrößern bzw. verkleinern. Um die Schriftgrößenänderungen nicht bei jedem Beispiel durchführen zu müssen, werden alle Einstellungen gespeichert und wirken sich auf alle bereits geöffneten und neuen Umgebungen aus.

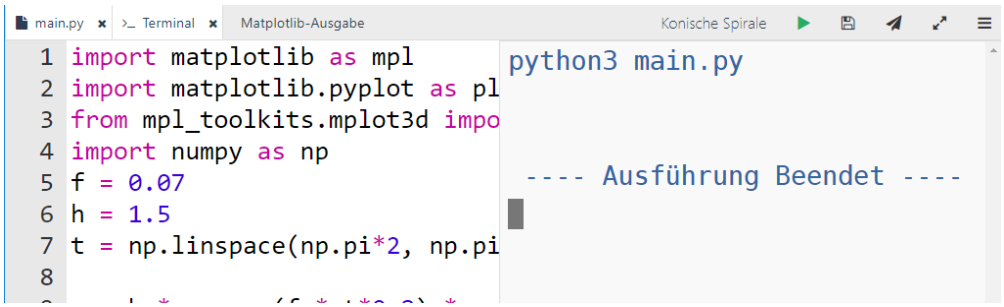



**Abbildung 44**: Vergrößerte Schrift bei einer Fenstergröße von 1024 × 768 Pixeln

Darüber hinaus verwendet die Darstellung ein adaptives Layout, welches die Elemente in Abhängigkeit zu der Auflösung und verfügbaren horizontalen und vertikalen Längen automatisch anpasst. Die Entwicklungsumgebung kann in andere Seiten, Präsentationen und Dokumente eingebettet werden, wodurch die Größe für die Verwendung innerhalb der Vorlesung ggf. zu klein ist. Deswegen wurde ein Vollbildmodus (vgl. Befehlsleiste in Abbildung 36) integriert, welcher die Umgebung bei Bedarf auf die volle Breite und Höhe ausweitet, sodass mehr Inhalte angezeigt werden können.

### 6.3.4 Interaktive Komponenten

Aus dem Konzept in Kapitel 5 wurde die Forderung nach anschaulichen bzw. fachbezogenen Beispielen abgeleitet. Diese dienen einerseits zur besseren Einführung von Problemen und andererseits lassen sich Lösungsversuche visuell besser verifizieren, da beispielsweise bei *Turtle Graphics* eine falsch gezeichnete Linie sofort auffällt. Gleichzeitig können solche Elemente die Motivation der Lernenden positiv beeinflussen, sodass diese sich gerne mit den Lerninhalten auseinandersetzen. Um dieser Forderung gerecht zu werden, wurden mehrere interaktive Elemente für die Verwendung in der Entwicklungsumgebung konzipiert.

#### Turtle

Das Beispiel in Kapitel 5.2.6 zeigt die Verwendung von *Turtle Graphics*, welche unter anderem in den interaktiven E-Books verwendet wird. Jedoch verwenden diese *skulpt* und damit eine reine JavaScript-Umgebung, welche somit nicht in Verbindung mit der entwickelten Sandbox-Umgebung verwendet werden kann. In Kapitel 6.3.2 wurde bereits die Möglichkeit zur Verwendung weiterer bidirektionaler *Streams* erwähnt, welche die Umsetzung von *Turtle Graphics* dennoch ermöglicht. Dabei findet die Ausführung des Programms innerhalb des Containers statt, der anschließend die

Zeichenbefehle an den Klienten schickt. Der Klient kann aber auch Nachrichten an den Container schicken, sodass Maus- und Tastatureingaben vom Programm verarbeitet werden können. Für die eigentliche Implementierung wurde ein ähnlicher frei verfügbarer Ansatz des Hasso Plattner Instituts verwendet und weiterentwickelt [Sta+16]. In der Sandbox-Umgebung wird dazu eine eigene Version der *Turtle Graphics* Bibliothek installiert, welche eine eigene Zeichenumgebung zur Verfügung stellt. Diese nutzt einen zusätzlichen *Stream*, der von der Sandbox-Umgebung zur Verfügung gestellt wird und darüber Nachrichten senden und empfangen kann. Alle Nachrichten werden vom Klienten verarbeitet und mittels JavaScript auf einem Canvas gezeichnet. Leider findet das Zeichnen der Linien teilweise verzögert bzw. nicht in einer einheitlichen Geschwindigkeit statt, was einen Nachteil darstellt. Ausschlaggebend dafür sind die Latenzen, welche aufgrund der ausgelagerten Ausführung und der Kommunikation über *Streams* gezwungenermaßen auftreten.

Die Entwicklungsumgebung unterstützt deshalb zwei verschiedene Ausführungsumgebungen, weswegen es zusätzlich möglich ist bei Bedarf *Turtle Graphics* Beispiele über *skulpt* laufen und anzeigen zu lassen. Somit kann je nach Beispiel die notwendige Unterstützung der gesamten *Turtle Graphics* Bibliothek genutzt oder die Darstellung ohne Unterbrechungen und Verzögerungen verwendet werden.

### Automatische Aufgabenbewertung

Für die interaktiven Lernmaterialien wurde zusätzlich eine automatische Aufgabenbewertung integriert, sodass Studierende die funktionale Korrektheit ihrer Lösung überprüfen können. Lehrende können für ein Beispiel Unit-Tests schreiben (vgl. Abbildung 45), welche anschließend von den Studierenden in der Entwicklungsumgebung ausgeführt werden können. Für die Verwendung der Tests muss die Sandbox-Umgebung entsprechend konfiguriert werden, sodass alle benötigten Bibliotheken zur Ausführung vorhanden sind. Im Falle der Programmiersprache Python wird ein eigens entwickelter *Test Runner* verwendet, der alle Tests ausführt und das Ergebnis im *JSON*-Format über einen weiteren *Stream* an den Klienten schickt, welcher die Ergebnisse anschließend visuell darstellt. Darüber hinaus werden spezielle Annotationen für das Schreiben von Tests bereitgestellt, sodass Testfälle gewichtet und ggf. ein Lösungshinweise ausgegeben, oder die Ausgaben unterdrückt werden. Zusätzlich können Eingaben simuliert und Ausgaben überprüft werden, wenn die Programme noch keine zu testenden Funktionen beinhalten. Die Erstellung der Testfälle findet direkt in der Entwicklungsumgebung statt, wie in Abbildung 45 abgebildet und kann dort ebenfalls getestet werden. Alle Testfälle werden anschließend im Datenmodell als Anhang gespeichert (vgl. Kapitel 6.3.3). Abbildung 46 zeigt die Ergebnisse eines Testdurchlaufs aus Sicht der Studierenden. Durch die Verwendung der Annotationen zum

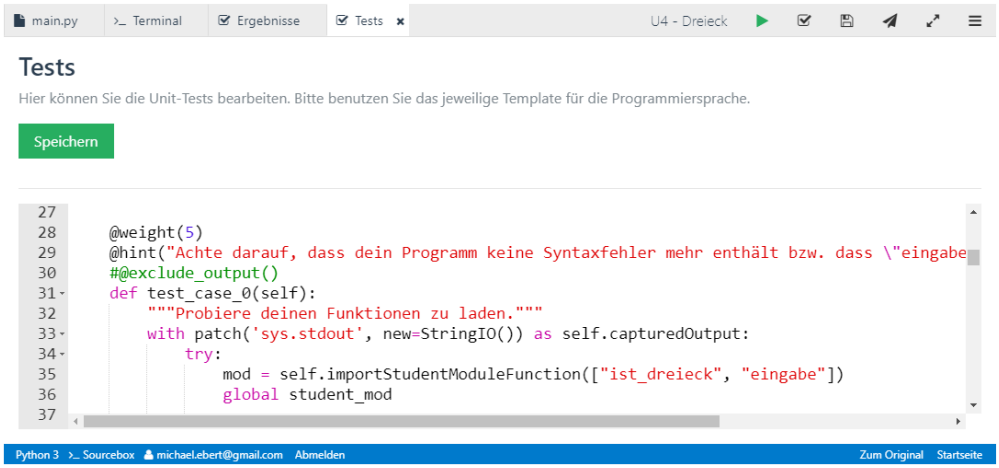



**Abbildung 45**: Erstellung von Testfällen für die automatische Bewertung

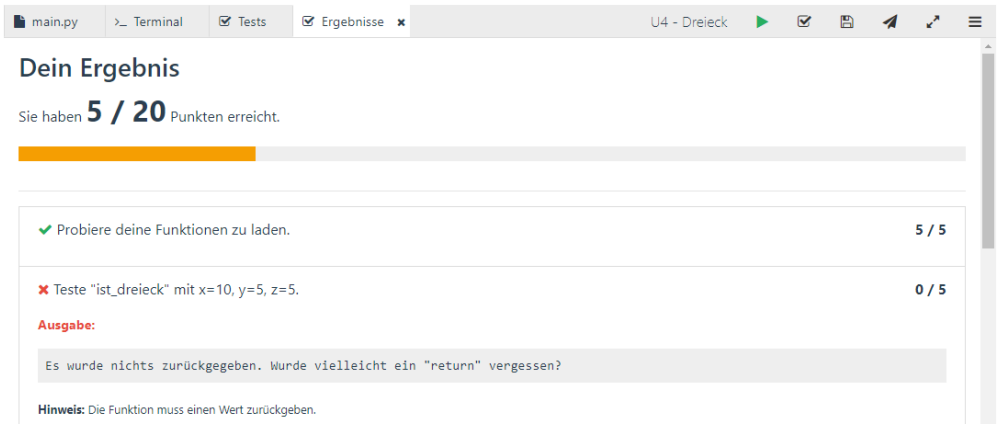



**Abbildung 46**: Ergebnisse der automatischen Bewertung in der Entwicklungsumgebung

Vergeben von Punkten, kann ein Punktestand für den Durchlauf berechnet werden. Fehlgeschlagene Testfälle werden mit der jeweiligen Ausgabe und Nachricht angezeigt.

Eine Verwendung zur Benotung, wie es andere Prüfsysteme durchführen, ist nicht intendiert, sondern das Testen dient als Hilfestellung und Selbsttest für die Studierenden. Durch die mögliche Bearbeitung von Aufgaben von überall aus, können Studierende Übungsaufgaben von zuhause aus erledigen und sich selbst prüfen und nur bei Bedarf in die Übungsstunden kommen. Das Angebot zur betreuten Übungsbearbeitung soll Studierenden einen Rahmen bieten Fragen zu klären und Hilfestellungen zu erhalten. Besonders bei einer heterogenen Studierendenschaft mit unterschiedlichem Vorwissen müssen nicht alle Studierenden anwesend sein. Dies hat für den Dozierenden den Vorteil, dass mehr Zeit für die schwächeren Studierenden zur Verfügung steht und zugleich die stärkeren sich nicht langweilen und die Zeit anderweitig nutzen können. Ein solches Konzept mit der freiwilligen Teilnahme muss sich allerdings in das Gesamtkonzept der Lehrveranstaltung einfügen und Studierenden einen Mehrwert bieten, da ansonsten die Gefahr besteht, dass die Übungsstunden nicht angenommen werden.

### Diagramme

Ein weiteres Thema der Lehrveranstaltung ist die wissenschaftliche Auswertung von Daten und deren Visualisierung in Diagrammen und Plots. Python ermöglicht die Verwendung von *NumPy*[30] und *matplotlib*[31] für numerische Berechnungen und für das Erzeugen von Diagrammen. Besonders *Jupyter Notebooks* ermöglichen eine Darstellung der erzeugten Diagramme direkt in der web-basierten Oberfläche. Eine ähnliche Funktion auf Basis eines zusätzlichen *Streams* wurde ebenfalls in die Entwicklungsumgebung integriert, sodass die erzeugten Bilder direkt angezeigt werden. Abbildung 47 zeigt die Darstellung der erzeugten Diagramme innerhalb der Entwicklungsumgebung. Diese werden automatisch angezeigt, ohne dass der Benutzer noch weitere Interaktionen vornehmen muss. Ziel ist es die Programme in der gleichen Art und Weise wie in lokalen Umgebungen auszuführen, ohne dass Änderungen im Quelltext nötig sind. Dies wird durch eine eigene Implementierung eines sogenannten *Backends* für *matplotlib* erreicht, welches sich um die Darstellung kümmert. Sobald das Diagramm mittels `plt.show` (vgl. Abbildung 47 Zeile 28) aufgerufen wird, erstellt das *Backend* eine Bilddatei und schickt diese an den Klienten, der das Bild wiederum anzeigt.

[30] Siehe http://www.numpy.org/
[31] Siehe https://matplotlib.org/

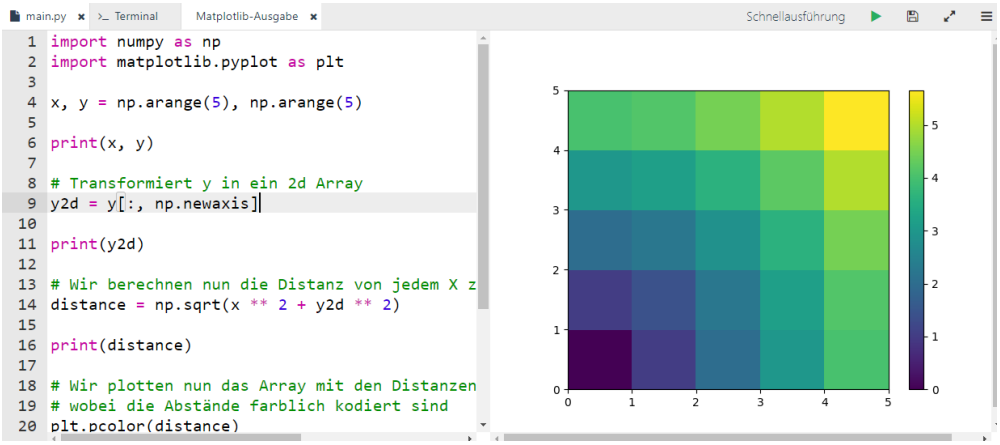



**Abbildung 47**: Darstellung von *matplotlib* Diagrammen in der Entwicklungsumgebung

### Simulationen

Die letzte interaktive Komponente stellt die Simulation von Hardware dar, um authentische und fachbezogene Probleme innerhalb der Vorlesung zu verwenden. In einem früheren Ansatz der Plattform wurde bereits eine web-basierte Simulation von Arduino-Aufbauten (vgl. Abbildung 48) umgesetzt [EH15b]. Diese ermöglicht die Interaktion mit LEDs, Schaltern und anderen Bausteinen, um Algorithmen und Probleme fachbezogen zu vermitteln. Für die Umsetzung wurde *skulpt* verwendet, weswegen der Ansatz nur in Verbindung mit dieser Ausführungsumgebung funktioniert. Ein Nachteil dieser Lösung ist die Verwendung starrer Aufbauten, die nicht verändert werden können, weswegen das Hinzufügen weiterer Aufbauten ein aufwendiger Prozess ist.

Um das Problem der starren Aufbauten zu umgehen und eine breite Verwendung zu ermöglichen wurde in Kooperation mit der *Raspberry Pi Foundation*[32] und *Forkable Inc.*[33] eine Simulation des *Sense Hat* umgesetzt. *Sense Hat* ist ein Modul, welches auf einen *Raspberry Pi* Computer aufgesteckt werden kann und anschließend mittels Python oder C programmiert wird. Abbildung 49 zeigt die Darstellung der Simulation im Browser, welche im Gegensatz zu den Arduino-Aufbauten auch dreidimensional erfolgen kann. Der Aufsatz ermöglicht die Ansteuerung einer $8 \times 8$ LED-RGB-Matrix, eines Joystick und einer inertialen Messeinheit (engl. *IMU*), die drei Sensoren vereint. Dadurch können Programme auf ein Gyroskop, einen Kompass und Beschleunigungssensor zugreifen, um die Position der Hardware zu bestimmen. Außerdem kann

[32] Siehe https://www.raspberrypi.org/
[33] Siehe www.trinket.io

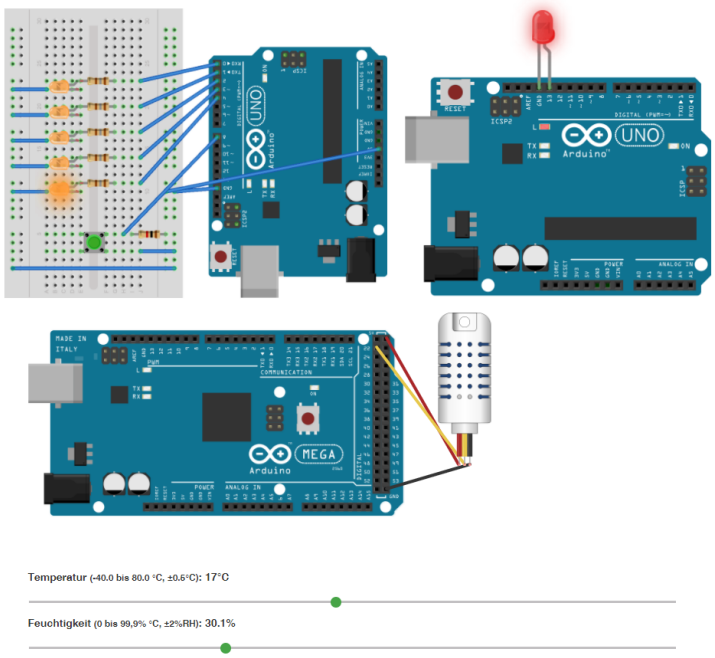



**Abbildung 48**: Beispiele für die web-basierte Simulation von Arduino-Aufbauten

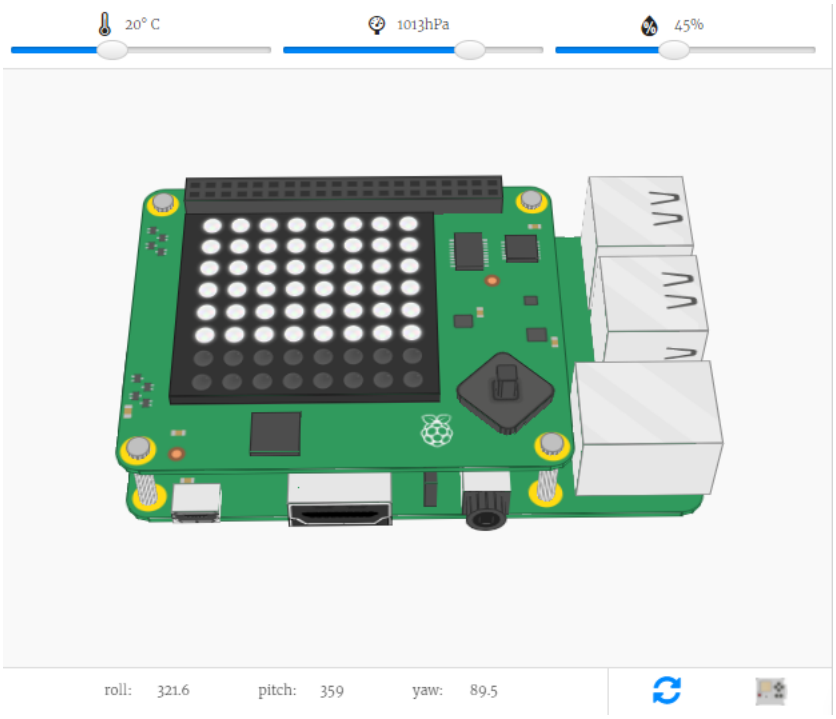



**Abbildung 49**: 3D-Simulation eines *RaspberryPis* mit aufgesetztem *Sense Hat*

die Temperatur, der Luftdruck und die Luftfeuchtigkeit simuliert werden, sodass Programme auf Veränderungen reagieren können. Das ermöglicht das Heranführen an die Programmierung von Hardware und verwandte Algorithmen zur Bestimmung der Orientierung auf Basis der inertialen Messeinheit durch eine sogenannte *Sensor Fusion*, wodurch Fehler bei der Orientierung durch eine Kombination der drei Sensoren reduziert werden. Mittels dieser Simulation können eine Vielzahl an fachbezogenen Aufgabenstellungen realisiert werden, welche Konzepte anhand authentischer Probleme adressieren. Zum Zeitpunkt dieser Arbeit kann die Simulation nur eingebunden werden und noch nicht direkt in der Entwicklungsumgebung genutzt werden.

## 6.4 Interaktive Lernmaterialien

Bei der Diskussion vorhandener Ansätze wurden die Vor- und Nachteile der interaktiven E-Books und der *Jupyter Notebooks* analysiert und endeten in der Forderung nach einem hybriden Ansatz. Aufgabe der Materialien ist die Durchführung des in Kapitel 5 beschriebenen Konzepts, welches interaktive Aufgaben und Diskussionen um diese in den Unterricht integriert. Um dies effizient zu gestalten, werden die zu bearbeitenden Aufgaben und Beispiele direkt in web-basierte Lernmaterialien eingebettet, sodass kein Medienwechsel (vgl. *Split Attention Effect* in Kapitel 3.4.3) notwendig ist. Zusätzlich ermöglichen die Funktionen der Entwicklungsumgebung einen schnellen Austausch von Lösungen sowie eine optimierte Darstellung auf *Beamern*. Bei Lernmaterialien wird zwischen dem Skript (interaktive Dokumente) für die Lernenden und Präsentationen für den Lehrenden unterschieden, um beiden eine sinnvolle Darstellung zu ermöglichen. Die Inhalte und der Dozierende können sich den positiven Effekt der visuellen und akustischen Vermittlung von Inhalten nutzen, um neue Konzepte einzuführen und vorzuführen ohne auf statische Beispiele zurückgreifen zu müssen (siehe *Modality Effect* in Kapitel 3.4.3). In den nächsten zwei Abschnitten werden die Erstellung und Besonderheiten beschrieben, welche für die Verwendung im Unterricht und zur Unterstützung der Durchführung des Konzeptes konzipiert wurden.

### 6.4.1 Dokumente

Ausgangspunkt für Materialien sind interaktive Dokumente, die mit den Kapiteln der interaktiven E-Books bzw. eines einzelnen *Jupyter Notebooks* (vgl. Kapitel 6.1.1) vergleichbar sind. Das Konzept sieht vor, dass der Unterricht auf die jeweiligen Lernenden, zumindest teilweise, angepasst wird. Darunter fällt beispielsweise die Ergebnissicherung oder die Bereitstellung weiterer Aufgaben und Beispiele. Bei den vorhandenen Ansätzen wurde bereits der Nachteil der interaktiven E-Books angesprochen,

die bei Veränderungen komplett neu generiert werden müssen. Im Falle von *Runestone* ist das zwar über eine Schaltfläche auf der Webseite möglich, allerdings müssen die Inhalte extern bearbeitet werden. Das Konzept hinter *Jupyter Notebooks* sieht die schrittweise Erstellung und Bearbeitung der Inhalte innerhalb der Oberfläche vor, wodurch Änderungen leicht vorzunehmen sind. Einzig die schwierige Verbreitung der Dokumente bzw. die Handhabung bei Aktualisierungen ist für den geplanten Einsatz von Nachteil. Eine zentrale Speicherung der Dokumente und eine Steuerung der Berechtigungen beim Zugriff darauf, ist somit wünschenswert. Aus diesem Grund wurde ein dokumentenbasierter Ansatz gewählt, der das in Kapitel 6.1.1 bereits beschriebene Konzept von Inhaltselementen verwendet.

### Format

Das von *Jupyter Notebooks* definierte Format dieser Inhaltselemente (`nbformat 4.0`) wird zur Speicherung aller Texte und interaktiven Elemente verwendet und darüber hinaus um einen Inhaltstyp erweitert. Quellcode 4 zeigt den exemplarischen Aufbau des Formats, welches einerseits Informationen über das Format selbst und in den Metadaten die Informationen über die Ausführungsumgebung (`kernel_info`) und über die zu verwendende Programmiersprache (`language_info`) abspeichert. Ein Dokument besteht dabei aus einer Vielzahl an Inhaltselementen (Zellen) (`cells`), die untereinander dargestellt werden. Jede Zelle besteht dabei aus einem Inhaltstyp, Metadaten und

```
{
  "metadata" : {
    "kernel_info": {
        # if kernel_info is defined, its name field is required.
        "name" : "the name of the kernel"
    },
    "language_info": {
        # if language_info is defined, its name field is required.
        "name" : "the programming language of the kernel",
        "version": "the version of the language",
        "codemirror_mode": "The name of the codemirror mode to use [optional]"
    }
  },
  "nbformat": 4,
  "nbformat_minor": 0,
  "cells" : [
      # list of cell dictionaries, see below
  ],
}
```

**Quellcode 4**: JSON-Format eines Dokuments

dem jeweiligen Inhalt selbst (vgl. Quellcode 5). Alle Inhalte werden in dem Feld `source`

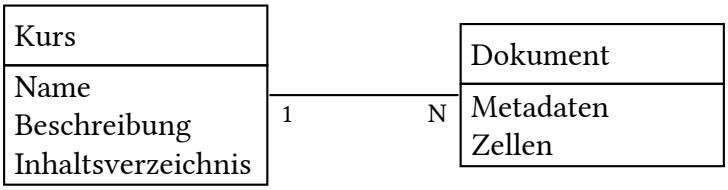



**Abbildung 50**: Zusammenhang zwischen Kursen und Dokumenten

hinterlegt und können entweder als Zeichenketten oder als Liste von Zeichenketten abgelegt werden. Der Standard sieht dabei, die in Tabelle 6 gezeigten, vier verschiedenen Inhaltselemente vor, die zusätzlich für die Einbettung von Aufgaben erweitert wurden. Ein Vorteil dieser Erweiterung ist die teilweise Erhaltung der Kompatibilität und Export sowie Import von Inhalten in diesem Format. Im Gegensatz zu *Jupyter*

```
{
  "cell_type" : "name",
  "metadata" : {},
  "source" : "single string or [list, of, strings]",
}
```

**Quellcode 5**: JSON-Format einer Zelle in einem Dokument

*Notebooks* wird das Dokument nicht lokal als Datei, sondern als *JSON* (JavaScript Object Notation [Bra14]) in einer dokumentenbasierten Datenbank abgelegt. Dokumente können dabei vom Ersteller und weiteren berechtigten Benutzern bearbeitet werden. Studierende können daraufhin über eine Internetadresse auf das Dokument zugreifen, welches vom Klienten anschließend dargestellt wird. Das Ausführen von Programmen und Beispielen wird über die festgelegte Ausführungsumgebung (*Sourcebox* bzw. *skulpt*) durchgeführt. Somit müssen Studierende keine weiteren Werkzeuge installieren und können innerhalb eines Browsers Programme erstellen und diese testen. Ein weiterer Vorteil ist die Möglichkeit zum Importieren und Exportieren von bereits vorhandenen Dokumenten, die bisher ggf. mit *Jupyter Notebooks* erstellt wurden.

Ein Lehrender hat zwei verschiedene Möglichkeiten seine Dokumente zu erstellen bzw. diese zu verknüpfen. Dazu wurde das Prinzip von Kursen, wie sie beispielsweise von *Moodle* bekannt sind, aufgegriffen. Einem Kurs können mehrere Dokumente zugeordnet werden, wobei eines der Dokumente als Inhaltsverzeichnis bzw. Übersichtseite dient (vgl. Abbildung 50). Sobald einem Dokument ein Kurs zugeordnet ist, wird automatisch eine Verlinkung zur Übersichtsseite erstellt, wodurch die Navigation erleichtert wird. Diese Verknüpfung ist jedoch nicht zwingend, weswegen Dokumente eigenständig verteilt werden können.

### Inhaltstypen

Im Folgenden werden die unterschiedlichen Inhaltselemente näher beschrieben und wie diese zur direkten Bearbeitung verwendet werden können. Bei der Erstellung eines neuen Dokuments sind in diesem noch keine Inhalte vorhanden, weswegen diese schrittweise erstellt werden müssen. Dazu wurde je Inhaltstyp eine Leseansicht und ein Bearbeitungsmodus implementiert, sodass alle Inhalte web-basiert bearbeitet werden können. Bei der Bearbeitung wird jeweils eine Zelle bzw. ein Element ausgewählt und der Bearbeitungsmodus aktiviert. Im Anschluss können Änderungen durchgeführt und diese anschließend gespeichert werden. Sobald die Benutzer die jeweilige Seite neu laden, stehen die neuen Inhalte bereits zur Verfügung. Dadurch lassen sich Änderungen und das Einfügen weiterer Inhalte effizient und somit im Unterricht umsetzen.

**Text** Zur Verwendung von (formatiertem) Text existiert ein Inhaltstyp, welcher in *Markdown* geschriebene Inhalte formatiert darstellen kann. *Markdown* [Joh16] ist ähnlich wie *RestructedText* ein Text-zu-HTML Filter, der die Formatierungen explizit im Text erwartet und zur Darstellung in HTML-Elemente umwandelt. Die eigentliche Formatierung bzw. Steuerung des Aussehens einzelner Elemente erfolgt über *CSS* (*Cascading Style Sheets*), wodurch ein einheitlicher Stil erzwungen werden kann. Ab-

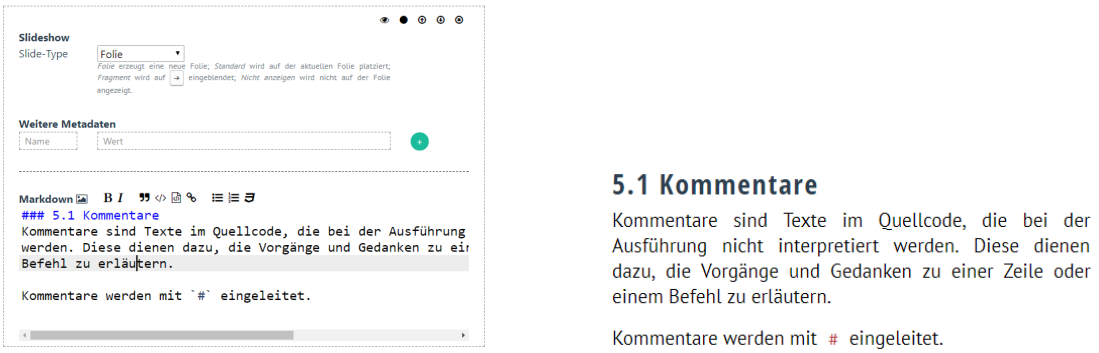



**Abbildung 51**: Bearbeitungsmodus (links) und Leseansicht (rechts) von Textelementen

bildung 51 zeigt den Bearbeitungsmodus (links) inklusive eines Beispiels in *Markdown* und das daraus erstellte Resultat in der Leseansicht (rechts). Der Bearbeitungsmodus ist dabei in zwei Bereiche unterteilt: Im oberen können die Metadaten bearbeitet und Einstellungen zur Erstellung von Präsentationen, auf die später noch näher eingegangen werden, vorgenommen wird. Darunter kann der Inhalt in einem Editor bearbeitet werden, wobei die Bearbeitung durch eine Werkzeugleiste erleichtert wird. Zudem

können Bilder und Grafiken innerhalb des Editors hochgeladen und eingefügt werden. Auf eine Beschreibung der *Markdown* Syntax wird an dieser Stelle verzichtet. Stattdessen werden nachfolgend die unterstützten Formatierungen und Textelemente aufgezählt, um einen Überblick über die Möglichkeiten zu geben:

- Bilder
- Überschriften
- Verlinkungen und Inhaltsverzeichnisse
- Zitate
- Listen
- Paragraphen
- Quelltext innerhalb eines Absatzes
- Quelltextblöcke
- Mathematische Ausdrücke und Formeln mittels LaTeX-Formatierung
- HTML (kann zur erweiterten Formatierung verwendet werden)

Die Bilder werden automatisch für ein Dokument und für den entsprechenden Kurs gespeichert und können in anderen Zellen ebenfalls eingebunden werden. Quelltexte die innerhalb von Textelementen eingebettet werden, können nicht ausgeführt werden. Diese dienen dazu auch kleinere Beispiele oder Sachverhalte darzustellen und diese explizit als nicht ausführbar zu kennzeichnen.

Mittels der Befehlsleiste in der oberen rechten Ecke (vgl. Abbildung 51 (links)) kann der Bearbeitungsmodus verlassen werden, was zur Folge hat, dass die Oberfläche in die Leseansicht zurückspringt und die neuen Änderungen sofort anzeigt. Zusätzlich können Zellen nach oben bzw. unten verschoben oder gelöscht werden. Darüber hinaus können Zellen ausgeblendet werden, um ggf. unfertige Inhalte nicht zu veröffentlichen oder Musterlösungen zu einem späteren Zeitpunkt erst zur Verfügung zu stellen.

**Code** Die in Kapitel 6.3.3 beschriebene Entwicklungsumgebung eignet sich für die Bearbeitung und Darstellung von Aufgaben und umfangreicheren Beispielen. Oftmals werden nun aber auch kleinere Beispiele zum Erklären verwendet, bei denen die Verwendung der Umgebung zu viele Ressourcen und Platz wegnimmt. Nicht jedes Beispiel ist als Aufgabe konzipiert und muss von allen Studierenden ausgeführt oder bearbeitet werden, teilweise ist dies nur zum Nacharbeiten oder zum Präsentieren im


**Abbildung 52:** Ausführbare Programme innerhalb eines Dokuments

Unterricht notwendig. Aus diesem Grund wurde zusätzlich eine einfachere Darstellung von kleineren Beispielen umgesetzt, die sich ähnlich zu *Jupyter Notebooks* bzw. *Runestone* direkt im Skript ausführen lassen. Abbildung 52 zeigt die Darstellung dieser Inhaltselemente in der Leseansicht, die sich in zwei Bereiche aufteilt. Die Ausgabe bzw. grafische Ausgabe erscheint erst nach dem Ausführen des Beispiels, ansonsten wird nur der obere Teil mit dem Quelltext angezeigt. Zudem kann der Quelltext ebenfalls in einem Editor bearbeitet, jedoch nicht von Studierenden gespeichert werden. Somit kann auf Fragen oder Rückmelden direkt eingegangen werden bzw. Studierende können Beispiele direkt im Dokument selbst ausprobieren bzw. ändern und die Ergebnisse beobachten. Wie bereits angemerkt, wird in dieser Ansicht ebenfalls die Darstellung von *Turtle Graphics* und *matplotlib* unterstützt. Derzeit ist einzig die Limitation auf eine einzelne Datei bei diesen Beispielen gegeben, da sonst ggf. der Vorteil der kompakten Darstellung verloren geht.

Im Gegensatz zu *Jupyter Notebooks* kann die Ausführungsumgebung und Programmiersprache pro Zelle verändert werden, um in einem Dokument verschiedene Sprachen einzubetten. Dies geschieht über die Verwendung spezieller Metadatenschlüssel, die einerseits die Programmiersprache und die Ausführungsumgebung spezifizieren können (vgl. Abbildung 53). Sobald diese in den Metadaten eines Elements angegeben werden, wird damit die standardmäßige Umgebung, die auf Ebene des Dokumentes definiert ist, überschrieben. Aus didaktischer Sicht kann somit die Universalität eines Programmierkonzepts anhand unterschiedlicher Sprachen gezeigt werden. Der Quelltext des Beispiels kann dabei in einem Editor vom Ersteller verändert und gespeichert werden. Weiterhin ist es möglich das Beispiel in der vollständigen Entwicklungsum-

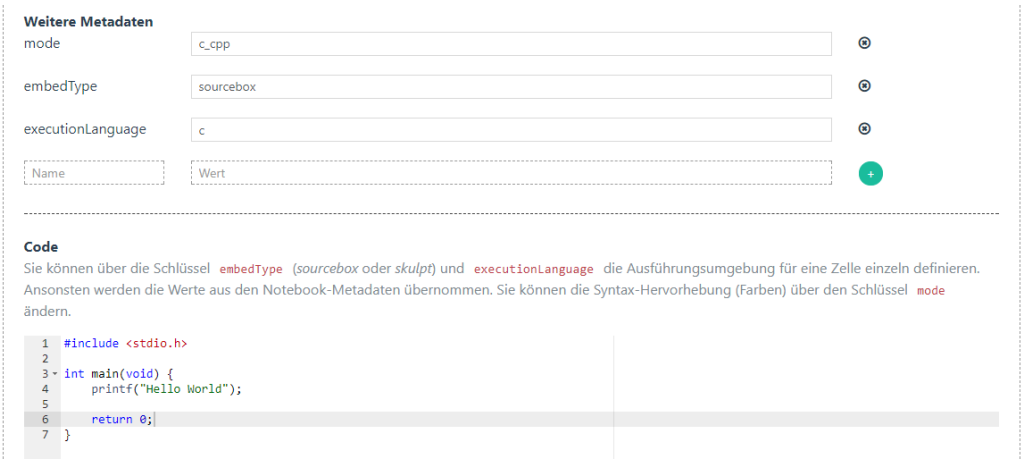



**Abbildung 53**: Bearbeitungsmodus des Inhaltstyps *Code* mit Änderung der Programmiersprache

gebung zu öffnen.

**Aufgabe** Für Aufgaben im Unterricht sowie für Übungsveranstaltung kann die Entwicklungsumgebung in das Dokument eingebettet werden. Dazu wurde der Standard (`nbformat 4.0`) um einen neuen Inhaltstyp `codeembed` erweitert, der im Feld `source` nur einen eindeutigen Bezeichner einer Aufgabe speichert. Über diesen werden für die Anzeige alle relevanten Informationen geladen und in der Entwicklungsumgebung angezeigt. Somit können alle Funktionen, die in Kapitel 6.3.3 beschrieben sind, wie beispielsweise das Einreichen von Lösungen, genutzt werden. Die Entwicklungs-

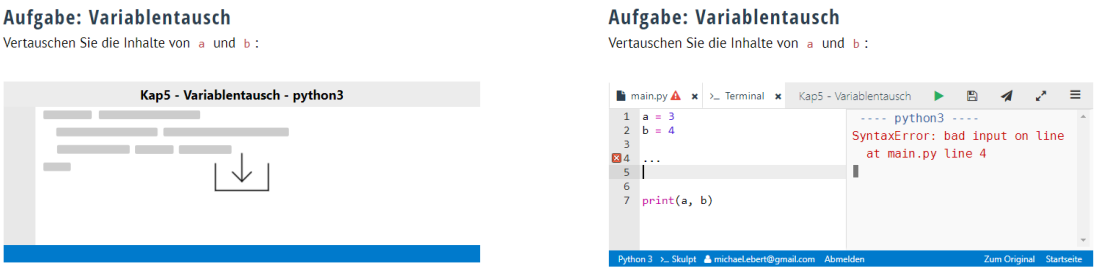



**Abbildung 54**: Ungeladene (links) und geladene (rechts) Entwicklungsumgebung

umgebung kann dabei als *iFrame* (Webseite in Webseite) geladen werden, jedoch hat dies einen negativen Einfluss auf die Ladezeit und Geschwindigkeit eines Dokuments. Deswegen wurde eine alternative Darstellung konzipiert, bei der die Aufgaben erst explizit geladen werden müssen, was den Seitenaufbau beschleunigt und die Ressourcen schont. Abbildung 54 zeigt eine solche Aufgabe, bevor die Entwicklungsumge-

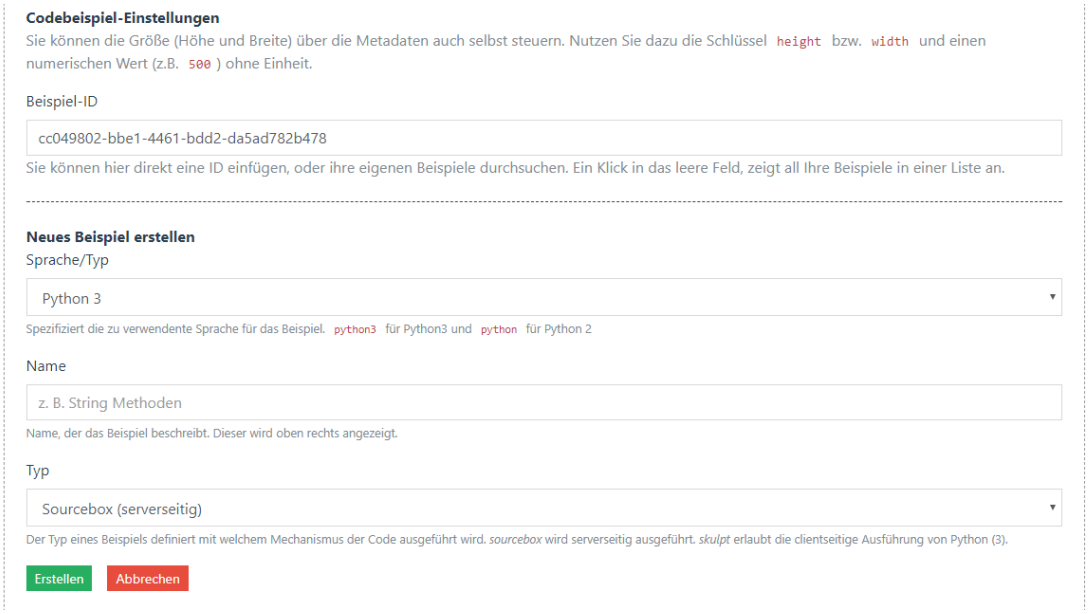



**Abbildung 55**: Bearbeitungsmodus des Inhaltstyps *Aufgabe*

bung geladen wurde (links). In dieser Darstellung werden nur die Metainformationen angezeigt, und die restlichen Daten (vgl. Abbildung 37) erst nach einem Klick auf das Download-Symbol (in der Mitte) geladen. Nach diesem Klick wird die Umgebung im Dokument geladen und angezeigt ohne auf *iFrames* zurückzugreifen (vgl. Abbildung 54 (rechts)). Im Gegensatz zu dem Inhaltstyp *Code* können Studierende in der Entwicklungsumgebung ihre eigenen Versionen speichern und wieder abrufen.

Neue Beispiele können direkt im Dokument im Bearbeitungsmodus erstellt oder bereits vorhandene Beispiele erneut eingebunden werden. Letzteres eignet sich für die Besprechung von Übungsaufgaben oder das Wiederverwenden von Beispielen an anderen Stellen bzw. in weiteren Dokumenten. Im Falle einer neuen Aufgabe, kann diese direkt im Bearbeitungsmodus erstellt und anschließend in der Entwicklungsumgebung bearbeitet werden. Abbildung 55 zeigt dabei das Formular für die Erstellung eines neuen Beispiels und das Eingabefeld für bereits vorhandene Beispiele, welches zugleich als Suche verwendet werden kann. Über die Metadaten kann die Höhe (und Breite) in der die Entwicklungsumgebung angezeigt werden soll, konfiguriert werden. Besonders bei kleineren Aufgaben kann der Platz der Umgebung angepasst werden, um den Textfluss nicht zu stören und zur Steuerung der Darstellung in Präsentationen verwendet werden.

**HTML-Inhalte (Externe Inhalte)** Der letzte Inhaltstyp kann für die Einbindung externer Inhalte (*Raw* vgl. Tabelle 6) und *HTML* verwendet werden. Ziel ist es das

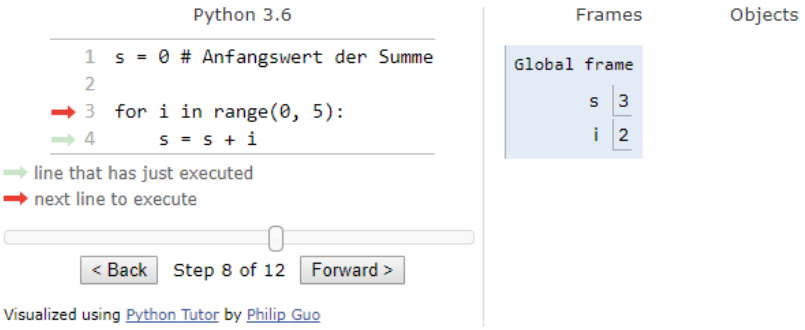



**Abbildung 56**: *PythonTutor* eingebettet in ein Dokument

vorhandene Format einerseits für weitere Inhaltstypen zu erweitern und zugleich eine einfache Möglichkeit zum Einbinden bereits vorhandener Inhalte zur Verfügung zu stellen. Ein Beispiel dafür ist die Verwendung von *PythonTutor* (vgl. Abbildung 56) für die schrittweise Ausführung und Visualisierung des Stacks und Heaps von Python und anderen Programmiersprachen [Guo13]. Zusätzlich kann beliebiger *HTML*-Quelltext verwendet werden, um weitere externe Inhalte, wie beispielsweise Quizze oder Online-Videos einzubinden.

### Anmerkungen

Die beschriebenen Mechanismen zur Erstellung von Dokumenten und zur Verwaltung eines Kurses ermöglichen ein komfortables Erstellen web-basierter, interaktiver Inhalte. Mittels der beschriebenen Inhaltselemente können interaktive Lernmaterialien konzipiert werden, mit denen Studierende von überall interagieren können.

Das zugrundeliegende Format kann zudem für weitere Inhaltstypen erweitert werden, wie z. B. das in Kapitel 4.4.1 beschriebene Quiz mit speziellen Fragetypen [EH15a], was jedoch im Rahmen dieser Arbeit aufgrund des Semesteranfangs nicht mehr umgesetzt werden konnte. Aus diesem Grund wurde die Unterstützung für beliebige externe Inhalte integriert, sodass die Plattform ein flexibles Instrument zur Gestaltung der Lernmaterialien bietet. Um den Lesern ein zusammenhängendes Bild eines Dokuments zu gewähren, wurde im Anhang A ein Ausschnitt eines längeren Dokuments angehängt.

### 6.4.2 Präsentationen

Eine weitere Forderung ist die Verwendung von Präsentationen im Unterricht, sodass eine optimale Darstellung der Inhalte unter Verwendung eines *Beamers* möglich ist. In Kapitel 6.1.3 und Kapitel 6.2 wurde die Möglichkeit zur Verwendung aller interaktiven Elemente innerhalb von Präsentationen gefordert. *Runestone* ermöglicht dies in der derzeitigen Form nicht und benötigt, wie weitere Ansätze auf Basis von *Sphinx*, eigene Quellen für die Erstellung der Präsentation. Im Kontrast dazu steht die einfache Generierung der Präsentation aus einem *Jupyter Notebook*, wie es mit der Erweiterung *RiSE* möglich ist. Der Nachteil dieser Lösung liegt bei der Darstellung und Verwendung der zugrundeliegenden Bibliothek (*reveal.js*) zur Erstellung der Präsentation[34].

#### Probleme bei der Darstellung

Zu beachten ist, dass für die Optimierung der Darstellung von Quelltexten und Oberflächen die Auflösung des Beamers für jede Präsentation einzeln konfiguriert werden muss. Darüber hinaus können weitere interaktive Elemente nur über Erweiterungen der Bibliothek hinzugefügt werden. *RiSE* löst dieses Problem, indem zu lange Inhalte nur über die Verwendung einer Scrollleiste zu sehen sind und die Textgröße auf eine bestimmte Größe festgelegt wird. Dessen ungeachtet wurde die Bibliothek *reveal.js* für einen möglichen Einsatz evaluiert und aufgrund von Problemen im Zusammenhang mit Quelltexteditoren verworfen. Präsentationen vergrößern bzw. verkleinern zur Anpassung der Größe die Text- und Inhaltselemente, sodass möglichst alle Inhalte dargestellt werden können. Zu Beginn dieser Arbeit verwendete *reveal.js* die Layouteigenschaft `zoom` und anschließend `transform` zur Skalierung der Inhalte. Dies führt jedoch bei der Verwendung von direkt eingebetteten Editoren zu Positionierungsfehlern des Mauszeigers, da die Berechnung der Position von diesen selbst durchgeführt wird. Für die Berechnung werden die Breiten der einzelnen Zeichen in einer Zeile benötigt, um die jeweilige Position beim Klicken in den Quelltext zu bestimmen. Derzeit werden die Breiten in Abhängigkeit einer Schriftart, welche immer die gleiche Breite besitzt (*monospaced*), berechnet. Eine Veränderung der Skalierung durch die genannten Eigenschaften, wird bei der Berechnung nicht berücksichtigt, da die Bestimmung des richtigen Faktors aufgrund der Verschachtelungen eines HTML-Dokuments erschwert wird. Denn es können auf unterschiedlichen Ebenen eines Dokuments verschiedene Skalierungsangaben erfolgen, welche Auswirkungen auf deren Kindelemente haben. Aus diesem Grund deaktiviert *RiSE* die automatische Anpassung, was wiederum zu Problemen bei der Darstellung führt. Zum Zeitpunkt der

[34] *RiSE* verwendet *reveal.js* (siehe https://github.com/hakimel/reveal.js) zur Darstellung der Präsentationen.

**Tabelle 10**: Darstellungsmöglichkeiten der Inhaltselemente in Präsentationen

| Art | Beschreibung |
|---|---|
| Folie | Das Element wird auf einer neuen Folie platziert. |
| Fragment | Das Element wird nach einem Klick auf der bisherigen Folie eingeblendet. |
| Standard | Das Element wird auf der bisherigen Folie platziert |
| Nicht anzeigen | Das Element wird nicht in der Präsentation angezeigt. |

Erstellung dieser Arbeit existierten nur zwei Quelltexteditoren, die den Anforderungen für die Entwicklungsumgebung genügten. Erst zum Ende der Entwicklung wurde von Microsoft ein weiterer veröffentlicht, der mit dieser Skalierung umgehen kann[35]. Es ist zwar möglich die Editoren über *iFrames* einzubinden, da dies die Anwendung der Skalierung verhindert, allerdings kann so keine optimale Darstellung garantiert werden. Aus diesem Grund wurde ein anderer Ansatz bzw. eine andere Bibliothek zur Erstellung der Präsentationen verwendet. *Spectacle*[36] verwendet zur Skalierung der Inhalte nicht die problematischen Layouteigenschaften, sondern einen Algorithmus zur Bestimmung einer optimalen Schriftgröße, sodass alle Inhalte auf eine Folie passen. Andere Inhalte können über die Verwendung eigener Komponenten zur Darstellung selbst optimiert werden. Dieser Möglichkeit wurde für diese Plattform genutzt, um interaktive Komponenten kompakt in Präsentationen darzustellen.

### Generierung der Präsentationen

Um die Pflege von mehreren Dokumenten zu vermeiden ist ein weiteres Ziel, die Präsentationen aus den bereits geschriebenen Dokumenten zu generieren, sodass die Inhalte nur an einer Stelle gepflegt werden müssen. Zudem soll es möglich sein, nur bestimmte Inhalte in die Präsentation mit aufzunehmen, und zugleich ein ausführlicheres Skript zur Verfügung zu stellen. Zusätzlich können so Redundanzen (vgl. Kapitel 3.4.3) bei der Präsentation der Inhalte vermieden werden, welche die Studierenden anfangs zu stark ablenken bzw. verwirren könnten. Inspiriert von *RiSE* wurde das Konzept zur Annotation der Inhaltselemente für die Darstellung in Präsentationen übernommen und angepasst. In Tabelle 10 sind alle möglichen Darstellungsarten und deren Beschreibung zusammengefasst. Eine Präsentation besteht dabei aus mehreren Folien, auf welchen die Inhaltselemente platziert werden können. In Abbildung 51 ist in der oberen linken Seite das Menü zum Auswählen der Darstellungsart sichtbar.

[35] Siehe https://microsoft.github.io/monaco-editor/
[36] Siehe http://formidable.com/open-source/spectacle/

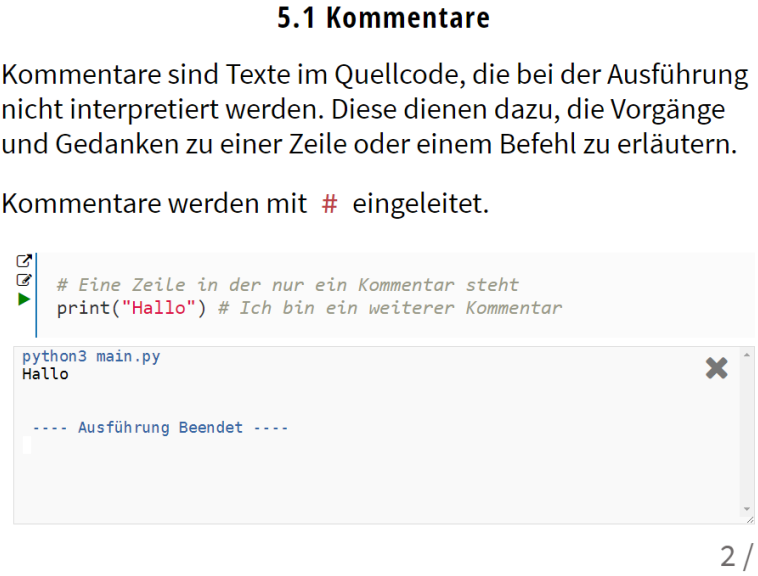



**Abbildung 57**: Darstellung einer Folie in einer Präsentation

Wird vom Benutzer keine Auswahl getroffen bzw. diese verändert, so wird automatisch die Auswahl *Standard* verwendet und das Element auf der aktuellen Folie platziert. Mehrere Elemente auf einer Folie werden untereinander dargestellt, können aber auch mittels weiterer Formatierungsangaben nebeneinander platziert werden. Abbildung 57 zeigt die Darstellung der Inhaltselemente auf Folien, welche sich je nach Auflösung vergrößern bzw. verkleinern. Trotz der automatischen Skalierung der Inhalte ist eine Überprüfung der Darstellung notwendig sowie eine nur begrenzte Anzahl an Inhaltselementen auf einer Folie möglich.

## 6.5 Indikatoren zur schnellen Einschätzung der Studierenden

Für den letzten Teil der Umsetzung wurden mehrere Indikatoren zur Einschätzung der Studierenden im Unterricht und zur weiteren nachträglichen Analyse erarbeitet. Darüber hinaus stellen diese eine Möglichkeit zur Evaluation der Plattform und des Konzepts dar, um empirische Auswirkungen erfassen zu können (vgl. Kapitel 2.3). Ebenfalls sollen diese bei der Überprüfung des Erreichens der intendierten Ziele, wie beispielsweise die Annahme des Konzepts, unterstützten. Bei der Vermittlung von Programmierkonzepten durch die Einbettung von Aufgaben anhand des in Kapitel 5

vorgestellten Konzepts, muss im Unterricht eine Entscheidung über die Anpassung der eigenen Lehre getroffen werden. Einen möglichen Hinweis auf den studentischen Fortschritt bzw. die Probleme, kann eine Analyse der aufgetretenen Fehler geben.

### 6.5.1 Analyse von Fehlern

Die Analyse der Fehler von Programmieranfängern für eine bessere Einschätzung wurde bereits mehrmals in der Forschung aufgegriffen und unterschiedlich betrachtet. Zunächst werden diese bisherigen Ansätze und daraus gewonnene Ansätze diskutiert und der im Rahmen dieser Arbeit verwendete Ansatz dargelegt.

Die Identifizierung der häufigsten Fehler und anschließende Kategorisierung wurde bereits in mehreren Studien für unterschiedliche Programmiersprachen durchgeführt [AB15; Too11]. Solche Ansätze verwenden meist die Ergebnisse bzw. Daten von Systemen für die automatische Bewertung von Programmieraufgaben (siehe Kapitel 6.1.2), die anschließend ausgewertet werden. Amjad et al. haben dazu 37 Millionen Datensätze ausgewertet und gezeigt, dass die Anzahl der syntaktischen und semantischen Fehler im Laufe eines Semesters abnehmen [AB15]. Zusätzlich wurde aus den Datensätzen die jeweilige Dauer bzw. Anzahl der Versuche bis zum Beheben der Fehler berechnet. Daraus ergibt sich, dass syntaktische Fehler im Vergleich zu semantischen Fehlern in kürzerer Zeit gefunden und verbessert werden können. Diese Aussage unterstützt in gewisse Weise auch die Zielsetzung dieser Arbeit, die bewusst mehr Gewicht auf das informatische Lösen mittels Algorithmen legt. Denn in Kapitel Kapitel 3.1 wurde bereits gezeigt, dass die Syntax von Programmiersprachen nur ein Teil des Problems sind und vielmehr das Verständnis der Zusammenhänge und das Problemlösen zu fördern sind.

McCall und Kölling haben in einer Studie aus ungefähr 20000 fehlerhaften Kompilierungen 197 Quelltexte näher untersucht, um eine aussagekräftige Kategorisierung von Anfängerfehlern durch eine qualitative Analyse zu erreichen. Eine reine quantitative Analyse von Fehler bzw. Diagnosemeldungen der Werkzeuge gibt nicht immer Rückschluss auf den dahinter liegenden Fehler. Besonders im Falle kaskadierender Fehler oder irreführender Meldungen kann die eigentliche Ursache oder eine falsche Vorstellung von Studierenden nicht von den Werkzeugen erkannt werden. Doch genau diese falschen Vorstellungen sind in der Lehre von Interesse, da diese Ansatzpunkte zur Wiederholung bzw. Verbesserung des Unterrichts geben. Die drei häufigsten Fehler bzw. Kategorien sind: (1) nicht deklarierte Variablen, (2) vergessene Semikolons (3) und falsch geschriebene Variablennamen. Diese Ergebnisse decken sich zusätzlich mit den in Jackson et al. und Denny et al. identifizierten Kategorien [JCC05; DLT12]. In den betrachteten Studien wurden bisher Syntaxfehler als häufigste Fehlerursache identifiziert. Denny et al. haben in einer weiteren Studie nachgewiesen, dass

eine Verbesserung der Fehlermeldungen samt Beispielen und Hinweisen zur Verbesserung keinen signifikanten Einfluss auf die Anzahl der Fehler hat [DLC14]. Pettit et al. konnten zudem in einer weiteren Studie diese Ergebnisse nicht messbar widerlegen, dennoch empfanden Studierende die Hinweise als hilfreich [PHG17]. Die Autoren argumentieren, dass die Lernenden die Hinweise und Fehlerausgaben nicht richtig lesen, weswegen keine Verbesserung bzw. messbarer Einfluss nachgewiesen werden konnte. Ein erster Vorstoß, um dieses Problem anderweitig zu lösen wurde bereits in Kapitel 6.3.3 beschrieben. Besonders wie die Entwicklungsumgebung dafür mittels solcher Hinweise und weiterer spezieller Unterstützung erweitert werden könnte. Allerdings ist das Ziel, eine schnelle Übersicht bzw. eine Einschätzung für den Lehrenden zu ermöglichen, sodass die Durchführung von Übungen im Unterricht unterstützt wird. Dementsprechend ist die summative Auswertung der Daten und Kategorisierung nur bedingt hilfreich, da diese keinen Überblick zu einen bestimmten Zeitpunkt ermöglicht.

Darüber hinaus wurden in diesen Studien meist die dahinter liegenden falschen Vorstellungen nur oberflächlich betrachtet, da zumeist ein quantitativer Ansatz gewählt und die Daten erst nach der Lehrveranstaltung ausgewertet wurden. Berges et al. haben in einem neuen Ansatz versucht, ausgehend von Problemen in studentischen Einreichungen, fehlende Kompetenzen der Studierenden zu identifizieren [Ber+16]. Dafür wurde eine qualitative Inhaltsanalyse auf den Quelltexten durchgeführt und daraus Kategorien und falsche Vorstellungen bzw. fehlende Kompetenzen ermittelt. Bei der Analyse wurden bisher primär Fehler aus der objektorientierten Entwicklung betrachtet, welche in der intendierten Verwendung des Systems bzw. im Rahmen der geplanten Evaluierung nicht zutreffen [SBH17]. Zudem kann diese Analyse nicht automatisiert werden, sondern ist ein zeitintensiver manueller Prozess, weswegen er sich nicht für die schnelle Einschätzung eignet.

### 6.5.2 Fehler je Beispiel

Das in Kapitel 5 beschriebene Konzept adressiert jeweils ein bestimmtes Programmierkonzept, welches anhand von Aufgaben direkt im Unterricht eingeübt und in Übungsveranstaltungen weiter vertieft wird. Die bisherigen Ansätze sind von einer Auswertung am Ende ausgegangen und haben dabei die jeweiligen Aufgaben und den Kontext nicht erkennbar einfließen lassen. Aber genau dieser Kontext und das zu vermittelnde Konzept sollte bei der Auswertung bzw. Einschätzung berücksichtigt werden, um Missverständnisse rechtzeitig zu erkennen und entsprechend zu reagieren. Programmieren und Codieren ist fehleranfällig, weswegen bei der Bearbeitung von Aufgaben unterschiedliche Arten von Fehlern auftreten können. Dies reicht vom einfachen Syntaxfehler hin zum Laufzeitfehler und semantischen Fehler, welche auf algorithmische

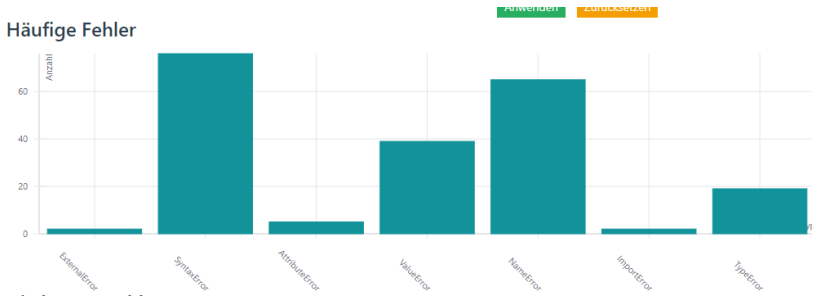



**Abbildung 58**: Darstellung der häufigsten Fehler für ein bestimmtes Beispiel

Fehlannahmen zurückzuführen sind. Im Laufe des Semesters steigert sich zudem die Komplexität der zu bearbeitenden Übungen, da die Konzepte aufeinander aufbauen und in Kombination verwendet werden. Zusätzlich werden die Aufgaben ebenfalls umfangreicher, um authentischere Probleme zu bearbeiten. Daraus kann abgeleitet werden, dass Studierende zu den jeweiligen Zeitpunkten unterschiedliche Fehler machen, weswegen ein rein summativer und quantitativer Ansatz zur Betrachtung von Fehlern nicht ausreicht. Fehler treten situativ bei den zu verwendenden Konzepten und Aufgaben auf. Daher müssen diese bei der Interpretation der Ursachen mit einbezogen werden müssen. Diese können dabei auf größere Missverständnisse hindeuten, aber ggf. auch nur auf Probleme bei der ersten Anwendung eines neuen Konzeptes.

Aus diesem Grund wurden zwei verschiedene Indikatoren bzw. Auswertungsmöglichkeiten in die Plattform bzw. Entwicklungsumgebung integriert. Studentische Interaktionen mit der Umgebung, wie beispielsweise Ausführungen, werden dabei inklusive des vollständigen Quelltextes erfasst und abgespeichert. Im Gegensatz zu den bisherigen Ansätzen wird zudem der Kontext, wie z. B. die Aufgabe und weitere Metadaten, erfasst. Dadurch können die häufigsten Fehler pro Beispiel, inklusive dem jeweiligen Quelltext in dem der Fehler aufgetreten ist, den Lehrenden zur Verfügung gestellt werden.

In der Entwicklungsumgebung können dazu jederzeit Statistiken zu dem jeweiligen Beispiel abgerufen werden, die sich zusätzlich in Echtzeit aktualisieren. Dazu werden die häufigsten Fehler für dieses spezifische Beispiel berechnet und darunter eine Möglichkeit zur Analyse angeboten (vgl. Abbildung 58). Standardmäßig werden die letzten zehn Fehler in einer Tabelle angezeigt, die zusätzlich den Fehlertext und Stelle umfasst. Die Tabelle kann zusätzlich nach Fehlertypen gefiltert werden, sodass die Analyse sich auf diese beschränkt (vgl. Abbildung 59 (oben)). Mittels eines Klicks auf eine Tabellenzeile wird der fehlerhafte Quelltext darüber in einem Editor angezeigt, um die Hintergründe des Fehlers zu erfassen. Dadurch lassen sich ggf. falsche Vorstellungen und semantische Fehler identifizieren, die für den angezeigten Fehler verantwortlich sind. Das in Abbildung 59 dargestellte Beispiel eines fehlerhaften Quellcodes, ermöglicht

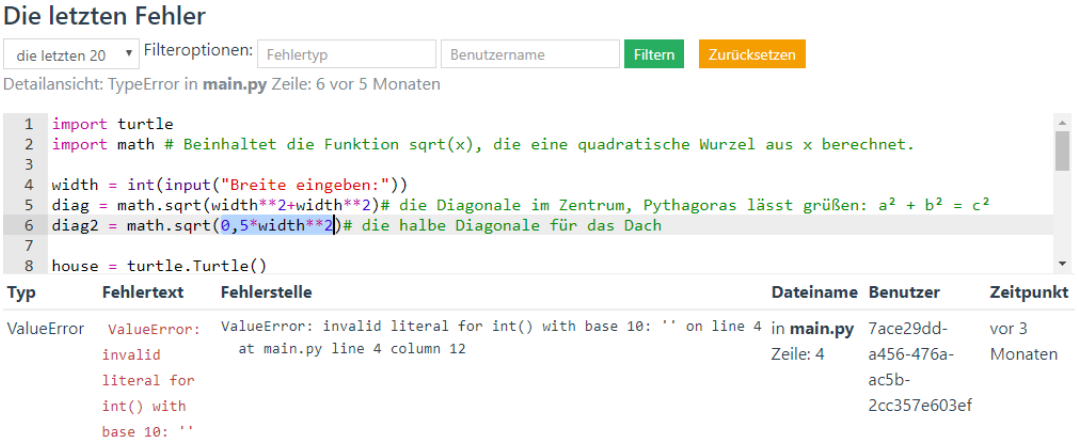



**Abbildung 59**: Auswertungsmöglichkeit der studentischen Fehler in der Entwicklungsumgebung

den Blick auf den eigentlichen Fehler. Der Python-Interpreter gibt für die blau markierte Stelle (`0,5*width**2`) folgende Fehlermeldung aus: `TypeError: sqrt() takes exactly 1 arguments (2 given)`. Der Fehler gibt an, dass beim Aufruf der Funktion zum Ziehen der quadratischen Wurzel zu viele Argumente übergeben wurden. Der Studierende hat in diesem Fall vergessen, dass die Programmiersprache nicht Kommata für die Dezimalzeichentrennung, sondern Punkte erwartet. Dementsprechend interpretiert die Sprache `0,5` als zwei Argumente und nicht als Dezimalzahl. Würde nur die Fehlermeldung zur Auswertung zur Verfügung stehen, kann die eigentliche Ursache unter Umständen, wie in diesem Beispiel, nicht ermittelt werden. Aus diesem Grund können die Häufigkeiten von Fehlern nur auf ein grundlegendes Problem hindeuten, ohne diese zu überinterpretieren. Das eigentlich das Ziel ist es Lehrenden einen schnellen Überblick des Kenntnisstands der Studierenden und wie sie die Aufgabe bearbeiten zu geben.

Mittels der beschriebenen Funktionen kann während der Durchführung rechtzeitig auf häufig auftretende Fehler reagiert und diese ggf. später weiter analysiert werden. Denn diese Daten können ebenfalls als Vorbereitung auf die nächste Vorlesung genutzt werden, um aus fehlerhaften Beispielen der Studierenden Wiederholungsfragen und Aufgaben zu erstellen. Zudem können auf Basis der Ergebnisse weitere Materialien zur Verfügung gestellt bzw. das Skript angepasst werden, sodass diese Fälle besser abgedeckt werden. Wofür sich die in Kapitel 6.4 beschriebene Möglichkeit zur Erstellung und Bearbeitung der Inhalte eignet, da die bereits vorhandenen Materialien ohne Weiteres ergänzt werden können.

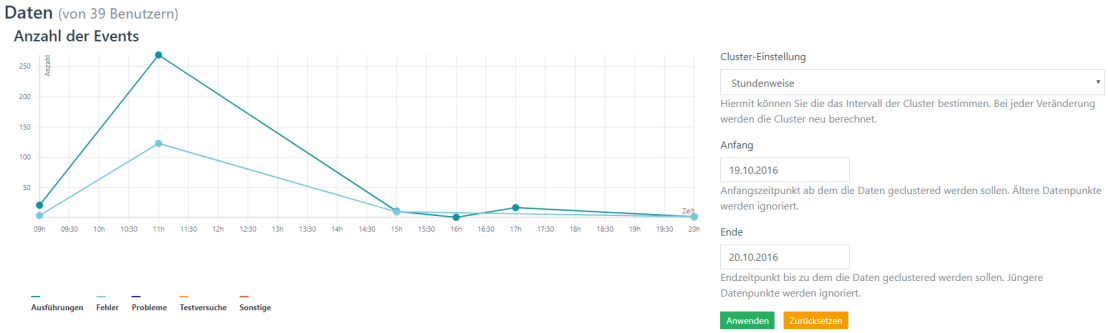



**Abbildung 60**: Darstellung der Ausführungen für ein Beispiel

### 6.5.3 Studentische Partizipation

Für die spätere Evaluation und im Unterricht ist ebenfalls die studentische Partizipation von Interesse. Einerseits können daraus quantitative Daten abgeleitet werden, um diese beispielsweise mit Noten zu korrelieren. Andererseits kann während der Durchführung der Aufgabe die Beteiligung der Studierenden bestimmt werden. Dazu werden alle Ausführungen, Fehler, Probleme (Fehler in der Entwicklungsumgebung), Testversuche (automatische Prüfung von Aufgaben) erfasst und diese an einer Zeitachse dargestellt. Abbildung 60 zeigt die berechneten Anzahlen der erfassten Interaktionen der Studierenden. Die schrittweise und iterative Entwicklung ist ein Merkmal des vorgestellten Prozesses des Programmierens (vgl. Kapitel 3.1), weswegen die reine Anzahl an Ausführungen nicht alleine aussagekräftig ist. Diese muss in Abhängigkeit der Anzahl der Lernenden, welche die Datenpunkte erzeugt haben, betrachtet werden. Aus diesem Grund muss die Anzahl der Benutzer berechnet und zur Verfügung gestellt werden. Doch auch diese Zahl muss in Abhängigkeit der Studierenden und der zur Verfügung stehenden Geräte im Unterricht interpretiert werden. In Kapitel 5 wurden bereits die Möglichkeit zum kooperativen und kollaborativen Lernens beschrieben, da nicht zu erwarten ist, dass alle Lernenden einen Computer oder Tablet besitzen oder dieses mitbringen. Deswegen kann dieser Indikator nur unter Berücksichtigung dieser Aspekte verwendet werden, da er Studierende, welche die Aufgaben auf Papier lösen, nicht erfasst.

Darüber hinaus werden die Daten ebenfalls in Echtzeit aktualisiert, sodass zugleich abgeschätzt werden kann, inwieweit die Studierenden noch an den Aufgaben arbeiten oder sich mit anderen vorlesungsfremden Inhalten beschäftigen. Für eine spätere Auswertung können die Zeiträume und die Intervallgröße eingestellt werden, um die Analyse auf bestimmte Zeiten zu begrenzen (siehe Abbildung 60 (rechts)).

### 6.5.4 Automatische Bewertungen

Der letzte Indikator zur Einschätzung der Studierenden kann aus den automatisch bewerteten Aufgaben berechnet werden. Ausgehend von erstellten Tests (vgl. Kapitel 6.3.4) können Studierende ihre eigenen Programme überprüfen und sich entsprechend selbst einschätzen. Diese Ergebnisse können jedoch zusätzlich zur Auswertung verwendet werden, da diese ebenfalls Aufschluss über den Bearbeitungsfortschritt geben können. Alle Testdurchführungen und die daraus entstandenen Resultate werden erfasst und daraus ein Durchschnittswert und die Standardabweichung zu bestimmten Zeitpunkten berechnet. Somit kann zum Zeitpunkt der Durchführung der Aufgaben, gesetzt den Fall, dass die Studierenden ihre Aufgaben überprüfen lassen, der Fortschritt bei der Bearbeitung analysiert werden.

Bei der Verwendung von Aufgaben im Unterricht ist die verfügbare Zeit für die Bearbeitung begrenzt, sodass trotz der Aktivitäten ausreichend Zeit für die Diskussion und weitere Inhalte zur Verfügung stehen. Dieser Indikator kann in Verbindung mit den bereits beschriebenen zu Verlängerung oder Verkürzung der Zeit genutzt werden. Zusätzlich kann ebenfalls die Schwierigkeit der Aufgabe eingeschätzt werden, wenn beispielsweise nur ein Bruchteil der Studierenden in der Lage ist, die Aufgabe vollständig zu lösen.

## 6.6 Zusammenfassung

In Kapitel 6 wurden die technischen Aspekte für die Erstellung interaktiver Lernmaterialien zur Umsetzung des in Kapitel 5 erarbeiteten Konzepts beschrieben. Zu Beginn wurden bereits vorhandene Ansätze und Plattformen bezüglich der intendierten Nutzung diskutiert, da bereits eine Vielzahl an Systemen existiert. Da sich diese Systeme teilweise stark ähneln, wurden jeweils bekannte Vertreter, die in bereits in einem großen Umfang eingesetzt werden, ausgesucht. Dabei wurden positive wie auch negative Aspekte der jeweiligen Systeme erfasst und in Tabelle 7 tabellarisch zusammengefasst. Anschließend wurden bisherige web-basierte Entwicklungsumgebungen diskutiert, wobei keines der beschriebenen Systeme explizit für die Verwendung in Präsentationen konzipiert wurden. Dies hat zur Folge, dass die Darstellung nicht für *Beamer* und in Präsentationen optimiert ist.

Angetrieben von der optimierten Darstellung und einfachen Erstellung von Inhalten, wurde ein hybrider Ansatz aus interaktiven E-Books und *Jupyter Notebooks* entwickelt. Um eine breite Anwendung zu garantieren, wurde eine Sandbox-Umgebung konzipiert und im Rahmen einer Masterarbeit implementiert. Mittels dieser Umgebung konnte eine web-basierte Entwicklungsumgebung, die für die Nutzung von Auf-

gaben optimiert ist, konzipiert und implementiert werden. Diese Elemente dienen der Umsetzung von anschaulichen Problemstellungen anhand denen neue Programmierkonzepte eingeführt werden können. Darüber hinaus wurde ein Mechanismus zum Einreichen von Lösungen im Unterricht entwickelt, sodass eine effiziente Umsetzung des Konzepts und der dazugehörigen Diskussionen im Unterricht gewährleistet ist. Zudem wurden drei Indikatoren auf Basis der studentischen Interaktionen ausgehend von bisherigen Forschungen abgeleitet und verbessert. Somit können Lehrende während der Bearbeitungsphase von Aufgaben, bereits erste Fehler und die studentische Partizipation überprüfen und entsprechend reagieren. Aus Sicht der Evaluation können diese Daten ebenfalls für eine spätere Bewertung und tiefere Analyse des Konzepts verwendet werden.

# Kapitel 7

# Evaluation

Der letzte Schritt des forschungsmethodischen Vorgehens (vgl. Kapitel 2.3) ist die Evaluation des Konzepts und der dafür eigens entwickelten Plattform, die einerseits die Umsetzung ermöglicht und andererseits Daten für die Auswertung erhebt. Die Evaluation des Konzeptes und der Plattform erfolgt dabei aus mehreren Perspektiven. Einerseits wird das Vorlesungskonzept als Ganzes am Ende des Semesters mittels eines anonymen Fragebogens evaluiert. Andererseits wird die Partizipation und die daraus resultierende Auseinandersetzung mit Vorlesungsinhalten und Konzepten im Zusammenhang mit den Prüfungsergebnissen untersucht. Dazu wird zuerst die Lehrveranstaltung und die umgesetzten Lehrhandlungen beschrieben, um das Konzept zu evaluieren.

## 7.1 Beschreibung der Durchführung

Das Konzept wurde im Rahmen der Lehrveranstaltung Programmieren 1 für Studierende der Elektrotechnik evaluiert. Die Lehrveranstaltung findet dabei im ersten Semester statt und wird für drei Vertiefungsrichtungen: Elektro- und Informationstechnik, Automatisierungstechnik und Robotik und Erneuerbare Energien angeboten. Zum Start des Semesters waren ca. 120 Studierende mit ca. 40 Studierenden je Studiengang immatrikuliert, wobei ein Teil bereits nach wenigen Wochen nicht mehr anwesend war. Ähnlich zu anderen naturwissenschaftlichen Studiengängen ist eine hohe Abbrecherquote zu Beginn bzw. während des ersten Semesters zu beobachten.

### 7.1.1 Organisatorisches

Die Lehrveranstaltung richtet sich an Programmieranfänger, die bisher noch keine Erfahrungen gesammelt haben. Frühere Befragungen haben ergeben, dass ein Großteil der Studierenden nur wenig bis kein Vorwissen im prozeduralen Programmieren gesammelt hatte. Der LV stehen vier Semesterwochenstunden (SWS) zur Verfügung, die in je zwei Stunden Vorlesung und Übung aufgeteilt sind. Aufgrund der Anzahl der Studierenden werden die Übungen in drei Gruppen durchgeführt und vom Dozierenden sowie einem zusätzlichen Labormitarbeiter betreut. Die erste Übung fand erst nach den ersten zwei Vorlesungen statt, jedoch sind zu Beginn Probleme bei der Raumplanung aufgetreten. Deswegen die Übungen Donnerstag am späten Nachmittag und Freitag nachmittags stattfanden, bis diese auf Dienstag nachmittags verschoben werden konnten. Weder die Anwesenheit während der Übungen oder der Vorlesung ist verpflichtend und stellt keine Voraussetzung zur Prüfungszulassung dar. Durchschnittlich waren 60 bis 70 Studierende in der Vorlesung anwesend. Eine 90-minütige Prüfung wird in schriftlicher Form abgehalten, bei der alle Hilfsmittel außer kommunizierende Geräte bzw. Computer erlaubt sind.

### 7.1.2 Inhalte und Ziele

Primäres Lehrziel stellen das Programmieren und die damit verbunden theoretischen Grundlagen und Konzepte dar. Die Lehrveranstaltung dient, wie bereits in Abbildung 17 gezeigt, als Vorbereitung auf konsekutive Module, die einen stärkeren Bezug zur hardwarenahen Programmierung haben. Aus diesem Grund werden in der ersten Lehrveranstaltung die Konzepte der Programmierung fokussiert (siehe auch Diskussion in Kapitel 2.1), ohne zu sehr auf Besonderheiten der hardwarenahen Programmierung einzugehen. Es soll in erster Linie ein allgemeines Verständnis für die Vorgänge beim Programmieren vermittelt und in den anschließenden Semestern fachspezifische Inhalte adressiert werden. Dies schließt jedoch die Verwendung von fachlich relevanten Beispielen und Problemen nicht aus.

Thematischer Rahmen der LV ist das Programmieren als unterstützendes Werkzeug bei beruflichen Tätigkeiten, sowie die wissenschaftliche Auswertung und Visualisierung, wobei die Programmiersprache *Python* in Version 3 für die Beispiele und Übungen verwendet wird. Der motivationale Rahmen wird durch aktuelle Beispiele in der Industrie adressiert, indem die Relevanz des Programmierens und der Informatik im Bereich der Elektrotechnik betont wird. *Python* wurde aufgrund der einfachen Syntax und aussagekräftigeren Fehlermeldungen gewählt, um Studierenden einen leichten Einstieg in das Thema zu ermöglichen und zugleich den Fokus auf die Konzepte zu legen. Inhaltlich orientiert sich die LV an dem Buch *Think Python* von Allen B.

Downey [Dow15], jedoch wird davon teilweise abgewichen, um eigene Schwerpunkte zu setzen. Beispielsweise werden zu Beginn der Aufbau und die Funktionsweise von Computern sowie Zahlensysteme behandelt, um Studierenden die Grenzen der Berechenbarkeit aufzuzeigen. Anschließend wird an das Thema Algorithmen herangeführt, deren Eigenschaften anhand alltäglicher Handlungen, wie z. B. Reifenwechseln, definiert werden. Dabei werden fünf grundlegende Anweisungen beschrieben mit denen jegliche Algorithmen dargestellt und implementiert werden können:

- Eingabe,
- Ausgabe,
- mathematische Anweisungen (z. B. Operatoren),
- bedingte Ausführung
- und die Wiederholung.

Diese bilden zugleich einen thematischen Rahmen, auf den in den einzelnen Kapiteln zu den Konzepten jeweils verwiesen wird. Im Laufe des Semesters werden für all diese grundlegenden Anweisungen Programmierkonzepte vorgestellt und diese entsprechend geübt. Für die Lehrveranstaltung wurden interaktive Lernmaterialien und Präsentationen für alle Kapitel erstellt. Tabelle 11 zeigt die Titel und Inhalte der Kapitel, die jeweils als eigenständige und interaktive Dokumente erstellt wurden. Die einzelnen Kapitel haben einen unterschiedlichen Umfang, weswegen in manchen Vorlesungen mehrere Kapitel auf einmal behandelt wurden. Zu Beginn werden in den Kapiteln *1* bis *11* grundlegende Programmierkonzepte vermittelt. Kapitel *12* beschäftigt sich ausschließlich mit der Vorgehensweise beim Programmieren und wie unterschiedliche Strategien bei der Programmkomposition verfolgt werden können. Das letzte grundlegende Konzept stellt die infinite Iteration dar (Kapitel *13*). Im Anschluss folgen weitere aufbauende und vertiefende Themen und komplexere Datenstrukturen, wie z. B. Zeichenketten und Listen. Diese dienen dazu die bisherigen Konzepte anhand neuer Themen zu wiederholen und somit die breite Anwendung eines einzelnen Konzepts aufzuzeigen.

Kapitel *20* ist die Vorbereitung auf die Einführung numerischer Berechnungen mit *Python*. Einerseits wird gezeigt, wie mehrere Werte aus einer Funktion zurückgegeben werden können, aber auch wie Tupel allgemein verwendet werden können. Darunter fällt auch die Darstellung von Vektoren als Tupeln, die anschließend in Kapitel *21* näher diskutiert und visualisiert werden.

Das letzte Kapitel dient als Zusammenfassung aller bisherigen Konzepte, wobei die Abstraktheit und die universale Anwendbarkeit betont werden. Unter anderem werden die jeweiligen Konzepte anhand von Python und zusätzlich anhand von *C* ge-

**Tabelle 11**: Übersicht der Kapitel und Themen

| # | Kapitel | Inhalte |
|---|---|---|
| 1 | Einführung | Relevanz der Informatik |
| 2 | Computeraufbau | Komponenten und Funktionsweise eines PCs |
| 3 | Python der Taschenrechner | Berechnungen mit dem Interpreter |
| 4 | Algorithmen | Einführung in die Algorithmen, Eigenschaften, Laufzeiten und Fehlerarten |
| 5 | Grundelemente von Python (1) | Werte, Variablen, Ausdrücke und Anweisungen |
| 6 | Grundelemente von Python (2) | Ausgabe, Eingabe, Literale und Typkonvertierung |
| 7 | Mathematische Funktionen | Funktionen, Konstanten und Zufallsgeneratoren |
| 8 | Turtle und die Iteration | Einführung in die finite Iteration anhand *Turtle Graphics* |
| 9 | Zahlensysteme | Zahlensysteme, -bereiche und -darstellung |
| 10 | Funktionen | Aufrufe, Parameter, Rückgabewerte |
| 11 | Bedingungen | Bedingte Ausführung und Bedingte Anweisungen |
| 12 | Komposition | Programmkomposition und Vorgehensweisen |
| 13 | Infinite Iteration | Iteration mittels `while` |
| 14 | Strings | Eigenschaften und Algorithmen |
| 15 | Listen | Referenzen und Funktionsweise |
| 16 | Matrizen | Darstellung und Anwendung |
| 17 | Traversieren | Iteration mit `while` |
| 18 | Dateien | Lesen und Schreiben von Dateien |
| 19 | Interpreter | Unterschiede zwischen kompilierten und interpretierten Sprachen |
| 20 | Tupel und *list-comphrensions* | Tupel-Zuweisung, Tupel-Rückgabewerte und Tupel-Argumente |
| 21 | Numerische Berechnungen mit Python | NumPy und Matplotlib |
| 22 | Programmierkonzepte in Python und in C | Vergleich der abstrakten Konzepte anhand der beiden Sprachen |

zeigt, um Studierenden zu zeigen, dass die Unterschiede hauptsächlich im Bereich der Syntax liegen. Zugleich wird damit auf die konsekutive Vorlesung *Programmieren 2* übergeleitet, die sich auf die Vermittlung von *C* konzentriert. Diese baut auf den bereits vermittelten Programmierkonzepten auf und adressiert hauptsächlich die neue Syntax, die manuelle Speicherverwaltung und Werkzeugkette von *C*.

Neben den in der Vorlesung durchgeführten Aufgaben, wurden elf wöchentliche Übungen veröffentlicht, die jeweils die neuen Inhalte und Konzepte abdecken. Ab der siebten Übung wurde zusätzlich die Entwicklungsumgebung *PyCharms* eingeführt und in einigen Aufgaben explizit auf deren Verwendung hingewiesen. Die Entwicklungsumgebung wurde auf den Computern im Labor installiert und konnte zu den Übungszeiten genutzt werden. Jedoch haben nach wie vor einige Studierenden trotzdem die Möglichkeit zur Programmierung mittels der web-basierten Entwicklungsumgebung weiterhin genutzt. Am Ende des Semesters wurde in der letzten Übung ein Lego Mindstorms EV3 Roboterarm programmiert.

### 7.1.3 Verwendung der Plattform

Alle Kapitel und Übungen wurden mittels der Plattform als interaktive Dokumente zur Verfügung gestellt. Dazu wurde die Plattform unter einer eigenen Internetadresse [37] veröffentlicht, sodass ein Zugriff innerhalb wie außerhalb der Hochschule möglich ist. Der Zugriff auf die Plattform sowie das Ausführen der Beispiele und Übungen mittels *skulpt* konnte ohne Anmeldung erfolgen. Für alle anderen Aufgaben war eine Registrierung und Anmeldung sowie die Zustimmung der Datenschutzerklärung notwendig, um einen Missbrauch der Sandbox-Umgebung zu verhindern bzw. nachverfolgen zu können. Studierende, die sich nicht registrieren wollten, wären nicht verfügbare Inhalte in anderer Form bereitgestellt worden. In diesem Fall war das allerdings nicht notwendig, da sich alle Studierenden dazu bereiterklärten die Plattform zu nutzen bzw. sich niemand dagegen aussprach. Zum Ende des Semesters wurde zusätzlich ein Skript in einer Druckversion zur Verfügung gestellt.

Darüber hinaus wurde die Plattform zum Sammeln der Benutzerinteraktionen (vgl. Kapitel 6.5) verwendet, die pro Beispiel und Aufgabe erfasst wurden. Insgesamt wurden 24 Aufgaben (vgl. Tabelle 12) in das Skript bzw. in die Vorlesung eingebettet und durchgeführt. Die Integration dieser in die Vorlesung erfolgte dabei nicht strikt nach dem entwickelten Konzept, denn sie sollen teilweise zur Verdeutlichung anderer Inhalte dienen. Das in Kapitel 5 beschriebene Konzept dient nicht dazu für jeden zu vermittelten Inhalt methodisch abzudecken, allerdings konnte die Plattform für weitere Methoden verwendet werden. Dementsprechend werden zur Evaluation des Konzepts

[37] *www.trycoding.io*

**Tabelle 12:** Aufgaben im Unterricht

| Kapitel | Aufgabe | Adressiertes Konzept |
|---|---|---|
| 5 | Variablentausch | Variablen |
| 5 | Tasked Based Programming Learning (vgl. [Fig+16]) | Operatoren |
| 6 | Umwandlung in Fahrenheit | Operatoren |
| 7 | Aufgabe math. Funktionen | Funktionsaufrufe und Operatoren |
| 8 | Polygon mit *Turtle Graphics* (Einführung) | Problembewusstsein für Iteration |
| 8 | Polygon mit `for` | Iteration mit `for` |
| 8 | Haus vom Nikolaus | Iteration mit `for` |
| 10 | Aufgabe Rechteck | Funktionen |
| 10 | Programmablauf | Programmablauf (Funktionen) |
| 10 | Liegt Wert dazwischen? | Funktionen |
| 11 | Negation | Boolesche Ausdrücke |
| 11 | Aussagen formulieren | Boolesche Ausdrücke |
| 11 | Mehrfache Verzweigungen | Bedingte Ausführung |
| 11 | Schaltjahr | Bedingte Ausführung |
| 12 | Inkrementelle Vorgehensweise | Programmkomposition |
| 13 | Restdivision | Iteration mit `while` |
| 14 | Traversierung | Iteration mit `while` |
| 14 | Aufgabe Trav. rückwärts | Iteration mit `while` |
| 14 | Häufigkeit eines Zeichens | Iteration mit `while` |
| 15 | Aufgabe: Eingabe in eine Liste | Iteration mit `for` |
| 20 | Tupel-Argumente | Iteration mit `for` |
| 20 | List-Comprehension | Iteration mit `for` |
| 21 | Vektoren in Python | Iteration mit `for` |

**Tabelle 13:** Finite Iteration: Aufgaben im Unterricht

| Schritt | Realisierte Handlungen |
|---|---|
| Problem | Studierende sollen ein 20-seitiges Polygon mit *Turtle Graphics* zeichnen (vgl. Kapitel 5.2.6). |
| Konzepteinführung | Einführung der `for` Schleife inklusive Veränderungen des Programmablaufs und *Worked Examples* (vgl. Kapitel 5.2.6). |
| **Aufgaben innerhalb der Vorlesung** | |
| Angeleitete Aufgabe | Zeichnen eines Hauses mittels Schleifen (Haus vom Nikolaus) und Diskussion einer Lösung |
| Transferaufgabe | Programmierung eines Lauflichts auf dem *SenseHat* und Diskussion zweier Lösungen |
| Übungen | Zeichnen einer *Mario-Pyramide* (vgl. Kapitel 3.2.2) |

nur einige ausgewählte Beispiele analysiert, bei denen alle Schritte abgedeckt sind.

### 7.1.4 Realisierte Lehrhandlungen

Das in Kapitel 5 beschriebene Konzept wurde an mehreren Stellen in unterschiedlicher Ausführung umgesetzt. Jedoch wurde nicht in jedem Fall das entwickelte Konzept direkt übertragen, sondern an die jeweiligen Situationen und Lernziele angepasst. Besonders zu Beginn der Vorlesung wurden zwar Aufgaben im Unterricht integriert, aber die Konzepte noch nicht strikt problemorientiert eingeführt. Aus diesem Grund werden nur die Lehrhandlungen beschrieben, welche nach dem Konzept umgesetzt wurden. Für die nachfolgende Auswertung werden anschließend die erfassten Daten näher untersucht.

#### Finite Iteration

Die finite Iteration (mit bekannter Anzahl an Durchläufen) stellt das erste Programmierkonzept dar, welches vollständig anhand der Schritte des entwickelten Konzepts eingeführt wurde. Zugleich wurde das Beispiel bereits in Kapitel 5.2.6 ausführlich beschrieben, weswegen an dieser Stelle auf die Beschreibung der Aufgaben verzichtet wird. Tabelle 13 listet die einzelnen Schritte des Konzepts und die realisierten Hand-

lungen im Unterricht auf.

### Funktionen

Das Konzept der Funktionen und wie diese zur Programmkomposition verwendet werden, wurde ebenfalls mittels des entwickelten Konzepts vermittelt. Nachfolgend werden die Realisierungen der einzelnen Schritte kurz beschrieben.

**Problem** Ausgangspunkt der problemorientierten Einführung ist diesmal kein explizites Beispiel, sondern ein Gedankenexperiment. Studierende mussten in den vorherigen Übungen bereits mehrmals mit *Turtle Graphics* verschiedene Elemente, wie beispielsweise Rechtecke und Dreiecke zeichnen. Die Studierenden sollten dabei über die Wiederverwendbarkeit der Anweisungen für beliebige Rechtecke und Dreiecke nachdenken und wie dies mittels Anweisungen realisiert werden kann. Denn eigentlich wiederholen sich beim Zeichnen zweier Rechtecke, nur die Parameter mit denen die jeweiligen Methoden aufgerufen werden. Synthese der Diskussion ist das Kombinieren von logischen Bausteinen und deren Parametrisierung, um ähnliche Abläufe in einem allgemeinen Algorithmus abzubilden.

**Konzepteinführung** Zunächst wurde die Definition einer neuen Funktion vom Lehrenden vorgestellt und anschließend die einzelnen Bestandteile einer Funktion näher erläutert. Anschließend wurde mittels *Live Coding* die Erstellung einer Funktion

```
1  import turtle
2  # Funktionsdefinition
3  def quadrat(stift, laenge):
4      stift.forward(laenge)
5      stift.right(90)
6      stift.forward(laenge)
7      stift.right(90)
8      stift.forward(laenge)
9      stift.right(90)
10     stift.forward(laenge)
11
12 stift1 = turtle.Turtle()
13 quadrat(stift1, 30) # Aufruf
14 quadrat(stift1, 40) # Aufruf
15 quadrat(stift1, 50) # Aufruf
```

**Quellcode 6**: Funktion zum Zeichnen eines Quadrats mit beliebiger Länge

zum Zeichnen von Quadraten mittels *Turtle Graphics* und die Ergebnisse bei unterschiedlicher Parametrisierung gezeigt (vgl. Beispiel in Quellcode 6). Danach wurde

die Änderung des Programmablaufs anhand mehreren Beispielen und unter Verwendung von *PythonTutor* gezeigt. Die Geltungsbereiche von Variablen und Parametern wurden wiederum anhand von Beispielen und mittels *PythonTutor* visualisiert. Abschließend wurde der Unterschied zwischen Funktionen mit einem Rückgabewert und ohne diskutiert und anhand zweier Beispiele vorgeführt.

**Aufgaben innerhalb der Vorlesung** Nach der Einführung wurden die Studierenden aufgefordert, eine Funktion zum Zeichnen eines Rechtecks zu definieren. Dabei wurde bereits Quellcode mit *Subgoal Labels* in der Entwicklungsumgebung zur Verfügung gestellt, sodass die Studierenden nur die Funktionsdefinition und die Anweisungen zum Zeichnen des Rechtecks implementieren mussten (vgl. Quellcode 7). Im

```
1  import turtle
2
3  # Funktion rechteck definieren
4
5
6  # Aufruf der Funktion
7  laenge = 40
8  breite = 60
9  stift = turtle.Turtle()
10 rechteck(stift, laenge, breite)
```

**Quellcode 7:** Funktion zum Zeichnen eines Rechtecks

Anschluss wurde eine studentische Lösung im Plenum diskutiert.

Nachfolgend wurde die `return` Anweisung und deren Verwendung in Bezug auf Funktionen sowie Möglichkeiten zur Verwendung temporärer Variablen zur Fehlersuche gezeigt. Aus Zeitmangel konnte keine weitere Übung in den Unterricht integriert werden. Zuletzt wurde ein Beispiel zum Zeichnen einer Blume mit mehreren Blättern erklärt, welches anschließend in den Übungen wieder aufgegriffen wurde.

**Übungen** In den dazugehörigen Übungen wird das Thema Funktionen in verschiedenen Aufgaben wieder aufgegriffen. Zuerst wird eine Aufgabe aus der vorherigen Übung mittels einer Funktion gelöst. Anschließend müssen zwei weitere Aufgaben mit dem Ziel neue Funktionen zu definieren und Rückgabewerte zu verwenden, bearbeitet werden. Schlussendlich muss eine Funktion zum Zeichnen von Blumen umgesetzt werden, wobei auf das bereits im Unterricht angesprochene Beispiel zurückgegriffen werden soll (vgl. Abbildung 61). Alle Aufgaben steigern dabei den jeweiligen Umfang und Komplexität, indem die Anzahl der Parameter erhöht wird und verschachtelte Aufrufe notwendig werden.

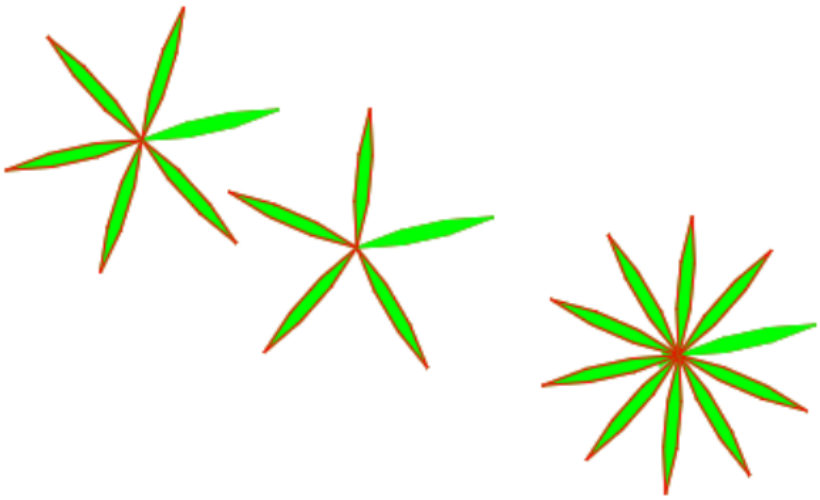


**Abbildung 61**: Mehrere *Blumen* mit unterschiedlicher Blätteranzahl

### Boolesche Ausdrücke

Das nächste Konzept bzw. stellen die Booleschen Ausdrücke dar, die für das darauf aufbauende Konzept der Bedingungen benötigt werden. Beide Konzepte wurden in derselben Vorlesung behandelt und in der dazugehörigen Übung durch mehrere Aufgaben wiederholt.

**Problem** Ähnlich zu den Funktionen wurde das Problem diskursiv eingeführt. Ausgehend von Anwendungen aus dem echten Leben, wie z. B. Fernbedingungen oder Programme, die unterschiedliche Berechnungen in Abhängigkeit von Eingabewerten durchführen sollen, wurden die Notwendigkeit zur Formulierung logischer und relationaler Aussagen abgeleitet. Zugleich wurde die Abhängigkeit bzw. die enge Verbindung zu der bedingten Ausführung vermittelt, da diese mittels Boolescher Ausdrücke formuliert werden.

**Konzepteinführung** In dieser Phase wurden zuerst die beiden Wahrheitswerte *Wahr* und *Falsch* eingeführt und anschließend der Datentyp *bool* in Python gezeigt. Python ist eine dynamisch typisierte Sprache, welche jedoch die explizite Typumwandlung erfordert, weswegen die jeweiligen Typen immer anhand der Funktionen zur Umwandlung erklärt werden. Danach wurden die relationalen Operatoren gezeigt, welche teilweise aus der Mathematik bekannt sind. Einzig die Operatoren für Gleichheit und Ungleichheit und deren syntaktische Darstellung ist dabei neu. Als Nächstes wur-

den die drei logischen Operatoren in Python eingeführt und anhand eines Beispiels in Verbindung mit relationalen Operatoren vorgeführt.

**Aufgaben innerhalb der Vorlesung** In einer ersten angeleiteten Übung mussten Studierenden den Booleschen Ausdruck `a = not (x < y)` invertieren. Eine kurze händische Abstimmung hat gezeigt, dass dem Großteil der Studierenden bereits die boolesche Algebra und Regeln aus dem Mathematikunterricht bekannt war. Hierbei wurde explizit die Zusammenarbeit in Gruppen betont, sodass sich Studierende gegenseitig helfen. Bei der Besprechung der Lösung wurde über eine Meldung diskutiert, ohne den Mechanismus zum Einreichen zu verwenden. Anschließend fasste der Lehrende die Lösung nochmals kurz zusammen.

Nachfolgend wurde eine schwierigere Übung durchgeführt, bei der die Studierenden vier verschiedene Boolesche Ausdrücke formulieren mussten:

Aufgabe - Relationale und logische Operatoren

*Formulieren Sie folgende Aussagen und weisen Sie das Ergebnis jeweils einer Variablen zu:*

$$x < y \land x \geq 20 \tag{7.1}$$
$$0 < x < y \tag{7.2}$$
$$x > 10 \lor y < \frac{x}{2} \tag{7.3}$$
$$x \leq 3 \land \neg(x < 0) \tag{7.4}$$

Die Aussagen wurden bewusst in mathematischer Notation angegeben, sodass die Studierenden sich erst Gedanken über die Lösung machen und diese anschließend zu implementieren sollten.

Für die Diskussion der Lösung wurde eine studentische Einreichung verwendet und diese anschließend unter der Aufgaben als Musterlösung zur Verfügung gestellt. Dazu wurde die Möglichkeit zur schnellen Bearbeitung der Inhalte genutzt und diese sofort nach der Durchführung hinzugefügt.

**Übungen** Die Formulierung Boolescher Ausdrücke wurde dabei explizit in einer weiteren Aufgabe, welche die Umsetzung sechs verschiedener Aussagen erforderte, geübt. Studierende sollten dazu fünf Ganzzahlen einlesen und diese in die Variablen von *z1* bis *z5* abspeichern und nachfolgende Aussagen umsetzen:

1. $z$ enthält nicht den größten Wert

2. Kein $z$ liegt im Bereich von $5..10$ (inklusive)
3. Mindestens ein $z$ hat einen Wert $> 2$
4. Nicht alle $z$ haben das gleiche Vorzeichen (0 ist positiv)
5. Genau ein $z$ hat einen Wert $< 0$
6. Alle $z$, die $> 0$ sind, sind auch $> 10$

Zur Überprüfung wurden den Studierenden anschließend sechs verschiedene Wertereihen für die Eingaben und die erwarteten Ergebnisse bereitgestellt, sodass sie ihre Aussagen überprüfen konnten. Darüber hinaus wurden weitere Aufgaben zu Bedingungen zur Verfügung gestellt, die ebenfalls die Formulierung Boolescher Ausdrücke erfordern.

#### Bedingungen

Aufbauend auf den Booleschen Ausdrücken wurde die bedingte Ausführung als nächster Baustein mit Verweis auf die grundlegenden Anweisungen (vgl. Kapitel 7.1.2) eingeführt.

**Problem** Die problemorientierte Einführung wurde dabei bereits bei den Booleschen Ausdrücken vorgenommen, weswegen hier nur auf die Notwendigkeit eines Konstrukts zur Formulierung verschiedener Ablaufpfade in einem Programm eingegangen wurde.

**Konzepteinführung** Anhand einfacher Beispiele und *Live Coding* wurden die bedingte Ausführung mittels `if`, der alternative Programmablauf mittels `else` und mehrfacher Verzweigungen gezeigt. Dabei wurden wiederum die unterschiedlichen Pfade, die durchlaufen werden, betont. Besonders der Aspekt, dass jeweils nur ein einzelner Pfad durchlaufen wird, wurde in mehreren Beispielen verdeutlicht.

**Aufgaben innerhalb der Vorlesung** Zum Einüben dieses Konzepts wurde eine Aufgabe gestellt, bei der die Studierenden einen Buchstaben von der Konsole einlesen und dann gemäß einer Tabelle Texte auf der Konsole ausgeben mussten. Entspricht der eingelesene Buchstabe keiner der Möglichkeiten in der Tabelle, so sollte *Ungültige Eingabe* ausgegeben werden. Somit mussten die Studierenden alle bereits vorgestellten Ausprägungen der bedingten Ausführung zur Lösung dieser Aufgabe kombinieren. Jedoch wurde bereits in der Konzepteinführung der mehrfachen Verzweigung ein

```
eingabe = input("Zeichen:")

if eingabe == "F":
    print("Foxtrott")
elif eingabe == "U":
    print("Uniform")
elif eingabe == "K":
    print(Kilo)
elif eingabe == "L":
    print("Lima")
```

**Quellcode 8**: Studentische Lösung zur bedingten Ausführung

Beispiel gezeigt, welches auf diese neue Aufgabe übertragen werden konnte. Quellcode 8 zeigt eine der diskutieren studentischen Lösungen, bei der die letzte Abfrage fehlt und sie somit nicht die Nachricht *Ungültige Ausgabe* ausgibt. Eine Musterlösung wurde im Anschluss an die Aufgabe im Skript unter der Aufgabe angehängt.

Nach der Diskussion wurde die Möglichkeit zur Verschachtelung von Bedingungen anhand mehrerer Beispiele gezeigt und wie diese sich ggf. mittels logischer Operatoren auflösen lassen. Zum Ende der Vorlesung sollten die Studierenden eine Bedingung bzw. Booleschen Ausdruck formulieren, der prüft, ob ein eingegebenes Jahr ein Schaltjahr ist. Dabei wurden die Studierenden mit folgender Aufgabenstellung konfrontiert:

> Ein Jahr dauert nach dem Sonnenkalender etwas mehr als 365 Tage - nämlich 365 Tage, 5 Stunden, 48 Minuten und 45,25 Sekunden. Das ist knapp ein Viertel Tag länger als die 365 Tage eines normalen Kalenderjahres. Um den Unterschied auszugleichen, wird alle 4 Jahre ein zusätzlicher Tag (Schalttag) im Kalenderjahr (der 29. Februar) eingefügt. Da das aber etwas zu viel ist, verzichtet man alle 100 Jahre auf den Schalttag. Alle 400 Jahre weicht man von dieser Verzichtregel ab und fügt den Schalttag doch ein. [Kla15]

Eine mögliche Lösung wurde im Unterricht besprochen, aber aus Zeitmangel konnten keine weiteren studentischen Lösungen besprochen werden. Die Studierenden hatten jedoch die Möglichkeit ihre Lösung automatisch überprüfen zu lassen, da bei dieser Aufgaben Testfälle hinterlegt wurden.

**Übungen** In der Übung wurde die bedingte Ausführung in unterschiedlichen Aufgaben mit steigender Schwierigkeit thematisiert. Zuerst sollte überprüft werden, ob eine Zahl *gerade* bzw. *ungerade* ist. Anschließend sollten zwei Zahlen miteinander verglichen werden und ausgegeben werden, ob die erste Zahl gleich der zweiten Zahl, größer oder kleiner ist. In der nächsten Aufgabe mussten textuelle Bewertungen in Abhängigkeit von Klausurnoten ausgegeben werden (mehrfache Verzweigungen). Zum

Schluss mussten die Studierenden nach Eingabe dreier Seitenlängen überprüfen, ob mit diesen ein Dreieck gebildet werden kann. Dies erfordert die Kombination von Funktionen, Booleschen Ausdrücken und ggf. bedingter Ausführung.

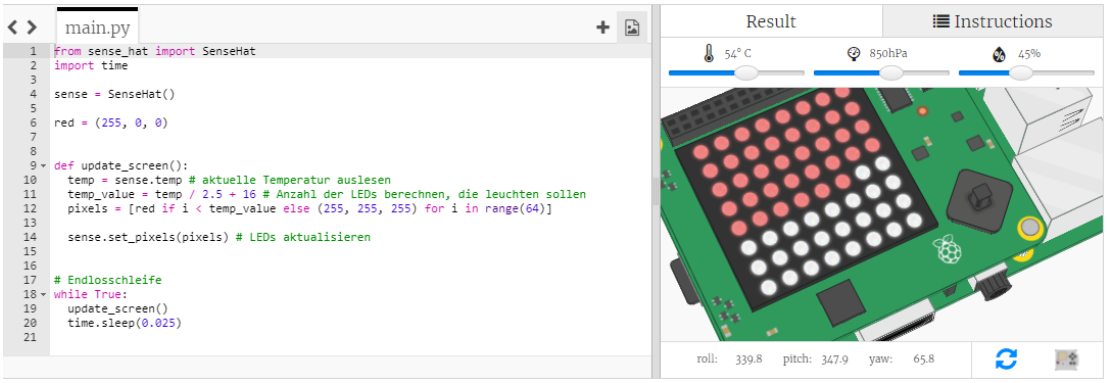
#### Infinite Iteration

Das letzte Programmierkonzept, welches ebenfalls anhand des entwickelten Konzepts eingeführt wurde, ist die infinite Iteration. Bei dieser Art der Wiederholung ist in der Regel die Anzahl der Durchläufe zu Beginn nicht eindeutig bzw. iterieren solange bis eine Bedingung zutrifft oder die Iteration abgebrochen wird. Eine Besonderheit dieser Iteration ist, dass die finite Iteration vollständig über diese abgedeckt werden kann.

**Problem** Bei der problemorientierten Einführung wurde wiederum auf ein Programmbeispiel verzichtet. Stattdessen wurde eine Diskussion über Programme, die z. B. integriert in Hardware laufen angestoßen. Dabei wurden der *SenseHat* und *Arduino* wieder aufgegriffen, die nach dem Start bzw. Bootvorgang ein bestimmtes Programm in einer Schleife wiederholen. Erst mit dem Auslösen eines Ereignisses wird dieser Ablauf unterbrochen und ggf. etwas anderes ausgeführt. Die Studierenden sollen dabei weitere Beispiele nennen, bei denen dieses Verhalten erkennbar ist.

**Konzepteinführung** Die Konzepteinführung nimmt aufgrund der Komplexität und der vielfältigen Möglichkeiten einen größeren Raum ein. Zu Beginn wurden Themen wie das Inkrementieren und Dekrementieren erläutert und gezeigt, da diese bisher nicht benötigt wurden. Besonders bei der finiten Iteration mit `for` wird über eine Reihe an Werten iteriert und es bedarf keiner Veränderung einer Laufvariablen wie es beispielsweise bei *C* erforderlich ist. Diese werden dann anhand mehrerer Beispiele zur Erstellung eines *Countdowns* gezeigt, wobei der Programmablauf anhand von Kontrollflussdiagrammen gezeigt wird. Anschließend wird ein weiteres Beispiel erklärt, welches eine Endlosschleife zeigt, die in Abhängigkeit der Temperatur LEDs auf einer LED-Matrix ein- bzw. ausschaltet. In Abbildung 62 ist die LED-Matrix, die in Abhängigkeit der ausgelesenen Temperatur die LEDs rot einfärbt und der Quelltext abgebildet.

Ausgehend von der Endlosschleife wird die `break` Anweisung, die das Beenden der Schleife ermöglicht, vorgestellt. Dabei wird das wiederholte Einlesen einer Zeichenkette, bis diese den Wert `fertig` beinhaltet, exemplarisch vorgeführt. Dies ist der Übergang zu einer längeren *Live Coding* Sequenz, in der schrittweise das Newtown-Verfahren zum Berechnen von Quadratwurzeln algorithmisch dargelegt und implementiert wird.




```python
from sense_hat import SenseHat
import time

sense = SenseHat()

red = (255, 0, 0)


def update_screen():
    temp = sense.temp # aktuelle Temperatur auslesen
    temp_value = temp / 2.5 + 16 # Anzahl der LEDs berechnen, die leuchten sollen
    pixels = [red if i < temp_value else (255, 255, 255) for i in range(64)]

    sense.set_pixels(pixels) # LEDs aktualisieren


# Endlosschleife
while True:
    update_screen()
    time.sleep(0.025)
```

**Abbildung 62**: Simulation des Auslesens der Temperatur in einer Endlosschleife (*SenseHat*)

## Aufgabe - Newton-Verfahren

**Aufgabe**: Schleifen werden in Programmen oft dazu verwendet, numerische Ergebnisse dadurch zu berechnen, dass von einer ungefähren Antwort ausgegangen wird, die dann schrittweise optimiert wird.
Das Newton-Verfahren ist zum Beispiel eine Möglichkeit zur Berechnung von Quadratwurzeln. Angenommen, Sie möchten die Quadratwurzel von a wissen. Wenn Sie mit einer beliebigen Schätzung $x$ beginnen, können Sie mit der folgenden Formel eine bessere Schätzung berechnen.

$$y = \frac{x + \frac{a}{x}}{2}$$

```python
# Näherung mit einem Schritt
a = 4.0
x = 3.0
y = (x + a/x) / 2
print(y)
```

```python
# Näherung mit n Schritten
# bis gewünschte Genauigkeit
# erreicht wurde
a = 4
x = 3
epsilon = 0.0000001
while True:
    print(x)
    y = (x + a/x) / 2
    if abs(x - y) < epsilon:
        break
    x = y
```

**Quellcode 9**: Mehrschrittige Entwicklung einer Lösung (rechts) zum Newton-Verfahren

Quellcode 9 zeigt dabei den Ausgangsentwurf (links), welcher die Näherung in einem Schritt zeigt. Diese wird dann im Unterricht durch *Live Coding* bis zur mehrschrittigen

main.py | Terminal | Turtle-Ausgabe | Ü5 - Spiralen

```
1  import turtle
2
3  def spirale(stift, n, winkel):
4      for step in range(n):
5          stift.forward(step)
6          stift.left(winkel)
7
8  def spirale2(stift, n, winkel):
9      # mit i=0 startend wird die Spirale von
10     # innen gezeichnet
11     i = 0
12     while i < n:
13         stift.forward(i)
14         stift.left(winkel)
15         i = i + 1
16
17 t = turtle.Turtle()
18 t.speed('fast')
19 spirale2(t, 140, 61)
```

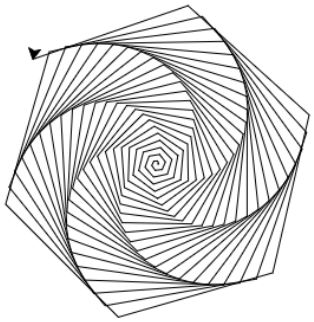


**Abbildung 63**: Aufgabe zur Umformung einer *for* Schleife in eine *while* Schleife

Lösung (rechts) entwickelt.

**Aufgaben innerhalb der Vorlesung** Im Anschluss wurde eine angeleitete Aufgabe im Unterricht durchgeführt. Ziel bei der Aufgabe ist es, eine Ganzzahl in das Binärsystem (Zeichenkette) umzuwandeln und auszugeben. In der Aufgabenstellung wurde dazu auf die Restdivision im Kapitel der Zahlensysteme verwiesen und bereits eine teilweise Lösung vorgegeben (vgl. Quellcode 10). Im Anschluss wurde eine stu-

```
1  zahl = 343
2  dual = ''
3
4  while zahl > 0:
5      pass
6
7  print(dual)
```

**Quellcode 10**: Studentische Lösung zur bedingten Ausführung

dentische Lösung im Unterricht diskutiert und auf weitere Fragen geantwortet.

**Übungen** In der dazugehörigen Übung wurden mehrere Aufgaben mit steigender Schwierigkeit zur Iteration mit *while* angeboten. Darunter fällt unter anderem die Umformung einer Schleife mit *for* in eine *while* Schleife (vgl. Abbildung 63). Hierbei ist bereits die Kombination der neuen Konzepte mit bereits eingeführten Konzepten, wie beispielsweise Funktionen, zu sehen. Eine Steigerung zur vorherigen Aufgabe stellt die Umsetzung eines Algorithmus, um sämtliche nichttriviale Teiler einer beliebigen Zahl zu finden und auszugeben, dar.

#### Vorbereitung der Vorlesung

Die in der Übung gesammelten Daten (siehe Kapitel 6.5) wurden zur Vorbereitung der nächsten Vorlesungen verwendet und daraus jeweils mehrere Wiederholungsfragen erstellt. Diese wurden zu Beginn der Vorlesung als aktivierendes Element eingesetzt. In den Fragen wurden bewusst häufige Fehler und identifizierte falsch verstandene Konzepte thematisiert, welche durch die Quelltextanalyse in der Entwicklungsumgebung gefunden wurden. Darüber hinaus wurden die Aufgaben aus den Übungen in der darauffolgenden Vorlesung durchgesprochen und ggf. studentische Lösungen diskutiert.

## 7.2 Untersuchung der studentischen Partizipation

Für die Evaluation der Plattform und des Konzeptes soll die studentische Partizipation im Zusammenhang mit den Prüfungsergebnissen untersucht werden. Ziel des Konzepts ist die Steigerung der studentischen Partizipation im Unterricht und die studentische Auseinandersetzung mit Programmieren. Aus diesem Grund soll auf Basis der gesammelten Daten durch die Bearbeitung von Aufgaben während des Unterrichts zuerst festgestellt werden, ob die Studierenden dies zur Partizipation genutzt bzw. sich mit dem Programmieren auseinandergesetzt haben. Ausgehend von der dritten Forschungsfrage (siehe Punkt 3), welche die Integration von Schreib- und Leseaktivitäten im Unterricht thematisiert, wurde folgende Hypothese abgeleitet, die anschließend untersucht wird.

### 7.2.1 Beschreibung der Datenerfassung

Zur Überprüfung der studentischen Partizipation werden die in den Unterricht durchgeführten Aufgaben näher betrachtet. Die Interaktionen von angemeldeten Benutzern mit der web-basierten Entwicklungsumgebung wurden dabei erfasst und abgespeichert. Insgesamt konnten so **47806** Datenpunkte gesammelt werden. Für jeden Datenpunkt wurden folgende Merkmale erfasst, sofern diese zur Verfügung standen:

- Eindeutige Kennung der Aufgabe bzw. des Beispiels
- Name der Aufgabe bzw. des Beispiels
- Art des Events
  - Fehler bei der Ausführung
  - Erfolgreiche Ausführung

  - Fehler in der Entwicklungsumgebung
  - Durchführung der automatisierten Aufgabenüberprüfung
- Fehlernachricht (wenn verfügbar)
- Fehlertyp (wenn verfügbar)
- Zeitstempel
- Eindeutige Kennung des Benutzers

Die Aufgaben aus den Vorlesungen und Übungen wurden zusätzlich aufbereitet, sodass jedem Datenpunkt weitere Informationen zugeordnet werden können. Jeder Aufgabe wurde, wenn möglich, das jeweilige primäre Programmierkonzept zugeordnet (vgl. Tabelle 12), um später detaillierte Analysen durchführen zu können. Die Zuordnung der Konzepte zu den Aufgaben in den Übungen ist Tabelle 49 zu entnehmen.

Weiterhin wurde jedem Datenpunkt die jeweilige Note aus den Ergebnissen der Prüfung zugeordnet, sofern der Studierende sich auf der Plattform registriert bzw. an der Prüfung teilgenommen hatte. Tabelle 14 fasst die Anzahl der Studierenden, die an der

**Tabelle 14**: Übersicht der Teilnahme an der Prüfung

| | **Anzahl** | **Registriert** | **Nicht Registriert** |
|---|---|---|---|
| Angemeldet | 102 | 87 | 15 |
| Teilgenommen | 73 | 66 | 7 |
| Nicht teilgenommen | 29 | 21 | 8 |

Prüfung teilgenommen haben und zugleich auf der Plattform registriert waren, zusammen. Davon ausgehend stehen Datenpunkte und Prüfungsergebnisse für insgesamt **66** Studierende zur Verfügung. Sieben Studierende haben an der Prüfung teilgenommen, ohne auf der Plattform registriert zu sein. Davon haben fünf Studierende nicht bestanden und die anderen beiden konnten die Noten *2,7* und *3,3* erreichen. Keiner der nicht registrierten Studierenden hat nach nicht angezeigten Inhalten und Aufgaben, wie beispielsweise Quelltextvorgaben, nachgefragt. Für die weitere Auswertung wurden die Noten den jeweiligen Datensätzen zugeordnet und pseudonymisiert, wobei die Identität durch eine Zahlen- und Buchstabenkombination ersetzt wurde.

### 7.2.2 Ergebnisse

Für die Auswertung werden verschiedene Untermengen der Datensätze betrachtet, um ein möglichst vollständiges Bild über die Aktivitäten der Studierenden im Unterricht,

aber auch in den dazugehörigen Übungen, zu geben. Denn das in Kapitel 5 entwickelte Konzept hat zum Ziel die starke Trennung zwischen Übungen und Vorlesung zu minimieren. Aus diesem Grund sind die Aktivitäten außerhalb des Unterrichts genauso relevant, besonders da die Plattform die Bearbeitung von überall aus ermöglicht. Die Auseinandersetzung mit Programmieren sollte idealerweise nicht nach dem Ende der Vorlesung aufhören, sondern kontinuierlich über das Semester stattfinden, weswegen diese Daten nachfolgend detaillierter betrachtet werden. Darunter fällt zuerst die Beteiligung bei den Aufgaben im Unterricht, mögliche Korrelationen zwischen der Beteiligung und der Note und ein Vergleich zwischen den bearbeiteten Konzepten und Notenstufen.

### Beteiligung im Unterricht

Für eine Einschätzung der studentischen Partizipation im Unterricht wird zuerst die Anzahl der Studierenden ermittelt, die sich während der Vorlesung mit den Aufgaben beschäftigt haben. Dabei zählt jede Art der Interaktion mit der Entwicklungsumgebung als Beschäftigung, wobei nur die Aufgaben betrachtet werden, die in Kapitel 7.1.4 beschrieben sind und nach dem entwickelten Konzept im Unterricht durchgeführt wurden. Ausgehend von durchschnittlich 70 anwesenden Studierenden in der Vorlesung, haben ein Drittel bis die Hälfte jeweils die Aufgaben digital bearbeitet. Zusätzlich hat ein weiterer Teil die Aufgaben auf Papier bearbeitet und ggf. weitere Studierende welche eine lokal installierte Entwicklungsumgebung genutzt haben. Die Anzahl der Studierenden, welche mit den Aufgaben interagiert haben, steigt zusätzlich, wenn die Anzahl über das komplette Semester ermittelt wird.

Das Konzept sieht die Wiederholung der neuen Konzepte in den Übungen vor (vgl. Abbildung 19). Aus diesem Grund werden ebenfalls die Aktivitäten der Studierenden in den Übungsaufgaben und insgesamt betrachtet, da diese eine breitere Datenbasis darstellen, um Aussagen abzuleiten. Eine Übersicht über die Aktivitäten und Beteiligung für alle Aufgaben, die im Unterricht durchgeführt wurden, befindet sich im Anhang in Tabelle 50. Ebenfalls wurden zusätzlich die Noten aus der Klausur mit in die Auswertung aufgenommen, um ggf. Unterschiede zwischen einzelnen Gruppen zu identifizieren.

### Anzahl der Aktivitäten je Note

In den folgenden Diagrammen wurden die Anzahlen der Aktivitäten und bearbeiteten Aufgaben pro Studierender gezählt und im Kontext der erreichten Prüfungsleistung dargestellt. Dabei wurden einerseits die Aufgaben im Unterricht und andererseits al-

**Tabelle 15**: Anzahl der Studierenden je Aufgabe

| Konzept & Aufgabe | Unterricht[1] | Insgesamt[2] |
|---|---|---|
| **Finite Iteration** | | |
| Polygon mit *Turtle* | 22 | 38 |
| Polygon mit *for* | 25 | 41 |
| Haus vom Nikolaus | 31 | 39 |
| **Funktionen** | | |
| Aufgabe Rechteck | 34 | 46 |
| **Boolesche Ausdrücke** | | |
| Negation | 9 | 22 |
| Aussagen formulieren | 36 | 44 |
| **Bedingungen** | | |
| Mehrfache Verzweigungen | 34 | 42 |
| Schaltjahr | 13 | 20 |
| **Infinite Iteration** | | |
| Restdivision | 33 | 41 |

[1] Anzahl der Studierenden, welche die Aufgabe im Unterricht bearbeitet haben.
[2] Gesamte Anzahl der Studierenden, die die Aufgabe bearbeitet haben.

le Aufgaben zusammen betrachtet. In den folgenden Streudiagrammen wurden allen Studierenden die Note *6* zugewiesen, die nicht an der Prüfung teilgenommen haben.

Abbildung 64 zeigt dabei die Anzahl der Aktivitäten und der bearbeiten Aufgaben, die in der Vorlesung durchgeführt wurden, je Note und Studierenden an. Bei den Studierenden, welche die Prüfung nicht bestanden bzw. nicht an dieser teilgenommen haben, ist eine geringe Varianz bei der Anzahl der Aktivitäten sichtbar. Auffällig ist, dass einige Studierende sehr gute Prüfungsleistungen erreicht haben, ohne die Plattform und die bereitgestellten Aufgaben web-basiert zu bearbeiten. Die Anzahl der Aktivitäten der Studierenden, die eine Prüfungsnote von *4,0* oder *3,7* erreicht haben, bewegen sich eher unter *100* Aktivitäten. Studierende, welche die Prüfung nicht bestanden haben, liegen, bis auf eine Ausnahme, unter *50* Aktivitäten im Unterricht. Im Bereich der Noten ab *3,3* schwankt die Anzahl der Aktivitäten stärker und liegen teilweise über *100* Aktivitäten. Die Anzahl der bearbeiteten Beispiele (vgl. Abbildung 64 unten) zeigt ein ähnliches Bild. Während die Studierenden mit einer Note von *4,4* eine sehr breit gestreute Anzahl an bearbeiteten Aufgaben zeigen, ist bei den besseren Studierenden eine Tendenz zu einer höheren Anzahl erkennbar. Auffällig ist in beiden Diagrammen, dass Studierende mit einer Note von *3,7* sich wenig beteiligt und wenige Aufgaben im Unterricht bearbeitet haben. Besonders im Vergleich mit den Studierenden mit einer Note von *3,3* ist ein größerer Unterschied in den Anzahlen zu erkennen. Gleichzeitig

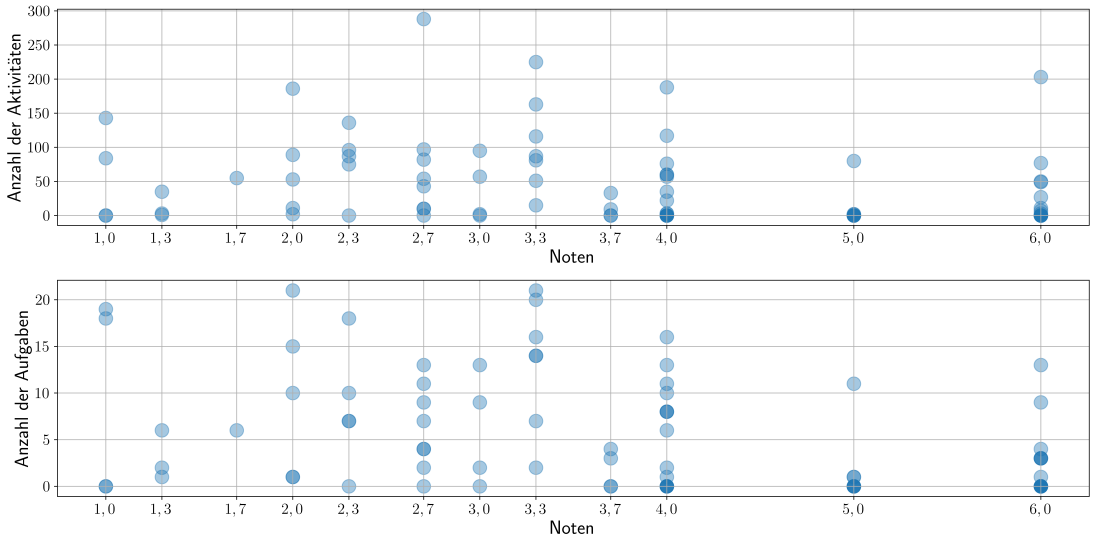



**Abbildung 64:** Anzahl der Aktivitäten (oben) und bearbeiteten Aufgaben (unten) im Unterricht je Note

gibt es Studierende mit guten Leistungen, die sich ebenfalls nur wenig beteiligt haben.

Abbildung 65 zeigt die Anzahl aller Aktivitäten und bearbeiteten Aufgaben je Note und Studierenden. Bei der Anzahl der Aktivitäten (vgl. Abbildung 65 oben), die über alle Aufgaben berechnet ist, sind Unterschiede etwas deutlicher zu erkennen. Studierende mit einer Note von *5,0*, *4,0* oder *3,7* liegen meist unter der Marke von *500* Aktivitäten. Studierende mit besseren Noten liegen teilweise darüber, wobei es hier eine starke Varianz gibt. Diese Tendenz wird bei der Anzahl der Aufgaben deutlicher, da ab der Notenstufe *3,3* nur bei wenigen Studierenden die Anzahl unter *30* liegt.

Zusätzlich zur Aufgliederung der einzelnen Notenstufen wurden die Studierenden anhand der Note in drei verschiedene Gruppen aufgeteilt. Bei der vorherigen Aufgliederung stehen bei manchen Teilnoten nur wenige Datenpunkte zur Verfügung, weswegen diese in andere Gruppen mit aufgenommen wurden. Dabei wurden die erste Gruppe von *1,0* bis *1,7*, die zweite von *2,0* bis *3,3* und die restlichen von *3,7* bis *5,0* zusammengefasst. Studierende, die nicht an der Prüfung teilgenommen haben, wurden für diese Auswertung nicht berücksichtigt. Abbildung 66 zeigt die Anzahl der bearbeiteten Aufgaben in Bezug auf die gruppierte Notenstufe, die über die Punkte berechnet wurde. Im Diagramm ist ein linearer Zusammenhang zwischen der Anzahl der Aufgaben und der Note erkennbar, wenn auch in den einzelnen Gruppen eine gewisse Varianz vorhanden ist. Dieser Zusammenhang wird im nächsten Kapitel tiefer gehend betrachtet, indem die Korrelationen zwischen Aktivitäten, der Note und weiteren Merkmalen berechnet und auf ihre Signifikanz hin überprüft werden.

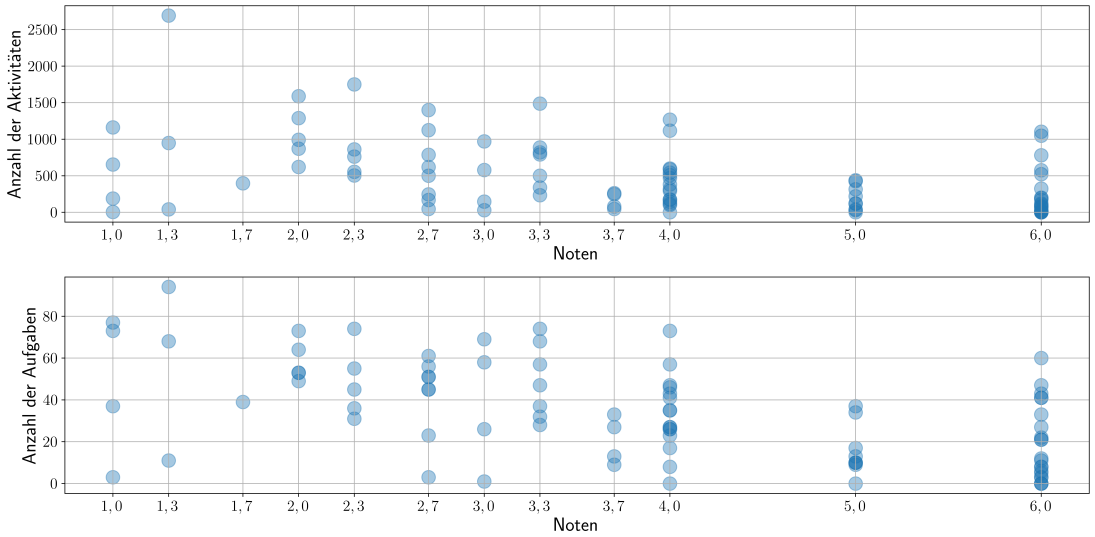



**Abbildung 65**: Anzahl der Aktivitäten und bearbeiteten Aufgaben insgesamt je Note

### Korrelationen

Ausgehend von den Unterschieden zwischen der Anzahl der Aktivitäten bzw. der Aufgaben und der Note, wurden weitere Merkmale aus den Daten abgeleitet. Anschließend wurde der Korrelationskoeffizient zwischen den einzelnen Merkmalen berechnet, um den Grad des Zusammenhangs zu bestimmen. Studierende, die nicht an der Prüfung teilgenommen haben, werden bei dieser Auswertung nicht berücksichtigt. Bei den Daten handelt es sich um metrische Daten, bei denen ein Zusammenhang (vgl. Kapitel 2.4) vermutet wird, weswegen der Rangkorrelationskoeffizient verwendet wird [BS10, S.156]. Es kann hier jedoch kein Pearson-Korrelationskoeffizient verwendet werden, da nicht alle zugrundeliegenden Daten normal verteilt sind.

Neben den Aktivitäten wurden weitere Merkmale aus den Datenpunkten für jeden Studierenden berechnet:

- Letzter Login auf der Plattform (kurz *Letzter Login*),
- Differenz zwischen der Registrierung auf der Plattform und dem letzten Login (kurz *Differenz LL. R.*),
- Anzahl der bearbeiteten Aufgaben insgesamt (kurz *Aktivitäten Vorlesung*),
- Anzahl der bearbeiteten Aufgaben in der Vorlesung (kurz *Anzahl Aufgaben Vorl.*),
- Anzahl der unterschiedlichen Konzepte, die bearbeitet wurden (kurz *Anzahl Konzepte*),

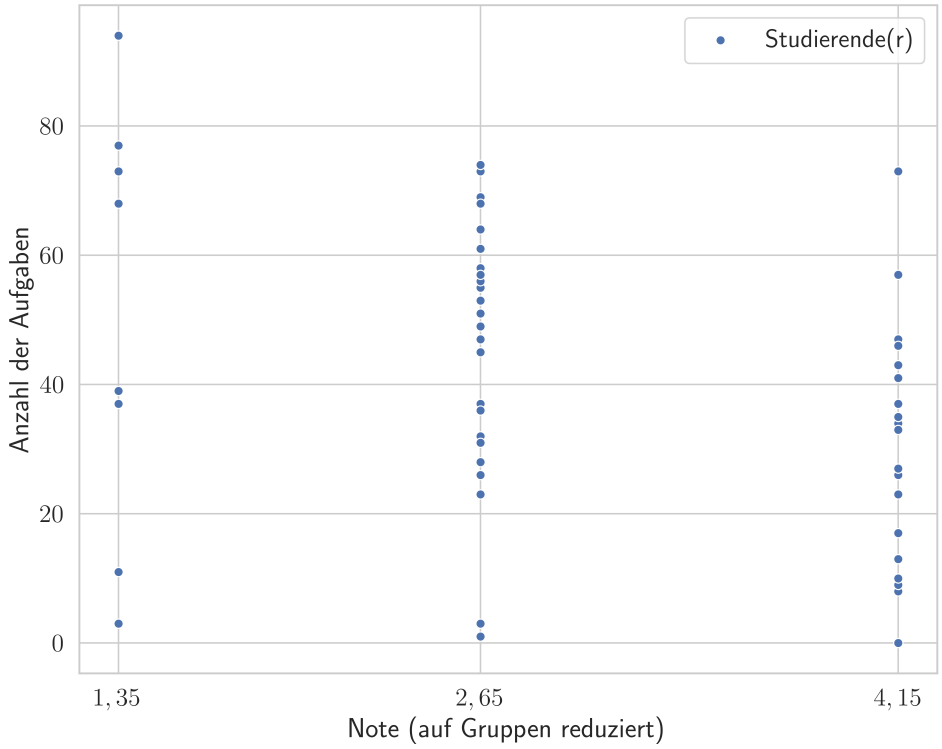



**Abbildung 66**: Anzahl der bearbeiteten Aufgaben insgesamt je Gruppe

| Merkmal | Aktivitäten | Note | Letzter Login | Differenz LL. R. | Anzahl Aufgaben | Aktivitäten Vorlesung | Anzahl Aufgaben Vorl. | Anzahl Konzepte | Tage |
|---|---|---|---|---|---|---|---|---|---|
| Aktivitäten | 1,00 | *-0,47* | 0,40 | 0,64 | 0,90 | 0,79 | 0,77 | 0,89 | 0,89 |
| Note | -0,47 | 1,00 | -0,59 | -0,58 | -0,53 | -0,45 | -0,49 | -0,47 | -0,56 |
| Letzter Login | 0,40 | *-0,59* | 1,00 | 0,68 | 0,41 | 0,37 | 0,40 | 0,38 | 0,48 |
| Differenz LL. R. | 0,64 | *-0,58* | 0,68 | 1,00 | 0,64 | 0,54 | 0,60 | 0,61 | 0,67 |
| Anzahl Aufgaben | 0,90 | *-0,53* | 0,41 | 0,64 | 1,00 | 0,81 | 0,89 | 0,93 | 0,92 |
| Anzahl Aktivitäten Vorl. | 0,79 | *-0,45* | 0,37 | 0,54 | 0,81 | 1,00 | 0,93 | 0,77 | 0,77 |
| Anzahl Aufgaben Vorl. | 0,77 | *-0,49* | 0,40 | 0,60 | 0,89 | 0,93 | 1,00 | 0,83 | 0,84 |
| Anzahl Konzepte | 0,89 | *-0,47* | 0,38 | 0,61 | 0,93 | 0,77 | 0,83 | 1,00 | 0,86 |
| Tage | 0,89 | *-0,56* | 0,48 | 0,67 | 0,92 | 0,77 | 0,84 | 0,86 | 1,00 |

**Tabelle 16**: Korrelationen der Aktivitäten der Studierenden

- und Anzahl der Tage, an denen Aktivitäten ausgeführt wurden (kurz *Tage*).

Alle Merkmale sind numerisch, weswegen vor der Berechnung der Korrelationskoeffizienten eine Normalisierung auf das Intervall $[0, 1]$ durchgeführt wurde. Dazu wurden für jedes Merkmal die Werte mit der maximalen und minimalen Ausprägung gesucht und anschließend normalisiert. Der letzte Login und das Datum der Registrierung auf der Plattform wurden dabei als Zeitstempel erfasst, weswegen auch hier eine Normalisierung durchgeführt werden konnte.

In Tabelle 16 wurden die Merkmale miteinander korreliert und wichtige Zusammenhänge farblich hervorgehoben. Bei der Betrachtung der Tabelle muss angemerkt werden, dass im Falle eines positiven Zusammenhangs zwischen der Note und eines anderen Merkmals der Koeffizient eine negative Ausprägung besitzt. Das liegt an der Skalierung der Noten, da eine kleinere Note einer besseren Prüfungsleistung entspricht.

Die Korrelationskoeffizienten zeigen einen Zusammenhang zwischen der Note und der Anzahl der Aktivitäten bzw. die Aktivitäten in der Vorlesung, die nach [Coh82] eine mittlere Ausprägung besitzen. Eine etwas stärkere Korrelation ist zwischen der Anzahl der bearbeiteten Aufgaben und der Note zu sehen, bzw. zwischen den Aufgaben in der Vorlesung und der Note. Ebenfalls korreliert die Anzahl der unterschiedlichen bearbeiteten Konzepte und die Note. Darüber hinaus existiert ein Zusammenhang zwischen dem letzten Login bzw. der Differenz zwischen Registrierung und letz-

| Merkmal | Aktivitäten | Note | Letzter Login | Differenz LL. R. | Anzahl Aufgaben | Aktivitäten Vorlesung | Anzahl Aufgaben Vorl. | Anzahl Konzepte | Tage |
|---|---|---|---|---|---|---|---|---|---|
| Aktivitäten | 0 | 0.0001 | 0.001 | 0 | 0 | 0 | 0 | 0 | 0 |
| Note | 0.001 | 0 | 0.001 | 0 | 0.001 | 0 | 0 | 0.001 | 0 |
| Letzter. Login | 0.0001 | 0.001 | 0 | 0 | 0.001 | 0.004 | 0.001 | 0.003 | 0 |
| Diff. LL. R. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Anzahl Aufgaben | 0 | 0.001 | 0.001 | 0 | 0 | 0 | 0 | 0 | 0 |
| Aktivitäten Vorl. | 0 | 0 | 0.004 | 0 | 0 | 0 | 0 | 0 | 0 |
| Anz. Aufgaben Vorl. | 0 | 0 | 0.001 | 0 | 0 | 0 | 0 | 0 | 0 |
| Anz. Konzepte | 0 | 0.001 | 0.003 | 0 | 0 | 0 | 0 | 0 | 0 |
| Tage | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Tabelle 17**: p-Werte für die Korrelationskoeffizienten aus Tabelle 16

tem Login und der Note. Je größer die Differenz zwischen den beiden Zeitpunkten ist, desto besser (negative Korrelation) ist die Note. Dies gilt ebenfalls für den letzten Login, der zugleich in die Differenz mit einfließt. Schlussendlich korreliert die Anzahl der Tage an denen Aktivitäten durchgeführt wurden mit der Note. Erwartungsgemäß sind Korrelationen zwischen der Anzahl der Aktivitäten, den Aufgaben, Tagen und Anzahl der Konzepte stärker ausgeprägt.

Für die Interpretation wurde das Signifikanzniveau von $\alpha = 0,01$ festgelegt, da die Streudiagramme der studentischen Partizipation (vgl. Abbildung 65 und Abbildung 66) eine breitere Streuung in den Daten aufzeigen [vgl. BS10, S.100f]. Die Werte in Tabelle 17 zeigen an, dass die Korrelationen als signifikant für das gewählte Niveau bezeichnet werden können. Die niedrigen p-Werte sind durch den Zusammenhang zwischen den einzelnen berechneten Anzahlen zu erklären. Eine detaillierte Interpretation der Hypothesen in Bezug auf diese Ergebnisse wird in Kapitel 7.5 beschrieben, sodass bei der Bewertung mehrere inhaltliche Aspekte berücksichtigt werden können. Denn erst diese inhaltliche Betrachtung ermöglicht eine Überprüfung der Hypothesen und der Plausibilität der gefundenen Zusammenhänge.

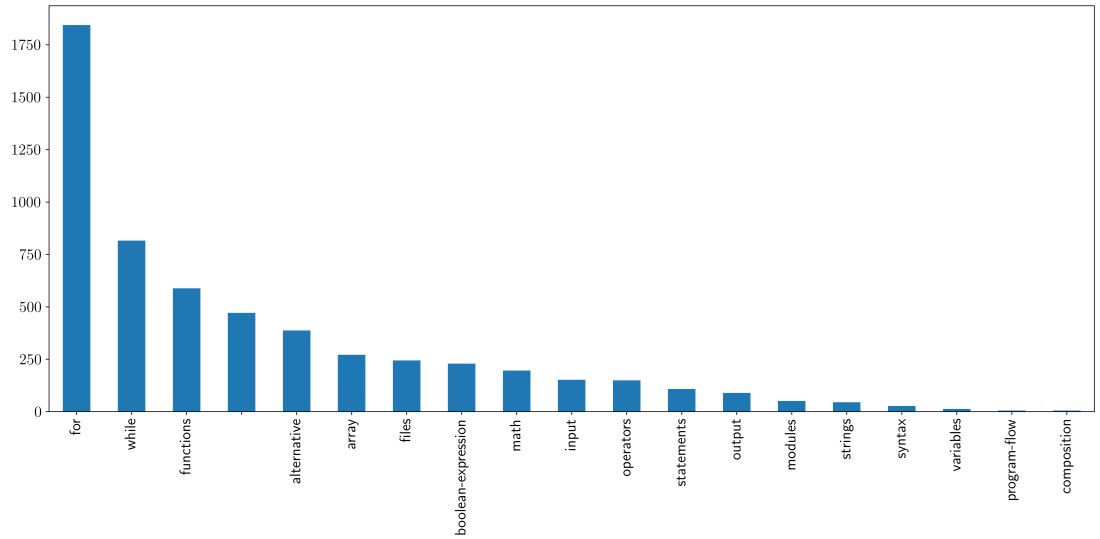



**Abbildung 67**: Anzahl der Aktivitäten je Konzept für Notenstufe 1,0 (von 7 Studierenden)

### Konzepte je Notenstufe

Die Streudiagramme in Abbildung 64 zeigen eine höhere Varianz in den einzelnen Notenstufen und eine teilweise geringe Beteiligung im Bereich der sehr guten Noten. In Abbildung 66 wurden deswegen bereits die Noten in Gruppen zusammengefasst dargestellt, da zu einzelnen Notenstufen nur sehr wenige Punkte existieren. Jedoch ist auch hier ein Unterschied zwischen der Anzahl der Aufgaben bei den sehr guten Noten zu sehen, weswegen die unterschiedlichen Aktivitäten je Konzept (und damit Aufgabe) für mehrere Notenstufen berechnet werden. Dadurch werden die Anzahl der Aufgaben und Aktivitäten hinsichtlich ihrer Entstehung in Bezug auf die Konzepte dargestellt, um hier ggf. Unterschiede zu identifizieren.

Abbildung 67 zeigt die Anzahl der Aktivitäten je Konzept für die Notenstufen von *1,0* bis *1,3*. Die meisten Aktivitäten wurden bei Aufgaben für die Konzepte der Iteration und Funktionen erzeugt. Konzepte wie Operatoren (*operators*), Anweisungen (*statements*) und Variablen (*variables*) sind eher weiter rechts angeordnet und wurden weniger stark bearbeitet. Die Notenstufe *2,0* (von *1,7* bis *2,3*) ist hinsichtlich der ersten 4 Konzepte mit der ersten Notenstufe vergleichbar, wie in Abbildung 68 abgebildet. Einzig die Anzahl der Aktivitäten für Funktionen und die bedingte Ausführung (*alternative*) sind vertauscht. Darüber hinaus ist die Anzahl der Aktivitäten bei Operatoren (*operators*) das fünfthäufigste Konzept, während dieses bei der vorherigen Stufe erst an elfter Stelle vorkommt. Die Reihenfolge der Aktivitäten der Konzepte für die Notestufe *3,0* (vgl. Abbildung 69) unterscheidet sich stärker von den vorherigen. Be-

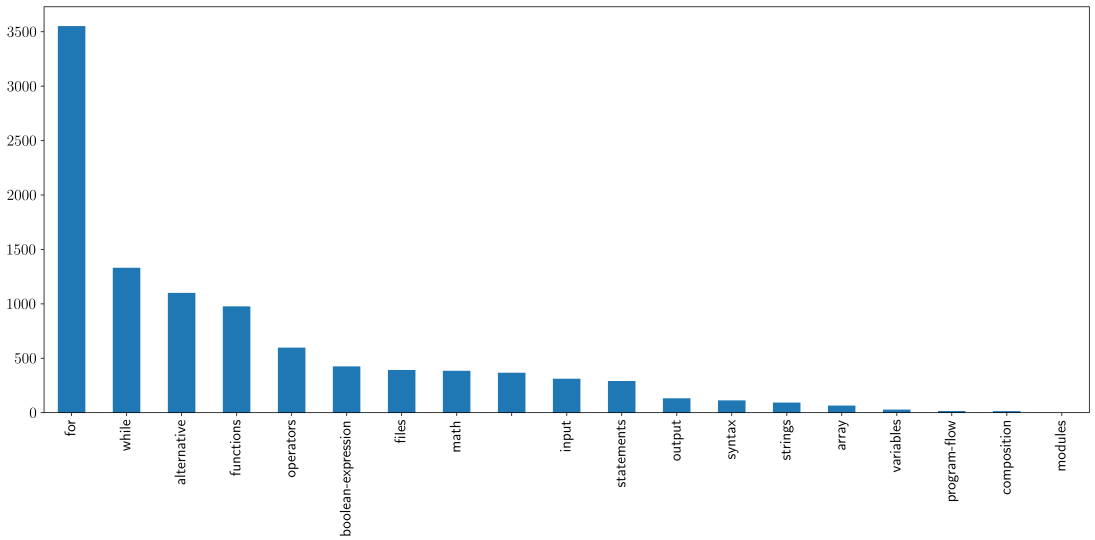



**Abbildung 68**: Anzahl der Aktivitäten je Konzept für Notenstufe 2,0 (von 11 Studierenden)

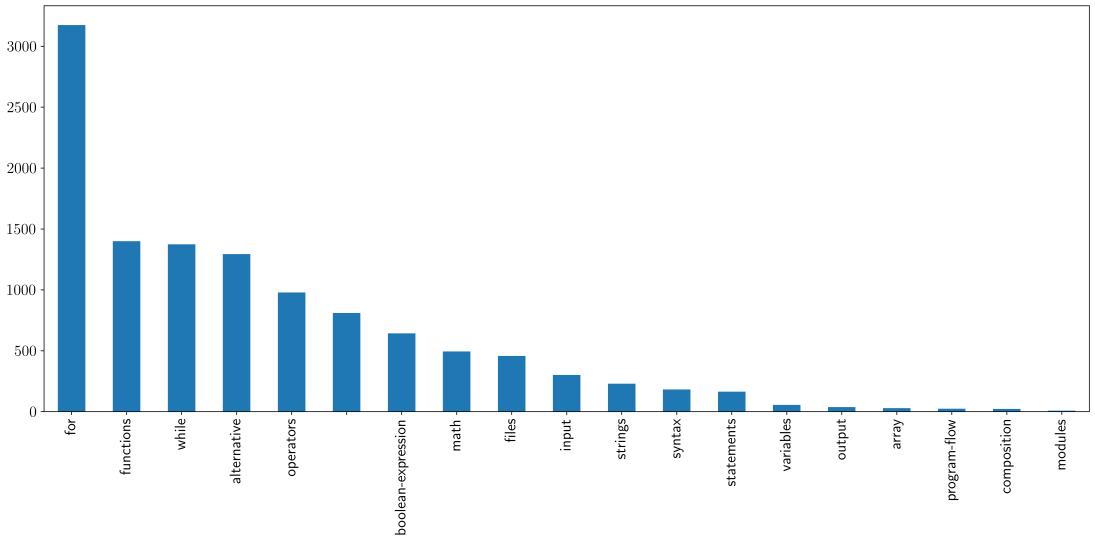



**Abbildung 69**: Anzahl der Aktivitäten je Konzept für Notenstufe 3,0 (von 19 Studierenden)

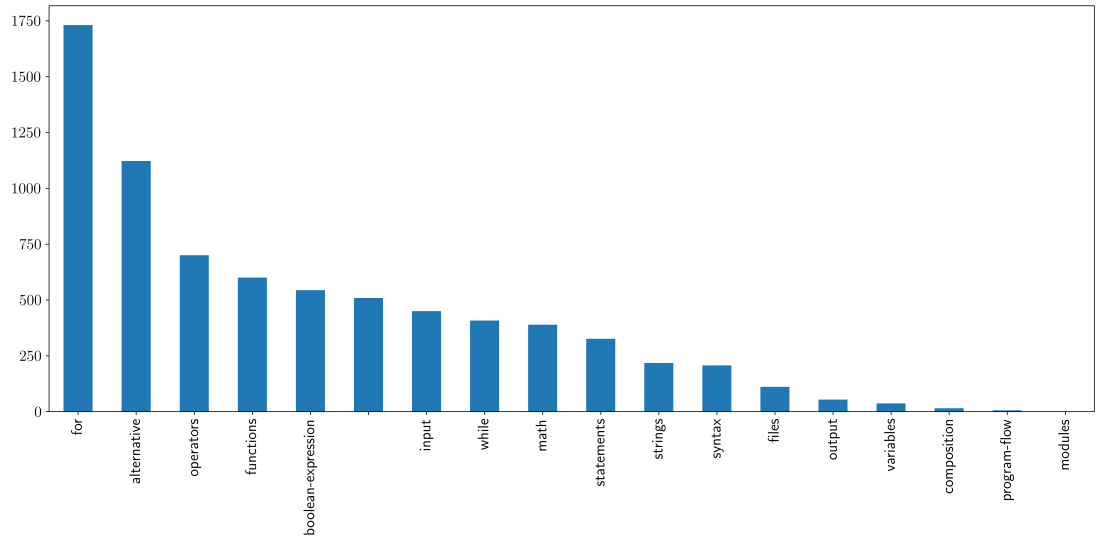



**Abbildung 70**: Anzahl der Aktivitäten je Konzept für Notenstufe 4,0 (von 19 Studierenden)

sonders das Konzept der Funktionen steht an zweiter Stelle, wobei die Differenz zur unbestimmten Iteration (*while*) und zur bedingten Ausführung (*alternative*) nur sehr gering ist.

In Notenstufe *4,0* ist die unbestimmte Iteration (*while*) im Vergleich zu den vorherigen Stufen deutlich weiter rechts angeordnet (vgl. Abbildung 70). Dieses Konzept ist nur noch das siebthäufigste Konzept, während es in den vorherigen Stufen noch an den Stellen eins bis drei platziert war. Ebenfalls sind die Funktionen im Vergleich zu vorher weiter nach hinten verschoben und dafür Operatoren (*operators*) an die dritte Stelle gekommen.

Im Vergleich zu den bisherigen ist für die Notenstufe *5,0* eine geringere Anzahl an Konzepten aus der Abbildung (vgl. Abbildung 71) zu erkennen. Auffällig ist die Platzierung der Operatoren, die nun an der ersten Stelle stehen und anschließend von der Iteration und bedingten Ausführung gefolgt werden.

Die ausgewerteten Aktivitäten stammen von insgesamt 82 Studierenden (vgl. Tabelle 18). Vergleicht man diese Zahl mit der Anzahl der Studierenden, die zur Prüfung angemeldet und auf der Plattform registriert sind aus Tabelle 14, ergibt sich eine Differenz von fünf. Diese kommt zustande, da diese zwar auf der Plattform registriert sind, jedoch nie eine Aktivität durchgeführt haben und somit bei der Auswertung nicht mitgezählt werden.

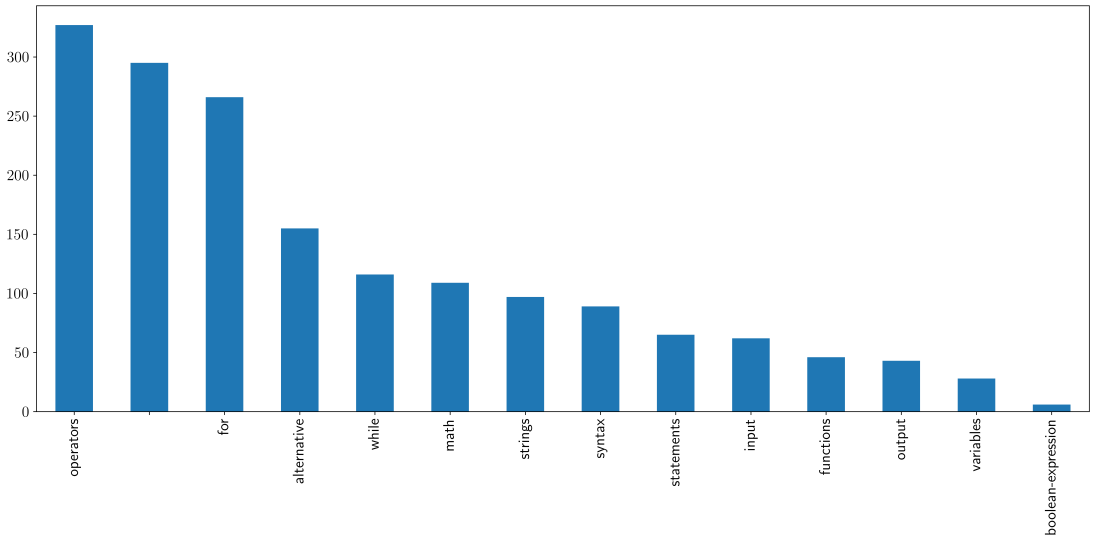



**Abbildung 71**: Anzahl der Aktivitäten je Konzept für Notenstufe 5,0 (von 8 Studierenden)

**Tabelle 18**: Verteilung der Aktivitäten je Notenstufe (gerundet)

| Stufe | Studierende | Anzahl | Anteil in % |
|---|---|---|---|
| *1,0* | 7 | 5690 | 14% |
| *2,0* | 11 | 10189 | 24% |
| *3,0* | 19 | 11673 | 28% |
| *4,0* | 19 | 7428 | 18% |
| *5,0* | 8 | 1704 | 4% |
| *n* | 18 | 5423 | 13% |
| Gesamt | 82 | 42107 | |

**Tabelle 19**: Die ersten zehn Aufgaben mit den meisten Aktivitäten

| # | Aufgabe | Konzept | Aktivitäten | Fehler |
|---|---|---|---|---|
| 1 | Ü5 - Polygon | for | 2203 | 614 |
| 2 | Ü3 - Turtle-Mario | for | 2023 | 435 |
| 3 | Ü5 - Boolesche Algebra | boolean-expression | 1820 | 382 |
| 4 | Ü7 - Lotto | for | 1704 | 416 |
| 5 | Ü4 - Dreieck | alternative | 1642 | 461 |
| 6 | Ü2 Aufgabe 1 | math | 1391 | 412 |
| 7 | Ü2 Aufgabe 2 Mario | for | 1311 | 317 |
| 8 | Ü2 Aufgabe 3 (Turtle) | statements | 1279 | 188 |
| 9 | Ü6 - Listen | for | 1238 | 414 |
| 10 | Ü5 - Primzahlen | while | 1217 | 292 |

### Betrachtung der Beispiele

Eines der Ziele der Arbeit ist die Verwendung von authentischen und motivierenden Beispielen, um einerseits den Praxisbezug herzustellen und andererseits den Spaß am Programmieren zu fördern. Aus diesem Grund sollen an dieser Stelle die häufigsten ausgeführten Programme (mit den meisten Aktivitäten) identifiziert werden. Im Anhang befindet sich die Tabelle 51, welche die Anzahl der Aktivitäten und Fehler je Aufgabe auflistet. Die zehn Aufgaben mit den meisten Aktivitäten werden zur Übersicht hier noch einmal tabellarisch dargestellt (vgl. Tabelle 19). Die ersten beiden Aufgaben (*Ü5 - Polygon* und *Ü3 - Turtle Mario*), sowie die Aufgabe *Ü2 Aufgabe 3 (Turtle)* verwenden *Turtle Graphics* (vgl. Kapitel 6.3.4 und Kapitel 5.2.6). Quantitativ gesehen, haben sich die Studierenden mit diesen Aufgaben am meisten auseinandergesetzt bzw. waren motiviert diese in mehreren Iterationen und Versuchen zu lösen oder darüber hinaus mit diesen eigene Ideen umzusetzen. Die anderen Aufgaben und insbesondere *Ü7 - Lotto*, *Ü6 - Listen* und *Ü5 - Primzahlen* stellen die schwierigeren und umfangreicheren Aufgaben dar, die ebenfalls mehr Aktivitäten erzeugt haben. Auffällig sind die geringen Aktivitäten für die Aufgaben in der 10. Übung (vgl. Tabelle 51), die nur von wenigen Studierenden bearbeitet wurden.

Die in Kapitel 6.3.4 beschriebene Hardware-Simulation des *Sense Hat Shields* wurde auf einem externen Dienst durchgeführt, weswegen keine vollständigen Daten gesammelt wurden. Laut Auskunft des externen Dienstes wurden *1495* Interaktionen bei der Aufgabe erfasst und würde somit auf Platz *6* der Tabelle (Tabelle 19) stehen.

Tabelle 52 zeigt die häufigsten Fehler für jede Aufgaben in aufsteigender Sortierung. Erwartungsgemäß finden sich bei vielen Aufgaben syntaktische Fehler als primäre

Fehlerart, aber es gibt Aufgaben, die von diesem Schema abweichen. Beispielsweise ist bei der Aufgabe *Ü7 - Lotto* die häufigste Fehlerursache eine falsche Indizierung von Listen. Oder die Verwendung eines falschen Literals (*ValueError*) in Aufgabe *Ü3 - Turtle-Mario*, welche auf die notwendige Eingabe einer Ganzzahl zurückzuführen ist. Wie in den in Kapitel 6.5.1 diskutierten Studien, decken sich die Ergebnisse mit denen auf der Plattform erhobenen Daten (vgl. Tabelle 53). Syntaktische Fehler kommen dabei am häufigsten vor, gefolgt von falsch geschriebenen Variablen oder Funktionsnamen. Erst auf dem dritten Platz befinden sich Fehler, welche die Verwendung eines falschen Datentyps andeuten. Jedoch wie bereits in Kapitel 6.5.2 angedeutet, kann teilweise aus den Fehlermeldungen des Interpreters nicht der eigentliche zugrundeliegende Fehler identifiziert werden. Dies kann nur mit weiteren Informationen und des Quelltextes erfolgen, weswegen an dieser Stelle auf eine genauere Analyse der Häufigkeiten der Fehler verzichtet wird.

### 7.2.3 Zusammenfassung & Diskussion

Im Hinblick auf die studentische Partizipation können mehrere Aussagen aus den Ergebnissen abgeleitet werden. Diese werden zuerst in Hinblick auf das forschungsmethodische Vorgehen (vgl. Kapitel 2.3) interpretiert und später in Bezug auf die Hypothesen diskutiert. Tabelle 15 zeigt die Beteiligung der Studierenden im Unterricht an den Aufgaben, welche nach dem in Kapitel 5 beschriebenen Konzept umgesetzt wurden. Bei durchschnittlich 70 anwesenden Studierenden haben sich zwei Drittel bis die Hälfte mit eigenen Geräten an den Übungen beteiligt. Zusätzlich konnten Effekte beobachtet werden, dass Studierende oft zu zweit an einem Gerät gearbeitet haben und sich dabei miteinander ausgetauscht haben. Diskussionen wurden hierbei auch mit der hinteren oder vorderen Sitzreihe geführt, sodass sich die intendierten kooperativen und kollaborativen Lernformen von den Studierenden selbst gebildet haben. Eine weitere Beobachtung ist die Beteiligung von Studierenden, die keine Geräte in der Vorlesung mitgeführt haben, jedoch die Aufgaben auf Papier lösten. Somit kann davon ausgegangen werden, dass mindestens die Hälfte der Studierenden die Aufgaben in der Vorlesung bearbeitet hat. Dadurch hat ein größerer Anteil der Studierenden bei der anschließenden Besprechung die Möglichkeit zur Reflexion und dem Vergleich der eigenen Lösung. Der Durchführung und der Beteiligung mittels eigenen Geräten, kam die Bereitstellung von Steckdosen im Vorlesungssaal zugute, da Studierende die Möglichkeit hatten ihre ggf. älteren Geräte mit schwachen Akkumulatoren zu laden. In Tabelle 15 fallen die Aufgaben *Negation* und *Schaltjahr* mit einer vergleichsweise geringen Beteiligung auf. Beide Aufgaben wurde zum Ende der Vorlesung durchgeführt und konnten nur teilweise besprochen werden. Außerhalb der Vorlesung wurden beide Aufgaben von weiteren Studierenden bearbeitet, allerdings immer noch deutlich weniger als bei den anderen Aufgaben. Die Gründe dafür können bei der Aufgabe *Ne-*

*gation* in der geringen Schwierigkeit liegen und bei *Schaltjahr*, dass eine Musterlösung am Ende der Vorlesung bereits veröffentlicht wurde und somit kein Anreiz zum Ausprobieren mehr vorhanden war. Im Vergleich dazu wurden die anderen Aufgaben aus dem Unterricht von durchschnittlich 40 Studierenden insgesamt über den Lehrveranstaltungszeitraum bearbeitet. Das bedeutet zudem auch, dass von 87 auf der Plattform registrierten (vgl. Tabelle 14) Studierenden nicht einmal die Hälfte der Aufgaben im Anschluss nachbearbeitet haben. Möglicherweise haben sich diese entweder gar nicht mit den Inhalten beschäftigt, diese nur in eigens installierten Entwicklungsumgebungen ausgeführt, oder nur die bereitgestellten Musterlösungen gelesen. Vergleicht man diese Zahl mit den Studierenden, die auf der Plattform registriert waren und an der Prüfung teilgenommen (66 vgl. Tabelle 14) haben, hat sich durchschnittlich mehr als die Hälfte der Studierenden mit den Aufgaben beschäftigt. Die Anzahl der Aktivitäten und Aufgaben in Abbildung 65 zeigt, einen Zusammenhang zur erreichten Note. Dieser ist in Abbildung 66 anhand von Notengruppen deutlicher zu erkennen. Besonders Studierende mit eher schlechteren Prüfungsergebnissen haben tendenziell weniger Aufgaben bearbeitet und weniger Aktivitäten verursacht. Bislang ist dieser Zusammenhang nur eine Vermutung, die von den Daten in Tabelle 18 gestützt wird. Dazu können auch die fünf Studierenden, welche in der Prüfung durchgefallen sind, jedoch auf der Plattform registriert und keinerlei Aktivitäten verursacht haben, als weiteres Indiz gewertet werden.

Daten zu weiteren Aufgaben, die nicht nach dem erarbeiteten Konzept im Unterricht durchgeführt wurden, da Sie teilweise nur syntaktische Eigenheiten adressiert haben, zeigen ein ähnliches Bild bei der Beteiligung (vgl. Tabelle 50). In dieser Tabelle ist die Abnahme der Beteiligung im Laufe des Semesters zu sehen, die sich auch bei der Anzahl der anwesenden Studierenden bemerkbar gemacht hat. Dieser Abfall ist teilweise durch aktuelle organisatorische Probleme in den Elektrotechnikstudiengängen bedingt. Für die Vorlesung und Übungen besteht für die Lehrveranstaltung Programmieren keine Anwesenheitspflicht und keine weiteren Prüfungszulassungsvoraussetzungen. Dies ist jedoch bei anderen Fächern der Fall, in denen die Studierenden mehrere Praktikumsberichte (z. B. Grundlagen Elektrotechnik 1) ausarbeiten müssen, die gleichzeitig als Voraussetzung zur Zulassung der Prüfung gelten. Ab diesem Zeitpunkt ist ein Rückgang der anwesenden Studierenden beobachtbar, sowie eine schlechtere Beteiligung in den Übungsgruppen. Ein weiterer Grund ist der geplante Einsatz einer vollständigen Entwicklungsumgebung, die ab ca. der Hälfte des Semesters eingeführt wurde. Die Daten deuten jedoch darauf hin, dass nach wie vor die web-basierte Entwicklungsumgebung verwendet wurde (vgl. Tabelle 51).

Die berechneten Korrelationen in Bezug auf die Note (vgl. Tabelle 16) deuten auf einen Zusammenhang zwischen den Aktivitäten bzw. der Anzahl der bearbeiteten Aufgaben und der erreichten Prüfungsnote hin. Bei der Bewertung ist zu berücksichtigen,

dass die Note den gesamten Lehrinhalt widerspiegelt und nicht nur einzelne Konzepte abfragt. Ausgehend von der Beteiligung in der Vorlesung (vgl. Tabelle 15) lässt sich die etwas geringere Korrelation bei der Anzahl der Aktivitäten und Aufgaben in der Vorlesung erklären, da eben nur ein gewisser Teil der Studierenden digital mitgearbeitet hat. Darüber hinaus zeigen die Korrelationen zwischen dem letzten Login, der Differenz zwischen Registrierung und letztem Login sowie die Anzahl der Tage einen positiven Einfluss auf die Note an. Studierende, die über das gesamte Semester Aufgaben bearbeitet und somit geübt haben, erreichten eine tendenziell bessere Prüfungsnote. Die Korrelation zwischen dem letzten Login und der Note lässt sich mit der Prüfungsvorbereitung und der Bearbeitung von Aufgaben vor der Prüfung erklären. Einen ähnlichen Einfluss hat die Anzahl der Konzepte auf die Note, welche erwartungsgemäß mit der Anzahl der Aufgaben ($r = 0,93$) korreliert. Dies ist plausibel, da die Wahrscheinlichkeit mehr verschiedene Konzepte zu üben mit der Anzahl unterschiedlicher Aufgaben steigt. Weiterhin korreliert die Anzahl der Aktivitäten in der Vorlesung mit den Aktivitäten insgesamt ($r = 0,79$), was darauf hindeutet, dass Studierende, die sich in der Vorlesung beteiligt haben, auch sich auch darüber hinaus mit den Aufgaben und Übungen beschäftigen. Hierbei kann jedoch nur eine Aussage für die Studierenden, die digital mitgearbeitet haben, getroffen werden, da die anderen nicht erfasst wurden.

Die Vermutung, dass sich bessere Studierende mit mehr Aufgaben und Konzepten beschäftigen, wird durch die Ergebnisse der Auswertung der Anzahl der Konzepte je Notenstufe unterstützt (siehe Kapitel 7.2.2). Studierende in der Notenstufe *1,0* haben sich im Vergleich zu den anderen stärker mit weiterführenden Themen beschäftigt. In Kontrast dazu stehen die Notenstufen *4,0* und *5,0*, bei denen an den vorderen Stellen eher grundlegende Konzepte wie beispielsweise *Operatoren* stehen. Betrachtet man dies in Verbindung mit der nachlassenden Beteiligung an den Aufgaben, haben besonders die Studierenden mit schlechteren Prüfungsleistungen die weitergehenden Konzepte nur teilweise oder gar nicht bearbeitet.

Verglichen mit den Ergebnissen der Studie von Hassinen et al. (siehe Kapitel 4.4.1) können die gleichen Effekte beobachtet werden. Einerseits gibt es sehr gute Studierende, die nur wenig aktiv gewesen sind und andererseits schlechte Studierende, die sehr aktiv waren. Der Unterschied (vgl. Abbildung 66) ist durch die geringe Anzahl der Studierenden nicht auf jeder einzelne Zwischennote sichtbar, aber in den zusammengefassten Gruppen besser zu sehen. Betrachtet man die Korrelationen besteht ein stärkerer Zusammenhang zwischen der Anzahl der bearbeiteten Aufgaben und der Note. Dies kann der Vorgehensweise geschuldet sein, da ggf. Studierende nur wenige erfasste Aktivitäten verursachen, wenn sie einen Ansatz mit mehr Vorüberlegungen und weniger die Strategie *Versuch und Irrtum* verwenden.

Aus Sicht der Forschungsmethodik sollten folgende Fragen beantwortet werden: (1) Wur-

den die Lernenden in der angestrebten Weise aktiv? (2) Zeigten sich unter Umständen erwünschte und unerwünschte Nebenwirkungen? (3) Welche Lernaktivitäten sind auf Seiten der Lernenden zu beobachten? Die Tabelle 15 zeigt, dass die Studierenden sich im Unterricht mit den Aufgaben zu den einzelnen Konzepten im Rahmen der realisierten Lehrhandlungen (siehe Kapitel 7.1.4) beschäftigt haben. Zusammenarbeit und die Bildung von Gruppen wurde zu Beginn angesprochen, aber es bildeten sich von alleine Paare und Gruppen, welche sich innerhalb der Vorlesung bei der Bearbeitung der Aufgaben ausgetauscht haben. Darüber hinaus deuten die Daten darauf hin, dass sich die Studierenden auch neben der Vorlesung mit den interaktiven Inhalten beschäftigt haben. Besonders die Auswertung der am häufigsten bearbeiteten Beispiele sprechen zusätzlich für die Verwendung authentischer und *greifbarer* Beispiele (vgl. Tabelle 19).

Eine Verwendung der Fehler für die schnelle Einschätzung ist nur teilweise notwendig, da diese wie bereits erwartet höchstens eine Tendenz angibt und konkrete Probleme erst im Zusammenhang mit dem auftretenden Quelltext und Kontext ersichtlich werden. Im Laufe der Durchführung wurde für die Aufgaben im Unterricht eher weniger auf die Fehler geachtet und stattdessen bei Bedarf mit den Studierenden diskutiert. Für die Vorbereitung der Vorlesung wurden die Möglichkeit zu Analyse der Fehler und der Quelltexte verwendet.

Die Durchführung der einzelnen Schritte im Rahmen des entwickelten Konzepts, konnte einerseits die Studierenden aktivieren sich bereits im Unterricht mit den Inhalten zu beschäftigen. Zusätzlich wurde durch die Diskussionen eine Atmosphäre geschaffen, in der Studierende andererseits den Unterricht durch Fragen steuern konnten. Um die Auswirkungen der Lehrhandlungen auf die Programmierfähigkeit zu untersuchen, wird im nächsten Kapitel eine Analyse der Implementierungsaufgaben der Prüfung durchgeführt.

## 7.3 Untersuchung der Aufgabenbewertung

Bisher galt der Blick den reinen Aktivitäten der Studierenden, jedoch erfasst die Note nicht immer vollständig die Fähigkeit zu programmieren. Die Lehrveranstaltung thematisiert neben der Funktionsweise eines Computers unter anderem Zahlensysteme und weitere theoretische Grundlagen. Aus diesem Grund wird das in Kapitel 3.3.3 beschriebene Instrument zur Messung der Programmierfähigkeit anhand dreier Implementierungsaufgaben in der Klausur für jeden Studierenden angewendet. Im Anschluss können diese neuen Punktzahlen, mit den Aktivitäten für die einzelnen Konzepte korreliert werden. Dadurch soll gezeigt werden, dass die Bearbeitung der Aufgaben zu bestimmten Konzepten einen positiven Einfluss auf die Programmierfähigkeit besitzt.

### 7.3.1 Datenerhebung

Für die Datenerhebung wird das Bewertungsschema zuerst auf die Aufgaben angewendet und anschließend werden alle Aufgaben entsprechend bewertet. Die Bewertung unterscheidet sich dabei von der eigentlichen Bewertung für die Notengebung, da syntaktische Fehler und Weiteres nicht mit einfließen. Darüber hinaus ist die Benotung und die Vergabe der Punkte für die Prüfungsleistung gleichzeitig an die Zeitdauer der Prüfung angelehnt, welche beim neuen Bewertungsschema nicht mit einfließen.

Bei der Bewertung nach dem Schema werden nur die Konzepte beachtet, welche nach dem in Kapitel 5 erarbeiteten Konzepts eingeführt und geübt wurden. Dies ermöglicht eine gezieltere Auswertung zwischen den Lehrhandlungen und den Ergebnissen bei den Studierenden. Aus diesem Grund wurden Elemente wie *Ein- und Ausgabe* nicht beachtet, sondern der Fokus vielmehr auf *Bedingungen* und *Iterationen* gesetzt.

#### Implementierungsaufgabe 1

Die erste Aufgabe erfordert strenggenommen nicht die Auswahl des richtigen Konzepts, sondern fokussiert dessen Verwendung.

**Aufgabe 1 - Produktbildung mit `while`**

Geben Sie eine `while` Schleife an, welche das Produkt über die Zahlen von 1 bis 42 (inklusive) berechnet.

```
1 i = 1
2 p = 1
3
4 while i <= 42:
5         i +=1
6         p *= i
```

**Quellcode 11:** Eine mögliche Lösung zur Implementierungsaufgabe 1

Quellcode 11 zeigt eine mögliche Lösung der Aufgabe. Insbesondere prüft diese Aufgabe das Verständnis des Programmablaufs, Gültigkeitsbereiche von Variablen und die notwendige Vorbelegung mit Werten von Variablen, wenn diese im Schleifenrumpf verändert werden.

Für diese Aufgabe wurde die Indikatorbildung wie in Tabelle 20 dargestellt, durchgeführt und auf die studentischen Lösungen angewendet.

**Tabelle 20**: Anwendung Bewertungsschema für Implementierungsaufgabe 1

| Wert | Punkte | Beschreibung |
|---|---|---|
| Konzeptauswahl | 0 | Bereits explizit vorgegeben |
| Stelle des Konzeptes | 0 | |
| Verwendung des Konstruktes | 2 | Initialisierung der Zählvariable und Produktvariable vor der Schleife |
| | 1 | Boolescher Ausdruck für die Schleifenbedingung (`<= 42`) |
| Gesamt | 3 | |

### Implementierungsaufgabe 2

Aufgabe 2 umfasst die Definition einer Funktion zur Überprüfung von der Gültigkeit von Variablennamen.

**Aufgabe 2 - Überprüfung von Variablennamen**

Erstellen Sie die Funktion `is_valid_name(s)`, welche Variablennamen auf deren Gültigkeit überprüft. Dabei gibt die Funktion entweder wahr oder falsch zurück. Es sollen dabei folgende Regeln für die Variablennamen gelten.

- Es sind nur **a-z** (Kleinbuchstaben) und _ (Unterstrich) als Zeichen erlaubt
- Der Name darf **nicht** mit einem **Unterstricht** anfangen
- Ein Name muss **mindestens** 1 Zeichen lang sein

**Beispiele**:

- `äbcd"` gibt wahr zurück
- `ä_bcd"` gibt wahr zurück
- `"_ab_cd"` gibt falsch zurück
- `Äbcd"` gibt falsch zurück

Die Lösung dieser Aufgabe erfordert die Kombination von Schleifen und Bedingungen, wobei unterschiedliche Lösungen möglich sind. Das Bewertungsschema und besonders die Stelle, an der das Konzept angewendet wird, muss dabei in Abhängigkeit von der Gesamtlösung betrachtet werden. In den Übungen wurden bereits Aufgaben zur Verfügung gestellt, die sich ebenfalls mit der Überprüfung von Zeichen(-ketten) beschäftigten. Weiterhin sind die Aufgaben so konzipiert, dass der Umfang der Lösung klein gehalten wird, sodass die Lösung auf ca. einer halben Seite Platz finden (vgl. Quellcode 12). Für die Bewertung wurden drei verschiedene Teilprobleme identifiziert, welche die Studierenden lösen müssen:

```python
def is_valid_name(s):
        # Mindestens 1 Zeichen lang
        length = len(s)
        if length < 1:
                return False

        # Muss mit einem Kleinbuchstaben beginnen
        if s[0] < "a" or s[0] > "z":
                return False

        # Erlaubte Zeichen a-z, _
        for i in range(1, length):
                if s[i] != "_" and not (s[i] >= "a" or s[i] <= "z"):
                        return False

        return True
```

**Quellcode 12:** Eine mögliche Lösung zur Implementierungsaufgabe 2

1. Überprüfung der Länge der Zeichenkette

2. Überprüfung der Gültigkeit des ersten Zeichens anhand der Sonderregel (nur Kleinbuchstaben)

3. Überprüfung der Gültigkeit aller Zeichen anhand der Regeln

Tabelle 21 enthält die für die Aufgabe durchgeführte Indexbildung zur Messung. Wie bereits erwähnt, wurde hier der Fokus auf die Konzepte der Iteration und der Bedingungen gelegt. Durch die Verwendung dieser Bewertung können verschiedene Lösungsansätze miteinander verglichen werden, da der Index die Lösung und Kombination der Teilprobleme abbildet und dadurch unterschiedliche Lösungen vergleichen kann. Denn die Überprüfung der Teilprobleme kann bei dieser Aufgabe an unterschiedlichen Stellen erfolgen und nach wie vor den geforderten Anforderungen entsprechen. Beispielsweise kann die Überprüfung des ersten Zeichens zu Beginn oder eben auch nach Überprüfung aller Zeichen erfolgen.

### Implementierungsaufgabe 3

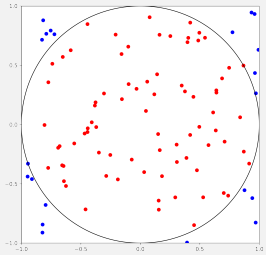
Die letzte Aufgabe, die im Rahmen dieser Arbeit untersucht wurde, thematisiert die Bestimmung von $\pi$ mittels des Monte-Carlo-Algorithmus.

**Tabelle 21**: Anwendung Bewertungsschema für Implementierungsaufgabe 2

| Konzept | Beschreibung | Punkte |
|---|---|---|
| **Länge der Zeichenkette** | | (3) |
| Auswahl | **Bedingung** | 1 |
| Stelle | Kann am Anfang oder am Ende erfolgen | 1 |
| Verwendung | Boolescher Ausdruck | 1 |
| **Erstes Zeichen** | | (3) |
| Auswahl | **Bedingung** | 1 |
| Stelle | Vor der Schleife, in der Schleife oder am Ende | 1 |
| Verwendung | Boolescher Ausdruck | 1 |
| **Iteration über alle Zeichen** | | (3) |
| Auswahl | **Infinite Iteration** Anzahl der Durchläufe ist bekannt, weswegen sich `for` eignet | 1 |
| Stelle | Nach Überprüfung der Länge oder ggf. davor | 1 |
| Verwendung | Iteration über alle Zeichen oder von 1 bis n | 1 |
| **Überprüfung der Zeichen** | | (3) |
| Auswahl | **Bedingung** | 1 |
| Stelle | Innerhalb der Schleife | 1 |
| Verwendung | Boolescher Ausdruck | 1 |
| Gesamt | | 12 |

**Aufgaben 3 - Bestimmung von Pi**

Schreiben Sie ein Programm, welches die Zahl Pi probabilistisch durch die Anwendung des Monte-Carlo-Algorithmus bestimmt. Dazu werden zufällige Punkte $(x, y|x \in [-1..1] \wedge y \in [-1..1])$ ausgewählt und anschließend überprüft, ob diese innerhalb des Einheitskreises liegen (siehe untenstehende Grafik). Ein Punkt $(x, y)$ liegt dabei im Einheitskreis, wenn $x^2 + y^2 \leq 1$ gilt.



Der Algorithmus lautet wie folgt:

1. Anzahl der zu berechnenden Punkte $n$ eingeben (Ganzzahl)
2. Überprüfung, dass $n > 0$ ist. Ansonsten soll eine Fehlermeldung ausgegeben werden.
3. N-maliges Generieren eines zufälligen Punktes $(x, y)$ und Überprüfung, ob dieser im Einheitskreis liegt. Falls der Punkte im Einheitskreis liegt, soll eine Zählvariable um 1 erhöht werden.
4. Anschließend lässt sich Pi mittels $pi = \frac{PunkteimKreis}{n} * 4$ berechnen.

Geben Sie anschließend Pi im Fließkommaformat mit 8 Nachkommastellen aus.

Auch bei dieser Aufgabe ist der Umfang der Lösung klein genug, sodass diese auf eine halbe Seite passt (vgl. Quellcode 13).

Bei dieser Aufgabe liegt die Besonderheit in den verschachtelten Bedingungen bzw. bei der Initialisierung der Zählvariable außerhalb der Schleife. Das Generieren der Zufallszahl wird im Zuge der Bewertung nicht weiter beachtet, sondern wie bei der vorherigen Aufgabe auf die Bedingungen und Iterationen fokussiert.

Tabelle 22 listet die einzelnen Teilprobleme und die Gewichtung der einzelnen Aspekte auf. Hier wurden ebenfalls die Ein- und Ausgabe weggelassen, da diese nicht nach dem entwickelten Konzept eingeführt wurden.

### Bewertung

Anhand der Bewertungsschemata für die einzelnen Aufgaben wurden alle Prüfungsaufgaben bewertet und die Teilergebnisse für jedes Teilproblem festgehalten. Die Teil-

**Tabelle 22:** Anwendung Bewertungsschema für Implementierungsaufgabe 3

| **Konzept** | **Beschreibung** | **Punkte** |
|---|---|---|
| **Überprüfung der Eingabe** $n > 0$ | | (4) |
| Auswahl | **Bedingung** mit Alternative (`if` und `else`) | 2 |
| Stelle | Muss nach der Eingabe erfolgen | 1 |
| Verwendung | Boolescher Ausdruck | 1 |
| **N-maliges Generieren** | | (4) |
| Auswahl | **Finite Iteration** (`for`) | 1 |
| Stelle | Innerhalb der Bedingung | 1 |
| Verwendung | Anzahl der Durchläufe | 1 |
| | Zählvariable außerhalb (Punkte im Kreis) | 1 |
| **Überprüfung Einheitskreis** | | (3) |
| Auswahl | **Bedingung** `if` ohne Alternative | 1 |
| Stelle | Innerhalb der Schleife nach der Generierung der Zufallszahlen | 1 |
| Verwendung | Boolescher Ausdruck | 1 |
| Gesamt | | 11 |

```
from random import random

n = int(input("n : "))

if n > 0:
        # Monte Carlo Simulation für Pi
        punkteImKreis = 0
        for i in range(n):
                zufallsZahlX = random() * 2 - 1
                zufallsZahlY = random() * 2 - 1

                if (zufallsZahlX ** 2 + zufallsZahlY ** 2) <= 1.0:
                        punkteImKreis += 1

        pi = (punkteImKreis / n) * 4
        print("Pi: %.8f" % pi)
else: #1
        print('Ungültige Eingabe für n')
```

**Quellcode 13**: Eine mögliche Lösung zur Implementierungsaufgabe 3

ergebnisse wurden äquivalent wie die Note den Studierenden zugeordnet und anschließend pseudonymisiert, um Rückschlüsse auf die Identitäten zu verhindern. Für jeden Studierenden wurden dabei folgende Daten erhoben und den Datensätzen zugeordnet:

- Punktzahl Aufgabe 1 (`while`)
- Punktzahl Aufgabe 2 - Bedingung (Länge der Zeichenkette)
- Punktzahl Aufgabe 2 - Bedingung (Erstes Zeichen)
- Punktzahl Aufgabe 2 - Finite Iteration (Iteration über alle Zeichen)
- Punktzahl Aufgabe 2 - Bedingung (Überprüfung eines Zeichens)
- Punktzahl Aufgabe 3 - Bedingung (Überprüfung von $n > 0$)
- Punktzahl Aufgabe 3 - Finite Iteration (N-maliges Generieren)
- Punktzahl Aufgabe 3 - Bedingung (Überprüfung Einheitskreis)

Zusätzlich wurden aus diesen Daten die jeweilige Gesamtpunktzahl berechnet und diese normalisiert, sodass die Daten mit den Aktivitäten korreliert werden können.

### 7.3.2 Ergebnisse

Um Rückschlüsse zwischen den Aktivitäten und den Ergebnissen bei der Anwendung der Konzepte erhalten und somit auf die Programmierfähigkeit, wurde ein ähnlicher Ansatz wie bereits in Kapitel 7.2 (Korrelationen) durchgeführt. Für jeden Studierenden wurden die neuen Bewertungen auf Basis des entwickelten Messinstruments jedem Datensatz zugeordnet. Zusätzlich wurden folgende Merkmale für jeden Studierenden berechnet:

- Anzahl der Aktivitäten für die infinite Iteration (`while`)
- Anzahl der Aktivitäten für die finite Iteration (`for`)
- Anzahl der Aktivitäten für Iteration (beide Arten)
- Anzahl der Aktivitäten für Bedingungen (`alternative`) inklusive Anzahl der Aktivitäten für *Boolesche Ausdrücke*
- Erreichte Punkte über alle Teillösungen mittels finiter Iteration
- Erreichte Punkte über all Teillösungen Iteration (infinite und finite)
- Erreichte Punkte über all Teillösungen Bedingung

In Tabelle 23 wurden die in den Aufgaben 1 bis 3 erreichten Punkte (*Aufgabe 1*, *Aufgabe 2* und *Aufgabe 3*) mit den jeweiligen Aktivitäten korreliert (Rangkorrelationskoeffizient). Für die Interpretation wird ebenfalls wie bei der studentischen Partizipation (vgl Kapitel 7.2) das Signifikanzniveau auf $\alpha = 0,01$ festgelegt. Zusätzlich wurden wie bereits in der Aufzählung erwähnt die Punkte von allen Teillösungen, welche die Konzepte *Bedingung*, *finite Iteration* und *Iteration* (finit und infinit) für jeden Studierenden berechnet und ebenfalls korreliert.

Zwischen den Aktivitäten der finiten und infiniten Iteration sowie die Summe aus beiden, ist ein positiver Zusammenhang mit den jeweiligen erreichten Punktzahlen vorhanden. Auffällig ist die deutlich schlechtere Korrelation zwischen den *Aktivitäten der Bedingungen* und den erreichten Ergebnissen in den Aufgaben.

Im Vergleich tritt eine höhere Korrelation zwischen den Aktivitäten der Iteration und *Punkte Iteration* auf. Auch hier ist die deutlich niedrigere Korrelation zwischen der *Aktivitäten Bedingungen* und *Punkte Bedingungen* erkennbar. Im Gegensatz dazu korrelieren die Punkte der finiten und infiniten Iteration um den Faktor 2 stärker mit den *Aktivitäten der Bedingungen*.

| Merkmale | Aufgabe 1 | Aufgabe 2 | Aufgabe 3 | Aktivitäten finite It. | Aktivitäten Infinite It. | Aktivitäten Iteration | Aktivitäten Bedingungen | Punkte finite It. | Punkte Iteration | Punkte Bedingungen |
|---|---|---|---|---|---|---|---|---|---|---|
| Aufgabe 1 | 1,00 | 0,44 | 0,43 | 0,44 | 0,39 | 0,46 | 0,42 | 0,41 | 0,66 | 0,47 |
| Aufgabe 2 | 0,44 | 1,00 | 0,52 | 0,43 | 0,48 | 0,46 | 0,41 | 0,76 | 0,76 | 0,87 |
| Aufgabe 3 | 0,43 | 0,52 | 1,00 | 0,34 | 0,34 | 0,37 | 0,24 | 0,81 | 0,80 | 0,81 |
| Aktivitäten finite It. | 0,44 | *0,43* | *0,34* | 1,00 | 0,85 | 0,99 | 0,75 | *0,40* | *0,47* | *0,41* |
| Aktivitäten Infinite It. | 0,39 | *0,48* | *0,34* | 0,85 | 1,00 | 0,91 | 0,82 | *0,47* | *0,52* | *0,44* |
| Aktivitäten Iteration | 0,46 | *0,46* | *0,37* | 0,99 | 0,91 | 1,00 | 0,78 | *0,44* | *0,51* | *0,45* |
| Aktivitäten Bedingungen | 0,42 | *0,41* | *0,24* | 0,75 | 0,82 | 0,78 | 1,00 | *0,43* | *0,49* | *0,33* |
| Punkte finite It. | 0,41 | 0,76 | 0,81 | 0,40 | 0,47 | 0,44 | 0,43 | 1,00 | 0,95 | 0,78 |
| Punkte Iteration | 0,66 | 0,76 | 0,80 | 0,47 | 0,52 | 0,51 | 0,49 | 0,95 | 1,00 | 0,79 |
| Punkte Bedingungen | 0,47 | 0,87 | 0,81 | 0,41 | 0,44 | 0,45 | 0,33 | 0,78 | 0,79 | 1,00 |

**Tabelle 23**: Korrelationen der Ergebnisse der Aufgabenbewertung und der Aktivitäten für jedes Konzept

**Tabelle 24**: p-Werte für die Korrelationskoeffizienten in Tabelle 23

| Merkmale | Aufgabe 1 | Aufgabe 2 | Aufgabe 3 | Aktivitäten finite It. | Aktivitäten Infinite It. | Aktivitäten Iteration | Aktivitäten Bedingungen | Punkte finite It. | Punkte Iteration | Punkte Bedingungen |
|---|---|---|---|---|---|---|---|---|---|---|
| Aufgabe 1 | 0 | 0.0002 | 0.0003 | 0.0002 | 0.0012 | 0.0001 | 0.0005 | 0.0006 | 0 | 0.0001 |
| Aufgabe 2 | 0.0002 | 0 | 0 | 0.0003 | 0 | 0.0001 | 0.0005 | 0 | 0 | 0 |
| Aufgabe 3 | 0.0003 | 0 | 0 | 0.0059 | 0.0052 | 0.0024 | 0.0557 | 0 | 0 | 0 |
| Aktivitäten finite It. | 0.0002 | 0.0003 | 0.0059 | 0 | 0 | 0 | 0 | 0.001 | 0.0001 | 0.0006 |
| Aktivitäten infinite It. | 0.0012 | 0 | 0.0052 | 0 | 0 | 0 | 0 | 0.0001 | 0 | 0.0002 |
| Aktivitäten Iteration | 0.0001 | 0.0001 | 0.0024 | 0 | 0 | 0 | 0 | 0.0002 | 0 | 0.0002 |
| Aktivitäten Bedingungen | 0.0005 | 0.0005 | 0.0557 | 0 | 0 | 0 | 0 | 0.0004 | 0 | 0.0063 |
| Punkte finite It. | 0.0006 | 0 | 0 | 0.001 | 0.0001 | 0.0002 | 0.0004 | 0 | 0 | 0 |
| Punkte Iteration | 0 | 0 | 0 | 0.0001 | 0 | 0 | 0 | 0 | 0 | 0 |
| Punkte Bedingungen | 0.0001 | 0 | 0 | 0.0006 | 0.0002 | 0.0002 | 0.0063 | 0 | 0 | 0 |

| Merkmale | Anzahl Beispiele | Aktivitäten | p-Wert Anzahl Beispiele | p-Wert Aktivitäten |
|---|---|---|---|---|
| Aufgabe 1 | 0,46 | 0,37 | 0,0001 | 0,0021 |
| Aufgabe 2 | 0,41 | 0,34 | 0,0007 | 0,0056 |
| Aufgabe 3 | 0,34 | 0,31 | 0,005 | 0,0123 |
| Punkte finite It. | 0,40 | 0,40 | 0,0009 | 0,0008 |
| Punkte Iteration | 0,48 | 0,45 | 0 | 0,0001 |
| Punkte Bedingungen | 0,42 | 0,33 | 0,0004 | 0,0064 |

**Tabelle 25**: Korrelationen der Ergebnisse der Aufgabenbewertung und der Aktivitäten für jedes Konzept

Die berechneten *p-Werte* in Tabelle 24 deuten in Bezug auf das gewählte Signifikanzniveau auf eine nicht signifikante Korrelation zwischen den *Aktivitäten für Bedingungen* und den Punkten bei den *Bedingungen* an. Tabelle 25 zeigt die Korrelationen zwischen den erreichten Punkten in den Aufgaben und den Aktivitäten bzw. den bearbeiteten Aufgaben (Beispielen). Die Anzahl der Beispiele korreliert dabei mit der erreichten Punktzahl der Aufgaben und den Punkten je Konzept. Ausgehend von den Ergebnissen aus Kapitel 7.2, welcher bereits die Prüfungsnoten in Zusammenhang mit den Aktivitäten untersucht hat, ist dieses Ergebnis plausibel. Bei dem Signifikanzniveau von $\alpha = 0,01$ muss jedoch die Korrelation zwischen den Aktivitäten und der erreichten Punktzahl bei *Aufgabe 3* als nicht signifikant zurückgewiesen werden. Im Vergleich existiert dennoch ein Zusammenhang zwischen der *Anzahl der Beispiele* und der erreichten Punktzahl bei dieser Aufgabe. Eine weitere Interpretation dieser Ergebnisse wird ebenfalls erst in Kapitel 7.5 erfolgen.

Neben den Korrelationen werden zusätzlich die Ergebnisse der unterschiedlichen Bewertungsschemata verglichen. Die Indexbildung zeigt dabei die grundlegende Korrektheit der Auswahl, Anwendung und Kombination der richtigen Konzepte für die einzelnen Teillösungen, ohne syntaktische Fehler (außer sie sind stark sinnverfälschend) zu berücksichtigten. Für den Vergleich wurden alle Ergebnisse auf das Intervall $[0..1]$ normalisiert, sodass keine Unterschiede zwischen den maximal erreichbaren Punkten existieren. Des Weiteren wird dadurch der grobe Lösungsentwurf bewertet und nicht jede einzelne Anweisung.

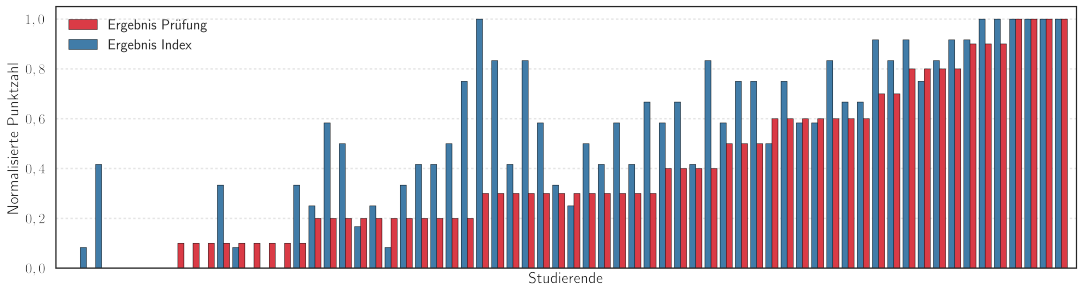



**Abbildung 72:** Vergleich der Prüfungsergebnisse mit dem neuen Bewertungsschema für Aufgabe 2

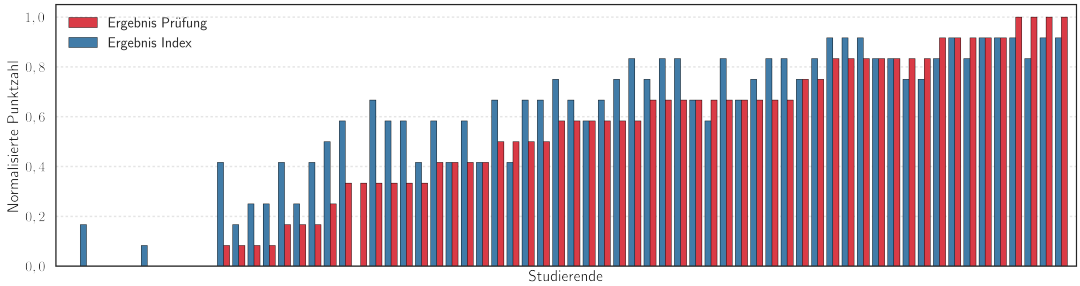



**Abbildung 73:** Vergleich der Prüfungsergebnisse mit dem neuen Bewertungsschema für Aufgabe 3

Das Balkendiagramm in Abbildung 72 vergleicht die erreichten Punkte in der Prüfung mit dem jeweiligen erreichten Index nach dem neuen Schema (vgl. Kapitel 3.3.3). Jeweils ein Paar bestehend aus einem blauen und einem roten Balken symbolisiert hierbei einen einzelnen Studierenden und dessen Ergebnisse bei der Bewertung. Die Aufgabe wurde in der Prüfung mit maximal 10 Punkten bewertet, wobei hier syntaktische Fehler eingeflossenen sind. Bei Aufgabe 2 ist ein deutlicher Unterschied zwischen den erreichten Punkten bei schlechteren Ergebnissen in der Prüfung und dem neuen Index sichtbar. Teilweise konnten Studierende eine Vielzahl der Punkte für die Bewertung der Konzepte und deren Kombination erlangen, jedoch bei der Prüfungsbenotung deutlich weniger Punkte erreichen.

Ebenso konnten Studierende vereinzelt (siehe Abbildung 72 links) Punkte bei der Prüfungsbewertung erlangen, jedoch ohne Punkte bei der Indexbewertung zu erzielen. Dies kann dadurch erklärt werden, dass bei der Indexbewertung nur die Programmierkonzepte nach dem in Kapitel 5 eingeführten Konzept bei der Bewertung berücksichtigt wurden. Somit konnten nicht alle Punkte aus der Prüfungsbewertung abgedeckt werden, wobei dies nicht das Ziel gewesen ist.

Beim Vergleich für Aufgabe 3 (vgl. Abbildung 73 ist die Differenz zwischen den zwei Bewertungsschemata nicht so stark ausgeprägt wie bei Aufgabe 2. Eine mögliche Antwort auf diese Unterschiede wird in der nachfolgenden Diskussion gegeben.

### 7.3.3 Diskussion

Die Korrelationen der Aufgabenbewertungen und Teilbewertungen (für die Konzepte) mit den Aktivitäten deuten auf einen positiven Zusammenhang hin. Studierende die mehr Aktivitäten für die einzelnen Konzepte durchgeführt haben, konnten dabei bessere Bewertungen erreichen. Dabei kann ebenfalls eine signifikante Korrelation zwischen den Aktivitäten der Iteration und der Bewertung bei der Bearbeitung der Bedingungen beobachtet werden. Diese kann auf die Aufgaben selbst zurückgeführt werden, da den Aufgaben das jeweils primäre Konzept zugeordnet wurde, allerdings erfordern diese Aufgaben meist die Verwendung und Kombination mehrerer Konzepte.

Zwischen den *Aktivitäten der Bedingungen* und der Aufgabenbewertung sowie der *Punkte Bedingungen* konnte keine signifikante Korrelation festgestellt werden. Dies kann durch mehrere verschiedene Ursachen erklärt werden. Erstens kann ein möglicher Zeitdruck in der Prüfung dazu geführt haben, dass ggf. nur eine der beiden Aufgaben bearbeitet wurde. Dies bewirkt eine deutliche Abschwächung der erreichten Indexpunkte unabhängig von der Anzahl der Aktivitäten. Besonders bei Aufgabe 2 (vgl. Abbildung 72) ist eine stärkere Differenz zwischen der erreichten Punktzahl und der Indexpunkte auffällig. Dies deutet auf eine Vielzahl von syntaktischen und kleineren Fehlern hin, die einen negativen Einfluss auf die Prüfungsleistung haben. Für die Bearbeitung der Prüfungsaufgaben hatten die Studierenden die Möglichkeit Musterlösungen und Anmerkungen zu verwenden. Zusätzlich basiert Aufgabe 2 sehr lose auf einer Übungsaufgabe, was zusammen mit den Musterlösungen die hohen Bewertungen beim Index erklären kann. Die Studierenden waren in der Lage die grundlegende Struktur durch Vergleich mit vorhandenen Lösungen (ggf. Umstrukturieren, vgl. Kapitel 3.2.2) zu erkennen, jedoch machten sie dabei eine Vielzahl an Fehlern, was eine Abwertung bei der Prüfungsleistung zur Folge hat. Somit können schlechte Studierende ein besseres Ergebnis beim Index erreichen, ohne mehr geübt zu haben (mehr Aktivitäten). Im Vergleich zu Aufgabe 3 ist hier die durchschnittlich erreichte Punktzahl geringer.

Weitere Unterstützung erhält diese Erklärung durch Betrachtung der Aufgabe 3, bei welcher dieses Problem deutlich geringer ausgeprägt ist. Im Gegensatz zu Aufgabe 2, ist diese an keine Übungsaufgabe angelehnt, sondern es wird nur ein mathematischer Algorithmus beschrieben, der anschließend von den Studierenden umgesetzt werden musste. Die beiden Ergebnisse der unterschiedlichen Bewertungsschemata liegen

deutlich näher zusammen, was darauf hindeutet, dass Studierende, welche einen Lösungsplan erarbeitet haben, diesen auch umsetzen konnten. Aber hier muss ebenfalls der Aspekt des Zeitdrucks berücksichtigt werden, da Studierende möglicherweise die Bearbeitung von Aufgabe 3 vorgezogen haben und somit bessere Ergebnisse erzielten.

Bei beiden Aufgaben kann aus den Vergleichen der Bewertungsschemata geschlossen werden, dass die Studierenden meist eine grundlegende Lösungsidee hatten, jedoch bei der Umsetzung noch Fehler machen. Im Kontext des in Kapitel 7.2 festgestellten Zusammenhangs zwischen Aktivitäten und Anzahl der Aufgaben zur Note kann auf zu wenig Übung (siehe auch Korrelation zwischen *Tage* und *Note* in Tabelle 16) geschlossen werden. Studierende sind nach einem Semester in der Lage grundlegende Lösungsalgorithmen zu formulieren, machen aber noch viele Fehler bei der Umsetzung. Die ermittelten Zusammenhänge können hierbei ebenfalls nur als Tendenz gewertet werden, da nur die Studierenden erfasst wurden, die sich digital beteiligt haben.

## 7.4 Bewertung des Vorlesungskonzeptes (studentische Sicht)

Die bisher beschriebenen Ergebnisse sind alle aus den Aktivitäten der Studierenden und deren Prüfungsergebnissen abgeleitet, weswegen noch eine weitere Perspektive für die abschließende Überprüfung der Hypothesen erfasst wurde. Eine Bewertung von Lehrmethoden und besonders von sozialen Interaktionen wird durch diese zweite Sichtweise unterstützt, da jene die Interpretation der Korrelationen und Plausibilität ermöglichen. Darüber hinaus können mit einem Fragebogen weitere relevante Aspekte abgefragt werden, die nicht digital erfasst werden können, da insbesondere eine Beteiligung ohne Zuhilfenahme eines digitalen Geräts nicht in den bisher betrachteten Daten widergespiegelt wird.

### 7.4.1 Beschreibung

Aus diesem Grund wurde am Ende des Semesters eine summative und anonyme Evaluation in Form eines Fragebogens durchgeführt, um die studentische Sichtweise auf die Lehrveranstaltung und der integrierten aktivierenden Elemente zu erfassen. Der Fragebogen wurde auf Basis des Heidelberger Inventars für Evaluation (HILVE) [RA94] konzipiert, wobei dabei eine Reduzierung der Fragen erfolgte. Zugleich wurden einige Fragen (auch *Items*) angepasst und Fragen hinzugefügt, um die Besonderheiten der Lehrveranstaltung zu erheben. Für die Befragung wurden Items aus den Dimensionen *Lehrerfolg* und *Merkmale der Studierenden* verwendet und zusätzlich um einzelne

**Tabelle 26:** Ergebnisse - Studiengang (Fragebogen)

| **Antworten** | Anzahl | Anteil |
|---|---|---|
| Automatisierung und Robotik | 13 | 36% |
| Erneuerbare Energien | 10 | 28% |
| Elektro- und Informationstechnik | 8 | 22% |
| Nicht angegeben | 5 | 14% |

Fragen aus dem Bereich des *Unterrichtshandelns des Dozenten* erweitert. Ziel der Befragung war es weitere unterstützende Informationen zu den Hypothesen zu erfassen und entsprechend auszuwerten.

Der Fragebogen ist in Kapitel B.5 angehängt, welcher folgende Themen umfasst:

- Mitarbeit und Beteiligung der Studierenden
- Verwendung von Beispielen und Herstellung des Praxisbezugs
- Verwendung der Plattform
- Selbsteinschätzung in Bezug auf das Programmieren und Sachverhalte
- Einstellung gegenüber Programmieren

Die Befragung wurde zu Beginn der letzten Vorlesung, die als Prüfungsvorbereitung genutzt wurde, ausgeteilt und am Ende eingesammelt.

### 7.4.2 Ergebnisse

Insgesamt wurden 36 Fragebögen ausgefüllt und für die Auswertung verwendet. Zunächst wurden für die einzelnen Fragen die Häufigkeiten für jede Antwortmöglichkeit ausgezählt, welche nachfolgend dargelegt werden.

Die Lehrveranstaltung wird derzeit für drei verschiedene Studiengänge der Elektrotechnik angeboten. In Tabelle 26 ist die Verteilung der Studierenden angegeben. Ausgehend von einer gleichen Verteilung am Semesterbeginn von je 40 Studierenden pro Studiengang würde dies einer Rücklaufquote von ca. 30% entsprechend. Da jedoch im Laufe des Semesters nur ca. 60 bis 70 Studierende die Vorlesung besucht haben, kann von einer Rücklaufquote von ungefähr 50% ausgegangen werden. Fünf Studierende haben keinen Studiengang auf dem Fragebogen angekreuzt.

**Tabelle 27**: Ergebnisse - Zum Mitdenken und Durchdenken des Stoffes wurde angeregt (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 16 | 44% |
| 4 | 18 | 50% |
| 3 | 1 | 3% |
| 2 | 1 | 3% |
| 1 (trifft nicht zu) | 0 | 0 % |

**Tabelle 28**: Ergebnisse - In der Lehrveranstaltung habe ich mich aktiv beteiligt (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 2 | 6% |
| 4 | 10 | 28% |
| 3 | 10 | 28% |
| 2 | 11 | 31% |
| 1 (trifft nicht zu) | 2 | 6 % |

### Mitarbeit

Zuerst werden die Fragen bzw. Items bezüglich der Mitarbeit der Studierenden in der Vorlesung betrachtet. Das Heidelberger Inventar zur Lehrevaluation umfasst dabei bereits mehrere Items, welche zusätzlich um eine Abfrage der Beteiligung an den Aufgaben in der Vorlesung erweitert wurden. Für die Fragen wurde eine fünfstufige Likert-Skala (*trifft nicht zu* bis *trifft voll zu*) verwendet (vgl. [BD16, S.269]). Dabei wurde für *trifft nicht zu* der Wert 1 und für *trifft voll zu* der Wert 5 zugeordnet. Die Zwischenwerte 2 bis 4 sind somit Abstufungen der Bewertung.

Tabelle 27 zeigt, dass der Großteil der Studierenden zum Mitdenken angeregt wurde. Im Gegensatz dazu steht die eigene aktive Beteiligung der Studierenden selbst, welche von sich aus Fragen gestellt bzw. ihre Lösung zur Diskussion zur Verfügung gestellt haben (vgl. Tabelle 28). Die Ergebnisse zeigen eine eher mittlere Beteiligung seitens der Studierenden, da der Großteil der Antworten sich auf den Bereich von 2 - 3 konzentriert. Bei der Beteiligung haben sich die Studierenden trotzdem meistens frei und äußerungsfähig gefühlt (vgl. Tabelle 29). Tabelle 30 zeigt ein ähnliches Bild wie bei der eigenen aktiven Beteiligung der Studierenden. Ungefähr die Hälfte der Studierenden (Antwort 5 und 4) geben an, dass sie die Aufgaben häufig in der Vorlesung bearbeitet haben. Während die andere Hälfte eher weniger Aufgaben im Unterricht bearbeitet

**Tabelle 29**: Ergebnisse - Beim Einbringen eigener Beiträge habe ich mich frei und äußerungsfähig gefühlt (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 5 | 14% |
| 4 | 17 | 47% |
| 3 | 9 | 25% |
| 2 | 1 | 3% |
| 1 (trifft nicht zu) | 1 | 3% |
| Keine Antwort | 3 | 8% |

**Tabelle 30**: Ergebnisse - Ich habe die in der Vorlesung gestellten Aufgaben häufig bearbeitet. (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 9 | 25% |
| 4 | 10 | 28% |
| 3 | 13 | 36% |
| 2 | 4 | 11% |
| 1 (trifft nicht zu) | 0 | 0% |

**Tabelle 31:** Ergebnisse - Es werden kommunikative Lehrformen, wie z. B. Diskussion, Aufgaben oder Quizzes eingesetzt (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 6 | 17% |
| 4 | 21 | 58% |
| 3 | 7 | 19% |
| 2 | 1 | 3% |
| 1 (trifft nicht zu) | 0 | 0% |
| Keine Antwort | 1 | 3% |

**Tabelle 32:** Ergebnisse - Die Dozentin/Der Dozent fördert Fragen und aktive Mitarbeit (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 12 | 33% |
| 4 | 21 | 58% |
| 3 | 2 | 6% |
| 2 | 1 | 3% |
| 1 (trifft nicht zu) | 0 | 0% |

haben (Antwort 3 und 2). Dennoch empfanden die meisten Studierenden, dass kommunikative Lehrformen im Unterricht eingesetzt wurden (vgl. Tabelle 31), wobei die Mehrheit Antwort 4 gewählt hat.

Aus diesem Grund wurden zusätzlich zwei Items zur Überprüfung zur Aufforderung zur Mitarbeit im Fragebogen verwendet. Die Mehrheit der Studierenden gab an, dass der Dozent die aktive Mitarbeit und das Stellen von Fragen fördert, was ein Bestandteil des entwickelten Konzepts ist (vgl. Tabelle 32).

**Tabelle 33:** Ergebnisse - Ich wurde zur Mitarbeit aufgefordert. (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 2 | 6% |
| 4 | 19 | 53% |
| 3 | 14 | 39% |
| 2 | 1 | 3% |
| 1 (trifft nicht zu) | 0 | 0% |

**Tabelle 34:** Ergebnisse - Ich fühle mich in der Lage, kleine Programme zu schreiben. (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 14 | 39% |
| 4 | 14 | 39% |
| 3 | 3 | 8% |
| 2 | 4 | 11% |
| 1 (trifft nicht zu) | 1 | 3% |

**Tabelle 35:** Ergebnisse - Es macht mir Spaß zu programmieren. (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) | Erneuerbare Energien |
|---|---|---|---|
| 5 (trifft voll zu) | 15 | 42% | 1 |
| 4 | 7 | 19% | 1 |
| 3 | 11 | 31% | 5 |
| 2 | 3 | 8% | 3 |
| 1 (trifft nicht zu) | 0 | 0% | 0 |

In der Kontrollfrage zur Aufforderung zur Mitarbeit (vgl. Tabelle 33), welche die individuelle Aufforderung betont, zeigt sich eine schwächere Ausprägung als in Tabelle 32. Somit wird zwar zur Mitarbeit seitens des Dozenten aufgefordert, aber es fühlen sich nicht alle Studierenden direkt bzw. weniger stark angesprochen.

### Selbsteinschätzung

Der Fragebogen enthält zusätzlich Fragen zur Selbsteinschätzung seitens der Studierenden über die Programmierfähigkeit. Tabelle 34 zeigt, dass sich der Großteil der Studierenden in der Lage fühlt, kleine Programme zu schreiben. Nur 4 Studierende fühlen sich nach Ende der Lehrveranstaltung eher nicht dazu in der Lage, wovon 2 Studierende keine Angabe zum Studiengang gemacht haben. Die anderen beiden Studierenden, die sich nicht in der Lage fühlen, stammen aus dem Studiengang *Erneuerbare Energien*.

Bei der Frage nach dem Spaß am Programmieren bilden sich zwei Gruppen in den Ergebnissen (vgl. Tabelle 35). Einerseits haben viele Studierende einen großen Spaß am Programmieren und, andererseits existiert ein fast ähnlich große Gruppe, welche eher eine neutrale Haltung hat. Auffällig ist dabei, dass niemand von den Befragten *trifft nicht zu* angekreuzt hat. Werden die Ergebnisse auf die einzelnen Studiengänge aufgeteilt, ergibt sich ein ähnliches Bild wie zuvor. Den Studierenden aus dem Studiengang

**Tabelle 36:** Ergebnisse - Ich habe die Möglichkeit genutzt, die Aufgaben auch später online zu bearbeiten (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 16 | 44% |
| 4 | 11 | 31% |
| 3 | 8 | 22% |
| 2 | 1 | 3% |
| 1 (trifft nicht zu) | 0 | 0% |

**Tabelle 37:** Ergebnisse - Ich habe die Übungen häufig von zuhause aus bearbeitet (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 5 | 14% |
| 4 | 12 | 33% |
| 3 | 12 | 33% |
| 2 | 3 | 8% |
| 1 (trifft nicht zu) | 4 | 11% |

der *Erneuerbaren Energien* macht Programmieren weniger Spaß als den anderen.

### Bearbeitung von Aufgaben

Als Nächstes wird die Art und der Umfang der Bearbeitung der Aufgaben näher analysiert. In der Auswertung der Aktivitäten im Bezug auf die studentische Partizipation wurde bereits festgestellt, dass einige Studierende die Möglichkeit genutzt haben, die Aufgaben später zu bearbeiten. Dies ist auch aus Sicht der abnehmenden Beteiligung in den angebotenen Übungen interessant. Aus Tabelle 36 kann abgeleitet werden, dass ein Großteil der Befragten die Möglichkeit zur späteren Bearbeitung, welche durch die entwickelte Plattform gegeben ist, genutzt haben.

Die Frage nach der Bearbeitung der Übungsaufgaben von zuhause aus, zeigt ein etwas anderes Bild (vgl. Tabelle 37). Es gibt einige Studierenden, die von dieser Möglichkeit sehr stark, oder keinen Gebrauch gemacht haben. Gleichzeitig gibt es Studierende (Antwortmöglichkeit 3), welche die Aufgaben nur teilweise zuhause bearbeitet haben. Schlussendlich die Frage nach der Bearbeitung der Aufgabe an den Terminen (vgl. Tabelle 38, dass die Mehrheit der Befragten die Aufgaben nicht an den Übungsterminen bearbeitet hat.

**Tabelle 38:** Ergebnisse - Ich habe die Übungen häufig an den Übungsterminen bearbeitet (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 3 | 8% |
| 4 | 5 | 14% |
| 3 | 5 | 14% |
| 2 | 9 | 25% |
| 1 (trifft nicht zu) | 14 | 39% |

**Tabelle 39:** Ergebnisse - Die/Der Lehrende benutzte oft Beispiele, die zum Verständnis der Lehrinhalte beitrugen (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 24 | 67% |
| 4 | 11 | 31% |
| 3 | 1 | 3% |
| 2 | 0 | 0% |
| 1 (trifft nicht zu) | 0 | 0% |

### Programmierumgebung

Die entwickelte Plattform und Programmierumgebung wurden ebenfalls im Fragebogen durch mehrere Items evaluiert, um Aussagen über die Verwendung und Annahme zu treffen. Ein elementarer Bestandteil des Konzepts sieht die Verwendung von situativen und passenden Beispielen vor, welche direkt in der Vorlesung ausgeführt und bearbeitet werden. Aus Tabelle 39 lässt sich ableiten, dass die verwendeten Beispiele aus Sicht der Studierenden sehr zum Verständnis der Lehrinhalte beigetragen haben. Die ausgegebenen Lehrmaterialien, welche primär aus den interaktiven Dokumenten bestehen, welche mittels der in Kapitel 6 vorgestellten Plattform den Studierenden zur Verfügung gestellt wurden, werden ebenfalls als hilfreich empfunden (vgl. Tabelle 40). Bei der Frage nach der Bedienung der Plattform (vgl. Tabelle 41) ergab die Auswertung, dass die Mehrheit der Befragten mit der Plattform gut zurechtgekommen ist. Einzig bei der Beurteilung zum Erstellen von Programmen in der web-basierten Programmierumgebung, gab ein Befragter an, dass er eher nicht damit zurechtgekommen ist (vgl. Tabelle 42). Der Großteil hatte jedoch keine Probleme beim Umgang mit der Entwicklungsumgebung.

**Tabelle 40**: Ergebnisse - Die von der/dem Lehrenden ausgegebenen Materialien (z. B. Literatur, Skript, E-Learning-Inhalte, etc.) haben mir sehr geholfen, den Stoff zu erarbeiten. (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 22 | 61% |
| 4 | 9 | 25% |
| 3 | 3 | 8% |
| 2 | 0 | 0% |
| 1 (trifft nicht zu) | 1 | 3% |

**Tabelle 41**: Ergebnisse - Ich bin mit der E-Learning Plattform (trycoding.io) gut zurechtgekommen. (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 20 | 56% |
| 4 | 9 | 25% |
| 3 | 5 | 14% |
| 2 | 0 | 0% |
| 1 (trifft nicht zu) | 0 | 0% |
| Keine Antwort | 2 | 6% |

**Tabelle 42**: Ergebnisse - Es war leicht, Programme im Webbrowser zu erstellen und auszuführen. (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 20 | 56% |
| 4 | 9 | 25% |
| 3 | 5 | 14% |
| 2 | 0 | 0% |
| 1 (trifft nicht zu) | 0 | 0% |
| Keine Antwort | 2 | 6% |

**Tabelle 43:** Ergebnisse - Die/Der Lehrende passte das Niveau der Lehrveranstaltung an den Wissensstand der Studierenden an (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| 5 (trifft voll zu) | 14 | 39% |
| 4 | 17 | 47% |
| 3 | 3 | 8% |
| 2 | 1 | 3% |
| 1 (trifft nicht zu) | 0 | 0% |
| Keine Antwort | 1 | 3% |

**Tabelle 44:** Ergebnisse - Die Anforderungen der Lehrveranstaltung waren: (Fragebogen)

| Antworten | Anzahl | Anteil (gerundet) |
|---|---|---|
| Viel zu hoch | 2 | 6% |
| Genau richtig | 33 | 92% |
| Viel zu niedrig | 1 | 3% |

### Weitere Ergebnisse und Korrelationen

Ein weiteres Ziel des Konzepts ist die Anpassung der Lehre an das aktuelle Niveau der Studierenden. Die Ergebnisse der Frage über die Anpassung der Lehrveranstaltung an den Wissensstand der Studierenden deuten auf eine gelungene Umsetzung hin. Über 80% der Befragten finden, dass der Lehrende das Niveau an den Wissensstand angepasst hat. Zudem bewerteten über 90% der Studierenden die Anforderungen der Lehrveranstaltung als *genau richtig* (siehe Tabelle 44).

Weiterhin wurden folgende signifikante ($\alpha = 0,01$) Korrelationen zwischen den Fragen ermittelt. Dabei wurde diesmal der Pearson-Korrelationskoeffizient berechnet. Studierende, die es leicht finden, Programme im Browser zu erstellen und auszuführen, wurden ebenfalls mehr zum Durchdenken des Stoffes angeregt und empfinden die ausgegebenen Materialien als hilfreich (vgl. Tabelle 45). Aufgrund des gewählten Signifikanzniveaus muss der Zusammenhang zwischen der einfachen Erstellung von Programmen und dem Zurechtkommen auf der Plattform als nicht signifikant zurückgewiesen werden. Befragte, die die eingeführten Begriffe gut wiedergeben können, empfinden ebenfalls, dass es leicht war die Programme web-basiert zu erstellen. Jedoch konnte kein signifikanter Zusammenhang mit dem Verstehen der Sachverhalte gefunden werden.

**Tabelle 45:** Korrelationen - Es war leicht, Programme im Webbrowser zu erstellen und auszuführen (Fragebogen)

| Frage | r (gerundet) | p (gerundet) |
|---|---|---|
| Zum Mitdenken und Durchdenken des Stoffes wurde angeregt. | 0,57 | 0,0001 |
| Die Lehrveranstaltung wurde in interessanter Form gehalten. | 0,51 | 0,0005 |
| Die von der/dem Lehrenden ausgegebenen Materialien (z. B. Literatur, Skript, E-Learning-Inhalte, etc.) haben mir sehr geholfen, den Stoff zu erarbeiten. | 0,63 | 0,0004 |
| Ich bin mit der E-Learning Plattform (trycoding.io) gut zurechtgekommen. | 0,52 | 0,0242 |
| Die in dieser Lehrveranstaltung eingeführten Begriffe/Sachverhalte kann ich wiedergeben. | 0,58 | 0,0086 |
| Die in dieser Lehrveranstaltung eingeführten Begriffe/Sachverhalte habe ich verstanden. | 0,42 | 0,1783 |

Neben den bisher zusammengefassten Ergebnissen zu den Fragen und der Plattform, sind aus Sicht des Konzepts die situativen Beispiele, welche zur Motivation der Studierenden dienen sollen, bei der Bewertung relevant. Die Korrelationen Tabelle 46 zeigen, dass sich der Spaß am Programmieren auf das Verständnis der eingeführten Sachverhalte und die Selbsteinschätzung auswirkt. Ebenfalls gibt es einen Zusammenhang zwischen den Beispielen, die zum Verständnis beitragen und dem Spaß.

Durch die einzelnen Phasen des Konzepts sollen die Studierenden zum Mitdenken und Durchdenken der neuen Konzepte und des Stoffes angeregt werden. Wie erwartet korreliert die Frage nach dem Mitdenken und Durchdenken (vgl. Tabelle 47) mit der Frage nach der Förderung von Fragen und aktiver Mitarbeit. Zudem gibt es zwischen dem Mitdenken und Durchdenken des Stoffes und der gewählten Form der Lehrveranstaltung, dem Erstellen von Programmen im Browser sowie den ausgegebenen Materialien einen positiven Zusammenhang. Einzig die persönliche Aufforderung zur Mitarbeit korreliert nicht mit der Aufforderung zum Durchdenken des Stoffes.

Schlussendlich wurden die Zusammenhänge zwischen dem ausgegebenen Material und den anderen Fragen untersucht. Studierende, welche die ausgegebenen interaktiven Materialien als hilfreich empfinden, konnten den Aufbau der Lehrveranstaltung besser nachvollziehen. Zusätzlich besteht ein Zusammenhang zwischen der Anregung zum Durchdenken des Stoffes und den Beispielen, die zum Verständnis beitragen. Schließlich sind diese Befragten mit der Plattform besser zurechtgekommen und empfinden das Erstellen und Ausführen von Programmen im Browser als leicht.

**Tabelle 46**: Korrelationen - Es macht mir Spaß zu Programmieren (Fragebogen)

| Frage | r (gerundet) | p (gerundet) |
|---|---|---|
| Die/Der Lehrende benutzte oft Beispiele, die zum Verständnis der Lehrinhalte beitrugen. | 0,57 | 0,013 |
| Die von der/dem Lehrenden ausgegebenen Materialien (z. B. Literatur, Skript, E-Learning-Inhalte, etc.) haben mir sehr geholfen, den Stoff zu erarbeiten. | 0,4 | 0,085 |
| Die in dieser Lehrveranstaltung eingeführten Begriffe/Sachverhalte kann ich wiedergeben. | 0,65 | 0,0001 |
| Die in dieser Lehrveranstaltung eingeführten Begriffe/Sachverhalte habe ich verstanden. | 0,67 | 0 |
| Ich habe die in der Vorlesung gestellten Aufgaben häufig bearbeitet. | 0,68 | 0,0001 |
| Ich fühle mich in der Lage, kleine Programme zu schreiben. | 0,58 | 0,0007 |

**Tabelle 47**: Korrelationen - Zum Mitdenken und Durchdenken des Stoffes wurde angeregt. (Fragebogen)

| Frage | r (gerundet) | p (gerundet) |
|---|---|---|
| Der inhaltliche Aufbau der Lehrveranstaltung war logisch/nachvollziehbar. | 0,61 | 0,0001 |
| Die Lehrveranstaltung wurde in interessanter Form gehalten. | 0,58 | 0,0007 |
| Die/Der Lehrende benutzte oft Beispiele, die zum Verständnis der Lehrinhalte beitrugen. | 0,52 | 0,0001 |
| Die Bedeutung/Der Nutzen der behandelten Themen wird vermittelt. | 0,56 | 0,001 |
| Die von der/dem Lehrenden ausgegebenen Materialien (z. B. Literatur, Skript, E-Learning-Inhalte, etc.) haben mir sehr geholfen, den Stoff zu erarbeiten. | 0,66 | 0 |
| Es war leicht, Programme im Webbrowser zu erstellen und auszuführen. | 0,57 | 0,0001 |
| Die Dozentin/Der Dozent fördert Fragen und aktive Mitarbeit. | 0,56 | 0,0014 |
| Ich wurde zur Mitarbeit aufgefordert. | 0,13 | 0,23 |

**Tabelle 48:** Korrelationen - Zum Mitdenken und Durchdenken des Stoffes wurde angeregt. (Fragebogen)

| Frage | r (gerundet) | p (gerundet) |
|---|---|---|
| Der inhaltliche Aufbau der Lehrveranstaltung war logisch/nachvollziehbar. | 0,65 | 0,0003 |
| Zum Mitdenken und Durchdenken des Stoffes wurde angeregt. | 0,66 | 0 |
| Die/Der Lehrende benutzte oft Beispiele, die zum Verständnis der Lehrinhalte beitrugen. | 0,58 | 0,0028 |
| Die Bedeutung/Der Nutzen der behandelten Themen wird vermittelt. | 0,58 | 0,0006 |
| Ich bin mit der E-Learning Plattform (trycoding.io) gut zurechtgekommen. | 0,59 | 0,0019 |
| Es war leicht, Programme im Webbrowser zu erstellen und auszuführen. | 0,63 | 0,0004 |

### 7.4.3 Diskussion

Fasst man die Ergebnisse der Befragung zusammen, ergeben sich mehrere wichtige Aussagen für die Evaluation des Konzepts. Die Auswertung hat unter anderem ergeben, dass die Studierenden vom Dozenten zur Mitarbeit aufgefordert werden. Jedoch fühlt sich dabei nicht jeder Studierende persönlich aufgefordert (siehe Tabelle 47 und Tabelle 33). Teilweise kann dies dem Umstand geschuldet sein, dass sich nicht alle Studierenden im Unterricht einbringen möchten (vgl. Tabelle 28), sondern jeweils nur ein Teil sich an den Aufgaben beteiligt. Dies deckt sich mit den Ergebnissen der Beteiligung, die aus den Aktivitäten der Studierenden berechnet wurde (vgl. Tabelle 15). Von den Studierenden wird ebenfalls das Konzept als kommunikative Lehrform aufgefasst, wobei nicht alle Programmierkonzepte strikt nach dem entwickelten Konzept eingeführt wurden.

Der Großteil der Befragten gab an, dass sie sich in der Lage fühlen, kleine Programme zu schreiben. Die Studierenden, die sich nicht in der Lage fühlen, stammen überwiegend aus dem Studiengang der Erneuerbaren Energien. Eben diese Gruppe hat auch deutlich weniger Spaß am Programmieren als Studierende der anderen beiden Studiengänge. Dies kann ggf. an den gewählten Beispielen liegen und, dass der Praxisbezug nicht im nötigen Umfang auf diesen Studiengang abgestimmt wurde. Auf der anderen Seite wurden die gewählten Beispiele zur Verdeutlichung der Sachverhalte als mehrheitlich hilfreich bewertet (vgl. Tabelle 39). Der Studiengang der Erneuerbaren Energien leidet derzeit an mehreren Problemen, die dem Umstand geschuldet

sind, dass Studierende mit anderen Vorstellungen anfangen diesen zu studieren. Dies führt zu einer hohen Durchfall- und Abbruchquote, da dieser Studiengang ebenfalls alle Grundlagen der Elektrotechnik beinhaltet und eher auf die technischen Aspekte fokussiert ist. Allgemein kann nicht vorausgesetzt werden, dass jeder Studierende der Elektrotechnik Spaß am Programmieren findet. Studierende, welche jedoch Spaß am Programmieren haben, bewerteten die verwendeten Beispiele besser und haben die Sachverhalte eher verstanden. Darüber hinaus habe diese im Vergleich häufiger die Aufgaben im Unterricht bearbeitet (vgl. Tabelle 46).

Die Fragen über die Bearbeitung der Aufgaben zeigen ein Problem bei den Übungsaufgaben auf. Ein größerer Anteil der Studierenden hat die Übungen nicht an den betreuten Übungsterminen bearbeitet. Es gaben zwar viele Studierende an, Aufgaben auch später online zu bearbeiten (siehe Tabelle 36), was sich ebenfalls in der Beteiligung bei den Aufgaben im Unterricht widerspiegelt (vgl. Tabelle 15), da einige Studierenden die Aufgaben auch später bearbeitet haben. Dennoch zeigen die Antworten bei der Frage nach Bearbeitung der Übungen von zuhause aus, dass dies auch nur von einem Teil gemacht wurde (vgl. Tabelle 37). Daraus folgt, dass einige Studierenden die Aufgaben gar nicht bearbeitet haben. Dies ist aber nicht unbedingt dem Konzept und der Plattform geschuldet, sondern eher der Freiwilligkeit zur Teilnahme an den Übungen und Vorlesungen. Die Plattform bietet vielmehr Studierenden die Möglichkeit die Inhalte später leicht nachzuarbeiten, was, wie bereits in den vorherigen Auswertungen gezeigt wurde, auch angenommen wird. Trotzdem kann an dieser Stelle überlegt werden, wie die leistungsschwachen Studierenden erreicht und motiviert werden können, um diese in den betreuten Übungen zu unterstützen bzw. diese zu besuchen.

Ein wichtiger Aspekt bei der Durchführung des Konzepts, ist einerseits die Interaktivität im Unterricht, aber ebenso die Möglichkeit die Inhalte nachzubereiten und zumindest diese teilweise selbstgesteuert zu erarbeiten. Die Befragung hat ergeben, dass die Beispiele als sehr hilfreich angesehen werden. Ebenfalls empfinden die Studierenden die entwickelte Plattform und die damit bereitgestellten Inhalte als hilfreich, um den Stoff zu erarbeiten (siehe Tabelle 40). Dies inkludiert sowohl die Präsenzphasen wie auch die Nachbereitung zuhause. Der Umgang mit der Plattform wurde von den meisten Befragten als problemlos bewertet, was besonders die Erstellung und das Ausführen von Programmen im Browser beinhaltet (siehe Tabelle 41 und Tabelle 42). Studierenden, denen das Erstellen der Programme leicht gefallen ist, bewerteten das Lehrkonzept und die Materialien besser. Zusätzlich fühlen diese sich zum Mitdenken und Durchdenken des Stoffes angeregt (vgl. Tabelle 45). Besonders dieser Zusammenhang zwischen der Anregung zum Durchdenken und der Bedienung der Plattform war eine der Annahmen, welche zur Entwicklung des Systems geführt haben (vgl. Kapitel 6). Neben Interaktivität und dem Durchdenken ist ebenfalls die Anpassung des Niveaus und des Tempos an den Wissensstand der Studierenden ein essentieller Be-

standteil. Diese Anpassung wurde von den Befragten als gut bewertet, was als positive Tendenz zur Bewährung des Konzepts gewertet werden kann. Über 80% der Befragten stimmten der Aussage zu, dass das Niveau der Lehrveranstaltung angepasst wurde.

## 7.5 Diskussion der Hypothesen

Zum Abschluss der Evaluation sollen nun die in Kapitel 2.4 aufgestellten Hypothesen betrachtet werden. Die einzelnen Hypothesen werden auf Basis der bisherigen Ergebnisse einerseits mit Hypothesentests und andererseits argumentativ überprüft und bewertet.

### 7.5.1 Einfluss auf die studentische Partizipation

Zuerst soll die Hypothese 1, welche einen positiven Einfluss auf die studentische Partizipation erwartet, überprüft werden. Ausgangspunkt für den Vergleich stellen die Lehrveranstaltungen in den vorherigen Jahren dar. In diese wurden nur wenige Aufgaben im Unterricht integriert, während das Einüben der neuen Programmierkonzepte und deren praktische Anwendung in die Übungen verlagert wurde. Es wurden zwar am Ende schriftliche Übungen und Zwischenfragen bereits eingesetzt, jedoch primär als Vorbereitung auf die Klausur und nicht um neue Konzepte zielgerichtet einzuüben. Aus diesem Grund gibt es keine quantitativen Vergleichsdaten, welche die studentische Teilnahme an Übungen im Unterricht aus den Vorjahren erfasst. Somit ist kein statistischer Hypothesentest möglich, weswegen zur Überprüfung die in den vorherigen Kapiteln beschriebenen Daten diskutiert werden.

So kann, ausgehend von der berechneten Anzahl der Studierenden, welche sich im Unterricht mit den Aufgaben beschäftigt haben, eine grundlegende Beteiligung der Studierenden an den neu eingeführten Aktivitäten belegt werden. Durchschnittlich hat etwa ein Drittel bis die Hälfte der Studierenden die Aufgaben im Unterricht digital bearbeitet (vgl. Kapitel 7.2). Bei der Durchführung der Lehrhandlungen hat sich gezeigt, dass die Studierenden oftmals in Gruppen bzw. Paaren zusammen arbeiten, was die Anzahl zusätzlich erhöht. Ebenfalls konnten für jede Aufgabe freiwillige Studierende gefunden werden, welche ihre Lösungen zur Diskussion zur Verfügung gestellt haben. Dabei hat sich jedoch bei der Umsetzung gezeigt, dass die zeitliche Planung für die Umsetzung teilweise entscheidend ist, da besonders Aufgaben zum Ende der Vorlesung nur sehr kurz besprochen werden konnten. Allgemein hat sich eine gute Beteiligung der Studierenden bei den im Unterricht durchgeführten Aufgaben gezeigt. Denn nicht nur die strikt nach dem in Kapitel 5 umgesetzten Aufgaben, sondern auch weitere (vgl. Tabelle 50) wurden von den Studierenden bearbeitet. Jedoch wurde im Laufe

des Semesters eine nachlassende Beteiligung bei den Übungsterminen beobachtet. Besonders die Aufgaben und Übungen zum Ende der Lehrveranstaltung, welche sich nur schwer in einer schriftlichen Prüfung abfragen lassen, wurden von deutlich weniger Studierenden bearbeitet. Darunter fallen Aufgaben, welche die numerische Auswertung und die Erstellung von Diagrammen thematisieren (vgl. Tabelle 50). Zudem haben andere Lehrveranstaltungen, welche Praktikumsberichte und teilweise Anwesenheitspflicht als Zulassungsvoraussetzung zur Prüfung vorschreiben, einen negativen Einfluss auf die Beteiligung. Diese werden von den Studierenden zum Teil als höher priorisiert als Programmieren, da hier keinerlei Voraussetzungen oder Pflichttermine, getreu der Vorgaben nach Bologna, eingesetzt werden. Besonders im ersten Studiensemester gibt es in naturwissenschaftlichen Fächern bereits einige Studierende, welche nach einigen Wochen nicht mehr zu den Veranstaltungen erscheinen oder das Studium abbrechen, welche aber statistisch nicht erfasst werden bzw. erfasst werden können.

Gleichwohl eignet sich das Konzept, um Studierende stärker im Unterricht zu beteiligen und somit den Unterricht entsprechend von den entstehenden Fragen und Rückmeldungen zumindest teilweise lenken zu lassen. Die Auswertung des Fragebogens zeigt, dass die Studierenden sich zur aktiven Mitarbeit aufgefordert fühlen, wenn auch nicht alle Studierenden dabei mitarbeiten möchten (vgl. Tabelle 27 und Tabelle 28). Der Großteil der Studierenden gibt an, dass zum Mitdenken und Durchdenken des Stoffes angeregt wurde. Die Frage nach der Verwendung von Beispielen, welche zum Verständnis beitragen, wurde von der Mehrheit der Studierenden bestätigt (vgl. Tabelle 39). Eben diese Beispiele, welche sofort von den Studierenden und vom Lehrenden ausgeführt und verändert oder erst entwickelt werden können, stellen ein fundamentales Element des entwickelten Konzepts dar. Tabelle 47 zeigt dabei, dass ein Zusammenhang zwischen der Anregung zum Mitdenken und den ausgegebenen Materialien (dem interaktiven Skript) und der Plattform existiert. Zusammengefasst kann somit eine positive Tendenz bei der Beteiligung der Studierenden festgestellt werden, denn die Studierenden nutzen die angebotenen Aktivitäten im Unterricht.

### 7.5.2 Einfluss auf die Programmierfähigkeit

Die zweite Hypothese stellt die Vermutung auf, dass Aktivitäten im Sinne des *Aktiven Lernens* einen positiven Einfluss auf die Programmierfähigkeit haben. Ausgehend von den gesammelten Daten kann eine statistische Überprüfung des Zusammenhangs zwischen den Aktivitäten der Studierenden und der erreichten Note in der Klausur vorgenommen werden. Darüber hinaus werden die Ergebnisse, welche mittels des in Kapitel 3.3.3 entwickelten Messinstruments zur Erfassung der Programmierfähigkeit diskutiert. Das entwickelte Instrument misst dabei jedoch nur die Kombination von Programmierkonzepten, um Probleme bzw. Teilprobleme zu lösen. Eine Anwendung

auf die Entwicklung von größeren Softwaresystemen oder den objektorientierten Entwicklungsansatz wurde dabei nicht beachtet, da ein prozedurales Programmierparadigma in der Lehrveranstaltung vermittelt wurde. Die verwendete Programmiersprache *Python* ist zwar objektorientiert, jedoch lag der Fokus dieser Lehrveranstaltung nicht auf dem objektorientierten Entwurf.

Für die Überprüfung wird zuerst die Nullhypothese $H_0$ formuliert, welche aussagt, dass kein Zusammenhang zwischen den Aktivitäten der Studierenden und der Prüfungsnote besteht. Um die Aussage zu überprüfen werden die bereits in Kapitel 7.2 dargelegten Ergebnisse betrachtet. In der Auswertung der durch die Plattform erhobenen Daten konnten signifikante Korrelationen zwischen der Anzahl der Aktivitäten und der Note festgestellt werden (siehe Tabelle 16 und Tabelle 17). Bei der Analyse der Daten wurde aber festgestellt, dass es Studierende gibt, die einerseits mit sehr wenigen Aktivitäten sehr gute Leistungen vollbringen und andererseits Studierende mit vielen Aktivitäten schlechte Leistungen erlangen. Aus diesem Grund wurden weitere Merkmale aus den Daten berechnet, wie beispielsweise die Anzahl der Aufgaben, welche ein Studierender bearbeitet hat. Diese Anzahl und auch die Anzahl der reinen Aufgaben, welche im Unterricht bearbeitet wurden, korrelieren stärker mit der erreichten Note als die Aktivitäten. Alle Korrelationen können als signifikant bezeichnet werden, da die p-Werte unter dem festgelegten Signifikanzniveau liegen. Die Ergebnisse decken sich mit denen von Hassinen et al. [vgl. HM06], welche einen positiven Zusammenhang zwischen Noten und der Anzahl an abgegebenen Übungsaufgaben nachgewiesen haben. Somit kann aus den Daten ein positiver Einfluss der Aktivitäten auf die Prüfungsnote belegt werden.

Dagegen sagt die Alternativhypothese ($H_1$) aus, dass die Aktivitäten einen positiven Einfluss auf die Programmierfähigkeit haben. Die Prüfungsleistung umfasst allerdings auch theoretische Grundlagen, da in der Prüfung verschiedene Lernziele abgefragt werden. Aus diesem Grund wurde bereits in Kapitel 3.1 eine Definition für Programmierfähigkeit anhand eines Programmierprozesses abgeleitet und anschließend ein Messinstrument entwickelt. Dieses Instrument wurde in Kapitel 7.3 für jeden Studierenden als Messinstrument (ein Punkteindex) zur Bewertung der Programmierfähigkeit angewendet. Dabei wurden drei Implementierungsaufgaben aus der Prüfung ausgewählt, für welche anschließend ein Bewertungsschema auf Basis des Instruments festgelegt wurde. Nachfolgend wurden die erreichten Punktzahlen der einzelnen Aufgaben und der Umsetzung und Anwendung der einzelnen Programmierkonzepte zum Lösen von Teilproblemen (vgl. Programmiervorgang in Kapitel 3.1) ermittelt. Diese Punkte wurden anschließend mit den Aktivitäten und Aufgaben je Konzept korreliert, um einen Zusammenhang nachzuweisen.

Die Ergebnisse deuten auf einen positiven Zusammenhang zwischen der Anzahl der Aktivitäten bei Iterationen (finite sowie infinite Iteration), aber nicht bei Bedingun-

gen hin (vgl. Tabelle 23). Ein Grund für das eher schlechte Abschneiden bei den Bedingungen kann in den Prüfungsaufgaben selbst liegen, da besonders Aufgabe 2 (vgl. Kapitel 7.3.1) mehrere Bedingungen zur Lösung voraussetzt. Diese konnten trotz der Aufgaben und Übungen teilweise nicht im Rahmen der schriftlichen Prüfung richtig gelöst werden, sodass eine Abwertung in den Punkten erfolgt. Im Vergleich mit Aufgabe 3 (vgl. Kapitel 7.3.1) wurden in Aufgabe 2 weniger Punkte erreicht. Dies kann auch daran liegen, dass die Studierenden sich auf Aufgabe 3 konzentriert haben und aufgrund eines Zeitmangels Aufgabe 2 nicht mehr vollständig bearbeiten konnten. Darauf deutet auch der Vergleich zwischen den erreichten Punkten in der Prüfung und mit dem neuen Bewertungsschema hin. Denn bei Aufgabe 2 erreichten die Studierenden bei der reinen Bewertung der Programmierkonzepte und deren Kombination meist eine (teilweise deutlich) höhere Punktzahl. In Aufgabe 3 ist dieses Phänomen deutlich schwächer ausgeprägt, was wiederum diese Vermutung unterstützt. Ein weiterer Grund für das schlechtere Abschneiden der Bedingungen kann auch an der Anzahl der Übungen, welche Bedingungen adressieren, liegen, da diese geringer sind als die Anzahl der Übungen für Iterationen. Oftmals inkludieren Übungen zur Iteration die Anwendung von Bedingungen, die im Gegensatz zu diesen korrelieren. Aus diesem Grund wurden die erreichten Punkte in den Aufgaben und für die einzelnen Konzepte mit den *Aktivitäten* und der *Anzahl der Beispiele* korreliert (vgl. Tabelle 25). Die Ergebnisse deuten auf einen positiven Zusammenhang zwischen diesen beiden Merkmalen und den erreichten Punkten für die einzelnen Konzepte hin. Hierbei fällt wiederum auf, dass die Anzahl der Beispiele stärker korreliert als die Anzahl der Aktivitäten. In Verbindung mit der Untersuchung der Konzepte je Notenstufe (siehe Kapitel 7.2.2) wird deutlich, dass bessere Studierende mehr unterschiedliche Konzepte geübt bzw. bearbeitet haben.

Für die abschließende Überprüfung der Nullhypothese $H_0$, werden die erreichten Punkte je Aufgabe und Konzept herangezogen, da diese die Programmierfähigkeit, wie sie für diese Arbeit definiert wurde, vergleichbar macht. Die einzelnen Punkte je Konzept korrelieren mit der Anzahl der Beispiele und den Aktivitäten. Ebenso existiert ein signifikanter Zusammenhang zwischen der Anzahl der Beispiele und den erreichten Punktzahlen der einzelnen Aufgaben. Nur bei Aufgabe 3 wurde die Korrelation zwischen der Anzahl der Aktivitäten bei dem gewählten Signifikanzniveau zurückgewiesen. Das in Kapitel 5 entwickelte Konzept verfolgt nun aber die Idee mehr Aufgaben und Beispiele in den Unterricht zu integrieren und dabei die Diskussionen und die Auseinandersetzung zu erhöhen. Ausgehend von den positiven Korrelationen konnte zwischen der Anzahl der Beispiele und den erreichten Punktzahlen bei den Aufgaben und Konzepten ein Zusammenhang belegt werden. Somit kann $H_0$ zurückgewiesen und die Alternativhypothese $H_1$ angenommen werden.

### 7.5.3 Auseinandersetzung mit Programmieren

Eine weitere zu Beginn aufgestellte Hypothese ist der Einfluss des *Aktiven Lernens* im Unterricht auf die studentische Auseinandersetzung mit dem Programmieren. Am Anfang der Arbeit wurde vermutet, dass die Aktivitäten des *Aktiven Lernens* und der einfache Zugang zum Programmieren Studierenden zu mehr Auseinandersetzung mit dem Thema Programmieren motivieren. Für einen möglichen Vergleich kann an dieser Stelle nur die vorherige Lehrveranstaltung herangezogen werden, welche keine Programmieraktivitäten seitens der Studierenden im Unterricht beinhaltete. Tabelle 15 zeigt die Anzahl der Studierenden, die im Unterricht bereits an den Aufgaben mitgearbeitet haben (Spalte Unterricht). Zusätzlich wurde die Anzahl der Studierenden insgesamt ermittelt, welche das Beispiel auch nach der Vorlesung nochmals bearbeitet haben. Aus dieser lässt sich ableiten, dass einige Studierenden die Aufgaben im Unterricht nachbereiten.

Die berechneten Daten geben jedoch alleine keinen Aufschluss über die Nutzung und die Wahrnehmung der angebotenen Plattform, welche Studierenden eine teilweise selbstgesteuerte Erarbeitung der Inhalte ermöglichen soll. Deswegen werden nachfolgend einige Fragen aus der am Ende durchgeführten Befragung diskutiert. Viele Befragte haben die Möglichkeit genutzt die Aufgaben auch später online zu bearbeiten (siehe Tabelle 36). Weiterhin empfinden viele Studierenden die ausgegebenen Materialien als hilfreich und stuften die Erstellung und Ausführung der Programme wurde als leicht ein. Die Korrelationen zwischen der leichten Erstellung und Ausführung von Programmen in der web-basierten Entwicklungsumgebung, deuten auf einen Zusammenhang beim Durchdenken und Mitdenken der Unterrichtsinhalte hin (siehe Tabelle 45). Besonders die Korrelationen zwischen dem Mitdenken und den eingesetzten Beispielen und der verwendeten Materialien (die interaktiven Dokumente und Präsentationen) weisen auf einen positiven Zusammenhang hin. Durch das neue Konzept und die entwickelte Plattform konnten die Studierenden stärker in den Unterricht einbezogen werden und sich somit öfters mit dem Thema Programmieren auseinandersetzen. Ebenfalls tragen die praxisbezogenen bzw. *greifbaren* Aufgaben dazu bei, welche von den Studierenden deutlich häufiger bearbeitet wurden. In Tabelle 19 sind die Aufgaben mit den meisten Aktivitäten aufgelistet. Diese wird von Aufgaben, welche *Turtle-Graphics* (siehe Kapitel 6.3.4) zur grafischen Darstellung verwenden, angeführt. Zudem wurde ebenfalls die Aufgabe, welche die Hardware-Simulation (vgl. Kapitel 6.3.4) verwendet, überdurchschnittlich oft ausgeführt bzw. mit dieser interagiert. Zusammengefasst kann eine höhere Auseinandersetzung aus den Daten der Plattform und Ergebnissen der Befragung abgeleitet werden. Dazu tragen besonders die einfach zu verwendende Plattform und die grafischen bzw. *greifbaren* Aufgaben bei. Die Studierenden beschäftigen sich, wie in der Hypothese vermutet, mehr mit Programmieren, da sie die Aufgaben im Unterricht aktiv bearbeiten und an den an-

schließenden Diskussionen teilnehmen.

### 7.5.4 Einfache Programmierumgebung

Die vierte Hypothese (vgl. Hypothese 4), vermutet einen positiven Einfluss auf die Auseinandersetzung mit dem Programmieren durch eine leicht zugängliche und nutzbare Programmierumgebung. In Kapitel 6 wurde die Konzeption und Entwicklung einer solchen Umgebung beschrieben, welche sich leicht im Unterricht und von überall aus nutzen lässt. Für die Überprüfung der Hypothese werden die erhobenen Daten und die Ergebnisse aus der Befragung herangezogen.

Die gesammelten Daten haben ergeben, dass ein Drittel bis die Hälfte der Studierenden die Aufgaben im Unterricht mittels der web-basierten Umgebung bearbeitet haben (vgl. Tabelle 50). Des Weiteren geben die Antworten auf die Frage zur Nutzung der Möglichkeit zur späteren Bearbeitung der Aufgaben Aufschluss über das Nutzungsverhalten der Studierenden (siehe Tabelle 36). Mehr als zwei Drittel der Befragten gaben dabei an, dass sie diese Option oft genutzt haben. Darüber hinaus haben einige Studierenden die Option wahrgenommen die Übungsaufgaben von zuhause aus zu bearbeiten (vgl. Tabelle 37).

Der Umgang mit der Plattform (vgl. Tabelle 41) und die Erstellung und Ausführung von Programmen (vgl. Tabelle 42) zeigt, dass die Studierenden gut mit der Umgebung zurechtgekommen sind. Die Korrelationen zwischen der leichten Erstellung und Ausführung von Programmen und des Verständnisses der Inhalte, deuten auf einen positiven Zusammenhang hin. Auch die Korrelation zwischen den ausgegebenen Materialien und der Programmierumgebung deutet auf die leichte Umgebung hin und das die interaktiven Inhalte zum Verständnis und erarbeiten des Stoffes beigetragen haben.

Einige Studierende nutzen die web-basierte Umgebung auch in späteren Übungen, obwohl auch eine vollständige Entwicklungsumgebung (*PyCharm*) eingeführt und zur Verwendung angeboten wurde. Trotzdem zeigt Tabelle 51, dass die Studierenden die Umgebung weiterhin genutzt haben. Daraus kann geschlossen werden, dass die Studierenden die web-basierte Umgebung als einfacher erachtet haben.

Auf Grund dieser Beobachtungen und Ergebnissen kann ein Einfluss der Programmierumgebung auf die Auseinandersetzung und Beschäftigung mit dem Programmieren abgeleitet werden. Die Studierenden verwenden die Umgebung, um die Aufgaben einerseits an den Übungsterminen, aber auch von zuhause aus zu bearbeiten.

### 7.5.5 Indikatoren

Die letzte und fünfte Hypothese (vgl. Hypothese 5), besagt, dass die Analyse der Rückmeldungen bei der Bearbeitung von Aufgaben (im Unterricht) eine Steuerung der Lehrveranstaltung ermöglicht. In der Auswertung der Daten und der Evaluation wurden dabei mehrere Aspekte näher betrachtet. Bei der Umsetzung der Lehrhandlungen konnten studentische Lösungen diskutiert und eine offene Atmosphäre für Rückmeldungen geschaffen werden. Dabei hat sich gezeigt, dass eine genaue Analyse der Fehler nur eingeschränkt im Unterricht möglich ist, und allenfalls als grober Indikator dienen kann. Häufige Fehler und zu schwierige Aufgaben können somit bereits bei der Durchführung erkannt werden, jedoch ist eine nähere Betrachtung der Fehler aufgrund des Zeitmangels und dem damit verbundenen Aufwand nur schwer möglich. Wie in Kapitel 6.5 bereits angemerkt, können aus reinen Fehlermeldungen nur schwer die dahinter liegenden Probleme bzw. Missverständnisse seitens der Studierenden abgeleitet werden. Dies erfordert immer eine Betrachtung des Kontextes, in welchem der Fehler aufgetreten ist.

In der durchgeführten Befragung, gaben die Studierenden an, dass der Lehrende das Niveau der Lehrveranstaltung an den Wissensstand der Studierenden anpasste (vgl. Tabelle 43). Aus den Fehlern der Übungsaufgaben wurden im Rahmen der Lehrveranstaltung Wiederholungsfragen für die nächste Vorlesung abgeleitet. Jedoch ist dies nur die Sicht eines einzelnen Dozenten und nicht einer größeren Gruppe, weswegen die Validität und Reliabilität der Ergebnisse in Frage gestellt werden können. Für die abschließende Überprüfung dieser Hypothese müssen noch weitere Evaluationen mit mehreren Lehrenden durchgeführt werden, die jedoch im Rahmen dieser Arbeit nicht erfolgen konnten.

### 7.5.6 Fazit

Für eine abschließende Bewertung werden die im forschungsmethodischen Vorgehen beschriebenen Fragestellungen, welche nach Tulodziecki und Herzig zur Beurteilung des Konzepts berücksichtigt werden sollen, kurz diskutiert (vgl. Kapitel 2.3). Dazu soll zuerst ein Blick auf die Einschätzung der Lernvoraussetzungen gelegt werden. Das Konzept und die Plattform wurden für Programmieranfänger mit keinem oder wenig Vorwissen konzipiert und umgesetzt. Die zu Beginn vorgenommene Einschätzung des Niveaus des Vorwissens des Studierenden hat die Mehrheit der Befragten als angemessen bewertet (vgl. Tabelle 44). Lediglich die Gruppe der Studierenden der Erneuerbaren Energien konnten im Vergleich zu den anderen weniger stark motiviert werden. Besonders der Spaß, wie in Tabelle 46 gezeigt, hat einen Einfluss auf das Lernen. Somit müssen teilweise die gewählten Beispiele und die Praxisbezüge noch stärker ausgear-

beitet werden, um diese Gruppe besser zu motivieren.

Die realisierten Lehrhandlungen (vgl. Kapitel 7.1.4) konnten im Rahmen des entwickelten Konzepts wie geplant durchgeführt werden. Ausgehend von der Bewertung der Hypothesen und der Beteiligung und Evaluation der Veranstaltung kann diese Umsetzung als sinnvoll bezeichnet werden. Einzig die Einschätzung der Studierenden auf Basis von Indikatoren erwies sich im Laufe der Arbeit nur als grobe Tendenz, da reine Fehlermeldungen nur wenig über die eigentlichen Probleme und Fehlvorstellungen aussagen. Zwei Aufgaben wurden allerdings nur von wenigen Studierenden bearbeitet, sodass diese in Hinblick auf die Umsetzung und die Aufgabestellung selbst überprüft werden müssen. Die Durchführung von Aufgaben am Ende der Vorlesung kann dazu führen, dass die Diskussion entweder nicht oder erst nur in sehr geringem Maße durchgeführt werden kann. Hierzu muss eine verbesserte zeitliche Planung der Aufgaben erfolgen, sodass die reflexiven Phasen (vgl. Kapitel 5) in einem ausreichenden Umfang durchlaufen werden.

Einen weiteren Aspekt bei der Beurteilung des entwickelten Konzepts stellen die beobachteten Lernaktivitäten seitens der Studierenden dar. In den Vorlesungen konnte eine gute Beteiligung festgestellt und mittels Daten (vgl. Kapitel 7.2) und bei der abschließenden Befragung (siehe Kapitel 7.4.3) belegt werden. Trotzdem hat die Befragung gezeigt, dass sich nicht alle Studierenden im Unterricht aktiv beteiligen wollen oder dazu motiviert wurden. Erfreulicherweise konnte bei der Durchführung die Bildung von Gruppen und Paaren bei der Bearbeitung beobachtet werden, die gemeinsam versucht haben die gestellten Aufgaben zu lösen. Wie erwartet haben sich die Studierenden stärker und häufiger mit den *greifbaren* Aufgaben mit grafischer Ausgabe beschäftigt. Dies stützt die Vermutung, dass diese Art der Beispiele die Studierenden stärker motivieren und sie sich schlussendlich mehr mit Programmieren auseinandersetzen. Schließlich konnte beobachtet werden, dass einige Studierenden, sich trotz des Angebots der Verwendung einer vollständigen Entwicklungsumgebung, weiterhin die web-basierte Umgebung verwendet haben. Einerseits ist dies erfreulich, da dies als Akzeptanz der Umgebung gewertet werden kann, andererseits sollten die Studierenden jedoch auch den Umgang mit einer richtigen Entwicklungsumgebung lernen.

Allerdings konnten auch Nebenwirkungen bei der Durchführung festgestellt werden. Im Laufe des Semesters konnte eine nachlassende Beteiligung der Studierenden beobachtet werden. Besonders Studierende, welche eine intensivere Betreuung, wie sie in den Übungen angeboten wird, benötigen, nehmen diese nicht wahr. Die Lehrveranstaltung verzichtet auf jegliche Anwesenheitspflichten und Zulassungsvoraussetzungen, weswegen einige Studierende diese nicht ernst nehmen bzw. diese besuchen. Zwar können diese aufgrund der entwickelten Plattform die Inhalte jederzeit nachholen und nachbereiten, was aber nur von einem gewissen Anteil der Studierenden vollzogen wird. Aus den ermittelten Korrelationen zwischen den Merkmalen und der

Note ist ersichtlich, dass die Anzahl der Tage, an denen mit der Plattform interagiert wurde, einen positiven Einfluss auf die erreichte Note hat. Je öfter und damit regelmäßiger in der Umgebung gearbeitet wurde, desto besser fallen die Prüfungsleistungen aus. Dabei wird an dieser Stelle auf eine Diskussion der Vor- und Nachteile von Anwesenheitspflicht und Zulassungsvoraussetzungen verzichtet, da diese außerhalb des Rahmens dieser Arbeit liegt. Trotzdem kann als Konsequenz eine bessere Kommunikation mit diesen Studierenden gefordert werden, um diese für die Teilnahme zu motivieren bzw. zu appellieren.

Zuletzt werden die erreichten Lernergebnisse betrachtet. Ausgehend von den ermittelten Daten kann ein Zusammenhang zwischen den Aktivitäten bzw. der Anzahl der bearbeiteten Beispiele und der erreichten Note aufgezeigt werden. Die Untersuchung der Programmierfähigkeit hat gezeigt, dass dieser Zusammenhang stärker bei den Konzepten der Iteration auftritt. Ein Vergleich der unterschiedlichen Bewertungsschemata und den erreichten Punktzahlen deutet darauf hin, dass die Studierenden zwar das richtige Konzept auswählen, jedoch bei der Umsetzung und Kombination noch Probleme haben. Die untersuchten Aufgaben wurden dabei noch auf Papier gelöst, was wiederum nur teilweise der Vorgehensweise beim Programmieren entspricht, da ein inkrementelles Vorgehen nur schwer umzusetzen ist. Zusammengefasst ist ein positiver Zusammenhang zwischen der Einübung der Konzepte und dem Abschneiden in der Prüfung zu erkennen. Dieser kann aufgrund des Umfangs der Studie vorerst als Tendenz aufgefasst werden, da weitere Untersuchungen notwendig sind, um ein sicheres und valides Bild zu erlangen.

# Kapitel 8

# Zusammenfassung

Zum Ende der Arbeit sollen zunächst der wissenschaftliche Beitrag und die daraus abgeleiteten Schlussfolgerungen zusammengefasst werden. Des Weiteren werden die gewählten wissenschaftlichen Methoden kurz reflektiert, um eine Aussage über die Reliabilität und Validität der Untersuchung zu treffen. Anschließend folgt noch ein Ausblick über mögliche weitere Schritte und ein Fazit.

## 8.1 Wissenschaftlicher Beitrag und Schlussfolgerungen

Ziel der Arbeit ist die Konzeption eines Lehrkonzepts auf Basis des *Aktiven Lernens* und die Umsetzung der dafür notwendigen bzw. unterstützenden Werkzeuge, um die Programmierfähigkeit der Studierenden zu verbessern bzw. die Lücke zwischen der theoretischen Vermittlung der Inhalte und deren praktischer Anwendung zu schließen. Ausgangspunkt dafür war die Forderung nach einem eher seminaristisch geprägten Unterricht, der die Studierenden stärker in die Steuerung der Vorlesung mit einbezieht.

Zu Beginn wurde der Begriff Programmieren und das programmatische Problemlösen diskutiert und daraus eine Definition bzw. ein Programmiervorgang für diese Arbeit erarbeitet. Auf Basis dieser Definition wurde der Begriff Programmierfähigkeit abgeleitet, um diesen messen zu können. Dazu wurden vorhandene Ansätze zur Bewertung der Programmierfähigkeit diskutiert und weiterentwickelt und schlussendlich ein anpassbares Messinstrument erarbeitet. Dieses wurde mit dem Ziel der Überprüfung von Lösungen von Implementierungsaufgaben entwickelt, sodass einzelne Lösungsansätze nicht abgewertet werden. Primär werden dabei der Einsatz und die

Kombination von logischen Bausteinen bzw. Programmierkonzepten zur Lösung von (Teil-)Problemen überprüft und anhand eines Schemas bewertet.

Anschließend wurden ausgehend von lehr- und lerntheoretischen Überlegungen, besonders aus gemäßigt konstruktivistischen Theorien, mehrere Ansätze des *Aktiven Lernens* diskutiert. Nachfolgend wurde der Begriff *Aktives Lernen* und dessen Charakteristika näher betrachtet, um schlussendlich Aktivitäten des *Aktives Lernens* im Bereich der Programmierausbildung zu identifizieren und zu bewerten. Im Rahmen der Konzeptentwicklung wurde ein Ansatz zur Integration von Aufgaben zur besseren Vermittlung von Programmierkonzepten direkt im Unterricht erarbeitet. Dieser basiert auf den Erkenntnissen der lehr- und lerntheoretischen Überlegungen in Kapitel 3.4 und Kapitel 4. Als Grundlage wurde der *Cognitive Apprenticeship* Ansatz gewählt und an das Gebiet des Programmierens angepasst und erweitert. Allerdings existiert bei der Adaption eine Einschränkung, da in Vorlesungen aufgrund der Rahmenbedingungen keine direkte Eins-zu-eins-Betreuung im Sinne des Meister-Lehrlings-Verhältnisses ermöglicht werden konnte. Dadurch können zielgerichtete Übungen mit Reflektionsphasen in den Unterricht integriert werden, um frühzeitig die Anwendung theoretischer Konzepte und deren Einsatz zum Problemlösen aufzuzeigen. Und darüber hinaus kann man als Lehrender Rückmeldungen über den aktuellen Lernstand und mögliche Probleme erhalten.

Für eine reibungslose Umsetzung des Konzepts und der Lösung der identifizierten Probleme, wurde eine interaktive Lernplattform entwickelt, welche sowohl den Dozierenden wie auch die Studierenden im Unterricht bei der Durchführung unterstützt. Diese Plattform (vgl. Kapitel 6) ermöglicht erstens interaktive Übungen und bietet einen Kommunikationskanal zum Dozierenden, um studentische Lösungen zu besprechen. Zweitens können Studierende jederzeit und von überall aus die interaktiven Inhalte abrufen und selbstgesteuert bearbeiten. Die Plattform selbst wurde nach einer Analyse vorhandener Ansätze neu entwickelt, um Studierenden *greifbare* Beispiele und Aufgaben zu präsentieren. Besonders die Darstellung der Quelltexte in Präsentationen und mittels der webbasierten Entwicklungsumgebung konnte durch den Einsatz neuer Webtechnologien an die jeweiligen Gegebenheiten von Vorlesungsräumen optimiert werden. Zusätzlich wurden dabei grafische Elemente und die Simulation von Hardware für einen besseren Praxisbezug umgesetzt.

Die Evaluation des Konzepts auf Basis der erfassten Daten hat gezeigt, dass ein Zusammenhang zwischen der Anzahl der Aktivitäten bzw. Beispiele, welche auf der Plattform durchgeführt bzw. bearbeitet wurden, und der erreichten Prüfungsleistung bei Implementierungsaufgaben existiert. Studierende, welche sich häufiger und mit mehr Beispielen beschäftigen, erreichen dabei tendenzielle eine bessere Prüfungsleistung. Die Auswertung zeigt darüber hinaus, dass besonders Studierende, welche gleichmäßig über das Jahr Aufgaben bearbeitet haben, ebenfalls bessere Ergebnisse erzielen.

Das erarbeitete Konzept wurde von den Studierenden angenommen und eine gute Beteiligung im Unterricht festgestellt, da die Studierenden sich auch zur Mitarbeit und zum Durchdenken der Inhalte angeregt fühlten. Eben diese Anregung und Mitarbeit stellen eines der Elemente des entwickelten Konzepts dar.

Um eine Aussage über den Einfluss auf die Programmierfähigkeit zu treffen, wurde das entwickelte Messinstrument zur Bewertung eingesetzt. Dazu wurden mehrere Implementierungsaufgaben der Prüfung nach dem neuen Schema bewertet und anschließend mit den aus der Plattform erhobenen Daten verglichen. Dabei wurde festgestellt, dass ebenfalls ein Zusammenhang zwischen den Aktivitäten bzw. der Anzahl der Beispiele und der erreichten Punktzahl bei den bewerteten Aufgaben existiert. Studierende, die mehr Aufgaben bzw. Aktivitäten durchgeführt haben, erreichten auch hier eine tendenzielle bessere Bewertung. Auf einzelne Konzepte heruntergebrochen, konnte ein Zusammenhang zwischen den Übungen, die das Konzept der Iteration adressieren, und der erreichten Punktzahl beim Einsatz dieser Konzepte gezeigt werden. Lediglich beim Konzept der Bedingungen konnte kein signifikanter Zusammenhang belegt werden, wobei dies an mehreren Gründen liegen kann, die in Kapitel 7.5 dargelegt sind. Durch die Evaluation wird auch die zu Beginn in Kapitel 3.1 aufgestellte These, dass das Erlernen von Programmierkonzepten aufeinander aufbaut, gestützt. Denn besonders die Studierenden mit tendenziell schlechteren Leistungen haben deutlich weniger unterschiedliche Konzepte bearbeitet bzw. sind nicht zu den aufbauenden Konzepten vorgedrungen (siehe Kapitel 7.2.2). Ohne diese weiteren Konzepte und ein Fehlen des damit einhergehenden Einübens der Kombination dieser im Rahmen des Algorithmenentwurfs kann zu schlechteren Leistungen führen. An dieser Stelle muss die Vermittlung der Inhalte und die Motivierung sich kontinuierlich mit diesen zu beschäftigen noch weiter verbessert werden, um diese Kohorte bestmöglich zu unterstützen. Zusammengefasst kann die zu Beginn aufgestellte Vermutung, dass mehr (zielgerichtete) Übungen und eine praxisbezogene Vermittlung von Inhalten, zu besseren Leistungen führen kann.

Die entwickelte Plattform und die dadurch mögliche Verwendung von interaktiven Beispielen wurde von den Studierenden als besonders hilfreich für das Verständnis gewertet. Ebenso scheint der Einsatz einer web-basierten Entwicklungsumgebung, welche den Fokus auf das Programmieren und nicht die Bedienung einer Software legt, als sinnvoll. Die Auswertung der Befragung zeigt, dass der Großteil der Studierenden mit der Plattform und der Umgebung gut zurechtkamen. Zudem zeigen die Nutzungsdaten, dass die Studierenden die Umgebung trotz professioneller Alternativen nutzen, um die Aufgaben zu bearbeiten und sich vorzubereiten.

Die Untersuchung zeigt aber auch, dass es sehr wohl Studierende gibt, die mit hohen Aufwand trotzdem nur moderate Leistungen erreichen. Daraus kann abgeleitet werden, dass die Art der Beschäftigung mit den Inhalten und die Qualität der Aufga-

ben und Beispiele wichtig ist. Zwar wurden die Beispiele von den Befragten als sehr hilfreich eingeschätzt, jedoch können diese besonders für die Studierenden der Erneuerbaren Energien noch verbessert werden (vgl. Kapitel 7.4.3). Schlussendlich müssen weitere Anreize zur Motivierung der eher schlechteren Studierenden erarbeitet werden, sodass diese die angebotene Betreuung wahrnehmen.

## 8.2 Methodenbewertung

Die gewählten Methoden und das forschungsmethodische Vorgehen sollen hier noch einmal rückblickend reflektiert werden. Der gewählte Ansatz zur Entwicklung eines Lehrkonzepts auf Basis von lehr- und lerntheoretischen Überlegungen erscheint für die Zielsetzung der Arbeit als sinnvoll, da die theoretische Fundierung und Analyse die eigentlichen Probleme erst aufzeigt. Besonders die Auseinandersetzung mit dem Begriff Programmieren und einem erarbeiteten Vorgehen, hat die Probleme und die verbesserungswürdige Disposition bei den Studierenden deutlicher und greifbarer gemacht. Wie bereits in der Problemstellung vermutet, müssen die Studierenden zur Lösung von Problemen verschiedene logische Bausteine - die Programmierkonzepte - auswählen und miteinander kombinieren. Dies wurde in den vorherigen Lehrveranstaltungen nur bedingt explizit angesprochen, sondern vielmehr implizit in die Übungen verlagert und sich erhofft, dass sich die Studierenden dies bei der Bearbeitung von Aufgaben alleine aneignen. Erst diese Analyse und Diskussion hat das eigentliche Problem zum Vorschein gebracht, welches anschließend durch die theoretischen Überlegungen und dem daraus entwickelten Konzept zumindest teilweise gelöst werden konnte. Die in der Vorgehensweise beschriebene Reihenfolge hatte auch eine deutlich fokussierte Umsetzung des Konzepts und der dazu benötigten Werkzeuge zur Folge. Zwar wurde aufgrund der Abhängigkeit vom Lehrbetrieb das Konzept in mehreren Schritten erarbeitet, doch hat sich rückblickend gezeigt, dass ohne eine theoretische Grundlage und damit verbundene Ziele, kein sinnvoller Einsatz von den entwickelten interaktiven Lernmaterialien möglich ist.

Bei der abschließenden Evaluation kann an mehreren Stellen das Vorgehen in Bezug auf die Übertragbarkeit auf die komplette Informatikausbildung kritisch betrachtet werden. Der Begriff der Programmierfähigkeit und deren Operationalisierung lässt sich unter Umständen nicht auf die Lösung aller Probleme anwenden, insbesondere nicht auf die Erstellung von großen Softwaresystemen. Jedoch war das Ziel die Bewertung von Programmieranfängern, welchen ein prozedurales Paradigma vermittelt wurde. Dementsprechend kann mittels des Schemas eine Überprüfung der intendierten Lernergebnisse stattfinden, wobei das Instrument dabei einen gewissen Freiheitsgrad hat, da es auf jeweilige Aufgaben angepasst werden muss. Zusätzlich kann dieser

Kritik entgegen gestellt werden, dass die Entwicklung des Messinstruments von existierenden und bewährten Ansätzen abgeleitet und angepasst wurde.

Bei der Datenerhebung wurden nur die Studierenden erfasst, welche auch die Plattform und Umgebung verwendet haben. Dadurch sind die Daten nicht vollständig und die Ergebnisse können nur als Tendenz gewertet werden. Die Bewertung der Implementierungsaufgaben wurde von nur einer Person durchgeführt, was einen möglichen Einfluss auf die Reliabilität haben kann. Eine Bewertung durch mehrere unterschiedliche Personen hätte durch eine gute Beurteilerübereinstimmung (vgl. [BD16, S.255f]) die Objektivität der Messung verbessern können. Einen weiteren Aspekt stellen die untersuchten Programmieraufgaben selbst dar, da diese ggf. vom Schwierigkeitsgrad nicht den Anforderungen der Lehrveranstaltung entsprechen. Besonders bei der zweiten Aufgabe (vgl. Kapitel 7.3.1) wurde ein schlechteres Abschneiden im Vergleich zur dritten Aufgabe sichtbar. Dies kann auch zur nicht signifikanten Korrelation zwischen den Bedingungen und den Merkmalen geführt haben. Zusätzlich wurde zwar in der Befragung das Vorwissen erfasst, jedoch konnte dies aufgrund der Anonymität nicht den einzelnen Studierenden zugeordnet werden, um dessen Einfluss bei der Auswertung mit einfließen zu lassen.

Abgesehen davon ermöglichen die erfassten Daten und die pseudonymisierten Prüfungsaufgaben eine Wiederholung der Auswertung und die beschriebenen Aufgaben eine erneute Durchführung des Konzepts. Da bisher nur eine Durchführung in einer Lehrveranstaltung ausgewertet wurde, können die Ergebnisse ebenfalls nur als Tendenz gewertet werden. Jedoch zeigt der Vergleich mit Hassinen et al. ein ähnliches Bild, wenn auch das Untersuchungsziel ein anderes war ([HM06]). Die Diskussion der vorhandenen Ansätze zur Überprüfung der Programmierkenntnisse (vgl. Kapitel 3.3.2) hat auch gezeigt, dass eine Studie über mehrere Institutionen und Lehrveranstaltungen ebenfalls sehr anfällig in Bezug auf die Aussagekraft und Kontrolle der nicht kontrollierbaren Faktoren ist. Bei der Überprüfung des Konzepts wurde auch bewusst auf die Verwendung von Kontrollgruppen verzichtet, da dies besonders in der Pädagogik ethisch nicht unbedingt vertretbar ist. Der Einsatz neuer Konzepte sollte schließlich auf Basis eines wohlüberlegten Ansatzes erfolgen und nicht aus reiner Experimentierfreude. Dementsprechend sollte einer Studiengruppe nicht der Zugang zu einer potenziell besseren Lehre verwehrt, sondern allen zugänglich gemacht werden. Aus diesem Grund können das gewählte Vorgehen und die Evaluationsmethoden als sinnvoll bezeichnet werden, wenn auch einige Aspekte hätten verbessert werden können.

## 8.3 Fazit und Ausblick

Lehre lebt unter anderem durch Methodenvielfalt, wobei die einzelnen Methoden situativ und kontextbezogenen auf die jeweiligen Bereiche angepasst werden müssen. Es gibt keine didaktischen Allroundwerkzeuge, sondern der Einsatz muss zielgerichtet sein und ein bestimmtes Problem oder Lehr-/Lernziel adressieren. Das in dieser Arbeit erstellte Konzept beschränkt sich auf die Vermittlung von Programmierkonzepten bzw. das Vorgehen beim Problemlösen durch die Kombination von logischen Bausteinen. Der anfangs diskutierte Praxisbezug, konnte durch die Beispiele teilweise bereits direkt in der Vorlesung umgesetzt werden, wenn auch der Umgang mit echter Hardware in den Übungen wünschenswert wäre. Dabei sollte der Einsatz zu den jeweiligen Inhalten passen bzw. anhand eines didaktischen Konzepts integriert werden. Die dabei entstandene Plattform kann beliebig erweitert werden, um weitere interaktive Materialien für andere Inhalte zur Verfügung zu stellen. Andere Inhalte und Konzepte wie beispielsweise Listen oder Felder können auch durch anderen Aufgaben und Darstellungen direkt im Unterricht vermittelt werden, ohne den Einsatz von Programmieraufgaben. Deswegen soll an dieser Stelle nochmals explizit darauf hingewiesen werden, dass dieses Konzept eine Erweiterung von bereits vorhandenen Lehr- und Lernarrangements ist, wenn auch der Einsatz nicht auf ein einzelnes Thema beschränkt ist.

Ansatzpunkte für die Weiterentwicklung existieren sowohl im theoretischen Bereich wie auch in der Plattform selbst, welche besonders durch mehr Elemente für die Lernstandskontrolle erweitert werden kann. Die Auswertung gibt Hinweise darauf, dass nach wie vor die Kombination von logischen Bausteinen zur Lösung stärker betont werden kann. Schlussendlich ist es wichtig, dass sowohl die Lehrenden wie auch die Studierenden motiviert sind, sich mit den zu vermittelnden Themen zu beschäftigen und dies sehr wohl bereits im Unterricht erfolgen kann. Besonders an Hochschulen sollten auch Grundlagenveranstaltungen seminaristisch geprägt sein und Studierenden eine Möglichkeit zur Steuerung und Mitgestaltung des Unterrichts gewährt werden.

# Literatur


[AB15] Amjad Altadmri und Neil C.C. Brown. „37 Million Compilations“. In: *The 46th ACM Technical Symposium*. Hrsg. von Adrienne Decker u. a. 2015, S. 522–527.

[AC10] Ana Paula L. Ambrósio und Fábio M. Costa. *Evaluating the impact of PBL and tablet PCs in an algorithms and computer programming course*. 2010.

[Akg13] Faruk Akgul. *ZeroMQ*. Birmingham: Packt Publishing, 2013.

[AR95] Owen Astrachan und David Reed. „AAA and CS 1: The applied apprenticeship approach to CS 1“. In: *ACM SIGCSE Bulletin* 27.1 (1995), S. 1–5.

[Atk+00] R. K. Atkinson u. a. „Learning from Examples: Instructional Principles from the Worked Examples Research“. In: *Review of Educational Research* 70.2 (2000), S. 181–214.

[Bar12] Tino Bargel. „Bedeutung von Praxisbezügen im Studium“. In: *Studium nach Bologna: Praxisbezüge stärken?!* Hrsg. von Wilfried Schubarth. SpringerLink : Bücher. Wiesbaden: Springer Fachmedien Wiesbaden und Imprint: Springer VS, 2012, S. 37–46.

[BC08] Leland L. Beck und Alexander W. Chizhik. „An experimental study of cooperative learning in cs1“. In: *ACM SIGCSE Bulletin* 40.1 (2008), S. 205–209.

[BC13] Leland Beck und Alexander Chizhik. „Cooperative learning instructional methods for CS1: Design, implementation, and evaluation“. In: *ACM Transactions on Computing Education (TOCE)* 13.3 (2013), S. 10.

[BD16] Jürgen Bortz und Nicola Döring. *Forschungsmethoden und Evaluation: Für Human- und Sozialwissenschaftler*. 5., überarb. Aufl. Springer-Lehrbuch. Heidelberg: Springer, 2016.

[BE76] Benjamin Samuel Bloom und Max D. Engelhart, Hrsg. *Taxonomie von Lernzielen im kognitiven Bereich*. 5. Aufl., (17. - 21. Tsd.) Bd. 35. Beltz-Studienbuch. Weinheim: Beltz, 1976.

[BE91] Charles C. Bonwell und James A. Eison. *Active learning: Creating excitement in the classroom*. Bd. 1, 1991. ASHE-ERIC higher education report.

Washington, D.C.: School of Education and Human Development, George Washington University, 1991.

[Ber+16] M. Berges u. a. „Towards Deriving Programming Competencies from Student Errors“. In: *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. 2016, S. 19–23.

[BHS16] Joachim Breitner, Martin Hecker und Gregor Snelting. „Der Grader Praktomat“. In: *Automatisierte Bewertung in der Programmierausbildung*. Hrsg. von Oliver J. Bott u. a. 2016.

[BHT07] Sigrid Blömeke, Bardo Herzig und Gerhard Tulodziecki. „Zum Stellenwert empirischer Forschung für die Allgemeine Didaktik“. In: *Unterrichtswissenschaft* 35.4 (2007), S. 355–381.

[Bit16] Bitkom, Hrsg. *Industrie 4.0 – Status und Perspektiven*. Berlin, 2016.

[Bla06] Toni R. Black. „Helping novice programming students succeed“. In: *Journal of Computing Sciences in Colleges* 22.2 (2006), S. 109–114.

[Bör07] Jürgen Börstler. „Objektorientiertes Programmieren-Machen wir irgendwas falsch?“ In: *Didaktik der Informatik in Theorie und Praxis*. Hrsg. von Sigrid Schubert. 2007, S. 9–20.

[Bra14] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format: RFC 7159*. RFC Editor, 2014.

[Bru72] Jerome S. Bruner. *Der Prozeß der Erziehung*. 2. Aufl. Bd. 4. Sprache und Lernen. Berlin: Berlin-Verl. [u.a.], 1972.

[BS08] Tobina Brinker und Thomas Stelzer-Rothe. *Kompetenzen in der Hochschullehre: Rüstzeug für gutes Lehren und Lernen an Hochschulen*. 2., aktualisierte. Aufl. Das Kompendium. Rinteln: Merkur-Verl., 2008.

[BS10] Jürgen Bortz und Christof Schuster. *Statistik für Human- und Sozialwissenschaftler: Limitierte Sonderausgabe : mit 70 Abbildungen und 163 Tabellen*. 7., vollständig überarbeitete und erweiterte Auflage. Springer-Lehrbuch. Berlin: Springer, 2010.

[Bun] Bundesministerium für Wirtschaft und Energie. *Digitale Transformation in der Industrie*.

[Cal07] Jane E. Caldwell. „Clickers in the large classroom: current research and best-practice tips“. In: *CBE life sciences education* 6.1 (2007), S. 9–20.

[Cas07] Michael Edelgaard Caspersen. „Educating Novices in The Skills of Programming“. Diss. 2007.

[CB00] A. T. Chamillard und Kim A. Braun. „Evaluating Programming Ability in an Introductory Computer Science Course“. In: *SIGCSE Bull* 32.1 (2000), S. 212–216.

[CB07] Michael E. Caspersen und Jens Bennedsen. *Instructional design of a programming course: A learning theoretic approach*. 2007.

[CBN88] Allan Collins, John Seely Brown und Susan E. Newman. „Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics“. In: *Thinking: The Journal of Philosophy for Children* 8.1 (1988), S. 2–10.

[Cla03] Volker Claus. *Duden Informatik: Ein Fachlexikon für Studium und Praxis*. [Neuaufl.] Mannheim und Wien u.a.: Dudenverl., 2003.

[Coh82] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Rev. ed., 5. [print.] New York NY u.a.: Acad. Press, 1982.

[Dav93] Simon P. Davies. „Models and theories of programming strategy“. In: *International Journal of Man-Machine Studies* 39.2 (1993), S. 237–267.

[DBH16] Markus Dahm, Frano Barnjak und Moritz Heilemann. „5Code – An Integrated Programming Environment for Beginners“. In: *i-com* 15.1 (2016).

[DD12] Gabriella Dodero und Francesco Di Cerbo. *Extreme Apprenticeship Goes Blended: An Experience*. 2012.

[Dec07] Adrienne Decker. „How students measure up: An assessment instrument for introductory computer science“. Diss. The State University of New York at Buffalo, 2007.

[DLC14] Paul Denny, Andrew Luxton-Reilly und Dave Carpenter. *Enhancing syntax error messages appears ineffectual*. 2014.

[DLT12] Paul Denny, Andrew Luxton-Reilly und Ewan Tempero. *All syntax errors are not equal*. 2012.

[Dör79] Dietrich Dörner. „Kognitive Merkmale erfolgreicher und erfolgloser Problemlöser beim Umgang mit sehr komplexen Problemen“. In: *Komplexe menschliche Informationsverarbeitung*. Hrsg. von Hans Ueckert. Bern: Huber, 1979.

[Dör83] Dietrich Dörner. *Empirische Psychologie und Alltagsrelevanz*. Universität. Bamberg: Univ, 1983.

[Dow15] Allen Downey. *Think Python*. Second edition. Sebastopol, CA: O'Reilly Media, 2015.

[Dud13] Dudenredaktion. *Duden: Die deutsche Rechtschreibung*. 26., völlig neu bearb. und erw. Aufl. Mannheim: Bibliographisches Institut, 2013.

[Duf17] Jeffrey L. Duffany. „Application of Active Learning Techniques to the Teaching of Introductory Programming“. In: *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje* 12.1 (2017), S. 62–69.

[Dun74] Karl Duncker. *Zur Psychologie des produktiven Denkens*. Berlin: Springer, 1974.

[Ea02] Eugene Judson und and Daiyo Sawada. „Learning from Past and Present: Electronic Response Systems in College Lecture Halls“. In: *Journal of Computers in Mathematics and Science Teaching* 21.2 (2002), S. 167–181.

[Ebe17] Michael Ebert. „Increase active learning in programming courses“. In: *2017 IEEE Global Engineering Education Conference, EDUCON 2017, Athens, Greece, April 25-28, 2017*. IEEE, 2017, S. 848–851.

[Ede94] Walter Edelmann. *Lernpsychologie: [eine Einführung]*. 4., überarbeitete Aufl. Weinheim: Beltz, 1994.

[Edu17] Educational Testing Service. *ETS Major Field Test for Computer Science*. 2017.

[EGM15] Barbara J. Ericson, Mark J. Guzdial und Briana B. Morrison. *Analysis of Interactive Features Designed to Enhance Learning in an Ebook*. 2015.

[EH15a] M. Ebert und W. Haupt. „Leveraging Parson's Problems and Code-Fragment-Questions in a Quiz for an Interactive Programming EBook“. In: *EDULEARN15 Proceedings*. 7th International Conference on Education and New Learning Technologies. IATED, 2015, S. 7691–7698.

[EH15b] Michael Ebert und Wolfram Haupt. „Browserbasierte Programmierung und Interaktion mit Arduino-Aufbauten im Bereich Elektrotechnik für Informatik e.V. (GI), München, 1.-4. September 2015“. In: *DeLFI 2015 - Die 13. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. (GI), München, 1.-4. September 2015*. Hrsg. von Hans Pongratz und Reinhard Keil. LNI. GI, 2015, S. 333–335.

[Eri+15] Barbara Ericson u. a. *Usability and Usage of Interactive Features in an Online Ebook for CS Teachers*. 2015.

[ESC12] Sarah Esper, Beth Simon und Quintin Cutts. *Exploratory homeworks: An active learning tool for textbook reading*. 2012.

[Est+15a] Bernardo Estácio u. a. *Evaluating Collaborative Practices in Acquiring Programming Skills: Findings of a Controlled Experiment*. 2015.

[Est+15b] Bernardo Estácio u. a. *Evaluating the Use of Pair Programming and Coding Dojo in Teaching Mockups Development: An Empirical Study*. 2015.

[ETA14] Stephen H. Edwards, Daniel S. Tilden und Anthony Allevato. „Pythy: Improving the Introductory Python Programming Experience“. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. New York, NY, USA: ACM, 2014, S. 641–646.

[ETB05] Anna Eckerdal, Michael Thuné und Anders Berglund. *What does it take to learn 'programming thinking'?* ACM, 2005.

[Fär+16] Tommy Färnqvist u. a. *Supporting Active Learning by Introducing an Interactive Teaching Tool in a Data Structures and Algorithms Course*. 2016.

[FBH15] P. Figas, A. Bartel und G. Hagel. „Task-based programming learning in higher education“. In: *IEEE Global Engineering Education Conference (EDUCON), 2015*. Piscataway, NJ: IEEE, 2015, S. 648–653.

[FBT15] Charles Fadel, Maya Bialik und Bernie Trilling. *Four-dimensional education*. 2015.

[Fel+00] Richard M. Felder u. a. „The future of engineering education, II: teaching methods that work". In: *Chemical Engineering Education* 34.1 (2000).

[Fig+16] Paula Figas u. a. „Learning programming languages through input-providing tasks". In: *2016 IEEE Global Engineering Education Conference, EDUCON 2016, Abu Dhabi, United Arab Emirates, April 10-13, 2016*. IEEE, 2016, S. 419–424.

[Fun03] Joachim Funke. *Problemlösendes Denken*. 1. Aufl. Einführungen und Allgemeine Psychologie. Stuttgart: Kohlhammer, 2003.

[GE14] James G. Greeno und Yrjö Engeström. „Learning in Activity". In: *The Cambridge handbook of the learning sciences*. Hrsg. von Robert Keith Sawyer. Cambridge handbooks in psychology. Cambridge: Cambridge University Press, 2014, S. 128–148.

[Gon06] Graciela Gonzalez. „A systematic approach to active and cooperative learning in CS1 and its effects on CS2". In: *ACM SIGCSE Bulletin* 38.1 (2006), S. 133–137.

[Gre14] Green T.R.G. „Programming languages as information structures". In: *Psychology of Programming*. Hrsg. von J. M. Hoc. Elsevier Science, 2014, S. 117–136.

[Gri16] Jean M. Griffin. *Learning by Taking Apart: Deconstructing Code by Reading, Tracing, and Debugging*. 2016.

[Guo13] Philip J. Guo. „Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. New York, NY, USA: ACM, 2013, S. 579–584.

[HA13] Matthias Hauswirth und Andrea Adamoli. „Teaching Java programming with the Informa clicker system". In: *Science of Computer Programming* 78.5 (2013), S. 499–520.

[Hak98] Richard R. Hake. „Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses". In: *American Journal of Physics* 66.1 (1998), S. 64–74.

[HE15] Lars Josef Hook und Anna Eckerdal. „On the Bimodality in an Introductory Programming Course: An Analysis of Student Performance Factors". In: *2015 International Conference on Learning and Teaching in Computing and Engineering*. IEEE, 2015, S. 79–86.

[HHS13] Ludwig Huber, Julia Hellmer und Friederike Schneider, Hrsg. *Forschendes Lernen im Studium: Aktuelle Konzepte und Erfahrungen*. 2. Auflage. Bd. 10. Motivierendes Lehren und Lernen in Hochschulen. Bielefeld: UVW Universitäts Verlag Webler, 2013.

[Hil+08] Tatjana S. Hilbert u. a. „Learning to prove in geometry: Learning from heuristic examples and how it can be supported“. In: *Learning and Instruction* 18.1 (2008), S. 54–65.

[HM06] Marko Hassinen und Hannu Mäyrä. „Learning Programming by Programming: A Case Study“. In: *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*. Baltic Sea '06. New York, NY, USA: ACM, 2006, S. 117–119.

[HMS99] Thorsten Hampel, Johannes Magenheim und Carsten Schulte. *Dekonstruktion von Informatiksystemen als Unterrichtsmethode - Zugang zu objektorientierten Sichtweisen im Informatikunterricht*. 1999.

[How88] Michael J. A. Howe. „Intelligence as an explanation“. In: *British Journal of Psychology* 79.3 (1988), S. 349–360.

[Hub07] Peter Hubwieser. *Didaktik der Informatik: Grundlagen, Konzepte, Beispiele*. 3., überarbeitete und erw. Aufl. EXamen.press. Berlin, Heidelberg: Springer-Verlag, 2007.

[IK10] Petri Ihantola und Ville Karavirta. „Open Source Widget for Parson's Puzzles“. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '10. New York, NY, USA: ACM, 2010, S. 302.

[IT13] Ville Isomöttönen und Ville Tirronen. „Teaching programming by emphasizing self-direction: How did students react to the active role required of them?“ In: *ACM Transactions on Computing Education (TOCE)* 13.2 (2013), S. 6.

[JC11] Wei Jin und Albert Corbett. *Effectiveness of cognitive apprenticeship learning (CAL) and cognitive tutors (CT) for problem solving using fundamental programming concepts*. 2011.

[JCC05] J. Jackson, M. Cobb und C. Carver. „Identifying Top Java Errors for Novice Programmers“. In: *Frontiers in Education 35th Annual Conference*. 2005, S. 24–27.

[Jea11] Jeanette Wing. *Research notebook: Computational thinking - What and why?* Hrsg. von The Link Magazine. 2011.

[Jen+10] Jam Jenkins u. a. „JavaWIDE: Innovation in an online IDE“. In: *Journal of Computing Sciences in Colleges* 25.4 (2010), S. 6.

[Jen+12] Jam Jenkins u. a. *Perspectives on active learning and collaboration: JavaWIDE in the classroom*. 2012.

[JJ13] Jörn Loviscach und Jürgen Handke, Christian Spannagel. „Elemente und Aspekte des Inverted Classroom Model“. In: *E-Learning zwischen Vision und Alltag*. Hrsg. von Claudia Bremer. Medien in der Wissenschaft. Münster und München [u.a.]: Waxmann, 2013, S. 395–396.

[Joh03] Johannes Wildt. „Ein hochschuldidaktischer Blick auf Lehren und Lernen in gestuften Studiengängen“. In: *Studienreform mit Bachelor und Master*. Hrsg. von Ulrich Welbers. Hochschulwesen - Wissenschaft und Praxis. Bielefeld: Webler, 2003, S. 25–42.

[Joh15] Johannes Henninger. *Sourcebox: Konzeption und Implementierung einer serverseitigen Sandbox zum sicheren Ausführen von unbekanntem Code einschließlich Remote-Console zur Integration in Webseiten*. 2015.

[Joh16] John MacFarlane. *CommonMark Spec: Version 0.27*. 2016.

[Jos13] Joshua Bressers. *Is chroot a security feature?* 2013.

[Kap12] Karl M. Kapp. *The gamification of learning and instruction: Game-based methods and strategies for training and education*. Essential resources for training and HR professionals. San Francisco, Calif.: Pfeiffer, 2012.

[KB04] Michael Kölling und David J. Barnes. „Enhancing apprentice-based learning of Java“. In: *ACM SIGCSE Bulletin* 36.1 (2004), S. 286–290.

[KK87] John Keller und Thomas Kopp. „Application of the ARCS model to motivational design“. In: *Instructional Theories in Action: Lessons Illustrating Selected Theories*. Hillsdale, NJ: Erlbaum, 1987, S. 289–320.

[KKB14] Maria Knobelsdorf, Christoph Kreitz und Sebastian Böhne. *Teaching theoretical computer science using a cognitive apprenticeship approach*. 2014.

[KL09] Robin H. Kay und Ann LeSage. „Examining the Benefits and Challenges of Using Audience Response Systems: A Review of the Literature“. In: *Comput. Educ.* 53.3 (2009), S. 819–827.

[Kla15] Klaus-Peter Becker. *Beispiel - Schaltjahre*. 2015.

[Kla74] Wolfgang Klafki. *Didaktische Analyse*. 11. Aufl. Bd. 1. [Auswahl / A]. Hannover: Schroedel, 1974.

[Kli76] Friedhart Klix. *Information und Verhalten: Kybernetische Aspekte der organismischen Informationsverarbeitung : Einführung in naturwissenschaftliche Grundlagen der Allgemeinen Psychologie*. [3. Aufl.] Bern und Stuttgart: H. Huber, 1976.

[Kon14] Klaus Konrad. *Lernen lernen - allein und mit anderen: Konzepte, Lösungen, Beispiele*. Wiesbaden: Springer VS, 2014.

[Kor+13] Ari Korhonen u. a. „Requirements and Design Strategies for Open Source Interactive Computer Science eBooks“. In: *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports*. ITiCSE -WGR ’13. New York, NY, USA: ACM, 2013, S. 53–72.

[Kus12] Clifton Kussmaul. *Process oriented guided inquiry learning (POGIL) for computer science*. 2012.

[Lan10] Hans Petter Langtangen. *Python scripting for computational science*. 3rd ed. Bd. 3. Texts in computational science and engineering. Berlin: Springer, 2010.

[Lip14] Derrell Lipman. „LearnCS!: A New, Browser-based C Programming Environment for CS1“. In: *J. Comput. Sci. Coll.* 29.6 (2014), S. 144–150.

[Lis+04] Raymond Lister u. a. „A multi-national study of reading and tracing skills in novice programmers“. In: *Working group reports from ITiCSE*. Hrsg. von Henry M. Walker. 2004, S. 119.

[Lop+08] Mike Lopez u. a. *Relationships between reading, tracing and writing skills in introductory programming*. 2008.

[Mal13] David J. Malan. *From cluster to cloud to appliance*. 2013.

[Mar15] Markus Dahm , Frano Barnjak and Moritz Heilemann. „5Code - Eine integrierte Entwicklungsumgebung für Programmieranfänger“. In: *GI-Edition Proceedings Band 247 - DeLFI 2015 - Die 13. E-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V.* Hrsg. von Reinhard Keil, Bonn Gesellschaft für Informatik und Hans Pongratz. Bonn: Köllen, 2015.

[Maz97] Eric Mazur. *Peer instruction: A user's manual*. 2. print. Upper Saddle River NJ: Prentice-Hall, 1997.

[McC+01] Michael McCracken u. a. „A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students“. In: *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '01. New York, NY, USA: ACM, 2001, S. 125–180.

[McC+13] Robert McCartney u. a. *Can first-year students program yet? A study revisited*. 2013.

[McC17] Kevin McCullen. „Teaching embedded systems using the Raspberry Pi and sense HAT“. In: *Journal of Computing Sciences in Colleges* 32.6 (2017), S. 200–202.

[McG04] Andrew D. McGettrick. *Grand challenges in computing: Education*. Swindon: British Computer Society, 2004.

[MD14] Briana B. Morrison und Betsy DiSalvo. *Khan academy gamifies computer science*. 2014.

[Mey03] Bertrand Meyer. „The Outside-In Method of Teaching Introductory Programming“. In: *Perspectives of System Informatics*. Hrsg. von Manfred Broy und Alexandre V. Zamulin. Bd. 2890. Lecture Notes in Computer Science. Berlin und Heidelberg: Springer, 2003, S. 66–78.

[MHW03] Charlie McDowell, Brian Hanks und Linda Werner. „Experimenting with pair programming in the classroom“. In: *ACM SIGCSE Bulletin* 35.3 (2003), S. 60–64.

[Mic14] Michael Berger. *Was Bachelor Elektrotechnik können sollten: Eine Checkliste für Studierende*. 2014.

[Mie08] Gerd Mietzel. *Wege in die Psychologie*. 14. Aufl. Stuttgart: Klett-Cotta, 2008.

[MM16] Michael Ebert und Markus Ring. „A Presentation Framework for Programming in Programing Lectures". In: *2016 IEEE Global Engineering Education Conference (EDUCON)*. 2016.

[MMG15] Briana B. Morrison, Lauren E. Margulieux und Mark Guzdial. *Subgoals, Context, and Worked Examples in Learning Computing Problem Solving*. 2015.

[MOA17] David J. Malan, Nikolai Onken und Dan Armendariz. *A Web-Based IDE for Teaching with Any Language (Abstract Only)*. 2017.

[MPC15] Daniel Marchena Parreira, Andrew Petersen und Michelle Craig. *PCRS-C: Helping Students Learn C*. 2015.

[MR12] Bradley N. Miller und David L. Ranum. *Beyond PDF and ePub: toward an interactive textbook*. 2012.

[MW14] Michael Ebert und Wolfram Haupt. „Combining MOOC Methods and Blended Learning for an Introductory Programming Course for Electrical Engineers". In: *EDULEARN14 Proceedings*. 2014.

[Nat11] National Research Council (U.S.) *Report of a Workshop on the Pedagogical Aspects of Computational Thinking*. Washington, DC: The National Academies Press, 2011.

[NS72] A. Newell und H. A. Simon. *Human problem solving*. Prentice-Hall, 1972.

[Nuu+08] Esko Nuutila u. a. *Learning Programming with the PBL Method – Experiences on PBL Cases and Tutoring*. 2008.

[OGr12] Michael J. O'Grady. „Practical Problem-Based Learning in Computing Education". In: *Trans. Comput. Educ.* 12.3 (2012), 10:1–10:16.

[Pär+13] Martin Pärtel u. a. „Test My Code". In: *Int. J. Technol. Enhanc. Learn.* 5.3/4 (2013), S. 271–283.

[Pat+16] Elizabeth Patitsas u. a. *Evidence That Computer Science Grades Are Not Bimodal*. 2016.

[PBS13] Leo Porter, Cynthia Bailey Lee und Beth Simon. „Halving Fail Rates Using Peer Instruction: A Study of Four Computer Science Courses". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. New York, NY, USA: ACM, 2013, S. 177–182.

[Pea+07] Arnold Pears u. a. „A Survey of Literature on the Teaching of Introductory Programming". In: *SIGCSE Bull* 39.4 (2007), S. 204–223.

[PEM14] Marco Piccioni, Christian Estler und Bertrand Meyer. „SPOC-supported Introduction to Programming". In: *Proceedings of the 2014 Conference on*

*Innovation & Technology in Computer Science Education*. ITiCSE '14. New York, NY, USA: ACM, 2014, S. 3–8.

[PG07] Fernando Pérez und Brian E. Granger. „IPython: a System for Interactive Scientific Computing". In: *Computing in Science and Engineering* 9.3 (2007), S. 21–29.

[PH06] Dale Parsons und Patricia Haden. „Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses". In: *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*. ACE '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc, 2006, S. 157–163.

[PHG17] Raymond S. Pettit, John Homer und Roger Gee. *Do Enhanced Compiler Error Messages Help Students? Results Inconclusive*. 2017.

[Por+11] Leo Porter u. a. *Peer instruction: Do students really learn from peer discussion in computing?* 2011.

[Por+16] Leo Porter u. a. *A Multi-institutional Study of Peer Instruction in Introductory Computing*. 2016.

[PR00] Christine Pauli und Kurt Reusser. „Zur Rolle der Lehrperson beim kooperativen Lernen." In: *Schweizerische Zeitschrift für Bildungswissenschaften* 22.3 (2000), S. 421–442.

[Pri04] Michael Prince. „Does Active Learning Work? A Review of the Research". In: *Journal of Engineering Education* 93.3 (2004), S. 223–231.

[PWH15] Dale Parsons, Krissi Wood und Patricia Haden. „What Are We Doing When We Assess Programming?" In: *ACE*. 2015.

[RA94] Heiner Rindermann und Manfred Amelang. *Das Heidelberger Inventar zur Lehrveranstaltungs-Evaluation: (HILVE) ; Handanweisung*. Heidelberg: Asanger, 1994.

[Rei+08] Kristina Maria Reiss u. a. „Reasoning and proof in geometry: Effects of a learning environment based on heuristic worked-out examples". In: *ZDM* 40.3 (2008), S. 455–467.

[Ren11] Alexander Renkl. „Aktives Lernen: Von sinnvollen und weniger sinnvollen theoretischen Perspektiven zu einem schillernden Konstrukt." In: *Unterrichtswissenschaft* 39.3 (2011), S. 197–212.

[Rey09] Günter Daniel Rey. *E-Learning: Theorien, Gestaltungsempfehlungen und Forschung*. 1. Aufl. Psychologie Lehrbuch. Bern: Hans Huber, 2009.

[RFW12] Felix Raab, Markus Fuchs und Christian Wolff. „CodingDojo: Interactive Slides with Real-time Feedback". In: *Mensch & Computer 2012 - Workshopband*. Hrsg. von Harald Reiterer. München: Oldenbourg, 2012.

[RHS09] Alexander Renkl, Tatjana Hilbert und Silke Schworm. „Example-Based Learning in Heuristic Domains: A Cognitive Load Theory Account". In: *Educational Psychology Review* 21.1 (2009), S. 67–78.

[Rob11] Steven Robbins. *Beyond clickers: Using ClassQue for multidimensional electronic classroom interaction.* 2011.

[Ros15] Cyrille Rossant. *Learning IPython for Interactive Computing and Data Visualization - Second Edition.* Packt Publishing, 2015.

[RPB17] Fernando J. Rodríguez, Kimberly Michelle Price und Kristy Elizabeth Boyer. *Exploring the Pair Programming Process: Characteristics of Effective Collaboration.* 2017.

[RRH12] Juan Carlos Rodriguez-del-Pino, Enrique Rubio Royo und Zenón Hernández Figueroa. „A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features". In: (2012).

[RRR03] Anthony Robins, Janet Rountree und Nathan Rountree. „Learning and Teaching Programming: A Review and Discussion". In: *Computer Science Education* 13.2 (2003), S. 137–172.

[Rub13] Marc J. Rubin. „The Effectiveness of Live-coding to Teach Introductory Programming". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education.* SIGCSE '13. New York, NY, USA: ACM, 2013, S. 651–656.

[RW11] Alex D. Radermacher und Gursimran S. Walia. *Investigating the effective implementation of pair programming: An empirical investigation.* 2011.

[San10] Mary Sanseverino. *Pedagogy that clicks: Clickers in the CSC classroom.* 2010.

[SBH17] Philipp Shah, Marc Berges und Peter Hubwieser. *Qualitative Content Analysis of Programming Errors.* 2017.

[Sch+10] Carsten Schulte u. a. „An Introduction to Program Comprehension for Computer Science Educators". In: *Proceedings of the 2010 ITiCSE Working Group Reports.* ITiCSE-WGR '10. New York, NY, USA: ACM, 2010, S. 65–86.

[Sch07a] Sigrid Schubert, Hrsg. *Didaktik der Informatik in Theorie und Praxis: 12. GI-Fachtagung "Informatik und Schule - INFOS 2007", 19. - 21. September 2007 an der Universität Siegen.* 2007.

[Sch07b] Carsten Schulte. „Lesen im Informatikunterricht". In: *Didaktik der Informatik in Theorie und Praxis. INFOS 2007: 12. GI-Fachtagung Informatik und Schule, 19.-21. September 2007 in Siegen.* Hrsg. von Sigrid E. Schubert. Bd. 112. LNI. GI, 2007, S. 307–318.

[Sch91] Sigrid Schubert. „Fachdidaktische Fragen der Schulinformatik und (un)mögliche Anworten". In: *Informatik und Schule 1991: Informatik: Wege zur Vielfalt beim Lehren und Lernen GI-Fachtagung Oldenburg, 7.–9. Oktober 1991 Proceedings.* Hrsg. von Peter Gorny. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, S. 27–33.

[Sch93] Andreas Schwill. „Fundamentale Ideen der Informatik“. In: *Zentralblatt für Didaktik der Mathematik* 25 (1993), S. 20–31.

[Sel15] Cynthia C. Selby. *Relationships: Computational thinking, pedagogy of programming, and Bloom's Taxonomy*. 2015.

[SG09] Michael Striewe und Michael Goedicke. *Effekte automatischer Bewertungen für Programmieraufgaben in Übungs- und Prüfungssituationen*. 2009.

[SH15] Teemu Sirkiä und Lassi Haaranen. *Acos server: Towards smart learning content interoperability*. 2015.

[Sie05] Horst Siebert. *Pädagogischer Konstruktivismus*. 3. Aufl. Reihe Pädagogik und Konstruktivismus. [Weinheim]: Beltz Verlagsgruppe, 2005.

[Sim+10] Beth Simon u. a. *Experience report: Peer instruction in introductory computing*. 2010.

[Sim15] Simon. „Emergence of computing education as a research discipline“. In: *(Keine Angabe)* (2015).

[SK07] Wolfgang Schnotz und Christian Kürschner. „A Reconsideration of Cognitive Load Theory“. In: *Educational Psychology Review* 19.4 (2007), S. 469–508.

[SM09] Gerald A. Straka und Gerd Macke. „Neue Einsichten in Lehren, Lernen und Kompetenz“. In: *(Keine Angabe)* (2009).

[Spa92] Hans Spada, Hrsg. *Lehrbuch allgemeine Psychologie*. 2., korr. Aufl. Huber-Psychologie-Lehrbuch. Bern: Huber, 1992.

[Spi07] Manfred Spitzer. *Lernen*. München: Spektrum Akad. Verl, 2007.

[SPS13] Beth Simon, Julian Parris und Jaime Spacco. *How we teach impacts student learning: peer instruction vs. lecture in CS0*. 2013.

[SS13] Andreas Stefik und Susanna Siebert. „An Empirical Investigation into Programming Language Syntax“. In: *ACM Transactions on Computing Education (TOCE)* 13.4 (2013), S. 19.

[SS15] Amy Shannon und Valerie Summet. „Live coding in introductory computer science courses“. In: *Journal of Computing Sciences in Colleges* 31.2 (2015), S. 158–164.

[Sta+16] Thomas Staubitz u. a. „CodeOcean - A versatile platform for practical programming excercises in online environments“. In: *2016 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 2016, S. 314–323.

[SvP98] John Sweller, van Merrienboer, Jeroen J. G. und Paas, Fred G. W. C. In: *Educational Psychology Review* 10.3 (1998), S. 251–296.

[Swe05] John Sweller. „Implications of Cognitive Load Theory for Multimedia Learning“. In: *The Cambridge Handbook of Multimedia Learning*. Hrsg. von Richard Mayer. Cambridge: Cambridge University Press, 2005, S. 19–30.

[Ten+17] Josh Tenenberg u. a., Hrsg. *Proceedings of the 2017 ACM Conference on International Computing Education Research - ICER '17*. New York, New York, USA: ACM Press, 2017.

[TG10] Allison Elliott Tew und Mark Guzdial. *Developing a validated assessment of fundamental CS1 concepts*. 2010.

[TG11] Allison Elliott Tew und Mark Guzdial. *The FCS1: A language independent assessment of CS1 knowledge*. 2011.

[THB04] G. Tulodziecki, B. Herzig und S. Blömeke. *Gestaltung von Unterricht: Eine Einführung in die Didaktik*. Klinkhardt, 2004.

[The16] Thomas Theis. *Einstieg in Python*. 4., aktualisierte und erweiterte Auflage 2014, 2., korrigierter Nachdruck 2016. Rheinwerk Computing. Bonn: Rheinwerk, 2016.

[TI11] Ville Tirronen und Ville Isomöttönen. *Making teaching of programming learning-oriented and learner-directed*. 2011.

[Tib11] Victor Tiberius, Hrsg. *Hochschuldidaktik der Zukunftsforschung*. Wiesbaden: VS Verlag für Sozialwissenschaften, 2011.

[TN16] Yonglei Tao und Jagadeesh Nandigam. „Programming case studies as context for active learning activities in the classroom". In: *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2016, S. 1–4.

[Too11] Warren Toomey. „Quantifying the Incidence of Novice Programmers Errors". In: *School of IT, Bond University* (2011).

[Töp12] Armin Töpfer. *Erfolgreich Forschen: Ein Leitfaden für Bachelor-, Master-Studierende und Doktoranden*. 3., überarb. und erw. Aufl. 2012. Springer-Lehrbuch. Berlin, Heidelberg: Springer Berlin Heidelberg und Imprint: Springer Gabler, 2012.

[Tüc03] Manfred Tücke. *Grundlagen der Psychologie für (zukünftige) Lehrer*. Bd. Bd. 8. Osnabrücker Schriften zur Psychologie. Münster: Lit, 2003.

[Tur11] George Turner. *Hochschule von A - Z*. 2. Aufl. Berlin: BWV, Berliner Wiss.-Verl, 2011.

[Ulr16] Immanuel Ulrich. *Gute Lehre in der Hochschule: Praxistipps zur Planung und Gestaltung von Lehrveranstaltungen*. Wiesbaden: Springer, 2016.

[Utt+13] Ian Utting u. a. *A fresh look at novice programmers' performance and their teachers' expectations*. 2013.

[Vih+13a] Arto Vihavainen u. a. *Massive increase in eager TAs: experiences from extreme apprenticeship-based CS1*. 2013.

[Vih+13b] Arto Vihavainen u. a. *Scaffolding students' learning using test my code*. 2013.

[Vis90] Willemien Visser. „More or less following a plan during design: Opportunistic deviations in specification". In: *International Journal of Man-Machine Studies* 33.3 (1990), S. 247–278.

[VL13] Arto Vihavainen und Matti Luukkainen. „Results from a Three-Year Transition to the Extreme Apprenticeship Method“. In: *Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies*. ICALT ’13. Washington, DC, USA: IEEE Computer Society, 2013, S. 336–340.

[VPL11] Arto Vihavainen, Matti Paksula und Matti Luukkainen. *Extreme apprenticeship method in teaching programming for beginners*. 2011.

[Wah13] Diethelm Wahl. *Lernumgebungen erfolgreich gestalten. Vom trägen Wissen zum kompetenten Handeln. 3. Aufl. mit Methodensammlung*. Bad Heilbrunn: Klinkhardt, 2013.

[WB16] Phillip C. Wankat und Lisa G. Bullard. „The Future of Engineering Education – Revisited“. In: *Chemical Engineering Education* 50.1 (2016), S. 19–28.

[Wha+06] Jacqueline L. Whalley u. a. *An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies*. 2006.

[Wid09] Wolfgang Widulle, Hrsg. *Handlungsorientiert Lernen im Studium: Arbeitsbuch für soziale und pädagogische Berufe*. 1. Aufl. Wiesbaden: VS Verlag für Sozialwissenschaften / GWV Fachverlage GmbH Wiesbaden, 2009.

[Wil11] Johannes Wildt. „”Forschendes Lernen” als Hochform aktiven und kooperativen Lernens“. In: *Ökonomisierung der Wissensgesellschaft : wie viel Ökonomie braucht und wie viel Ökonomie verträgt die Wissensgesellschaft?* (2011).

[Wil15] Wilfried Schubarth. „Beschäftigungsfähigkeit als Bildungsziel an Hochschulen“. In: *Aus Politik und Zeigeschichte* 65 (2015), S. 23–30.

[Win06] Jeannette M. Wing. „Computational Thinking“. In: *Commun. ACM* 49.3 (2006), S. 33–35.

[Win08] Jeannette M. Wing. „Computational thinking and thinking about computing“. In: *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* 366.1881 (2008), S. 3717–3725.

[WK14] Wilfried Schubarth und Karsten Speck. *Employability und Praxisbezüge im wissenschaftlichen Studium: HRK-Fachgutachten ausgearbeitet für die HRK*. Hrsg. von HRK. 2014.

[Woo+13] Krissi Wood u. a. *It’s never too early: Pair programming in CS1*. 2013.

[Zin+13] Daniel Zingaro u. a. *Facilitating code-writing in PI classes*. 2013.

[Zum03] J. Zumbach. *Problembasiertes Lernen*. Waxmann Verlag, 2003.

# Anhang A

# Beispiele für interaktive Lernmaterialien

## A.1 Interaktives Dokument

Um den Leser dieser Arbeit ein besseres Bild über die verwendeten Dokumente zu geben, werden hier ein Beispiel in der Leseansicht sowie das verwendete Format zur Abspeicherung dieses gezeigt.

Grundelemente von Python 1 (Kapitel 5) Michael Ebert · Zuletzt aktualisiert vor 7 Monaten

# Kapitel 5 - Grundelemente von Python 1

Variablen, Ausdrücke und Anweisungen

## Inhalte

5.1 Kommentare
5.2 Werte
5.3 Variablen
5.4 Variablen und Schlüsselwörter
5.5 Operatoren und Operanden
5.6 Ausdrücke und Anweisungen
5.7 Rangfolge von Operatoren
5.8 String-Operationen
5.9 Ein paar Fehlerursachen
Glossar

### Wichtige Unterpunkte:

Dezimaltrennzeichen

## 5.1 Kommentare

Kommentare sind Texte im Quellcode, die bei der Ausführung nicht interpretiert werden. Diese dienen dazu, die Vorgänge und Gedanken zu einer Zeile oder einem Befehl zu erläutern.

Kommentare werden mit `#` eingeleitet.

```
# Eine Zeile in der nur ein Kommentar steht
print("Hallo") # Ich bin ein weiterer Kommentar
```

Kopieren

## 5.2 Werte

Ein **Wert** - auch *Literal* genannt - ist eines der grundlegenden Elemente, mit denen ein Programm arbeitet. Das kann z.B. ein Buchstabe oder eine Zahl sein.

Wir haben bereits ein paar Werte verwendet: `"Hallo Elektrotechnik"` oder `500` oder `3.4`

Jeder Wert gehört einem bestimmten Typ an.

- `500` ist ein **Integer** (eine Ganzzahl)
- `"Hallo .."` ist ein **String** (eine Zeichenkette) [Strings werden in Hochkommata gesetzt]
- `3.4` ist ein **Float** (eine Fließkommazahl)

Leseansicht eines Dokuments mit verschiedenen Inhaltselementen

## A.2 Beispiel Dokumentenformat

Das Kapitel A.2 verdeutlicht die Speicherung eines Dokuments in *JSON* und des zugrundeliegenden `nbformat 4.0` (Notebook-Format).

```
{
  "cells": [
    {
      "cell_type": "markdown",
      "metadata": {
        "slideshow": {
          "slide_type": "slide"
        }
      },
      "source": "# Kapitel 5 - Grundelemente von Python 1\n## <small>Variablen,
       ↪  Ausdrücke und Anweisungen</small>"
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "slideshow": {
          "slide_type": "skip"
        }
      },
      "source": "### Inhalte\n\n[5.1 Kommentare](#5-1-kommentare)  \n[5.2
       ↪  Werte](#5-2-werte)  \n[5.3 Variablen](#5-3-variablen)  \n[5.4 Variablen
       ↪  und Schlüsselwörter](#5-4-variablen-und-schluesselwoerter)  \n[5.5
       ↪  Operatoren und Operanden](#5-5-operatoren-und-operanden)  \n[5.6
       ↪  Ausdrücke und Anweisungen](#5-6-ausdruecke-und-anweisungen)  \n[5.7
       ↪  Rangfolge von Operatoren](#5-7-rangfolge-von-operatoren)  \n[5.8
       ↪  String-Operationen](#5-8-string-operationen)  \n[5.9 Ein paar
       ↪  Fehlerursachen](#5-9-ein-paar-fehlerursachen)  \n[Glossar](#glossar)
       ↪  \n\n\n#### Wichtige Unterpunkte:
       ↪  \n[Dezimaltrennzeichen](#dezimaltrennzeichen)\n"
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "slideshow": {
          "slide_type": "slide"
        }
      },
      "source": "### 5.1 Kommentare\nKommentare sind Texte im Quellcode, die bei
       ↪  der Ausführung nicht interpretiert \nwerden. Diese dienen dazu, die
       ↪  Vorgänge und Gedanken zu einer Zeile oder einem \nBefehl zu erläutern.
       ↪  \n\nKommentare werden mit `#` eingeleitet."
    },
    {
      "cell_type": "code",
```

```
      "metadata": {
        "slideshow": {
          "slide_type": ""
        }
      },
      "source": "# Eine Zeile in der nur ein Kommentar steht\nprint(\"Hallo\") #
       ↪  Ich bin ein weiterer Kommentar"
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "slideshow": {
          "slide_type": "slide"
        }
      },
      "source": "### 5.2 Werte\n\nEin **Wert** - auch *Literal* genannt - ist eines
       ↪  der grundlegenden Elemente, mit denen \nein Programm arbeitet. Das kann
       ↪  z.B. ein Buchstabe oder eine Zahl sein.\n\nWir haben bereits ein paar
       ↪  Werte verwendet: `\"Hallo Elektrotechnik\"` oder `500` oder `3.4`"
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "slideshow": {
          "slide_type": "slide"
        }
      },
      "source": "Jeder Wert gehört einem bestimmten Typ an.\r\n\r\n* `500` ist ein
       ↪  **Integer** (eine Ganzzahl)\r\n* `\"Hallo ..\"` ist ein **String** (eine
       ↪  Zeichenkette) [Strings werden in Hochkommata gesetzt]\r\n* `3.4` ist ein
       ↪  **Float** (eine Fließkommazahl)"
    },
    {
      "cell_type": "markdown",
      "metadata": {
        "slideshow": {
          "slide_type": "slide"
        }
      },
      "source": "Python verrät Ihnen auch den Typ eines Wertes:"
    },
    {
      "cell_type": "code",
      "metadata": {
        "slideshow": {
          "slide_type": ""
        }
      },
      "source": "print(type(\"Hallo Elektrotechnik!\"))\nprint(type(17))"
    }
```

```
75    ],
76    "metadata": {
77      "author": "Michael Ebert",
78      "kernelspec": {
79        "display_name": "Python 3",
80        "language": "python",
81        "name": "webbox"
82      },
83      "language_info": {
84        "name": "python",
85        "version": 3
86      },
87      "lastUpdate": "2016-10-12T06:54:24.834Z",
88      "newStateCreatedAt": "2016-10-12T06:52:24.720Z",
89      "title": "Grundelemente von Python 1 (Kapitel 5)"
90    },
91    "nbformat": 4,
92    "nbformat_minor": 0
93  }
```

Umfangreicheres Beispiel eines Dokuments im `nbformat 4.0`

## A.3 Informationen zur Plattform

Folgend sind die Plattform sowie die dazugehörigen Anleitungen und Repositories aufgelistet und verlinkt, welche für den Betrieb der beschriebenen Plattform verwendet wurden.

### A.3.1 trycoding.io

Die Plattform selbst ist unter www.trycoding.io verfügbar und kann, ohne eine Registrierung, eingeschränkt verwendet werden. Der für die Evaluation verwendete Kurs mit allen Beispielen ist unter www.trycoding.io/course/python-prog1 erreichbar.

### A.3.2 Repositories und Anleitungen

Alle zum Betrieb notwendige Informationen sind auf mehrere Repositories aufgeteilt und können öffentlich eingesehen werden.

**Plattform-Server** Die Plattform, welche die interaktiven Lehrmaterialien ausliefert sowie die webbasierte Entwicklungsumgebung und die dazugehörige Benutzer- und

Rechteverwaltung ist unter https://github.com/ebertmi/webbox verfügbar. Folgende Liste gibt einen kurzen Überblick auf die enthaltenen Komponenten:

- Installationsanleitung
- Zusätzliche Dokumentation und Hinweise zur Fehlerbehebung
- Konfigurationsdateien für Sandboxes
- Beispielkonfiguration von *nginx* für AWS
- CLI-Skripte zum administrieren der Plattform
- Backup-Skripte
- Datenbank-Migrationsskripte
- Plattform-Server
- Webbasierte Entwicklungsumgebung
- Komponenten für die Erstellung von Dokumenten und Präsentationen
- Oberfläche zur Verwaltung der Plattform
- Konfigurationen für verschiedene Programmiersprachen für die Verwendung mit der Sandbox

**Sandbox-Server** Der Server, welcher neue Sandboxes bei Benutzeranfragen erstellt und die bidirektionale Kommunikation zwischen einer Sandbox und einem Clienten verwaltet ist unter https://github.com/ebertmi/sandbox-server erreichbar.

**Sourcebox-Sandbox** Die im Rahmen der Masterarbeit erstellte Implementierung der konzipierten Sandbox zur sicheren Ausführung von Quellcode ist unter https://github.com/ebertmi/sourcebox-sandbox verfügbar.

**Sourcebox-Web** Diese Komponente enthält Bibliotheken für die Ansteuerung der Sandbox auf seitens des Servers und des Clients. Bei der Umsetzung des Sandbox-Servers und der Plattform wurden diese Bibliotheken verwendet und sind als Abhängigkeit eingebunden. Der Quellcode ist unter https://github.com/ebertmi/sourcebox-web verfügbar.

**Skulpt** Zuletzt ist unter https://github.com/ebertmi/skulpt eine eigenständige Version der Bibliothek *skulpt* öffentlich verfügbar. Diese enthält spezielle Anpassungen für den Betrieb auf trycoding.io in Bezug auf Benutzereingaben und das Terminieren von laufenden Programmen. Zusätzlich sind weitere nicht direkt verfügbare Funktionen und Module implementiert sowie Fehler für die Verwendung auf Rechnern mit deutschem Tastaturlayout behoben worden.

# Anhang B

# Daten der Evaluation

## B.1 Zuordnung der Konzepte zu Übungsaufgaben

Folgende Tabelle zeigt die Zuordnung der jeweiligen Programmierkonzepte zu den Aufgaben, die innerhalb der Übungen bearbeitet wurden.

| **Aufgabe** | **Konzept** |
|---|---|
| u1_hello_world | output |
| u1_rauminhalt | operators |
| u1_mittel | operators |
| u1_bmi | operators |
| u1_fehlersuche | syntax |
| u1_zirpen | operators |
| u1_sternchen | strings |
| u1_bmi_eingabe | input |
| Ü2 Aufgabe 1 | math |
| Ü2 Aufgabe 2 Mario | for |
| Ü2 Aufgabe 3 (Turtle) | statements |
| Ü2 Aufgabe 4 (Mario 2) | for |
| Ü3 Konvertierung | input |
| Ü3 - Turtle-Mario | for |
| Ü4 - BMI Teil 1 | functions |
| U4 - Gerade oder nicht | alternative |
| U4 - Groß & Klein | alternative |

Die Tabelle wird auf der nächsten Seite fortgesetzt.

| Aufgabe | Konzept |
|---|---|
| U4 - Klausurnoten | alternative |
| U4 - Dreieck | alternative |
| Ü4 - Fermat | functions |
| Ü4 - Blume | functions |
| Ü5 - Boolesche Algebra | boolean-expression |
| Ü5 - Polygon | for |
| Ü5 - Spiralen | while |
| Ü5 - Primzahlen | while |
| Ü6 - Tatort | for |
| Ü6 - Listen | for |
| Zusatz - Zensur | while |
| Zusatz Schleife 1 | for |
| Zusatz Schleife 2 | for |
| Zusatz Schleife 3 | while |
| Zusatz Schleife 4 | for |
| Ü7 - ist_sortiert | while |
| Ü7 - Lotto | for |
| Ü7 - Palindrom | functions |
| Ü8 - Lange Wörter | files |
| Ü8 - Wörter ohne e | for |
| Ü8 - Alphabetisch | functions |
| Ü8 - Caesar | functions |
| Ü9 - Voltmeter | files |
| Ü9 - Das erste Modul | modules |
| Ü10 - List Comphrension | for |
| Ü10 - Listen mit Python | for |
| Ü10 - Listen NumPy | array |
| Ü10 - Coulumb | array |
| Ü10 - Plot, Plot, Plot | array |
| Ü10 - Numpy-Array | array |
| Ü10 - Mehrere Linien | array |
| Ü10 - Determinante | array |

**Tabelle 49**: Zuordnung der Aufgaben und des jeweiligen adressierten Programmierkonzepts

## B.2 Anzahl der Aktivitäten und Benutzer je Beispiel

Folgende Tabelle listet alle Aktivitäten und Benutzer für jede Aufgabe, die im Unterricht durchgeführt wurde, auf. Dabei sind die Anzahlen einmal für die Zeit im Unterricht und unabhängig vom Unterricht angegeben.

**Tabelle 50**: Anzahl der Studierenden für alle Aufgaben im Unterricht

| Aufgabe | Aktivitäten Insgesamt[1] | Studierende Insgesamt | Aktivitäten im Unterricht | Studierende im Unterricht[2] |
|---|---|---|---|---|
| Variablentausch | 247 | 36 | 148 | 19 |
| Übung TBPL | 311 | 35 | 121 | 14 |
| Fahrenheit | 266 | 33 | 186 | 17 |
| Polygon mit Turtle | 145 | 38 | 69 | 22 |
| Polygon mit for | 206 | 41 | 80 | 25 |
| Aufgabe Math. Funktionen | 671 | 42 | 476 | 35 |
| Haus vom Nikolaus | 668 | 39 | 459 | 31 |
| Aufgabe Rechteck | 408 | 46 | 241 | 34 |
| Aufgabe Programmablauf | 63 | 21 | 22 | 9 |
| Liegt dazwischen | 274 | 41 | 188 | 28 |
| Negation | 85 | 22 | 48 | 9 |
| Aussagen formulieren | 181 | 44 | 163 | 36 |
| Mehrfache Verzweigungen | 783 | 42 | 607 | 34 |
| Schaltjahr | 217 | 20 | 84 | 13 |
| Inkrementelle Vorgehensweise | 73 | 14 | 23 | 6 |
| Restdivision | 473 | 41 | 303 | 33 |
| Traversierung (while) | 232 | 37 | 117 | 19 |
| Aufgabe Trav. rückwärts | 350 | 38 | 177 | 26 |
| Häufigkeit eines Zeichens | 387 | 39 | 240 | 30 |
| Aufgabe Eingabe mit Liste | 118 | 13 | 60 | 7 |
| Tupel-Args Minimum | 129 | 25 | 107 | 21 |
| Fahrenheit-ListComprehension | 221 | 25 | 199 | 20 |
| Num - Vektoren in Python | 111 | 14 | 61 | 10 |

[1] Gesamte Anzahl der Studierenden, die die Aufgabe bearbeitet haben.
[2] Anzahl der Studierenden, welche die Aufgabe im Unterricht bearbeitet haben.

## B.3 Anzahl der Aktivitäten je Aufgabe

Folgende Tabelle listet alle Vorlesungs- und Übungsaufgaben inklusive der Anzahlen der Aktivitäten und Fehler auf.

| Aufgabe | Konzept | Aktivitäten | Fehler |
|---|---|---|---|
| Zusatz - Schleife 2 | for | 32 | 1 |
| Zusatz Schleife 1 | for | 38 | 5 |
| Ü10 - Plot, Plot, Plot | array | 39 | 3 |
| Ü10 - Listen NumPy | array | 46 | 9 |
| Ü10 - Mehrere Linien | array | 49 | 10 |
| Zusatz - Schleife 3 | while | 50 | 10 |
| Zusatz - Schleife 4 | for | 53 | 14 |
| Kap10 - Aufgabe Programmablauf | program-flow | 63 | 12 |
| Ü9 - Das erste Modul | modules | 71 | 14 |
| Kap12 - Inkrementelle Vorgehensweise | composition | 73 | 17 |
| Ü10 - Determinante | array | 75 | 23 |
| Ü10 - Numpy-Array | array | 80 | 20 |
| Kap11 - Negation | boolean-expression | 85 | 28 |
| Ü10 - List Comphrension | for | 95 | 26 |
| Ü10 - Coulumb | array | 101 | 25 |
| Num - Vektoren in Python | for | 111 | 38 |
| Kap15 - Aufgabe Eingabe mit Liste | for | 118 | 48 |
| Tupel-Args Minimum | for | 129 | 22 |
| Kap9 - Polygon mit Turtle | for | 145 | 4 |
| Kap11 - Aufgabe Boolesche Ausdrücke | boolean-expression | 181 | 34 |
| Ü8 - Caesar | functions | 189 | 49 |
| Ü10 - Listen mit Python | for | 201 | 73 |
| Kap9 - Polygon mit for | for | 206 | 4 |
| Kap11 - Schaltjahr | alternative | 217 | 46 |
| Fahrenheit-ListComphrension | for | 221 | 32 |
| Kap14 - Traversierung (while) | while | 232 | 34 |
| Kap5 - Variablentausch | variables | 247 | 50 |
| Kap6 - Fahrenheit | operators | 266 | 58 |
| Kap10 - Liegt dazwischen | functions | 274 | 89 |
| Kap5 - Übung TBPL | operators | 311 | 102 |
| Zusatz - Zensur | while | 317 | 66 |
| Kap14 - Aufgabe Trav. rückwärts | while | 350 | 98 |

Die Tabelle wird auf der nächsten Seite fortgesetzt.

| Aufgabe | Konzept | Aktivitäten | Fehler |
|---|---|---|---|
| Kap14 - Häufigkeit eines Zeichens | for | 387 | 96 |
| Kap10 - Aufgabe Rechteck | functions | 408 | 138 |
| Ü8 - Alphabetisch | functions | 413 | 103 |
| Ü7 - Palindrom | functions | 420 | 120 |
| u1_zirpen | operators | 427 | 85 |
| Ü8 - Lange Wörter | files | 436 | 123 |
| u1_bmi | operators | 467 | 88 |
| Kap13 - Restdivision | while | 473 | 82 |
| u1_hello_world | output | 482 | 81 |
| Ü4 - Fermat | functions | 582 | 114 |
| Ü8 - Wörter ohne e | for | 621 | 165 |
| u1_bmi_eingabe | input | 631 | 123 |
| Kap9 - Aufgabe Nikolaus | operators | 668 | 208 |
| Kap8 - Aufgabe Math. Funktionen | math | 671 | 253 |
| U4 - Groß & Klein | alternative | 679 | 191 |
| u1_mittel | operators | 735 | 225 |
| Ü5 - Spiralen | while | 750 | 156 |
| Kap11 - Verzweigungen | alternative | 783 | 272 |
| Ü6 - Tator | for | 824 | 179 |
| u1_sternchen | strings | 824 | 229 |
| u1_fehlersuche | syntax | 839 | 329 |
| U4 - Gerade oder nicht | alternative | 878 | 275 |
| Ü9 - Voltmeter | files | 881 | 283 |
| Ü4 - BMI Teil 1 | functions | 942 | 198 |
| U4 - Klausurnoten | alternative | 961 | 241 |
| u1_rauminhalt | operators | 963 | 268 |
| Ü3 Konvertierung | input | 1025 | 329 |
| Ü2 Aufgabe 4 (Mario 2) | for | 1037 | 179 |
| Ü4 - Blume | functions | 1152 | 310 |
| Ü7 - ist_sortiert | while | 1184 | 366 |
| Ü5 - Primzahlen | while | 1217 | 292 |
| Ü6 - Listen | for | 1238 | 414 |
| Ü2 Aufgabe 3 (Turtle) | statements | 1279 | 188 |
| Ü2 Aufgabe 2 Mario | for | 1311 | 317 |
| Ü2 Aufgabe 1 | math | 1391 | 412 |
| Ü4 - Dreieck | alternative | 1642 | 461 |
| Ü7 - Lotto | for | 1704 | 416 |

Die Tabelle wird auf der nächsten Seite fortgesetzt.

| Aufgabe | Konzept | Aktivitäten | Fehler |
|---|---|---|---|
| Ü5 - Boolesche Algebra | boolean-expression | 1820 | 382 |
| Ü3 - Turtle-Mario | for | 2023 | 435 |
| Ü5 - Polygon | for | 2203 | 614 |

**Tabelle 51**: Anzahl der Aktivitäten und Fehler je Aufgabe

## B.4 Häufigster Fehler je Aufgabe

| Aufgabe | Häufigster Fehler | Anzahl | Anzahl insg. |
|---|---|---|---|
| Zusatz - Schleife 2 | SyntaxError | 1 | 1 |
| Ü10 - Plot, Plot, Plot | NameError | 2 | 3 |
| Kap9 - Polygon mit Turtle | SyntaxError | 4 | 4 |
| Kap9 - Polygon mit for | NameError | 3 | 4 |
| Zusatz Schleife 1 | SyntaxError | 4 | 5 |
| Ü10 - Listen NumPy | TypeError | 7 | 9 |
| Ü10 - Mehrere Linien | AttributeError | 5 | 10 |
| Zusatz - Schleife 3 | NameError | 7 | 10 |
| Kap10 - Aufgabe Programmablauf | ExternalError | 7 | 12 |
| Ü9 - Das erste Modul | TypeError | 7 | 14 |
| Zusatz - Schleife 4 | TypeError | 5 | 14 |
| Kap12 - Inkrementelle Vorgehensweise | SyntaxError | 11 | 17 |
| Ü10 - Numpy-Array | NameError | 6 | 20 |
| Tupel-Args Minimum | SyntaxError | 8 | 22 |
| Ü10 - Determinante | ValueError | 12 | 23 |
| Ü10 - Coulumb | ValueError | 8 | 25 |
| Ü10 - List Comphrension | SyntaxError | 10 | 26 |
| Kap11 - Negation | SyntaxError | 27 | 28 |
| Fahrenheit-ListComprehension | SyntaxError | 16 | 32 |
| Kap11 - Aufgabe Boolesche Ausdrücke | SyntaxError | 29 | 34 |
| Kap14 - Traversierung (while) | SyntaxError | 15 | 34 |
| Num - Vektoren in Python | TypeError | 18 | 38 |
| Kap11 - Schaltjahr | SyntaxError | 25 | 46 |
| Kap15 - Aufgabe Eingabe mit Liste | SyntaxError | 29 | 48 |
| Ü8 - Caesar | TypeError | 17 | 49 |
| Kap5 - Variablentausch | SyntaxError | 36 | 50 |

Die Tabelle wird auf der nächsten Seite fortgesetzt.

| Aufgabe | Häufigster Fehler | Anzahl | Anzahl insg. |
|---|---|---|---|
| Kap6 - Fahrenheit | NameError | 22 | 58 |
| Zusatz - Zensur | SyntaxError | 25 | 66 |
| Ü10 - Listen mit Python | SyntaxError | 39 | 73 |
| u1_hello_world | SyntaxError | 60 | 81 |
| Kap13 - Restdivision | SyntaxError | 44 | 82 |
| u1_zirpen | SyntaxError | 45 | 85 |
| u1_bmi | SyntaxError | 58 | 88 |
| Kap10 - Liegt dazwischen | SyntaxError | 67 | 89 |
| Kap14 - Häufigkeit eines Zeichens | SyntaxError | 36 | 96 |
| Kap14 - Aufgabe Trav. rückwärts | IndexError | 45 | 98 |
| Kap5 - Übung TBPL | ValueError | 96 | 102 |
| Ü8 - Alphabetisch | SyntaxError | 29 | 103 |
| Ü4 - Fermat | SyntaxError | 76 | 114 |
| Ü7 - Palindrom | SyntaxError | 27 | 120 |
| Ü8 - Lange Wörter | NameError | 46 | 123 |
| u1_bmi_eingabe | TypeError | 38 | 123 |
| Kap10 - Aufgabe Rechteck | SyntaxError | 85 | 138 |
| Ü5 - Spiralen | NameError | 52 | 156 |
| Ü8 - Wörter ohne e | SyntaxError | 57 | 165 |
| Ü6 - Tator | SyntaxError | 87 | 179 |
| Ü2 Aufgabe 4 (Mario 2) | SyntaxError | 83 | 179 |
| Ü2 Aufgabe 3 (Turtle) | NameError | 71 | 188 |
| U4 - Groß & Klein | SyntaxError | 141 | 191 |
| Ü4 - BMI Teil 1 | SyntaxError | 89 | 198 |
| Kap9 - Aufgabe Nikolaus | SyntaxError | 76 | 208 |
| u1_mittel | SyntaxError | 148 | 225 |
| u1_sternchen | SyntaxError | 162 | 229 |
| U4 - Klausurnoten | SyntaxError | 146 | 241 |
| Kap8 - Aufgabe Math. Funktionen | SyntaxError | 149 | 253 |
| u1_rauminhalt | SyntaxError | 113 | 268 |
| Kap11 - Verzweigungen | SyntaxError | 221 | 272 |
| U4 - Gerade oder nicht | SyntaxError | 189 | 275 |
| Ü9 - Voltmeter | SyntaxError | 79 | 283 |
| Ü5 - Primzahlen | SyntaxError | 168 | 292 |
| Ü4 - Blume | SyntaxError | 146 | 310 |
| Ü2 Aufgabe 2 Mario | SyntaxError | 226 | 317 |
| Ü3 Konvertierung | SyntaxError | 134 | 329 |

Die Tabelle wird auf der nächsten Seite fortgesetzt.

| Aufgabe | Häufigster Fehler | Anzahl | Anzahl insg. |
|---|---|---|---|
| u1_fehlersuche | TypeError | 172 | 329 |
| Ü7 - ist_sortiert | SyntaxError | 141 | 366 |
| Ü5 - Boolesche Algebra | SyntaxError | 234 | 382 |
| Ü2 Aufgabe 1 | SyntaxError | 227 | 412 |
| Ü6 - Listen | SyntaxError | 209 | 414 |
| Ü7 - Lotto | IndexError | 226 | 416 |
| Ü3 - Turtle-Mario | ValueError | 142 | 435 |
| Ü4 - Dreieck | SyntaxError | 182 | 461 |
| Ü5 - Polygon | SyntaxError | 352 | 614 |

**Tabelle 52:** Häufigster Fehler je Aufgabe

**Tabelle 53**: Häufigste Fehlerarten

| Fehlerart | Anzahl |
|---|---|
| io.UnsupportedOperation | 1 |
| SuspensionError | 1 |
| OverflowError | 1 |
| LookupError | 2 |
| KeyError | 3 |
| NotImplementedError | 3 |
| AssertionError | 3 |
| OSError | 4 |
| Error | 4 |
| numpy.linalg.linalg.LinAlgError | 6 |
| FileNotFoundError | 9 |
| turtle.TurtleGraphicsError | 12 |
| ParseError | 16 |
| UnicodeDecodeError | 18 |
| ExternalError | 36 |
| ZeroDivisionError | 37 |
| ImportError | 52 |
| UnboundLocalError | 153 |
| AttributeError | 300 |
| IndentationError | 444 |
| IndexError | 501 |
| ValueError | 1004 |
| TypeError | 1538 |
| NameError | 2114 |
| SyntaxError | 5552 |

## B.5 Fragebogen

Der in Kapitel 7 verwendete Fragenkatalog zur Erfassung der studentischen Sichtweise ist nachfolgend dargestellt.

## Fragebogen

Liebe Teilnehmerin, lieber Teilnehmer,
dieser Bogen wird maschinell ausgewertet. Markieren Sie eine Antwort bitte in der folgenden Weise: ○⊗○.
Wenn Sie eine Antwort korrigieren möchten, füllen Sie bitte den falsch markierten Kreis und noch etwas darüber hinaus aus, ungefähr so: ○●⊗.

Ziffern sollen ungefähr so aussehen: 0 1 2 3 4 5 6 7 8 9; Korrekturen so: 

# Elektrotechnik Evaluation WS16/17

### Allgemeine Daten

| | |
|---|---|
| Studiengang | ○ Automatisierung und Robotik ○ Elektro- und Informationstechnik ○ Erneuerbare Energien |

### Gestaltung der Lehrveranstaltung

| | trifft nicht zu | | | | trifft voll zu |
|---|---|---|---|---|---|
| Der inhaltliche Aufbau der Lehrveranstaltung war logisch/nachvollziehbar. | ○ | ○ | ○ | ○ | ○ |
| Zum Mitdenken und Durchdenken des Stoffes wurde angeregt. | ○ | ○ | ○ | ○ | ○ |
| Die Lehrveranstaltung wurde in interessanter Form gehalten. | ○ | ○ | ○ | ○ | ○ |
| Die/Der Lehrende benutzte oft Beispiele, die zum Verständnis der Lehrinhalte beitrugen. | ○ | ○ | ○ | ○ | ○ |
| Die Bedeutung/Der Nutzen der behandelten Themen wird vermittelt. | ○ | ○ | ○ | ○ | ○ |
| Ein Bezug zwischen Theorie und Praxis/Anwendung wird hergestellt. | ○ | ○ | ○ | ○ | ○ |
| Die von der/dem Lehrenden ausgegebenen Materialien (z. B. Literatur, Skript, E-Learning-Inhalte, etc.) haben mir sehr geholfen, den Stoff zu erarbeiten. | ○ | ○ | ○ | ○ | ○ |
| Die/Der Lehrende passte das Niveau der Lehrveranstaltung an den Wissensstand der Studierenden an. | ○ | ○ | ○ | ○ | ○ |
| Ich bin mit der E-Learning Plattform (trycoding.io) gut zurechtgekommen. | ○ | ○ | ○ | ○ | ○ |
| Es war leicht Programme im Webbrowser zu erstellen und auszuführen. | ○ | ○ | ○ | ○ | ○ |

### Inhaltliche Einschätzung

| | |
|---|---|
| Mein Vorwissen: | ○ Zu wenig, um folgen zu können ○ Genau richtig ○ Alles mir schon bekannt gewesen |
| Kreuzen Sie alle Sprachen an mit denen Sie bereits Erfahrung haben. | ○ C ○ C++ ○ Java ○ Python ○ Weitere: ..... |
| Die Anforderungen der Lehrveranstaltung: | ○ Viel zu niedrig ○ Genau richtig ○ Viel zu hoch |

| | trifft nicht zu | | | | trifft voll zu |
|---|---|---|---|---|---|
| Ich habe in der Veranstaltung häufig lauffähige Programme erstellt. | ○ | ○ | ○ | ○ | ○ |
| Es macht mir Spaß zu programmieren. | ○ | ○ | ○ | ○ | ○ |
| Mein Wissensstand ist nach der Lehrveranstaltung wesentlich höher als vorher. | ○ | ○ | ○ | ○ | ○ |
| Die in dieser Lehrveranstaltung eingeführten Begriffe/Sachverhalte kann ich wiedergeben. | ○ | ○ | ○ | ○ | ○ |
| Die in dieser Lehrveranstaltung eingeführten Begriffe/Sachverhalte habe ich verstanden. | ○ | ○ | ○ | ○ | ○ |
| Ich habe etwas Sinnvolles und Wichtiges gelernt. | ○ | ○ | ○ | ○ | ○ |

**Eigene Teilnahme**

| | trifft nicht zu | | | | trifft voll zu |
|---|---|---|---|---|---|
| In der Lehrveranstaltung habe ich mich aktiv beteiligt. | ○ | ○ | ○ | ○ | ○ |
| Beim Einbringen eigener Beiträge habe ich mich frei und äußerungsfähig gefühlt. | ○ | ○ | ○ | ○ | ○ |
| Ich habe die in der Vorlesung gestellten Aufgaben häufig bearbeitet. | ○ | ○ | ○ | ○ | ○ |
| Die Lehrveranstaltung hat mein Interesse am Studium gefördert. | ○ | ○ | ○ | ○ | ○ |
| Die Lehrveranstaltung motiviert mich dazu, mich selbst mit den Inhalten zu beschäftigen. | ○ | ○ | ○ | ○ | ○ |
| Die Dozentin/Der Dozent fördert Fragen und aktive Mitarbeit. | ○ | ○ | ○ | ○ | ○ |
| Es werden kommunikative Lehrformen eingesetzt (z.. B. Diskussion, Aufgaben, Quizzes) | ○ | ○ | ○ | ○ | ○ |
| Ich habe die Möglichkeit genutzt, die Aufgaben auch später online zu bearbeiten. | ○ | ○ | ○ | ○ | ○ |
| Mein Arbeitsaufwand ist, verglichen mit anderen Lehrveranstaltungen, hoch. | ○ | ○ | ○ | ○ | ○ |
| | zu leicht | | | | zu schwer |
| Wie schätzen Sie die Schwierigkeit der Übungen ein? | ○ | ○ | ○ | ○ | ○ |
| | 0 h | 1-2 h | 3-4 h | 5 - 6 h | mehr als 6 h |
| Schätzen Sie bitte den eigenen wöchentlichen zeitlichen Aufwand, den Sie insgesamt für die Vor- und Nachbereitung **inklusive** der Vorbereitung auf die Prüfungsleistung aufbringen. Angaben bitte **exklusive** der Vorlesung und der Terminübungsstunden. | ○ | ○ | ○ | ○ | ○ |

**Schlussfragen**

| | trifft nicht zu | | | | trifft voll zu |
|---|---|---|---|---|---|
| Ich habe die Übungen häufig von zuhause aus bearbeitet. | ○ | ○ | ○ | ○ | ○ |
| Ich habe die Übungen häufig an den Übungsterminen bearbeitet. | ○ | ○ | ○ | ○ | ○ |
| Jetzt finde ich das Thema interessanter als zu Beginn der Lehrveranstaltung. | ○ | ○ | ○ | ○ | ○ |
| Ich wurde zur Mitarbeit aufgefordert. | ○ | ○ | ○ | ○ | ○ |
| In dieser Lehrveranstaltung habe ich viel gelernt. | ○ | ○ | ○ | ○ | ○ |
| Ich fühle mich in der Lage, kleine Programme zu schreiben. | ○ | ○ | ○ | ○ | ○ |
| | 1 | 2 | 3 | 4 | 5 |
| Wenn man alles in einer Note zusammenfassen könnte, würde ich der Veranstaltung folgende Note geben: | ○ | ○ | ○ | ○ | ○ |

| Weitere Anmerkungen (z. B. Was hat Ihnen gut gefallen?, Was hat Ihnen nicht so gut gefallen? und Sonstiges) |
|---|
| |

| Anregungen und Verbesserungen für die E-Learning-Plattform |
|---|
| |

# Eigene Publikationen

Folgende Publikationen sind während meiner Promotion entstanden:

Michael Ebert und Wolfram Haupt. „Combining MOOC Methods and Blended Learning for an Introductory Programming Course for Electrical Engineers“. In: *EDULEARN14 Proceedings*. 2014

M. Ebert und W. Haupt. „Leveraging Parson's Problems and Code-Fragment-Questions in a Quiz for an Interactive Programming EBook“. In: *EDULEARN15 Proceedings*. 7th International Conference on Education and New Learning Technologies. IATED, 2015, S. 7691–7698

Michael Ebert und Wolfram Haupt. „Browserbasierte Programmierung und Interaktion mit Arduino-Aufbauten im Bereich Elektrotechnik für Informatik e.V. (GI), München, 1.-4. September 2015“. In: *DeLFI 2015 - Die 13. e-Learning Fachtagung Informatik der Gesellschaft für Informatik e.V. (GI), München, 1.-4. September 2015*. Hrsg. von Hans Pongratz und Reinhard Keil. LNI. GI, 2015, S. 333–335

Michael Ebert und Markus Ring. „A Presentation Framework for Programming in Programing Lectures“. In: *2016 IEEE Global Engineering Education Conference (EDUCON)*. 2016

Paula Figas u. a. „Learning programming languages through input-providing tasks“. In: *2016 IEEE Global Engineering Education Conference, EDUCON 2016, Abu Dhabi, United Arab Emirates, April 10-13, 2016*. IEEE, 2016, S. 419–424

Michael Ebert. „Increase active learning in programming courses“. In: *2017 IEEE Global Engineering Education Conference, EDUCON 2017, Athens, Greece, April 25-28, 2017*. IEEE, 2017, S. 848–851