Technische Universität Carolo-Wilhelmina zu Braunschweig

Fakultät für Architektur, Bauingenieurwesen und Umweltwissenschaften

Masterarbeit

Janis Zisenis

4104786

Entwicklung und Implementierung eines agilen Lehrkonzepts basierend auf dem Paradigma der Testgetriebenen Entwicklung

Institut für rechnergestützte Modellierung im Bauingenieurwesen

Erstprüfer: Prof. Dr.-Ing. habil. Manfred Krafczyk

Zweitprüfer: Prof. Dr. rer. nat. Martin Geier

Betreuer: Dr.-Ing. Jan Linxweiler

Braunschweig 2018

Technische Universität Braunschweig

Inst. für Rechnergestützte Modellierung im Bauingenieurwesen (iRMB)
Pockelsstr. 3, 38106 Braunschweig, Germany

Geschäftsstelle Fakultät 3
- im Hause -

iRMB
INSTITUT FÜR RECHNERGESTÜTZTE MODELLIERUNG IM BAUINGENIEURWESEN
INSTITUTE FOR COMPUTATIONAL MODELING IN CIVIL ENGINEERING
Prof. Dr.-Ing. habil. M. Krafczyk
Fon: +49-(0)531-391-94501
Fax: +49-(0)531-391-94511
irmb@tu-bs.de

Masterarbeit

für Herrn Janis Zisenis (Matrikelnr.: 4104786)

„Entwicklung und Implementierung eines agilen Lehrkonzepts basierend auf dem Paradigma der Testgetriebenen Entwicklung“

Aufgabenstellung

In den Bereichen der Programmierung und der Softwareentwicklung außerhalb des akademischen Lehrbetriebs haben sich insbesondere bei moderatem Ressourceneinsatz die sog. agilen Methoden bewährt, die auf dem Konzept der Selbstorganisation basieren und neben einer effizienteren Vermittlung von Fachinhalten einen größeren Fokus auf soziale Interaktion in der Lern- bzw. Entwicklungsphase legen. Agile Methoden bedienen sich i.d.R. visueller Gestaltungswerkzeuge, um den Teamfortschritt sichtbar zu machen. Die Entwicklung eines solchen Software-Werkzeugs kann dabei helfen, den Studierenden eine Art Lernfortschrittskontrolle zu ermöglichen und gleichzeitig dem Lehrenden potentielle Leistungsdefizite aufzeigen. Langfristig kann durch die erhobenen Daten auch die Komplexität des Lehrstoffs evaluiert und auf die durchschnittliche Leistungsfähigkeit der Studierenden angepasst werden. Zur Motivation der Lernenden, sich selbständig der Thematik zu nähern, soll prototypisch ein Werkzeug entwickelt werden, das den Studierenden unmittelbares Feedback beim Programmieren und der Bearbeitung von Aufgabenstellungen liefert. Der Ablauf der Veranstaltung soll dabei in “Mikrozyklen” unterteilt werden. Diese beinhalten als zentrales Element ein unmittelbares und fortlaufendes Feedback in Form unterschiedlicher Metriken zur Visualisierung des individuellen Teamfortschritts. Das Feedback soll dabei auf einem zentralen Versionskontrollsystem basieren, das sowohl die Aufgabenstellung als auch die Projektkonfiguration und die Lösung für die Studierenden vorhält. Dabei erfolgt eine automatisierte Kontrolle des Bearbeitungsfortschritts mit Hilfe zuvor definierter Tests vergleichbar mit denen der Testgetriebenen Entwicklung (TDD vgl. Beck 2002). Das Ausführen dieser Tests wiederum ermöglicht das Ableiten einer Visualisierung im Stil eines BurnDown-Charts.

Im Einzelnen beinhaltet die Arbeit folgende Arbeitspunkte:

1. Erarbeitung eines Konzepts zur Spezifikation von Programmieraufgaben mit Hilfe von Unit-Tests im Sinne der Testgetriebenen Entwicklung nach Beck 2002. Dabei dienen die Unit-Tests zum einen dazu, die Studierenden durch die Aufgabenstellungen zu führen und liefern anderseits ein stetiges Feedback über den derzeitigen Fortschritt. In diesem Zusammenhang ist insbesondere auf die „Kontinuität" aufeinanderfolgender Tests zu achten, um einen „unterbrechungsfreies" Bearbeiten der Aufgabenstellungen zu ermöglichen.
2. Exemplarische Umsetzung des erarbeiteten Konzepts anhand von existierenden Aufgabenstellungen aus dem Lehrbetrieb des Instituts.
3. Realisierung einer Softwareinfrastruktur zur Ausführung der beteiligten Komponenten wie Versionskontrolle (z.B. GitLab) und Continuous Integration (z.B. Jenkins) sowie einer Visualisierung des Fortschritts (Burn-Down-Chart) basierend auf bestehenden (frei verfügbaren) Lösungen.
4. Entkopplung der Softwareinfrastruktur von der verwendeten Hardwareplattform durch den Einsatz von Containervirtualisierung (z.B. Docker).
5. Integration der Fortschrittsvisualisierung als Plugin in die verwendete Entwicklungsumgebung (optional).
6. Ausführliche Dokumentation des Konzepts, der exemplarischen Umsetzung sowie der softwaretechnischen Infrastruktur.

Als Zweitprüfer benenne ich JProf. Dr. Martin Geier.

Als Betreuer benenne ich Dr.-Ing. Jan Linxweiler.

Prof. Dr.-Ing. habil. Manfred Krafczyk

ausgegeben am:	12.06.2018	verlängert bis:	18.10.2018
abzugeben am:	02.10.2018	abgegeben am:	

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingsverzeichnis

Abkürzungsverzeichnis

CI *Continuous Integration*

CSE *Computational Sciences in Engineering*

HTTP *Hypertext Transfer Protocol*

IDE *Integrated Development Environment*

iRMB Institut für rechnergestützte Modellierung im Bauingenieurwesen

LA Lernaktivität

LÜ Lernzielüberprüfung

LZ Lernziel

SUT *System Under Test*

TDD *Test-Driven Development*

UoW *Unit of Work*

PP *Pair Programming*

VCS *Version Control System*

XML *Extensible Markup Language*

KAPITEL 1

Einleitung

Die Qualität eines Lernangebots misst sich an der Lerneffektivität bzw. an den Lernerfolgen, die bei den Lernenden im Rahmen einer Lehrveranstaltung zu verzeichnen sind [18]. Dabei wird die Effektivität maßgeblich durch die verwendete Methodik beeinflusst. Aufgrund der individuellen Zusammensetzung ergeben sich bei der Anwedung der gleichen Methodik unterschiedliche Lernerfolge für verschiedene Lerngruppen. Aus dieser Kontextabhängigkeit leitet sich das Bedürfnis der stetigen Reflexion und Anpassung der Methodik ab. Zu diesem Zweck gilt es, Strukturen für regelmäßiges Feedback in das Lehrkonzept zu integrieren, die die Evaluation erleichtern [18, 32]. Da diese Art des Lehrens jedoch meistens mit einem erhöhten Ressourcenaufwand seitens der Lehrperson einhergeht, wird in der universitären Bildung häufig der Frontalunterricht bevorzugt.

Wie die Gestaltung eines Lernangebots unterliegt auch die Entwicklung von Software sich kontinuierlich verändernden Anforderungen. Um dem stetigen Anpassungsbedarf insbesondere bei moderatem Ressourceneinsatz gerecht zu werden, bewährten sich in den Bereichen der Softwareentwicklung die sogenannten agilen Methoden, die auf dem Prinzip der Selbstorganisation beruhen. Durch ihre iterativen und inkrementellen

Ansätze bieten die agilen Methoden ausreichend Gelegenheiten, um regelmäßiges Feedback zur Reflexion und Evaluation bereitzustellen. Weiterhin fokussieren sie die soziale Interaktion der beteiligten Individuen und fördern ebenfalls soziale Kompetenzen sowie Kollaboration [24, 38].

Für eine erhöhte Lehrqualität wird in dieser Arbeit ein innovatives Lehrkonzept zur Vermittlung der Programmierung erarbeitet, dass die Ansätze der agilen Methoden – insbesondere der testgetriebene Entwicklung – auf die Anwendung in der Lehre transferiert und in die Lehrveranstaltung einbindet. Weiterhin erfolgt die Realisierung eines technischen Systems, dass sich der Gestaltungswerkzeuge der agilen Methoden bedient, um die Lernfortschritte der Studierenden zu visualisieren. Infolgedessen steigt die Transparenz der Lehrveranstaltung und den Studierenden wird die Möglichkeit der Selbsteinschätzung gegeben. Darüber hinaus können die durch das technische System erhobenen Daten langfristig für die Evaluation des Curriculums herangezogen werden.

Da es sich bei didaktischen Problemstellungen wie zuvor beschrieben um kontextabhängige Probleme handelt, die stetiger Reflexion und iterativer Weiterentwicklung unterliegen sollten, geht der Gestaltung des agilen Lehrkonzepts die Analyse einer bestehenden Lehrveranstaltung voraus. Dadurch können bereits gewonnene Kenntnisse über die Vermittlung der Programmierkompetenz in die Entwicklung des agilen Lehrkonzepts einfließen. Beispielsweise können so positive Aspekte sowie effektive Ansätze übernommen und Unzulänglichkeiten verbessert werden. Als zu verbessernde Lehrveranstaltung betrachtet diese Arbeit die Vorlesung „Algorithmen und Programmieren“ des Instituts für rechnergestützte Modellierung (iRMB).

Zu Beginn dieser Arbeit werden die Aspekte der Unterrichtsplanung erläutert, die für die Entwicklung und Umsetzung des agilen Lehrkonzepts herangezogen werden. Zuerst werden dazu unterschiedliche Ansätze qualitativ hochwertiger Lehre betrachtet und die charakteristischen Merkmale „Guter Lehre“ zusammengestellt. Danach werden die geläufigsten Konzepte des didaktischen Designs vorgestellt, die die Gestaltung eines Lernangebots unterstützen. Darauf basierend wird im Anschluss die Vorlesung „Algorithmen und Programmieren“ analysiert und Verbesserungsansätze herausgearbeitet. Ein nächster Schritt leitet die Grundzüge der agilen Softwareentwicklung her und be-

schreibt die zur Verbesserung des Lehrkonzepts verwendeten agilen Vorgehensmodelle und Praktiken. Darunter befinden sich *Scrum*, die testgetriebene Entwicklung (*Test-Driven Development*; TDD), *Pair Programming* (PP) und *Continuous Integration* (CI). Danach wird die zukünftige Methodik des agilen Lehrkonzepts vorgestellt. Die zuvor genannten agilen Methoden werden auf die Lehre übertragen und die erwarteten Effekte beschrieben. Weiterhin wird in diesem Zuge auch eine zeitliche Konzeption der wöchentlichen Lehrveranstaltung vorgestellt. Um die Qualität der Lehrveranstaltung zu erhöhen und die Verbesserungsansätze zu adressieren, werden die Aufgabenstellungen im Sinne der testgetriebenen Entwicklung umgestaltet. Bei der Spezifikation gilt es, einige Aspekte zu beachten, die nachfolgend erklärt und anhand von Beispielaufgaben verdeutlicht werden. Abschließend wird die Realisierung des technischen Systems gezeigt, das die kontinuierliche Lernfortschrittsvisualisierung bereitstellt und die Lehrveranstaltung begleitet.

Bei dem in dieser Arbeit entstandenen Lehrkonzept handelt es sich um ein innovatives Konzept zur Vermittlung des Programmierens, das im kommenden Semester (WS2018/19) an der Technischen Universität durchgeführt werden soll. Zwar lassen sich die in dieser Arbeit beschriebenen positiven Effekte aus den betrachteten Ansätzen qualitativ hochwertiger Lehre ableiten, ob sich jedoch tatsächlich ein gesteigerter Lernerfolg der Studierenden einstellt, gilt es, im Anschluss an die Durchführung zu evaluieren.

Im Sinne des Genderns verwendet die vorliegende Arbeit geschlechtsneutrale Begriffe. So werden Studenten und Studentinnen im Folgenden als Studierende, Dozentinnen und Dozenten als Dozierende, Lehrer und Lehrerinnen als Lehrende oder auch als Lehrperson bezeichnet. Im Bereich der Softwareentwicklung gestaltet sich die Verwendung von geschlechtsneutralen Begriffen eher schwierig, da keine prägnante Bezeichnung für Entwicklerinnen und Entwickler oder auch Kunden und Kundinnen gefunden wurde. Hier gelten die Bezeichnungen Entwickler und Kunde gleichermaßen für alle Geschlechter.

KAPITEL 2

Aspekte der Unterrichtsgestaltung

Die Gestaltung eines Lernangebots entsteht aus der Motivation einer Lehrperson, ausgewählte Wissensinhalte an Lernende zu vermitteln. Dabei ist es von entscheidender Bedeutung, die Lehre auf die zu vermittelnden Inhalte und Kompetenzen auszurichten und das Lernen der Studierenden zu fördern. Der Entwurf eines Lernangebots fällt unter den Bereich der Didaktik (von gr. *didaskein*: lehren), der die Theorie vom Lehren und Lernen abbildet. Für den Begriff „Didaktik“ existiert in der Fachliteratur keine einheitliche Definition. Diese Arbeit legt das Verständnis nach Reinmann [32] zugrunde, das die Didaktik als gemeinsame Betrachtung der Lehrinhalte und der Lehrmethoden auffasst.

Das nachfolgende Kapitel erläutert die Aspekte der Unterrichtsgestaltung, die im Rahmen dieser Arbeit für die Analyse und Bewertung des bestehenden Lehrkonzepts (siehe Kapitel 3) sowie für die Gestaltung des zukünftigen agilen Lehrkonzepts (siehe Kapitel 5) relevant sind. Dazu wird zunächst auf die Bedeutung von Qualität in der Lehre eingegangen und beschrieben, wie diese erreicht wird. Anschließend werden die Konzepte des didaktischen Designs vorgestellt, die bei der Gestaltung eines Lehrkonzepts unterstützen.

2.1 Qualität in der Lehre

Nach Reinmann [32] handelt es sich bei dem Lernen um einen komplexen Prozess, den soziale, kognitive, emotionale und motivationale Ressourcen bedingen. Für eine hohe Qualität im Lernprozess, die durch einen effektiven Kompetenzerwerb und ertragreiche Lernerfolge charakterisiert wird, muss jener komplexe Prozess verstanden und berücksichtigt werden [18]. Durch die Anwendung von Ansätzen und Methoden, die den Lernprozess fördern, erhöht sich die Lernleistung. Infolgedessen steigt ebenfalls die Qualität des Lehrkonzepts.

Der nachfolgende Abschnitt diskutiert einige dieser Qualitätsansätze, die im Rahmen dieser Arbeit als Grundlage für die Gestaltung von hochwertigen Lehrkonzepten dienen. Im Einzelnen erläutert er die von dem Bologna-Prozess aufgegriffene Lernergebnis- und Kompetenzorientierung, verdeutlicht die Theorie des sichtbaren Lehrens und Lernens nach Hattie [18] und legt die Qualitätsansichten der Technischen Universität Braunschweig dar. Anschließend wird eine Definition „Guter Lehre“ herausgearbeitet, die die verschiedenen Ansätze kombiniert und die wichtigsten Aspekte zusammenfasst.

2.1.1 Lernergebnis- und Kompetenzorientierung

Ein konventionelles und häufig in Hochschulen anzutreffendes Konzept für Lehrveranstaltungen besteht in dem Frontalunterricht, da er im Vergleich zu kooperativen Lehrmethoden nur wenig Ressourceneinsatz fordert. Dabei trägt die Lehrperson das zu vermittelnde Wissen in einer darbietenden Weise vor. Diese Lehrstrategie des darbietenden Lehrens erzeugt nach Reinmann [32] einen rezeptiven Lernprozess, in dem die Vermittlung des fachlichen Wissens im Fokus steht. Diese Art des Lehrens regt die Studierenden nicht dazu an, das gelernte Wissen in ihrem Handeln einzubinden und erzeugt somit „träges Wissen“ [20, 32]. Der Frontalunterricht ist deshalb durch weitere Formen des Lehrens zu ergänzen [43].

Um dieses „träge Wissen“ zu reduzieren und die vermittelten Inhalte in dem Kompetenzprofil der Studierenden zu verankern, bedarf die Lehre eines Perspektivenwechsels – den *shift from teaching to learning* [20]. Hierbei soll wie in dem Bologna-Prozess an-

gestrebt, nicht die Vermittlung des Fachwissens in den Vordergrund gerückt werden. Vielmehr orientiert sich ein gutes Lehrkonzept demnach an dem Lernprozess selbst, an den Lernergebnissen (*Learning Outcomes*) und den Kompetenzen, die Studierende im Verlauf der Lehrveranstaltung erreichen sollen [1, 20]. Eine Kompetenz beschreibt die individuelle Voraussetzung zur Lösung komplexer Aufgaben [19]. Neben Fertigkeiten und (Fach-)Kenntnissen, also den kognitiven Ressourcen, spielen Kompetenzen auch auf motivationale und soziale Ressourcen an, die ebenfalls Gegenstand des Lernprozesses sein müssen [19, 32, 35].

2.1.2 Sichtbares Lehren und Lernen

Auch Hattie [18] fordert auf, den Lernprozess und die resultierenden *Learning Outcomes* in den Fokus der Lehre zu stellen. Dabei kritisiert er eine Lehrstrategie, die das Curriculum der Lehrveranstaltung als umzusetzende Pflicht verteidigt. Es kommt beim Lehren nicht darauf an, den Inhalt bis zum Ende der Veranstaltung „loszuwerden“ und dabei keine Rücksicht auf den Lernprozess der Studierenden zu nehmen, sondern auf die Art und Weise, wie die Lehrperson den Inhalt vermittelt. Dafür ist es jedoch erforderlich, das die Lehrperson ihre Funktion in der Lehre überdenkt.

Nach Hattie [18] besteht die wichtigste Funktion der Lehrperson darin, das Lehren und das Lernen „sichtbar zu machen“ (*Visible Learning*). Für ertragreiche Lernerfolge und eine hohe Effektivität der Lehrveranstaltung ist es von besonderer Bedeutung, dass die Lehrperson das Lernen der Studierenden verfolgen und interpretieren kann. Dadurch kann sie in den richtigen Momenten beabsichtigt und zu einem bestimmten Zweck eingreifen, um den Lernprozess zu lenken. Gleichzeitig müssen Lernende die Tätigkeit des Lehrenden als Grundlage ihres Lernprozesses erkennen und annehmen.

In der Studie, im Rahmen derer Hattie [18] die Aufforderung zum *Visible Learning* formuliert, synthetisiert er in 15-jähriger Arbeit die Ergebnisse aus über 800 Metaanalysen aus dem Bereich der Bildungswissenschaften der vorangegangenen 40 Jahre. Er arbeitet in diesem Zusammenhang die 138 wichtigsten Faktoren[1] heraus, die den Lern-

[1] Die Faktoren entsprechen dabei Einflussgrößen und sind nicht als multiplikative Faktoren der Mathematik zu verstehen. Für den Bezug zur Fachliteratur wird i.F. weiterhin der Begriff Faktor verwendet.

prozess und den Lernerfolg beeinflussen. Darüber hinaus stellt er ein Wertesystem – das „Hattie-Barometer“ (siehe Abbildung 2.1) – vor, das die Einflussfaktoren hinsichtlich ihrer Effektstärke (d [-]) klassifiziert [18, 46]. Dabei stellen negative Werte umkehrende Effekte dar, die sich negativ auf die Lerneffektivität auswirken. Alle Einflussfaktoren mit positiven Werten hingegen begünstigen das Lernen, dennoch ist eine weitere Unterteilung hinsichtlich ihrer Effektstärke sinnvoll. Bei Werten von 0,0 bis 0,15 handelt es sich um die sogenannten Entwicklungseffekte. Nach Hattie [18] sind die Entwicklungseffekte mit dem vergleichbar, was ein Lernender vermutlich auch ohne eine Lehrveranstaltung erreicht. Sie eignen sich deshalb nicht für die Umsetzung in einem Lernangebot. Werte zwischen 0,15 und 0,4 entsprechen einer Lerneffektivität, die eine durchschnittliche Lehrveranstaltung charakterisiert. Vor dem Hintergrund mit der Konzeption eines Lernangebots über dem durchschnittlichen Lernerfolg zu liegen, zählen Einflussfaktoren mit Werten größer als 0,4 zu den erwünschten Effekten. Sie sind daher in einer Lehrveranstaltung bevorzugt, dennoch mit Bedacht und in den richtigen Momenten, einzusetzen. Zu beachten gilt, dass sich die Studie von Hattie [18] auf den Unterricht in Schulen bezieht.

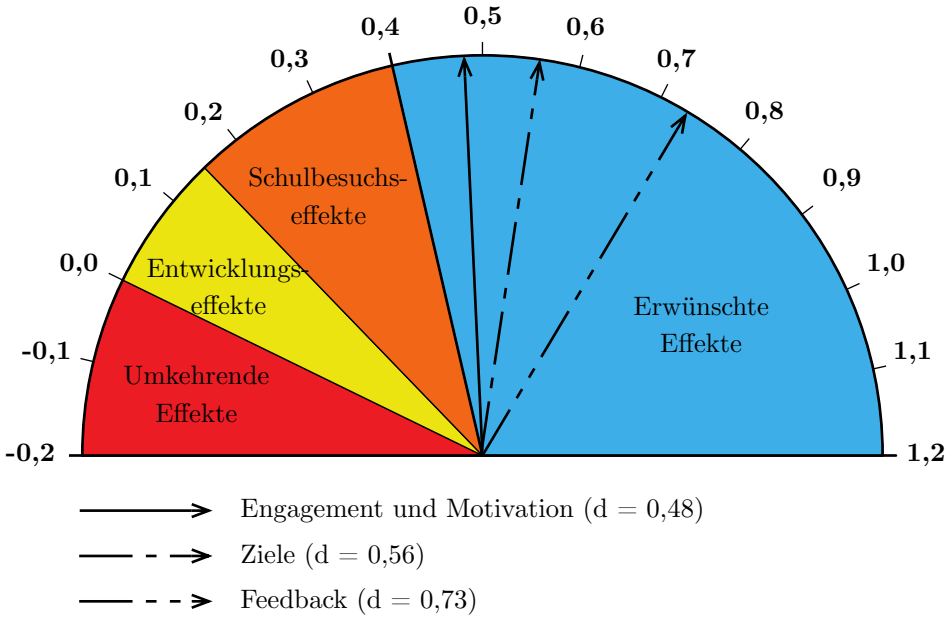

Abbildung 2.1: Das „Hattie-Barometer“ in Anlehnung an Hattie [18] und Zierer [46] und Einordnung der Faktoren Ziele, Feedback sowie Engagement und Motivation.

Bei ausreichend kleinen Lerngruppen bzw. bei Lehrveranstaltungen, deren Durchführung dem Unterricht im Klassenverband ähnelt, lassen sich die Ansätze des *Visible Learning* jedoch auch auf die Lehre an der Universität übertragen.

Wesentliche Faktoren des *Visible Learning*, die hohen Einfluss auf die *Learning Outcomes* der Studierenden hervorrufen, sind unter anderem die in Abbildung 2.1 dargestellten Faktoren Ziele ($d = 0{,}56$), Feedback ($d = 0{,}73$) zwischen Lehrenden und Studierenden sowie Engagement und Motivation ($d = 0{,}48$) der beteiligten Menschen. Das *Visible Learning* und die damit verbundene Lerneffektivität entsteht somit nach Hattie [18] in einer Lehrveranstaltung, wenn

- das Lernen das explizite Ziel ist,
- es ausreichend anspruchsvoll ist,
- sowohl die Lehrperson als auch die Lernenden das Erreichen der Ziele überprüfen,
- das Lernen wohlbedacht zu dem Zweck erfolgt, Ziele zu erreichen,
- Feedback gegeben und eingefordert wird und
- am Lernprozess aktive, leidenschaftliche und engagierte Menschen beteiligt sind.

Die 138 Einflussfaktoren ordnet Hattie [18] sechs verschiedenen Domänen zu. Sie lassen sich dadurch in Einflussfaktoren kategorisieren, die den Lernenden, das Elternhaus des Lernenden, die Schule (hier die Universität als Bildungseinrichtung), das Curriculum und das Unterrichten betreffen. Im Rahmen dieser Arbeit werden vorwiegend die Faktoren des Unterrichtens, bzw. der Unterrichtsansätze betrachtet. Abbildung 2.2 zeigt einen tabellarischen Auszug einiger wesentlicher Faktoren zusammen mit ihren ermittelten Effektstärken. Dabei handelt es sich um Faktoren, die in dieser Arbeit für die Gestaltung des agilen Lehrkonzepts von besonderer Bedeutung sind. Die gezeigten Einflussfaktoren werden nachfolgend näher erläutert und ihre Wirkung auf den Lernprozess verdeutlicht.

Ziele

Wie Locke et al. [22] beschreiben, werden menschliche Handlungen von bewussten Zielen und Absichten angetrieben. Daraus leitet sich die entscheidende Rolle von Zielen für

Faktor	**Effektstärke *d* [-]**
Lernende	
Konzentration, Ausdauer und Engagement	0,48
Motivation	0,48
Unterricht	
Formative Evaluation des Unterrichts	0,90
Feedback	0,73
Ziele	0,56
Peer-Tutoring	0,55

Abbildung 2.2: Auszug einiger Faktoren aus *Visible Learning* [18] und ihre Effektstärke in Anlehung an Zierer [46].

die Lernleistung ab. Nach Hattie [18] sind Ziele anspruchsvoll in Bezug auf die aktuellen Kompetenzen der Lernenden zu formulieren, da sie eine deutlichere Vorstellung von Erfolg geben. Durch anspruchsvolle Ziele lassen sich so die Aufmerksamkeit der Studierenden auf die relevanten Verhaltensweisen und Ergebnisse lenken. Weiterhin spornen anspruchsvolle Ziele an, die schwierigen Erfolgskriterien zu erreichen und wirken sich deshalb motivierend auf Lernende aus. Zusätzlich erhöht die Selbstverpflichtung (*Commitment*) der Studierenden gegenüber gesetzten Zielen das Engagement und fördert weiterhin die Lerneffektivität. Zwar steigt nach Hattie [18] die Lernleistung linear mit dem Schwierigkeitsgrad der Ziele an, dennoch ist bei der Formulierung der Erfolgskriterien zu berücksichtigen, dass sie im Rahmen des Erreichbaren bleiben.

Das Besprechen der Ziele mit den Studierenden sorgt ebenfalls für eine Verbesserung des Lernprozesses, da sie die zu erwerbenden Kompetenzen vor Augen führen und transparente Lernintentionen darstellen. Darüber hinaus lassen sich durch präzise formulierte Ziele die Lernfortschritte der Studierenden messen. Infolgedessen bieten Ziele eine Möglichkeit der Selbsteinschätzung für Studierende. Sie fördern somit die Sichtbarkeit des Lehrens und Lernens und sind deshalb wesentlicher Bestandteil von *Visible Learning*.

Feedback und Formative Evaluation des Unterrichts

Nicht nur das an Studierende gerichtete Feedback, das z.B. aus der zuvor beschriebenen Selbsteinschätzung durch Ziele entsteht, unterstützt das *Visible Learning*. Ebenfalls fördern Rückmeldungen an die Lehrperson das „Sichtbarmachen“ der Lehr- und Lernprozesse. So informiert beispielsweise die Erfassung der Lernfortschritte den Lehrenden über seinen Erfolg bzw. Misserfolg während der Lehrveranstaltung [18, 20, 35]. Dieses und auch weiteres Feedback von Studierenden (ggf. durch Dialoge oder anderweitige Interaktionen) kann die Lehrperson zur Evaluation des Unterrichts verwenden. Dabei ist zu beachten, dass dem Feedback für die Lehrperson eine höhere Bedeutung zugeschrieben wird als dem Feedback für die Studierenden, da infolge der fortwährenden Evaluation die Möglichkeit besteht, das Konzept wie auch den Lernprozess kontinuierlich zu verbessern. Besonders effektiv wirkt sich die formative Evaluation aus, wenn die Fortschritte der Studierenden in grafischer Form dargestellt werden [18].

Nach Hattie [18] werden durch verstärktes Feedback die Perspektiven getauscht, sodass die Lehrperson in Bezug auf die Lehre selbst die Position des Lernenden einnimmt. In Kombination mit eindeutig und anspruchsvoll definierten Zielen führt das Feedback durch die gleichzeitig gebotene Möglichkeit der Selbsteinschätzung dazu, dass Lernende eigenständig und selbstorganisiert lernen und die Rolle ihrer eigenen Lehrperson einnehmen. Dieser Effekt ist wünschenswert, da er selbstregulierende Merkmale zeigt und die überfachlichen Kompetenzen der Studierenden fördert. Je häufiger und intensiver dieser Rollentausch in der Lehrveranstaltung stattfindet, desto größer sind die zu erwartenden *Learning Outcomes*.

Motivation und Engagement

Die Umsetzung des *Visible Learning* hängt von der Motivation und dem Engagement ab, das beteiligte Personen aufbringen, um sich auf das Geben und Einfordern von Feedback sowie den daraus resultierenden Rollentausch einzulassen. Um Studierende zu motivieren und ihr Engagement zu fördern, ist es sinnvoll, sie positiv gegenüber der Lehrveranstaltung und den zu vermittelnden Lehrinhalten einzustellen. Motivierte

und engagierte Studierende verpflichten sich eher den Lernintentionen, wodurch ihre *Learning Outcomes* gefördert werden. Motivation und Engagement lassen sich häufig durch ausreichendes Feedback, Bestätigung durch andere und dem Gefühl der Kontrolle über das eigene Lernen erhöhen [13, 18].

Peer-Tutoring

Ein Ziel des *Visible Learning* besteht darin, Lernende in die Position ihrer eigenen Lehrperson zu versetzen, da sich dadurch ein selbstregulierender Prozess ergibt und die Studierenden eigenständig und selbstorganisiert lernen. Mit der Rolle der Lehrperson überträgt sich ebenfalls die Kontrolle des Lernprozesses auf die Lernenden, was sich positiv auf die Motivation und das Engagement auswirkt. Eine Lehrstrategie, um diesen Rollentausch zu forcieren, bildet das *Peer-Tutoring*, bei dem sich *Peers* (dt: „Gleichartige, Kollegen“) – hier die Studierenden – gegenseitig unterrichten. Neben einer höheren Lerneffektivität werden durch diese kooperative Lehrstrategie ebenfalls soziale Kompetenzen gefördert [18]. Die Lehrperson nimmt während des *Peer-Tutorings* eine beratende Rolle ein und begleitet die Studierenden in ihrem selbstorganisierten Lernprozess.

2.1.3 Hochwertige Lehre an der TU Braunschweig

Einige der von Hattie [18] geschilderten Ansätze und Faktoren lassen sich ebenfalls in dem Diskussionpapier „Gute Lehre an der TU Braunschweig“ [7] wiederfinden, das die Technische Universität Braunschweig als Denkanstoß für die Gestaltung guter Lernangebote anbietet. Das Diskussionspapier definiert zehn Aspekte, die bei der Gestaltung „Guter Lehre“ zu berücksichtigen sind. Nachfolgend werden einige dieser Aspekte des Diskussionpapiers genannt und erläutert.

> *»Gute Lehre an der TU Braunschweig orientiert sich an engagierten Studierenden.«*

Dieser Aspekt sagt aus, dass die Gestaltung eines Lernangebots auf diejenigen Studierenden ausgerichtet werden soll, die engagiert und motiviert an der Lehrveranstaltung teil-

nehmen. Um diese Studierenden in ihrem selbstgesteuerten Lernprozess zu unterstützen, erfordert ein gutes Lernangebot regelmäßige Rückmeldung über die erworbenen Kompetenzen und bietet dadurch die Möglichkeit der Selbsteinschätzung. Des Weiteren ist den Studierenden die Möglichkeit einzuräumen, durch Rückmeldungen betreffend der Lehrveranstaltung zu der Weiterentwicklung der Lehre beizutragen.

> *»Gute Lehre an der TU Braunschweig orientiert sich an steter Reflexion und der Suche nach besseren Konzepten.«*

Dieser Aspekt thematisiert die kontinuierliche Evaluation der Lehrveranstaltung durch die Lehrperson sowie die andauernde Überarbeitung des Lehrkonzepts. Er fordert Lehrende auf, regelmäßige Rückmeldungen betreffend der Lehrveranstaltungen durch Studierende zu erheben und sie in das bestehende Konzept einzuarbeiten oder sogar innovative Ideen daraus zu schöpfen. Um die stete Reflexion zu garantieren, müssen feste Strukturen in dem Lehrkonzept verankert werden, die die regelmäßige Erhebung von Feedback vorsehen und somit eine qualitätssichernde Wirkung aufweisen.

> *»Gute Lehre an der TU Braunschweig orientiert sich an der Entwicklung von sowohl fachlichen als auch überfachlichen Kompetenzen.«*

Dieser Aspekt unterstreicht den Stellenwert der überfachlichen Kompetenzen in der Lehre. Er fordert auf, den überfachlichen Kompetenzen ihren Platz in dem Lehrkonzept einzuräumen, da die Studierenden erst durch sie in der Lage sind, die fachlichen Kompetenzen in der Praxis anzuwenden.

> *»Gute Lehre an der TU Braunschweig berücksichtigt die Arbeit im Team.«*

Dieser Aspekt geht auf die Planung eines Lernangebots ein, die bei der Gestaltung „Guter Lehre“ in Teamarbeit geschieht. Neben dem kooperativen Austausch zwischen Lehrenden sind ebenfalls Studierende in den Planungsprozess einzubinden.

2.1.4 Definition „Gute Lehre“

Die vorangegangenen Abschnitte schildern mehrere Ansätze für die Gestaltung eines guten Lernangebots. Neben der geforderten Lernergebnis- und der Kompetenzorientierung (siehe Abschnitt 2.1.1) werden dabei auch Hatties Ansichten des *Visible Learning* (siehe Abschnitt 2.1.2) sowie hochschuldidaktische Diskussionsgrundlagen der Technischen Universität Braunschweig (siehe Abschnitt 2.1.3) erläutert. Definition 2.1 kombiniert die wichtigsten Punkte dieser Ansätze und bildet die im Rahmen dieser Arbeit verwendete Grundlage für das Verständnis „Guter Lehre“.

Definition 2.1. „Gute Lehre“: Qualitativ hochwertige Lehre ergibt sich, wenn Lehre allem voran auf den Lernprozess der Studierenden ausgerichtet ist und Strukturen bereitstellt, um diesen fortwährend zu verbessern. Sie fokussiert zu vermittelnde Kompetenzen als *Learning Outcomes*, adressiert neben fachlichen Aspekten auch motivationale und soziale Ressourcen und fördert die Selbstorganisation der Studierenden. Sie ist transparent, indem sie die intendierten Lernintentionen offenlegt und den Lernenden die Erfolgskriterien nahebringt. Sie ist in Hatties Sinne sichtbar und bietet ausreichendes Feedback, sodass beide Seiten – Lehrende sowie Lernende – ihre eigenen Handlungen reflektieren können. „Gute Lehre“ erfolgt, wenn die Lehrperson aufmerksam das Lernen von engagierten und motivierten Studierenden verfolgt und in den richtigen Momenten absichtsvolle Maßnahmen ergreift, um das Lernen zu lenken. Darüber hinaus bindet „Gute Lehre“ die Studierenden in die Konzeption des Lernangebots ein.

2.2 Konzepte des Didaktischen Designs

Für die Gestaltung eines Lernangebots bedarf es eines gestalterischen Prozesses, der die Planung, Konzeption und Umsetzung eines Lehrkonzepts betrifft. Dieser Prozess kann nach Reinmann [32] als didaktisches Design bezeichnet werden. Gegenstand des didaktischen Designs sind alle Konzepte, Modelle sowie Theorien, die den Planungsprozess des Lernangebots unterstützen und bei der Begründung didaktischer Entscheidungen helfen [32]. Zunächst stellt der nachfolgende Abschnitt die in dieser Arbeit verwendeten

Konzepte des didaktischen Designs vor. Darunter befinden sich die Lernziele, die Lernzieltaxonomien und das *Constructive Alignment*. Anschließend werden Vorgehensmodelle und ihre Anwendbarkeit diskutiert.

2.2.1 Lernziele

Zu Beginn des didaktischen Designs werden Ziele formuliert, die wegweisend für die Ausrichtung des Lernangebots sind. Sie beeinflussen das Lehrkonzept sowohl inhaltlich als auch in Bezug auf die verwendeten Methoden bei der Durchführung der Lehrveranstaltung [32].

Die sogenannten Lernziele orientieren sich an den Kompetenzen (siehe Abschnitt 2.1.1), die Studierende erwerben sollen. Dabei ist zu beachten, dass sich Kompetenzen aus der Auseinandersetzung mit komplexen Problemen und der damit verbundenen Problemorientierung heraus entwickeln und nicht nur durch die Vermittlung des Fachwissens selbst entstehen [19, 42]. Deshalb muss Studierenden die Möglichkeit gegeben werden, durch Lernaktivitäten mit dem fachlichen Wissen ins Handeln zu kommen und Erfahrungen zu sammeln. Infolgedessen wird der Lernprozess gefördert. Diese entdecken-lassende [32] oder erforschen-lassende [1] Lehrstrategie unterstützt die Studierenden dabei, die Kompetenz der Anwendung zu erwerben. Auch die überfachlichen Qualifikationen, die sich unter anderem in den sozialen Kompetenzen wie z.B. Teamfähigkeit oder in Selbstkompetenzen wie selbstgesteuertes bzw. selbstorganisiertes Lernen äußern, werden dadurch positiv beeinflusst [1, 32].

Da sich Kompetenzen wie zuvor beschrieben infolge der aktiven Auseinandersetzung mit fachlichen Problemen und Anwendung des Wissens entwickeln, werden Lernziele als ausführbare Handlungen formuliert, die die zu erwerbenden Kompetenzen voraussetzen [35]. Erreichen die Studierenden in der Lehrveranstaltung die beschriebenen Lernziele, macht das auch den Erwerb der zugrunde liegenden Kompetenzen wahrscheinlich.

2.2.2 Lernzieltaxonomien

Aus den zuvor erläuterten Zusammenhängen geht hervor, dass sich Lernziele nicht nur auf den zu vermittelnden Wissensinhalt beziehen, sondern ebenfalls eine Handlung definieren, die Studierende mit dem jeweiligen Inhalt durchführen können soll. Lernziele bestehen daher aus einer Inhaltskomponente und einer Handlungskomponente [23]. Die Handlungskomponente gibt dabei die Qualität bzw. die Tiefe des erlangten Wissens an [32]. Mit steigender Qualität des Wissens erhöht sich ebenfalls der Schwierigkeits- oder Komplexitätsgrad der zugrunde liegenden Kompetenz. Dabei gilt es zu beachten, dass der Erwerb von Kompetenzen stufenweise erfolgt. Es ist nicht möglich, eine Kompetenz hoher Komplexitätsstufe zu erwerben, ohne zuvor die weniger komplexen Stufen zu verinnerlichen [44].

In der Regel werden Lernziele nach Schaper et al. [35] entlang von Lernzieltaxonomien formuliert, die die stufenweise Aneignung von Kompetenzen bereits berücksichtigen. Lernzieltaxonomien sind im Allgemeinen Klassifikationsschemata um Lernziele anhand ihrer Komplexität zu ordnen und ihnen somit betreffend ihrer Schwierigkeit eine sinnvolle Reihenfolge zu geben. Dadurch unterstützen Lehrzieltaxonomien die Lehrperson dabei, sich des Ziels des Lernangebots bewusst zu werden und zu evaluieren, welche Erwartungen realistisch bzw. unrealistisch sind [32].

Zur Klassifizierung der Lernziele wird in dieser Arbeit die Lernzieltaxonomie nach Anderson et al. [2] verwendet, die auf der Lernzieltaxonomie nach Bloom et al. [9] beruht. Während Bloom et al. in kognitive, affektive und psychomotorische Lernziele unterscheidet, beschränkt sich die Lernzieltaxonomie nach Anderson et al. lediglich auf die kognitive Dimension [32]. Da im Rahmen dieser Arbeit ein Lehrkonzept entsteht, das auf dem Fachwissen der Programmierung beruht und somit vorwiegend kognitive Prozesse anspricht, resultieren aus der Vernachlässigung der nicht-kognitiven Dimensionen keine gravierenden Folgen.

Darüber hinaus entwickelten Anderson et al. die kognitive Dimension im Vergleich zu Bloom et al. weiter. Anstatt wie Bloom et al. nur die Handlungskomponente anhand der Komplexität verschiedener kognitiver Prozesse zu klassifizieren, wird auch die Inhaltskomponente entsprechend unterschiedlicher Schwierigkeitsgrade betrachtet. Es entsteht

somit die in Abbildung 2.3 gezeigte zweidimensionale Matrix, in der beide Komponenten der Lernziele aufgeschlüsselt werden [35].

Die Spalten der Matrix entsprechen der Klassifikation des Lernziels hinsichtlich der Komplexität des kognitiven Prozesses. Dabei ist „Erinnern“ die einfachste Stufe, da sie das bloße Wiedergeben des Inhalts betrifft. Mit steigendem Schwierigkeitsgrad folgen danach „Verstehen“, „Anwenden“, „Analysieren“, „Bewerten“ und „Erschaffen“. Die Zeilen der Matrix unterteilen die Inhaltskomponente mit ansteigender Komplexität in „faktisch“, „konzeptionell“, „prozedural“ und „metakognitiv“. Ein Lernziel in der Zelle **A1** bezieht sich somit auf das bloße Erinnern von faktischem Wissen und bildet das einfachste Lernziel. Demgegenüber steht Zelle **D6** als schwierigstes Lernziel.

Wissens-dimension	Kognitive Prozessdimension					
	1 Erinnern	**2** Verstehen	**3** Anwenden	**4** Analysieren	**5** Bewerten	**6** Erschaffen
A faktisch						
B konzeptionell						
C prozedural						
D metakognitiv						

Abbildung 2.3: Lehrzieltaxonomie nach Anderson et al. [2].

Aus der Theorie des stufenartigen Kompetenzerwerbs geht hervor, dass für das Erreichen eines definierten Lernziels ebenfalls die Zellen links und/oder oberhalb der durch das Lernziel adressierten Zelle beherrscht werden müssen. Gibt das gewünschte Lernziel beispielsweise vor, konzeptionelles Wissen zu verstehen (Zelle **B2**), muss der Studierende zuvor in der Lage sein, sich sowohl an konzeptionelles Wissen zu erinnern (Zelle **B1**), als auch faktisches Wissen zu verstehen (Zelle **A2**). Um wiederum die Zellen **B1** und **A2** zu erreichen, muss zuvor die Kompetenzstufe **A1** erlangt werden, die das Erinnern

von faktischem Wissen beinhaltet. Liegen die Vorstufen der in dem Lernangebot angestrebten Kompetenz nicht in dem Kompetenzprofil der Studierenden vor – z.B. als Vorwissen durch frühere Lehrveranstaltungen – sind auch diese bei der Gestaltung des Lernangebots zu berücksichtigen.

2.2.3 Constructive Alignment

Die zuvor erläuterten Lernzieltaxonomien unterstützen die Lehrperson, die Ausrichtung des Lernangebots zu erarbeiten, die zu erwerbenden Kompetenzen festzulegen sowie die damit verbundenen Lernziele zu formulieren. Die Lernziele allein sind jedoch nicht ausreichend, um den Lernerfolg der Studierenden zu garantieren. Eher dienen sie nach Reinmann [32] als Grundlage für die weitere Planung des Lehrkonzepts.

Wie in Abschnitt 2.1.1 beschrieben entstehen Fachkompetenzen durch die problemorientierte Auseinandersetzung mit dem fachlichen Wissen. Demnach üben die verwendeten Lehrmethoden – respektive die ausgeführten Lernaktivitäten des Studierenden – erheblichen Einfluss auf den Lernerfolg aus. Geben die ausgewählten Lehrmethoden den Studierenden nicht die Möglichkeit, die in den Lernzielen geforderten Kompetenzen zu erwerben, entsteht eine negative Rückkopplung auf die Lernziele. Einen weiteren wichtigen Faktor für den Erfolg der Lehrveranstaltung bildet das gewählte Format der Prüfung, die das Erreichen der Lernziele überprüft. Nach Schaper et al. [35] richten sich Studierende bei der Durchführung des Lernangebots nach anderen Aspekten als die Lehrperson. Während sich Lehrende an den intendierten Lernergebnissen orientieren, stehen für Studierende oftmals die Prüfungsanforderungen im Vordergrund. Wird ein Prüfungsformat verwendet, das auf ein anderes Kompetenzniveau abzielt als die formulierten Lernziele, beeinflusst das ebenfalls den Lernerfolg der Studierenden.

Daraus ergeben sich die in Abbildung 2.4 dargestellten wechselwirkenden Beziehungen zwischen den Lernzielen, den Lehrmethoden und der Prüfung. Diese Beziehungen sind Gegenstand des *Constructive Alignment* nach Biggs [8], das besagt, die Lehrmethoden, die Prüfungsaufgaben und die Lernziele aufeinander abzustimmen, um sie so in einen Gesamtzusammenhang zu bringen [4]. Infolge dieser Abstimmung entsteht eine konsistente Lernumgebung, die die Bedingungen für den Lernerfolg verbessern [8].

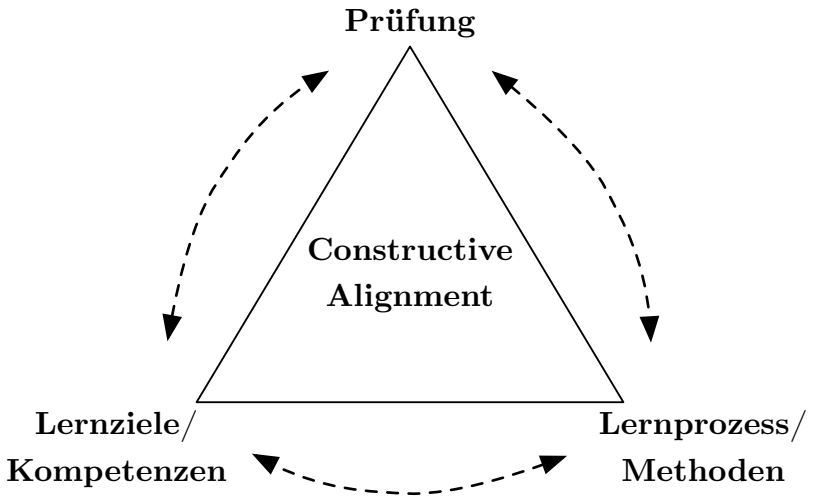

Abbildung 2.4: Beziehungen des *Constructive Alignment* in Anlehnung an Wildt et al. [44] und Baumert et al. [4].

Zur Umsetzung des *Constructive Alignments* bei der Gestaltung eines Lernangebots kann der TAMAS[2]-Ansatz der Universität Zürich herangezogen werden [3, 35]. Er basiert auf der in Abschnitt 2.2.2 beschriebenen Lehrzieltaxonomie nach Anderson et al. [2]. Grundgedanke dieses Ansatzes ist es, das Lernziel (LZ), die Lernaktivität (LA) und die Lernzielüberprüfung (LÜ) in die zweidimensionale Matrix einzuordnen und somit die Ausrichtung aufeinander zu überprüfen. Eine optimale Konzeption sieht dabei vor, dass sich alle drei Komponenten in derselben Zelle befinden. Kleinere Abweichungen, wie z.B. Einträge in benachbarten Zellen, sind dabei nach Schaper et al. [35] zu vertreten.

2.2.4 Vergehensmodelle

Wie in jeder gestalterischen Disziplin geht der Umsetzung eines Projekts stets ein Planungsprozess voraus, der durch eine Zielsetzung gelenkt wird. Im Anschluss an die Umsetzung dient die Zielsetzung als Kontrollwerkzeug, um den Erfolg der Umsetzung zu überprüfen [32]. Betrachtet man das umzusetzende Projekt als zu lösendes Problem, ergibt sich ein lineares Vorgehensmodell nach Abbildung 2.5.

[2]Taxonomie-Matrix zur Analyse und Selbstevaluation von Hochschullehre

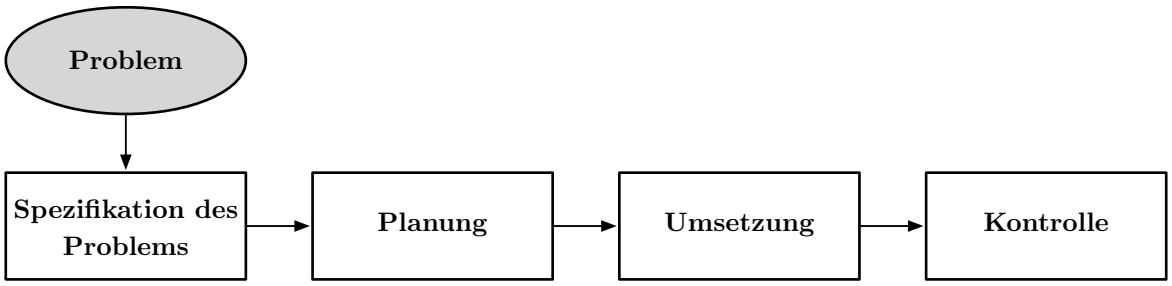

Abbildung 2.5: Lineares Vorgehensmodell in nach Richter et al. [33].

Das lineare Vorgehensmodell sieht vor, das Problem zu spezifizieren, einen Plan zu dessen Lösung zu entwerfen, diesen umzusetzen und anschließend das Ergebnis in Bezug auf seine Zielsetzung zu überprüfen. Für dieses Vorgehen müssen Ausgangs- und Zielzustand sowie nötige Lösungsschritte und -verfahren mit vorhersehbaren Ergebnissen eindeutig definiert sein. Bei diesen eindeutig spezifizierbaren Problemen handelt es sich um kontextunabhängige Probleme, die nicht von Merkmalen der realen Umwelt abhängen. Beispiele für kontextunabhängige Probleme sind nach Richter et al. [33] die Türme von Hanoi und das Acht-Damen-Problem. Beide Probleme sind durch feste Regeln und Rahmenbedingungen eingegrenzt, zu deren Lösung transparente Operationen mit vorhersehbaren Resultaten angewendet werden.

Weder Ausgangs- noch Zielzustand lassen sich in der Didaktik eindeutig bestimmen, da sie unter anderem von den Merkmalen der Lernenden abhängen (Faktoren des Lernenden, siehe Abschnitt 2.1.2). Ebenfalls existiert in der Didaktik nicht die eine Methodik, die während des Lehrens und Lernens vorhersehbare Ergebnisse hervorruft. Der Erfolg einer Lehrmethode wird vielmehr dadurch bestimmt, ob sie von den Lernenden angenommen und durchgeführt wird. Auch der soziale Hintergrund und das Vorwissen der Lernenden spielen dabei eine wichtige Rolle. Da didaktische Problemstellungen von Merkmalen der realen Umwelt abhängen, handelt es sich somit um inhärent kontextabhängige Probleme [32, 33].

Die Lösung kontextabhängiger Probleme bedarf ein iteratives Vorgehen, bei dem die Gestaltung des Projekts bzw. des Lernangebots schrittweise erfolgt. Anstatt den gestalterischen Prozess wie in dem linearen Vorgehensmodell nach der Kontrolle zu beenden, müssen für den Entwurf eines Lehrkonzepts die Ergebnisse der Evaluation zurück in

den Planungsprozess fließen, um das Lernangebot weiterhin zu verbessern. Dieses Vorgehen greift die formative Evaluation des Unterrichts nach Hattie [18] (siehe Abschnitt 2.1.2) und die von der Technischen Universität Braunschweig beschriebene stetige Reflexion des Lehrkonzepts auf (siehe Abschnitt 2.1.3). Vorgehensmodelle, die eine iterative Lösung des Problems vorsehen, sind beispielsweise das spiralförmige oder das zyklische Vorgehensmodell [32, 33].

KAPITEL 3

Analyse des bestehenden Lehrkonzepts

Angestoßen durch die Forderung nach kontinuierlicher formativer Evaluation des Unterrichts (siehe Abschnitt 2.1.2) und steter Reflexion des Lehrkonzepts (siehe Abschnitt 2.1.4) sowie durch die iterative bzw. zyklische Vorgehensweise bei der Gestaltung didaktischer Problemstellungen (siehe Abschnitt 2.2.4) geht dem Entwurf des agilen Lehrkonzepts (siehe Kapitel 4) die Analyse einer bestehenden Lehrveranstaltung voraus. Das zukünftige Lehrkonzept befindet sich dadurch in einer höheren Iterationsstufe und profitiert von bereits gewonnenen Erfahrungen eines derzeitigen Lernangebots. Basierend auf den in Kapitel 2 erläuterten didaktischen Grundlagen analysiert das nachfolgende Kapitel die Lehrveranstaltung „Algorithmen und Programmieren“ des Instituts für rechnergestützte Modellierung im Bauingenieurwesen (iRMB) der Technischen Universität Braunschweig. Angeboten wird die Lehrveranstaltung in den Masterstudiengängen Bauingenieurwesen und *Computational Sciences in Engineering* (CSE). Wie das zu entwerfende agile Lehrkonzept (siehe Kapitel 5) adressiert auch die betrachtete Lehrveranstaltung die Vermittlung von Kompetenzen im Fachbereich der Programmierung.

Zu Beginn der Analyse wird zunächst die derzeitige didaktische Konzeption der zuvor genannten Lehrveranstaltung dargelegt. Weiterhin werden einige Fehlannahmen und

-konzeptionen geschildert, die der didaktischen Konzeption zugrunde liegen. Diese Unzulänglichkeiten mindern die Qualität des bestehenden Lehrkonzepts, da sie seine Umsetzung – insbesondere die der Methodik – erschweren. Daraus werden anschließend Verbesserungsansätze abgeleitet, die es in der Gestaltung des zukünftigen agilen Lehrkonzepts zu berücksichtigen gilt.

3.1 Derzeitige didaktische Konzeption

Das Konzept eines bestehenden Lernangebots lässt sich anhand der Gestaltung sowie der gegenseitigen Abstimmung seiner didaktischen Elemente veranschaulichen. Zu diesen Elementen zählen unter anderem die Lernziele, die Lernaktivitäten sowie die Gestaltung der Prüfung (siehe Abschnitt 2.2.3).

Zur Veranschaulichung des bestehenden Lehrkonzepts der Veranstaltung „Algorithmen und Programmieren“ nutzt der nachfolgende Abschnitt die in Abschnitt 2.1 beschriebenen Konzepte des didaktischen Designs. Ausgehend von den Qualifikationszielen der Modulhandbücher [14, 15] werden in einem ersten Schritt die intendierten Lernziele formuliert und ihre Kompetenzstufen anhand der Lernzieltaxonomie nach Anderson et al. [2] ermittelt. Anschließend wird die geplante Methodik beschrieben, die verwendet werden soll, um die formulierten Lernziele zu erreichen. Dabei wird zum einen die geplante Methodik der Übungsveranstaltung erläutert, zum anderen die Art der Prüfungsaufgaben diskutiert. Ein weiterer Schritt überprüft das *Constructive Alignment* der Konzeption anhand des TAMAS-Ansatzes. Letztlich wird das bestehende Lehrkonzept in Bezug auf die in Abschnitt 2.1 erläuterten Ansätze und Ansichten qualitativ hochwertiger Lehre bewertet.

3.1.1 Lernziele

Den Modulhandbüchern der beiden Studiengänge ist zu entnehmen, dass die Veranstaltung als Ziel die Vermittlung der Kompetenz verfolgt, die grundlegenden Prinzipien und Konzepte der objektorientierten Programmierung anzuwenden, um mäßig komplexe

Simulationsaufgaben zu lösen, Algorithmen umzusetzen und im Hinblick auf ihre Effizienz zu spezifizieren sowie die wichtigsten Datenstrukturen zu implementieren. Dieses Qualifikationsziel beinhaltet mehrere Einzelziele, die auf der Anwendung der objektorientierten Programmierung bzw. der Programmierung im Allgemeinen beruhen. Das wichtigste Lernziel dieser Lehrveranstaltung fokussiert deshalb die Kompetenz des Programmierens selbst und lässt sich wie folgt formulieren:

> **Lernziel 1.** Studierende beherrschen und verinnerlichen die Grundlagen der Programmierung sowie die Konzepte und Prinzipien der objektorientierten Programmierung, sodass diese routiniert für die Lösung von mäßig komplexen Ingenieurproblemen angewendet werden können.

Wie beispielsweise bei der Formulierung wissenschaftlicher Arbeiten, bei denen die Sprache als Mittel verwendet wird, um Ergebnisse und Ansichten zu schildern, handelt es sich bei der Programmierung um das Instrumentarium, um Vorgänge und Prozesse zu strukturieren und zu modellieren. Fehlen dem Autor Kenntnisse und Routine in der verwendeten Sprache bezüglich des Vokabulars, der Grammatik oder der Stilmittel, gefährdet das den Schreibfluss, sodass sich der Autor neben dem fachlichen Anspruch ebenfalls mit den Herausforderungen der Sprache auseinandersetzen muss. Um sich vollständig auf das zu implementierende Problem konzentrieren zu können, bedarf es deshalb einer hohen Routine unter anderem in Bezug auf die Schlüsselwörter der verwendeten Sprache (vgl. Vokabular), die Syntax (vgl. Grammatik) sowie auf die Konzepte und Prinzipien der objektorientierten Programmierung (vgl. Stilmittel). Das wesentliche Ziel dieser Veranstaltung beruht darauf, den Studierenden diese Routine in der Programmierung zu vermitteln, sodass sie während der Umsetzung immer noch über ausreichend kognitive Ressourcen für die zu implementierende Problemdomäne verfügen. Im einzelnen bezieht sich dieses Lernziel unter anderem auf den Umgang mit Basisdatentypen, Variablen, Basisoperatoren, Ausdrücken, Anweisungsblöcken wie z.B. Verzweigungen oder Schleifen, auf die Umsetzung von Klassen und die Verwendung von Objekten inklusive ihrer Beziehungen sowie auf die rekursive Umsetzung von Prozeduren.

Neben dem Verstehen und Anwenden von Konzepten beinhaltet dieses Lernziel (LZ1) auch das Erlernen von Handlungsmustern und das Ausüben von Handlungsweisen und

ist deshalb in der Lernzieltaxonomie nach Anderson et al. [2] (siehe Abschnitt 2.2.2) als das Anwenden von prozeduralem Wissen (**C3**) einzuordnen.

Ein weiteres Lernziel befasst sich mit der Algorithmik, die nach Ottmann et al. [28] ein zentrales Gebiet der Informatik darstellt. Bei Algorithmen handelt es sich um schrittweise Lösungsverfahren für bestimmte Probleme, die mit Hilfe von Programmiersprachen implementiert und automatisiert werden können. Algorithmen lassen sich anhand ihrer Komplexität klassifizieren, die sich an der Anzahl der auszuführenden Schritte misst. Die Umsetzung effizienter Algorithmen sieht vor, die Anzahl der auszuführenden Schritte auf ein notwendiges Minimum zu reduzieren [28].

Zu lösende Probleme von Algorithmen sind beispielsweise das Sortieren und Suchen von Werten in einer listenartigen Ansammlung von Daten. Im Rahmen dieser Lehrveranstaltung werden einige bestimmte Algorithmen wie beispielsweise die einfachen Sortieralgorithmen *Bubble Sort*, *Selection Sort* und *Insertion Sort* sowie die rekursiven Algorithmen *Merge Sort* und *Quick Sort* implementiert. Weiterhin wird die Untersuchung der implementierten Algorithmen im Hinblick auf ihre Komplexität bzw. die Anzahl der auszuführenden Schritte thematisiert. Das entsprechende Lernziel formuliert sich wie nachfolgend gezeigt:

> **Lernziel 2.** Studierende beherrschen ausgewählte Algorithmen und können diese implementieren. Weiterhin sind Studierende in der Lage, beliebige Algorithmen in Bezug auf ihre Komplexität zu untersuchen und sie im Hinblick auf ihre Effizienz zu vergleichen.

Die Implementierung von Algorithmen erfordert es, das Verständnis über die Funktionsweise auf die Konzepte der Programmierung zu übertragen und so die Abläufe durch Programmstrukturen abzubilden. Dieses Lernziel geht somit über die kognitive Prozessdimension „Verstehen“ der Lernzieltaxonomie nach Anderson et al. [2] hinaus. Auch die Untersuchung von Algorithmen verlangt das Anwenden gelernter Konzepte zur Ermittlung der Anzahl an auszuführenden Schritten. Demnach ordnet sich das zweite Lernziel (LZ2) in der Zelle **B3** – konzeptionelles Wissen anwenden – ein.

Üblicherweise erfolgt während des Ablaufs der Algorithmen die Manipulation von Daten, die sich dafür je nach Algorithmus in einer geeigneten Datenstruktur befinden müssen. Nach Ottmann et al. [28] gehören deshalb Algorithmen und Datenstrukturen zusammen. Zu den wichtigsten Typen der Datenstrukturen zählen listenähnliche Strukturen wie beispielsweise *Arrays* und verkettete Listen, die eine eindimensionale Ansammlung von Daten ermöglichen. Da die Implementierung der Algorithmen aus LZ2 Kenntnisse über die Funktionsweise dieser Datenstrukturen erfordert, sind sie ebenfalls Teil des Lehrinhalts. Es ergibt sich somit ein weiteres Lernziel:

> **Lernziel 3.** Studierende beherrschen die Funktionsweise ausgewählter Datenstrukturen und können diese mit Hilfe der objektorientierten Programmierung implementieren.

Für die Implementierung der Datenstrukturen ist wie schon zuvor bei LZ2 die Übertragung der Funktionsweisen auf die Konzepte der Programmierung erforderlich. Das dritte Lernziel (LZ3) ordnet sich deshalb ebenfalls in die Zelle **B3** ein. Betrachtet man die drei Lernziele gemeinsam, ergibt sich eine Einordnung analog zu Abbildung 3.1.

Wissens-dimension	Kognitive Prozessdimension					
	1 Erinnern	**2** Verstehen	**3** Anwenden	**4** Analysieren	**5** Bewerten	**6** Erschaffen
A faktisch						
B konzeptionell			LZ2 LZ3			
C prozedural			LZ1			
D metakognitiv						

Abbildung 3.1: Klassifizierung der Lernziele LZ1 (Programmierung allgemein), LZ2 (Algorithmen) und LZ3 (Datenstrukturen) des bestehenden Konzepts, dargestellt in der Lehrzieltaxonomie nach Anderson et al. [2].

3.1.2 Methodik

Die Lehrveranstaltung unterteilt sich wie konventionell üblich in Vorlesung und zentrale Übung. Während der Vorlesung werden die Lehrinhalte in vorwiegend darbietender Weise vermittelt. Dabei findet auch eine gewisse Interaktion mit den Studierenden statt. Beispielsweise gibt die Lehrperson praxisnahe Beispiele, stellt Fragen an das Publikum oder es werden gemeinsam kurze Gedankengänge nachvollzogen. Somit bindet die Lehrperson Studierende in das Geschehen der Lehrveranstaltung aktiv ein und regt sie zu einer Auseinandersetzung mit den zu vermittelnden Lehrinhalten an. Des Weiteren sorgt das für eine Entschleunigung der Vorlesung, sodass Studierende dem Lehrinhalt besser folgen können. Durch die vorwiegend darbietende Lehrweise entsteht wie in Abschnitt 2.1.1 beschrieben ein rezeptiver Lernprozess, der nicht die Kompetenz der Anwendung vermittelt. Betrachtet man die Vorlesung isoliert, befindet sie sich daher maximal auf der Kompetenzstufe der Zelle **B2** – konzeptionelles Wissen verstehen. Sie bereitet die Studierenden allerdings für den weiteren Kompetenzerwerb vor.

Um die Kompetenzen der Studierenden über dieses Niveau hinaus auf die von den Lernzielen angestrebten Stufen anzuheben, wird die Vorlesung von zentralen Übungen ergänzt. In diesen Übungen sollen Studierende in kleinen Gruppenverbänden (siehe *Peer-Tutoring*, Abschnitt 2.1.2) Übungsaufgaben lösen, um die Vorlesungsinhalte selbstorganisiert und kooperativ anzuwenden. Neben der Fachkompetenz werden dadurch auch überfachliche Kompetenzen wie z.B. die Selbstkompetenz und soziale Kompetenzen gefördert.

Für die Lösung der Übungsaufgaben steht den Studierenden jede Art von Hilfsmitteln zur Verfügung. Neben dem Internet als Informationsquelle, dienen weiterhin die Vorlesungsunterlagen sowie Studierende der gleichen oder anderer Gruppen als Unterstützung bereit. Darüber hinaus sind neben der Lehrperson weitere Dozierende während der Übung anwesend, deren Aufgabe in erster Linie darin besteht, Studierende bei der Lösung der Aufgaben zu beraten und den selbstgesteuerten Lernprozess zu lenken und zu fördern. Im Anschluss an die Bearbeitung der Aufgaben im Gruppenverband erarbeitet die Lehrperson zusammen mit den Studierenden eine mögliche Lösung. Dadurch erhalten Studierende Feedback über die Korrektheit ihrer eigenen Lösung und können somit ih-

ren Lernfortschritt verfolgen. Infolge der Interaktion erhält dabei auch die Lehrperson Rückmeldung über den Lernprozess der Studierenden.

Die Übungsaufgaben sind so ausgelegt, dass Studierende zu deren Lösung selbst programmieren müssen und verfolgen daher LZ1. Zu Beginn der Veranstaltungsreihe beinhaltet die Übung eher leichtere Aufgaben, die die technische Umsetzung der grundlegenden Programmierkenntnisse (Basisdatentypen, Variablen, Anweisungen, Klassen etc.; siehe Abschnitt 3.1.1) vermitteln. Im weiteren Verlauf der Veranstaltungsreihe steigert sich die Komplexität der Aufgaben. So erfordern beispielsweise spätere Aufgabenstellungen die Implementierung von Suchalgorithmen oder Datenstrukturen und unterstützen somit neben LZ1 ebenfalls die Lernziele LZ2 und LZ3. Neben den durch LZ2 und LZ3 beschriebenen Problemdomänen sieht die Lehrveranstaltung weitere Übungsaufgaben vor, die den Studierenden einen Ausblick für die weitere Anwendung ihrer erworbenen Kenntnisse geben. Dabei handelt es sich um eine Taschenrechner-Anwendung mit grafischer Oberfläche sowie ein *TicTacToe*-Spiel, bei dem der Computer einen unbesiegbaren Gegner simuliert. Da diese Aufgaben nur dazu dienen, den Studierenden weiteren Anwendungsbezug zu ermöglichen, finden sich ihre Inhalte nicht in den formulierten Lernzielen wieder. Dennoch unterstützen sie bei dem Kompetenzerwerb des Programmierens (LZ1).

Die Studierenden schließen die Lehrveranstaltung mit einer schriftlichen Klausur ab, die vorwiegend Aufgaben in halb-offenem Format enthält. Dabei handelt es sich nach Reinmann [32] um Aufgaben, die mit einigen Stichworten beantwortet werden können. Die Prüfungsaufgaben fokussieren dabei hauptsächlich die Abfrage der grundlegenden Konzepte der objektorientierten Programmierung sowie die Funktionsweise der thematisierten Algorithmen und Datenstrukturen. Auch enthalten einige wenige Prüfungsaufgaben Ausschnitte von Quelltext, die Studierende während der Prüfung verstehen oder bearbeiten müssen. So sieht die Klausur auch vor, dass beispielsweise die Komplexität gegebener Algorithmen ermittelt oder kleinere Prozeduren programmiert (schriftlich auf Papier) werden.

3.1.3 Überprüfung des Constructive Alignments

Wie Abschnitt 2.2.3 erläutert, sind nach dem *Constructive Alignment* die Lernziele, die Lehrmethoden sowie die Prüfung aufeinander abzustimmen, sodass sich eine konsistente Ausrichtung auf eine gemeinsame Kompetenzstufe ergibt. Dadurch wird den Studierenden in Lernaktivitäten die Möglichkeit geboten, die von den Lernzielen geforderten Kompetenzen zu erwerben und diese in der Prüfung unter Beweis zu stellen. Überprüft man die konstruktive Abstimmung der drei didaktischen Elemente für das zuvor erläuterte Lehrkonzept anhand des TAMAS-Ansatzes (siehe Abschnitt 2.2.3), lassen sich die Lernaktivitäten (LA) sowie die Lernzielüberprüfungen (LÜ) aus Abschnitt 3.1.2 zu den entsprechenden Lernzielen (LZ) aus Abschnitt 3.1.1 analog zu Abbildung 3.2 einordnen.

Wissens-dimension	Kognitive Prozessdimension					
	1 Erinnern	**2** Verstehen	**3** Anwenden	**4** Analysieren	**5** Bewerten	**6** Erschaffen
A faktisch						
B konzeptionell		LÜ1 LÜ2 LÜ3	LZ2/LA2 LZ3/LA3			
C prozedural			LZ1/LA1			
D metakognitiv						

Abbildung 3.2: Überprüfung des *Constructive Alignments* des bestehenden Lehrkonzepts mit dem TAMAS-Ansatz.

Die Lernaktivitäten befinden sich dabei jeweils in der gleichen Zelle der Lernzieltaxonomie wie die entsprechenden intendierten Lernziele und sind somit vollständig aufeinander abgestimmt. Die Lernziele sowie die durchzuführende Methodik verfolgen demnach die gleichen Kompetenzstufen. Im Gegensatz dazu fokussiert die Prüfung wie in Abschnitt 3.1.2 beschrieben durch die Art der Aufgabenstellungen hauptsächlich die Abfrage von

konzeptionellem Wissen. Die Lernzielüberprüfungen befinden sich folglich in Zelle **B2**. Die Begründung dafür lässt sich in dem gewählten Prüfungsformat – einer schriftlichen Klausur – wiederfinden. Das erste Lernziel befasst sich damit, die Kompetenz des Programmierens zu vermitteln, sodass Studierende die Konzepte der Programmierung routiniert verwenden können, um verschiedene Probleme zu modellieren und zu implementieren. In der Klausur wird zwar versucht, die Kompetenz des Programmierens durch die handschriftliche Implementierung von Quelltext zu überprüfen, dies gibt jedoch keinen Aufschluss über die erlernten Handlungsmuster und Handlungsweisen in der realen Programmierumgebung. Daher ermöglicht das gewählte Prüfungsformat lediglich, das konzeptionelle Wissen der Programmierung anzuwenden (Zelle **B3**). Da diese Aufgaben (Bearbeiten von Quelltext) jedoch nur vereinzelt vorliegen und sich die Prüfung eher auf die Abfrage – also das „Erinnern“ und „Verstehen“ – von Konzepten konzentriert, ergibt sich ein durchschnittliches Kompetenzniveau von **B2**. Darüber hinaus überprüft die Klausur durch die geringe Anzahl von Programmieraufgaben ebenfalls nicht ausreichend die Übertragung der Funktionsweisen der Algorithmen und Datenstrukturen auf die Konzepte der Programmierung. Auch hier wird das angestrebte Kompetenzniveau von **B3** nicht erreicht.

3.1.4 Bewertung

Unter der Annahme von Schaper et al. [35], dass kleinere Abweichungen wie z.B. die Zuordnung von LZ, LA und LÜ in benachbarten Zellen zu vertreten sind, liegt in dem bestehenden Lehrkonzept der Veranstaltung „Algorithmen und Programmieren“ eine gute Abstimmung der didaktischen Elemente bezüglich des zweiten und dritten Lernziels vor. Da das erste Lernziel eine höhere Abweichung aufweist, ist eine Anpassung des Lehrkonzepts erforderlich. Dazu können entweder Lernziel und Lernzielaktivität auf das Niveau der Prüfungsaufgaben angepasst, oder das Format der Prüfung überarbeitet werden. Da die Modulhandbücher die in Abschnitt 3.1.1 erläuterten Lernziele festlegen und ihnen offiziellen Charakter verleihen, handelt es sich bei diesen Lernzielen um feste Bestandteile des Curriculums. Infolgedessen ergibt sich eine Veränderung des Prüfungsformats. Hierbei ist es nicht notwendig, die grundlegende Struktur der schriftlichen Klausur zu ersetzen, sondern lediglich durch weitere Formate zu ergänzen, die das Überprüfen der

Programmierkompetenz ermöglichen. Dazu bietet sich beispielsweise eine unbenotete Studienleistung an, im Rahmen derer Studierende Programmieraufgaben lösen und ihre Kompetenz darlegen. Eine weitere Möglichkeit bietet das Format der Klausur+, bei dem eine zu erbringende Studienleistung benotet und zu einem gewissen Prozentsatz in die Modulnote eingeht (siehe §9 Abs 1. in [31]).

Betrachtet man die Methodik des Lehrkonzepts (siehe Abschnitt 3.1.2) in Bezug auf die zuvor formulierte Definition „Guter Lehre“ (siehe Definition 2.1, Abschnitt 2.1.4) und die Effektstärken (d [-]) der verwendeten Hattie-Faktoren, lässt sich das bestehende Konzept positiv bewerten. Durch die Interaktion der Lehrperson mit den Studierenden in der Vorlesung sowie durch die Methodik in den Übungsveranstaltungen (*Peer-Tutoring* mit Lehrperson als Berater; $d = 0{,}55$) bietet das Konzept sowohl für Lehrende als auch für Studierende ausreichend Möglichkeiten, Feedback ($d = 0{,}93$) bereitzustellen und einzufordern. Beide Seiten der beteiligten Personen können somit den Lernfortschritt verfolgen und ihre Handlungen reflektieren und verbessern (formative Evaluation des Unterrichts; $d = 0{,}90$). Der Lernprozess wird damit wie von Hattie [18] gefordert „sichtbar“ und rückt in den Mittelpunkt des Lehrkonzepts. Weiterhin unterstützt die kooperative Bearbeitung der Programmieraufgaben in der zentralen Übung die Kompetenzorientierung und fördert auch die überfachlichen Kompetenzen der Studierenden. Darüber hinaus dienen die Übungsaufgaben als Zielvorgaben (Ziele; $d = 0{,}56$), die von den Studierenden erreicht werden sollen und sorgen damit für Transparenz.

Neben den in Abschnitt 2.1.2 erläuterten Faktoren berücksichtigt das Lehrkonzept weitere Hattie-Faktoren, die zu den erwünschten Effekten ($d > 0{,}4$) gehören oder sich im oberen Bereich der üblichen Schulbesuchseffekte ($d = 0{,}15 \ldots 0{,}4$) befinden. So wird den Studierenden durch die Programmieraufgaben in den zentralen Übungen „aktive Lernzeit“ ($d = 0{,}38$) ermöglicht. Durch den selbstorganisierten Lernprozess infolge des *Peer-Tutorings* findet des Weiteren „kooperatives Lernen“ ($d = 0{,}41$) statt, das „lautes Denken“ ($d = 0{,}64$) individuell oder im Gruppenverband unterstützt. Die Berücksichtigung der zuvor genannten Faktoren, wie auch das „Fragenstellen“ ($d = 0{,}64$) durch die Lehrperson in der Vorlesung, sind Bestandteil des Lehrkonzepts und sollen den Lernerfolg sowie die Lerneffektivität steigern.

3.2 Fehlannahmen und -konzeptionen

Ergänzt man das Prüfungsformat der Lehrveranstaltung wie in Abschnitt 3.1.4 beschrieben um die Überprüfung der Programmierkompetenz, ergibt sich ein Lehrkonzept, das gemessen an den in Kapitel 2 erläuterten Grundlagen der Didaktik eine konstistente Lernumgebung schafft und eine hohe Qualität aufweist. Es müsste demnach den Lernprozess der Studierenden unterstützen und ertragreiche Lernerfolge sowie eine hohe Lerneffektivität in Bezug auf die zuvor formulierten Lernziele (siehe Abschnitt 2.2.1) hervorrufen. Die Ergebnisse der zuvor erwähnten Prüfungsaufgaben, in denen Quelltext bearbeitet und verstanden werden muss, lassen jedoch darauf schließen, dass die Studierenden die Lernziele nicht vollständig erreichen (siehe Anhang A). Obwohl die Kompetenz des Programmierens als wichtigstes Lernziel in dem Curriculum der Lehrveranstaltung verankert ist und die geplanten Lernaktivitäten den Kompetenzerwerb ermöglichen (siehe Abschnitt 3.1.2), werden diese Aufgaben mehrheitlich falsch oder gar nicht beantwortet. Dabei befinden sich die Aufgaben wie in Abschnitt 3.1.3 beschrieben noch unter der angestrebten Kompetenzstufe des ersten Lernziels. Der Grund für diesen Misserfolg liegt in der zuvor genannten Kontextabhängigkeit didaktischer Problemstellungen (siehe Abschnitt 2.2.4). Der Erfolg einer Lehrveranstaltung hängt demnach unter anderem von den Merkmalen des Lernenden wie z.B. Vorwissen, Motivation und Engagement ab, die in der Gestaltung des Lernangebots zu berücksichtigen sind. Fehlannahmen bezüglich dieser Merkmale können zu der Konzeption einer Methodik führen, die Studierende nicht ausüben können oder wollen.

Auch in der Übung der betrachteten Lehrveranstaltung werden Unzulänglichkeiten des Konzepts deutlich, da die Studierenden nicht in der Lage sind, die Programmieraufgaben kollaborativ und eigenständig zu lösen. Der Versuch der Lehrperson, die Studierenden dennoch selbst zum Handeln zu bewegen und dabei unterstützend zu agieren, endet zumeist darin, die Programmieraufgaben gemeinsam im Plenum zu bearbeiten. Dabei obliegt die Aktivität des Programmierens der Lehrperson, während die Studierenden den Lösungsweg über Projektor und Leinwand verfolgen. Die geplante Kooperation, Interaktion und selbstgesteuerte Bearbeitung der Programmieraufgaben geht daher in eine darbietende Lehrweise über, sodass sich ein rezeptiver Lernprozess einstellt und der

Kompetenzerwerb der Anwendung vernachlässigt wird (siehe Abschnitt 2.1.1). Darüber hinaus verliert das Konzept gleichzeitig an Transparenz, da weder Lehrperson noch Studierende den Lernfortschritt verfolgen können. Auch entfällt die Möglichkeit, ausreichend Feedback zu erzeugen. Die Qualität des Lernangebots sinkt.

Im Gegensatz zu der geplanten Methodik des bestehenden Lehrkonzepts basiert die tatsächliche Methodik in der Übung ähnlich wie die Vorlesung auf Frontalunterricht mit einiger Interaktion zwischen der Lehrperson und den Studierenden. Wie die Vorlesung erreicht deswegen auch die Übung eine maximale Kompetenzstufe von **B2** (siehe Abschnitt 3.1.2). Da auch das bestehende Prüfungsformat nur diese Kompetenzstufe adressiert (siehe Abschnitt 3.1.3), absolvieren dennoch viele Studierende das Modul mit guten bis sehr guten Noten, ohne die Kompetenz des Programmierens zu beherrschen.

Um die Qualität der Lehrveranstaltung zu erhöhen, die Lerneffektivität zu steigern und die Ausrichtung des Lernangebots auf die angestrebten Lernziele zu lenken, sind die Fehlannahmen zu identifizieren und die Methodik zu überarbeiten. Eine der grundlegenden Annahmen des derzeitigen Lehrkonzepts besteht darin, dass die Studierenden eine ausreichende Homogenität aufweisen, die die kooperative und interaktive Zusammenarbeit begünstigt. Die Veranstaltung „Algorithmen und Programmieren“ wird für Studierende der Masterstudiengänge Bauingenieurwesen und CSE angeboten. Während es sich bei Bauingenieurwesen um einen deutschsprachigen Studiengang handelt, sind in dem englischsprachigen Studiengang CSE vorwiegend internationale Studierende anzutreffen. Neben sprachlichen und kulturellen Unterschieden ergibt sich deshalb zusätzlich eine motivationale Heterogenität, die sich in den verschiedenen Zielen der Studiengänge begründet. Eine weitere Disparität entsteht aus dem individuellen universitären und schulischen Hintergrund der Studierenden, der sich in stark diversifiziertem Vorwissen und Erfahrungsreichtum äußert. Zwar setzt das *Peer-Tutoring* einen Wissensunterschied der Studierenden voraus, ist dieser jedoch so groß, dass er die kollaborative und interaktive Zusammenarbeit erschwert, stellen die Effekte des *Peer-Tutorings* nicht in dem erwarteten Ausmaß ein. Zusammen mit den kulturellen, sprachlichen und motivationalen Barrieren wird daher der Gesamtfortschritt wird der Gruppe eingedämmt.

Des Weiteren ruft die geplante Methodik des bestehenden Lehrkonzepts eine hohe kognitive Belastung der Studierenden hervor. Sie erfordert von den Studierenden zeitgleich, die umzusetzende Logik (beispielsweise der Datenstrukturen oder Algorithmen) zu verstehen, sie in Programmstrukturen zu übertragen, in Quelltext umzusetzen und in der Gruppe zu kommunizieren. Studierende sind daher in fachlichen und sozialen Disziplinen zugleich gefordert. Da die Studierenden die dafür erforderlichen Kompetenzen erst im Laufe der Lehrveranstaltung erwerben, sie also noch nicht routiniert beherrschen, trägt jede dieser Disziplinen zur kognitiven Auslastung bei. Abbildung 3.3 zeigt die vier unterschiedlichen Prozesse, die Studierende während der Bearbeitung der Aufgabenstellungen zugleich bewältigen müssen. Erst nach der Bearbeitung der gesamten Aufgabe erfolgt das Besprechen der Lösung im Plenum. Die lange Zeitspanne von Bearbeitungsbeginn bis zur Validierung des Quelltexts und Feedback über die umgesetzte Lösung zwingt die Studierenden dazu, die Aufgaben als monolithisches Problem zu betrachten, das im Gesamten gelöst werden muss. Im Zusammenhang mit umfangreichen und komplexen Aufgabenstellungen steigt dadurch die momentan betrachtete Komplexität, die Studierende in einem sehr weit gespannten Arbeitsschritt (sowohl zeitlich als auch inhaltlich) beherrschen müssen.

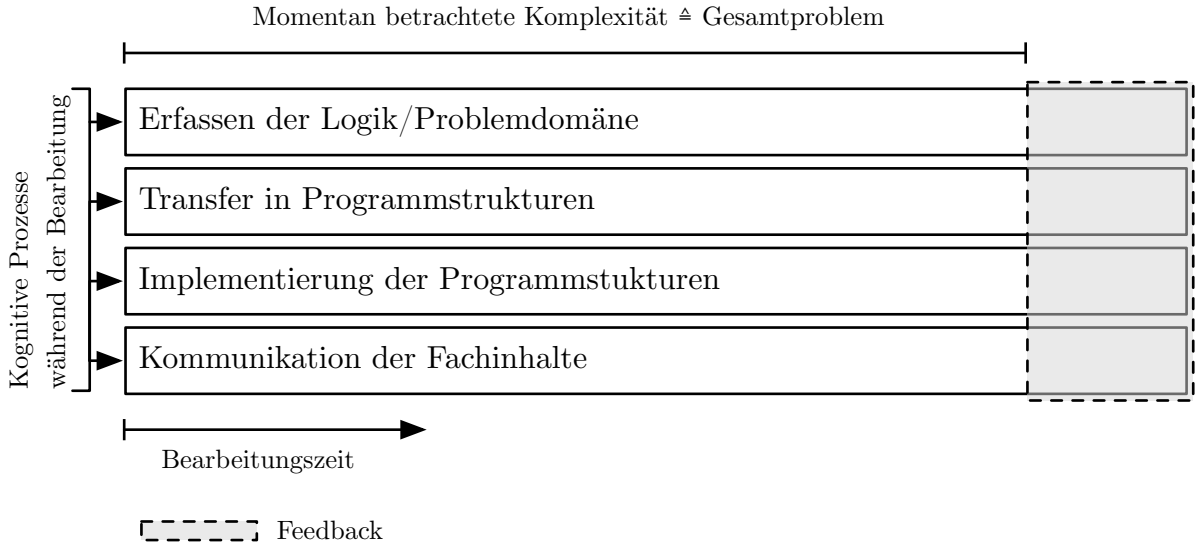

Abbildung 3.3: Zeitliche Abfolge der Prozesse während der Bearbeitung der Aufgabenstellungen in dem bestehenden Lehrkonzept.

Zusammen mit der Auslastung infolge der zeitgleichen Bewältigung der vier geforderten Prozesse entsteht durch das Ausmaß der momentan betrachteten Komplexität schnell eine Überforderung der Studierenden. Insbesondere in der zweiten Semesterhälfte, in der die Programmieraufgaben die komplexen Algorithmen und Datenstrukturen betreffen, ist allein die zu implementierende Logik der Problemdomäne (das Gesamtproblem) ausreichend für eine vollständige kognitive Auslastung. Der Transfer der Logik in Programmstrukturen, das zeitgleiche Umsetzen dieser Strukturen in Quelltext sowie die soziale Interaktion und Kommunikation der Fachinhalte lösen deshalb die zuvor genannte Überforderung aus. Die ursprünglich als anspruchsvolle Ziele gedachten Aufgabenstellungen verlassen damit den Bereich des Erreichbaren und wirken wie in Abschnitt 2.1.2 erläutert demotivierend. Dadurch sinken Engagement und Motivation der Studierenden, wodurch sich die Bearbeitung der Übungsaufgaben erschwert.

3.3 Verbesserungsansätze

Als maßgebende Aspekte für den eingeschränkten Erfolg der Lehrveranstaltung nennt Abschnitt 3.2 die Heterogenität der Studierenden, die eine kooperative und interaktive Zusammenarbeit unterbindet, sowie die Konzeption der Aufgabenstellungen, die aufgrund ihrer Schwierigkeitsgrade eine Überforderung auslösen und die selbstgesteuerte Bearbeitung verhindern. Beide Aspekte gilt es, in der Erarbeitung eines neuen Lehrkonzepts bzw. einer neuen Methodik zu verbessern, um die darbietende Lehrweise in der Übung in die ursprünglich geplante Lehrweise zu überführen.

Da im Rahmen der betrachteten Lehrveranstaltung kein Einfluss auf den sprachlichen und kulturellen Hintergrund, die Erfahrungen aus früheren Lernangeboten sowie auf die individuellen Beweggründe der Studierenden genommen werden kann, lässt sich die Heterogenität der Studierendengruppe nicht beseitigen. Es müssen Methoden angewendet werden, die trotz dieser Disparitäten die gewünschte Kooperation und Interaktion der Studierenden hervorrufen. Zu diesem Zweck könnte ein formeller Rahmen gestaltet werden, der Verhaltensregeln formuliert, um die Interaktion der Studierenden anzuregen. Dieser Leitfaden ist dabei als Unterstützung bzw. Hilfsmittel zu betrachten, auf

den die Studierenden zurückgreifen können, wenn die selbstorganisierte Gruppenstruktur, Kooperation und Interaktion nicht zum Ziel führen bzw. versagen. Insbesondere zu Beginn der Gruppenarbeit kann dieser formelle Rahmen dafür sorgen, die Barrieren zwischen den Studierenden zu überwinden und das gemeinsame Bearbeiten der Aufgaben zu fördern.

Auch sollte das zu erarbeitende Lehrkonzept die Umgestaltung der Aufgabenstellungen adressieren. Anstatt die Studierenden mit Aufgaben zu konfrontieren, die aufgrund ihrer unerreichbaren Zielsetzung zur Demotivation führen, könnten die Aufgabenstellungen nach dem altbewährten Prinzip *Divide-and-Conquer*[1] (dt: „teile und herrsche“) in mehrere kleine Teilprobleme zergliedert werden. Daraus ergibt sich eine verbesserte zeitliche Abfolge der kognitiven Prozesse während der Bearbeitung der Aufgabenstellungen entsprechend Abbildung 3.4.

Durch die Zergliederung der Aufgabenstellung lenkt sich die Konzentration der Studierenden von dem Gesamtproblem auf die Bearbeitung der Teilprobleme. Infolgedessen

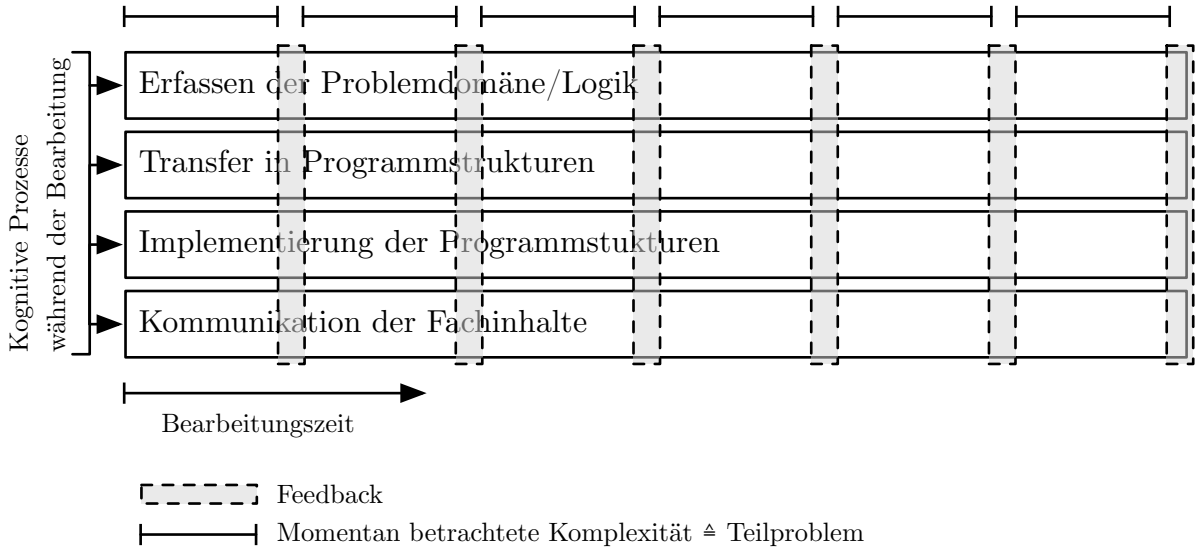

Abbildung 3.4: Verbesserungsansatz der zeitlichen Abfolge der Prozesse während der Bearbeitung der Aufgabenstellungen.

[1] Dem Prinzip Divide-and-Conquer liegt der Ansatz zugrunde, komplexe Aufgaben durch Dekomposition zu beherrschen. Dabei wird das Gesamtproblem in mehrere kleine Teilprobleme zergliedert, die für sich verstanden und umgesetzt werden können.

sinkt die momentan betrachtete Komplexität und es stehen mehr kognitive Ressourcen für die geforderten Prozesse zur Verfügung. Die Aufgabenstellungen rücken folglich zurück in den Bereich des Erreichbaren und definieren anspruchsvolle sowie motivierende Ziele. Darüber hinaus leitet kontinuierliches Feedback nach der Bearbeitung jeder Teilaufgabe die Studierenden durch die Übungsaufgaben und sorgt für Transparenz der Lernfortschritte. Durch die vielen kleinen Erfolgserlebnisse steigert sich ebenfalls das Selbstbewusstsein sowie das *Commitment* der Studierenden.

KAPITEL 4

Agile Methoden in der Softwareentwicklung

In der Programmierung bzw. in der Softwareentwicklung bewährten sich in der Vergangenheit die agilen Methoden für eine erfolgreiche und effiziente Umsetzung von Softwareprojekten. Wie die qualitativ hochwertige Lehre fokussieren auch die agilen Methoden das kooperative und selbstorganisierte Arbeiten, das sich durch soziale Interaktion, Transparenz und regelmäßiges Feedback auszeichnet. Die agilen Methoden weisen daher eine hohe Schnittmenge mit den Ansätzen „Guter Lehre“ auf und können daher im akademischen Bereich für die Lehre eingesetzt werden. Auch das in dieser Arbeit entstandene Lehrkonzept basiert auf der Anwendung der agilen Methoden der Softwareentwicklung. Das nachfolgende Kapitel erläutert die Grundzüge der agilen Softwareentwicklung und stellt die im Rahmen des zukünftigen Lehrkonzepts verwendeten agilen Methoden vor.

Zunächst wird dafür die sequentielle Vorgehensweise der traditionellen Softwareentwicklung dargelegt. Anschließend wird basierend auf dem Bedürfnis nach flexiblen und effizienten Entwicklungsprozessen die Notwendigkeit agiler Vorgehensmodelle verdeutlicht. Ein weiterer Abschnitt zeigt die grundlegenden Werte auf, die den agilen Methoden zugrunde liegen. Im weiteren Verlauf wird *Scrum* als agiles Vorgehensmodell vorgestellt.

Abschließend werden die im Rahmen dieser Arbeit verwendeten agilen Praktiken der Softwareentwicklung vermittelt.

4.1 Vorgehensweise traditioneller Softwareentwicklung

Der Entwicklungsprozess von Software unterliegt nach Martin et al. [24] einem kontinuierlichen Wandel der Anforderungen. Daher bedarf es eines flexiblen und anpassungsfähigen Prozesses für die Umsetzung von Softwareprojekten. Aufgrund falscher Metaphern zu anderen Disziplinen des Ingenieurwesens ergibt sich nach Poppendieck et al. [30] in der traditionellen Softwareentwicklung das Übertragen von Prozessen und Mechanismen, die nicht die Bedürfnisse der Softwareentwicklung bedienen.

Da die Effizienz der Softwareentwicklung aufgrund der geringen Materialkosten maßgebend durch die aufgebrachte Zeit der Entwickler beeinflusst wird, gilt es die in anderen Ingenieurdisziplin aufzufindende ganzheitliche Planung im Voraus zu vermeiden [30]. Dennoch findet sich sie in der traditionellen Softwareentwicklung das planungsorientierte Vorgehen anderer Ingenieurdisziplinen wieder [30]. Eines der bekanntesten Vorgehensmodelle ist das Wasserfall-Modell nach Royce [34], das die Softwareentwicklung als einen sequentiellen Prozess auffasst, in dem verschiedene Phasen nacheinander bearbeitet werden [34, 38, 39]. Dabei durchläuft der Prozess jede Phase nur einmal, Rückschritte zu vorherigen Phasen sind nicht vorgesehen. Es ergibt sich beispielsweise die in Abbildung 4.1 gezeigte Phaseneinteilung des Entwicklungsprozesses. Zunächst werden dabei die Anforderungen an die zu implementierende Software analysiert. Anschließend erfolgt eine ganzheitliche Planung (vgl. Planungsphase) des Produkts, in der Architektur- und Designentscheidungen getroffen werden. Ein nächster Schritt befasst sich mit der Implementierung der Software (vgl. Umsetzungsphase). Danach wird das entstandene Produkt auf seine Fehlerfreiheit überprüft und ausgeliefert [39].

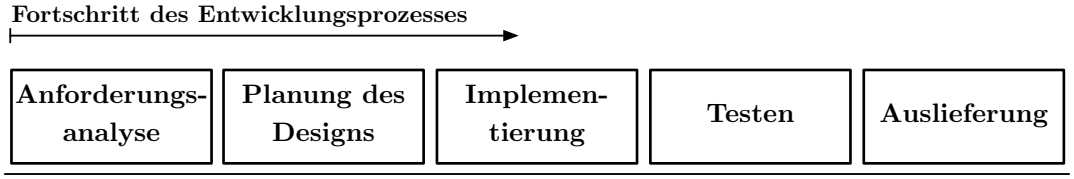

Abbildung 4.1: Sequentieller Prozess zur Entwicklung von Software nach dem Wasserfall-Modell in Anlehnung an Szalvay [39].

4.2 Notwendigkeit agiler Vorgehensweisen

Das Vorgehen der traditionellen Softwareentwicklung legt eine vollständige Analyse der Anforderungen zu Beginn der Entwicklung zugrunde, die jedoch aufgrund sich stetig ändernden Anforderungen nicht möglich ist. Die Unstetigkeiten entstehen beispielsweise durch sich fortwährend ändernde technische Anforderungen der zu implementierenden Software, die sich unter anderem aus Kundenwünschen, Marktänderungen oder dem Fortschritt verwendeter Technologien ergeben. Auch beeinflussen Veränderungen innerhalb des Entwicklerteams, wie etwa eine wechselnde Besetzung oder der Wissensgewinn im Verlauf der Produktion, die Umsetzung der Software [24]. Eine der größten Herausforderungen der Softwareentwicklung liegt deshalb darin, den Entwicklungsprozess so zu gestalten, dass er die Flexibilität in der Umsetzungsphase unterstützt und sich kontinuierlich an Veränderungen der Anforderungen anpasst [6, 24, 36].

Des Weiteren bietet die Entwicklung von Software nach Beck et al. [6] aufgrund ihrer inhärenten Komplexität [12] ein hohes Potenzial für Risiken, die den Entwurfsprozess erschweren. So können unter anderem Fehler in Planung oder Umsetzung auftreten, die bedingt durch das sequentielle Vorgehen erst spät in dem Entwurfszyklus bemerkt werden und daher eine Verzögerungen in der Auslieferung, einen erhöhten zeitlichen Ressourceneinsatz und die damit verbundene Kostensteigerung sowie eine geringe Qualität des Produkts hervorrufen [24, 38]. Um diese Risiken zu minimieren, beinhaltet die traditionelle Softwareentwicklung in der Planungsphase qualitätssichernde Vorgehensweisen und Mechanismen, die die Planungsfreiheit zur Risikovermeidung einschränken [24]. Wie die Vergangenheit zeigt, führt die Vielzahl an möglichen Risiken nach Martin et al. [24] dazu, dass Unternehmen in ihren Planungsprozess derart viele Vorgehensweisen

und Mechanismen integrieren (*Process Inflation*), dass sich ihr Entwurfsprozess deutlich verlangsamt. Infolgedessen erhöht sich einerseits der erforderliche Ressourceneinsatz in der Planungsphase, andererseits wird dadurch die geforderte Flexibilität verfehlt [36]. Stark einschränkende Vorgehensweisen schaden somit der Wettbewerbsfähigkeit der betroffenen Unternehmen [30, 24].

Durch den kontinuierlichen Wandel der Entwicklungsbedingungen handelt es sich bei der Softwareentwicklung wie bei didaktischen Problemstellungen um kontextabhängige Probleme, in denen Erkenntnisse als Feedback in vorherige Phasen zurückfließen müssen [36]. Takeuchi et al. [40] beschreiben diesen Ansatz in Bezug auf die allgemeine Produktentwicklung. Anstatt den Prozess zu sequenzieren und die Phasen getrennt nacheinander auszuüben, sollte die Vorgehensweise erlauben, flexibel zwischen den Phasen zu springen. In der Softwareentwicklung ermöglicht dieser Ansatz, flexibel auf die sich ändernden Entwicklungsbedingungen einzugehen, Unzulänglichkeiten in der Planung unverzüglich aufzudecken sowie Fehler in der Umsetzung zu beseitigen. Dadurch reduziert sich die Planungszeit des Entwicklerteams sowie die Notwendigkeit für qualitätssichernde Mechanismen in der Planung. Eine effizientere Vorgehensweise ergibt sich daher im Vergleich zur traditionellen Softwareentwicklung aus leichtgewichtigen und wenig einschränkenden Prozessen, die jedoch eine hohe Transparenz und ausreichend Feedback aufweisen, um die Qualität des Produkts trotz ihrer geringen Einschränkungen zu gewähren.

> *»Process and technology are a second-order effect on the outcome of a project. The first-order effect is the people.«*
> Alistair Cockburn [24]

Wie Brooks [11] in seinem Artikel *»No Silver Bullet«* erläutert, dass es keine absolute Vorgehensweise für hochwertiges Design (flexibel und wartbar) des Quelltexts gibt, existiert nach Poppendieck et al. [30] ebenfalls nicht die eine Methodik, die einen Entwicklungsprozess mit optimaler Anpassungsfähigkeit und Effizienz hervorruft. Eine Begründung dafür liefern nach Martin et al. [24] die individuellen Unterschiede in der Zusammensetzung verschiedener Entwicklerteams. Wie Alistair Cockburn vertritt auch Erich Gamma in seinem Vorwort zu *Agile principles, patterns, and practices in C#* [24] die Meinung, dass vorgegebene Prozesse bei der erfolgreichen Umsetzung von Softwa-

reprojekten nur eine untergeordnete Rolle spielen. So können festgelegte Vorgehensweisen, Praktiken und Werkzeuge den Entwurfsprozess unterstützen. Für den Erfolg eines Projekts müssen sie dennoch von den Teammitgliedern angenommen und ausgeführt werden. Ein größerer Einfluss auf den Erfolg ergibt sich daher im Vergleich zu festgelegten Prozessen aus den individuellen Entwicklern, die an dem Entwurfsprozess beteiligt sind. Daher führen gleiche Prozesse, die von verschiedenen Entwicklerteams ausgeführt werden, wahrscheinlich zu unterschiedlichen Ergebnissen hinsichtlich Effizienz und Erfolg des Projekts. Um Prozesse zu ermöglichen, die die individuellen Begebenheiten unterstützen, ist den Teams im Gegensatz zur traditionellen Softwareentwicklung die eigenständige Organisation ihrer Vorgehensweise zu überlassen [24].

Sowohl die geforderte Flexibilität des Prozesses, als auch die Selbstorganisation des Entwicklerteams gilt es für eine erfolgreiche und effiziente Umsetzung von Softwareprojekten zu adressieren. Einen Ansatz dafür bieten die agilen Vorgehensmodelle der Softwareentwicklung. Durch ihre iterativen, inkrementellen und wenig einschränkenden Vorgehensweisen [10, 21, 29], bieten sie ausreichend Transparenz und Feedback, um sich flexibel an veränderte Anforderungen anzupassen sowie Optimierungen des teaminternen Prozesses vorzunehmen. Nach Martin et al. [24] sind diese leichtgewichtigen Vorgehensmodelle daher der Schlüssel zu erfolgreichen und effizienten Softwareprojekten und verschaffen ihren Unternehmen einen entscheidenden Wettbewerbsvorteil.

4.3 Das Agile Manifest

Basierend auf der beobachteten *Process Inflation* sowie der Notwendigkeit der Selbstorganisation erarbeitet die *Agile Alliance* – eine Gruppe renommierter Softwareentwickler – im Jahr 2001 die grundlegenden *Values* (dt.: „Werte/Überzeugung“), die agile Vorgehensmodelle charakterisieren und auf die effiziente Implementierung von Software sowie die gleichzeitige Flexibilität des Entwicklungsprozesses abzielen [24]. Als Ergebnis entsteht das in Abbildung 4.2 dargestellte „Agile Manifest“ (*The Agile Manifesto*), dessen grundlegenden vier Werte nachfolgend näher erläutert werden.

Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen.

Durch diese Tätigkeit haben wir folgende Werte schätzen gelernt:

1. *Individuen und Interaktionen über Prozesse und Werkzeuge*
2. *Funktionierende Software über umfassende Dokumentation*
3. *Zusammenarbeit mit dem Kunden über Vertragsverhandlungen*
4. *Reagieren auf Veränderung über Befolgen eines Plans*

Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein.

Unterzeichnet von:

Kent Beck	James Greening	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

Abbildung 4.2: „Agiles Manifest“ (*The Manifesto of the Agile Alliance*) in Anlehnung an Martin et al. [24].

Individuen und Interaktionen über Prozesse und Werkzeuge

Wie in Abschnitt 4.2 beschrieben wird der Erfolg eines Softwareprojekts in hohem Maße von den beteiligten Entwicklern beeinflusst. So kann nach Martin et al. [24] beispielsweise ein effektiver Prozess nicht den Erfolg des Projekts sichern, wenn sich das ausführende Team aus Mitgliedern mit nur mittelmäßigen Fähigkeiten zusammensetzt. Im Gegensatz dazu sind Teams mit leistungsstarken Mitgliedern trotz eines ineffizienten Prozesses in der Lage dazu, die Umsetzung des Projekts – wenn auch wenig effizient – voranzutreiben. Dabei ist zu beachten, dass selbst eine Gruppe leistungsstarker Entwickler scheitert, wenn ihre Zusammenarbeit im Team nicht funktioniert. Die Stärke eines Entwicklers

misst sich deshalb nicht nur an dem Programmieren selbst, sondern an seiner Fähigkeit, kollaborativ und interaktiv zu arbeiten sowie im Team zu kommunizieren [24]. Weiterhin kann der Entwicklungsprozess durch unzählige Werkzeuge wie z.B. die integrierte Entwicklungsumgebung (*Integrated Development Environment*; IDE) oder Versionskontrollsysteme (*Version Control System*; VCS) unterstützt werden. Dennoch können zu viele oder im Verlgeich zu der Teamstruktur unpassende Werkzeuge die Interaktion der Teammitglieder stören und den Erfolg des Projekts beeinträchtigen [24].

Anstatt die Entwicklungsumgebungen durch möglichst professionelle Werkzeuge und feste Prozesse vorzugeben, bezieht sich der erste Wert des „Agilen Manifests“ darauf, das Zusammenwachsen des Teams sowie die teaminterne Interaktion und Kollaboration zu fördern, aus denen sich dann die notwendigen Prozesse und Werkzeuge ableiten [24].

Funktionierende Software über umfassende Dokumentation

Dokumentation spielt in der Softwareentwicklung eine wichtige Rolle, da sie die Intentionen des Entwicklers und die daraus resultierenden Funktionsweisen der Software in für den Menschen lesbaren Dokumenten festhält und eine Kommunikationsgrundlage bildet. Eine ausführliche Dokumentation bedarf jedoch eines hohen zeitlichen Aufwands, da sie kontinuierlichem Anpassungsbedarf an den stetigen Entwicklungsfortschritt der Software unterliegt. Eine umfangreiche Dokumentation kann daher den Entwicklungsprozess behindern [24].

Der zweite Wert des „Agilen Manifests“ fordert auf, sich bei der Entwicklung eher auf die Umsetzung funktionierender Software zu konzentrieren als auf deren Dokumentation. Die einzige verlässliche Quelle über die aktuelle Funktionsweise bildet der zugrunde liegende Quelltext, der zur Kommunikation des Systems – etwa zur Einarbeitung neuer Teammitglieder – verwendet werden sollte [24].

Zusammenarbeit mit dem Kunden über Vertragsverhandlungen

Die vollständige Analyse der Anforderungen an die zu implementierende Software zu Beginn des Entwicklungsprozesses ist aufgrund der sich stetig ändernden Rahmenbedin-

gungen nur eingeschränkt möglich (siehe Abschnitt 4.2). Dennoch liegen den Vertragsverhandlungen mit dem Kunden oftmals die Ergebnisse dieser Anforderungsanalyse zugrunde. Im Laufe der Entwicklung ergibt sich sowohl auf der Seite des Kunden als auch im Entwicklerteam ein Lernprozess, der Änderungen in den technischen Anforderungen auslöst [24].

Der dritte Wert des „Agilen Manifests“ adressiert die Zusammenarbeit von Kunden und Entwicklerteam. Anstatt sich starr an den zuvor vereinbarten Vertragsverhandlungen zu orientieren, sollten die Ergebnisse des Lernprozesses beider Seiten sowie das Feedback des Kunden eingearbeitet werden, um kollaborativ und über den vertraglichen Rahmen hinaus eine zufriedenstellende Lösung anzustreben [24].

Reagieren auf Veränderungen über Befolgen eines Plans

Einer des maßgebenden Erfolgsfaktoren liegt in der Anpassungsfähigkeit, mit der ein Entwicklerteam sich an geänderte Rahmenbedingungen anpassen kann (siehe Abschnitt 4.2). Diese Anpassungsfähigkeit bedarf flexibler Pläne, die sich auf neue Eventualitäten ausrichten lassen. Je detaillierter und weitreichender ein Plan ausformuliert ist, desto schwieriger gestaltet es sich, Abweichungen einzuarbeiten. Für eine hohe Flexibilität sollten detaillierte Pläne daher nur für eine überschaubare Zeitspanne ausgearbeitet werden [24].

Der vierte Wert des „Agilen Manifests“ besagt, die Flexibilität eines Projekts gegenüber der ausgiebigen Planung vorzuziehen, um sich schnell und problemlos auf die sich ändernden Rahmenbedingungen anzupassen [24].

4.4 Scrum als agiles Vorgehensmodell

Schon lange vor der Entstehung des „Agilen Manifests“, entwickelt Jeff Sutherland zusammen mit Ken Schwaber – Mitglieder der *Agile Alliance* – das Vorgehensmodell *Scrum*, das die Werte des Manifests (siehe Abschnitt 4.3) in die Praxis umsetzt. *Scrum* basiert auf der Idee des *Lean*-Ansatzes nach Ohno et al. [26], bei dem mit möglichst

wenig Ressourceneinsatz die maximale Qualität des Endproduktes erreicht werden soll. Nach Sutherland [38] sind dafür die Ausführung ineffizienter Prozesse sowie die Bearbeitung unzweckmäßiger Aufgaben zu vermeiden, die oftmals aus der unvollständigen und teilweise fehlerhaften Anforderungsanalyse sowie der ganzheitlichen Planung der traditionellen Vorgehensweise resultieren (siehe Abschnitt 4.2 und 4.1). Wie in Abschnitt 4.3 beschrieben ergibt sich während der Entwicklung ein Lernprozess unter anderem in Bezug auf die Anforderungen oder die technische Umsetzung des Produkts (siehe „Zusammenarbeit mit dem Kunden über Vertragsverhandlungen“). Der fortschrittsbedingte Lernprozess kann dazu führen, dass sich die Entwicklungsbedingungen ändern und die in der ganzheitlichen Planung definierten Aufgaben und Prozesse zu einem späteren Zeitpunkt nicht mehr zielführend sind. Um auf die Veränderungen frühzeitig reagieren zu können, weist *Scrum* inkrementelle und iterative Strukturen auf, die diesen kontinuierlichen Lernprozess berücksichtigen. Bei *Scrum* handelt es sich demnach um einen Ansatz des Projektmanagements, der gegensätzlich zum Wasserfall-Modell den während des Entwicklungsprozesses auftretenden Unsicherheiten gerecht wird. Das agile Vorgehensmodell bietet den Entwicklern die Möglichkeit nicht nur das entstandene Resultat regelmäßig zu überprüfen, sondern ebenfalls den dafür ausgeübten Prozess iterativ und kollaborativ zu evaluieren und zu optimieren. Daher unterstützt *Scrum* ebenfalls die geforderte Selbstorganisation der Entwicklerteams und trägt erheblich zur Effizienz der Umsetzung bei. Zugleich fördert *Scrum* verglichen mit der traditionellen Vorgehensweise durch nur geringe Einschränkungen die Kreativität und Freiheit der Entwickler und sorgt für einen leichtgewichtigen Entwicklungsprozess [36, 38].

Die Grundzüge von *Scrum* entnimmt Sutherland [38] dem Paper *The New New Product Development Game*, das Takeuchi et al. [40] im Jahr 1986 veröffentlichen. Anstatt wie in dem sequentiellen Wasserfall-Modell, bei dem die einzelnen Phasen nacheinander von unterschiedlichen spezialisierten Teams ausgeführt werden, setzt *Scrum* wie von Takeuchi et al. [40] gefordert auf multidisziplinäre Teams, in denen die Phasen gleichzeitig ablaufen können. So entsteht die Metapher von einem Rugby-Team, das geschlossen über den Platz läuft und sich den Ball gegenseitig zuspielt [40]. Der Name *Scrum* (dt: „Getümmel/Gedränge“) leitet sich dabei von dem *Rugby Scrum* zu Beginn jedes Spielzugs ab.

Zu beachten gilt es, dass *Scrum* lediglich ein Prozessrahmenwerk liefert, das dazu anhält, die Aktivitäten regelmäßig zu unterbrechen, um die Richtung des Entwicklungsprozesses zu überprüfen und Wissen innerhalb des Teams auszutauschen. Es beinhaltet keine konkreten Praktiken, die an die Softwareentwicklung gebunden sind und kann daher auch in anderen Ingenieurdisziplinen angewendet werden. Nachfolgend erläutert dieser Abschnitt die wichtigsten Elemente des *Scrum*-Prozesses. Dazu gehören unter anderem die Rollenverteilung in den Teams, Werkzeuge sowie der zyklische Workflow.

4.4.1 Rollenverteilung im Scrum-Team

Der iterative und inkrementelle Prozess von *Scrum* beruht auf der ständigen Evaluation, ob sich die Entwicklung analog zu den Anforderungen des Kunden bewegt. Dazu bedarf es eines Ansprechpartners (*Product Owner*), der die Visionen des Kunden vertritt, Anforderungen an das Produkt kommuniziert und Feedback über bereits umgesetzte Bestandteile gibt. Für eine anhaltende und ausgiebige Kollaboration sollte der *Product Owner* permanent zur Verfügung stehen und ist deshalb ein Mitglied des *Scrum*-Teams. Besteht nicht die Möglichkeit, den tatsächlichen Kunden in das Team einzubeziehen, wird der *Product Owner* durch eine Person repräsentiert, die engen Kontakt zu dem Kunden pflegt [38].

Die Umsetzung des Produkts übernimmt ein Entwicklerteam, das aus drei bis maximal sechs Personen besteht. Die Entwickler (*Developer*) dieses Teams weisen die entsprechenden Fähigkeiten auf, um aus den Vorstellungen des *Product Owners* ein vollwertiges Produkt zu erstellen [38].

Weiterhin beinhaltet ein *Scrum*-Team die Rolle eines *Scrum Masters*, dessen Aufgabe darin besteht, die korrekte Ausführung des iterativen und inkrementellen Prozesses zu gewährleisten. Der *Scrum Master* unterstützt die Entwickler, in dem er sie von dem bürokratischen Aufwand wie beispielsweise der Terminfindung für Besprechungen befreit. Gleichzeitig bewältigt er anstehende Hindernisse und Hürden wie z.B. technische Probleme, die die Arbeit des Entwicklerteams behindern und ebnet dadurch den Weg zu einer problemfreien Entwicklungsumgebung. Darüber hinaus hilft der *Scrum Master* dem Entwicklerteam durch Beobachtung und Moderation, Verbesserungspotentiale

aufzudecken und den Prozess zu optimieren [38]. Ein *Scrum*-Team weist demnach die folgenden Rollen auf:

- *Product Owner*
- *Developer*
- *Scrum Master*

4.4.2 Werkzeuge

Die Anwendung von *Scrum* fokussiert die möglichst effiziente Umsetzung von Projekten, die sich einerseits an der Umsetzungsgeschwindigkeit, andererseits an der erreichten Qualität im Vergleich zu den Bedürfnissen des Kunden misst. Um beide Aspekte kontinuierlich zu überwachen, beinhaltet der *Scrum*-Prozess Hilfswerkzeuge. So existiert beispielsweise ein *Backlog* (dt: „Rückstand“), bei dem es sich um eine listenartige Ansammlung aller Anforderungen und Kundenwünsche handelt, die das umzusetzende Produkt noch nicht erfüllt. Diese Liste wird zu Beginn jedes Iterationsschritts erneuert, sodass bereits umgesetzte oder nicht mehr aktuelle Einträge entfernt bzw. neue Anforderungen, die sich im Laufe des vorherigen Iterationsschritts ergaben, hinzugefügt werden. Ein *Backlog* umfasst je nach Komplexität des Produkts eine mehr oder weniger hohe Anzahl an Einträgen. Er kann unter Umständen hunderte Einträge aufweisen, die mit großer Wahrscheinlichkeit nicht alle in dem nächsten Iterationsschritt umgesetzt werden können. Um dennoch die effiziente Umsetzung des Produkts nicht aus den Augen zu verlieren, werden die Einträge des *Backlogs* so priorisiert, dass zunächst diejenigen Aufgaben bearbeitet werden, die die höchste Wertsteigerung für das Produkt liefern. Auch der erste Iterationsschritt beginnt mit der Aktualisierung des *Backlogs* und dem Priorisieren der Einträge. Die Erstellung eines initialen *Backlogs* ist deshalb der Einstiegspunkt zur Durchführung von *Scrum*. Die Aktualisierung des *Backlogs* am Anfang jedes Iterationsschritts erfolgt durch Zusammenarbeit des gesamten *Scrum*-Teams [38].

Zur Visualisierung des Projektfortschritts verwendet *Scrum* weitere Werkzeuge. Darunter befindet sich beispielsweise das in Abbildung 4.3 dargestellte *Burndown Chart*, das eine grafische Darstellung der umgesetzten *Backlog*-Einträge in Bezug auf die vergangenen Iterationsschritte darstellt. Da sich die *Backlog*-Einträge unter anderem in ihrer

Komplexität unterscheiden, werden sie zur besseren Vergleichbarkeit mit Punkten bewertet, die dem Aufwand ihrer Umsetzung entsprechen. Die vertikale Achse des *Burndown Chart* spiegelt somit den Arbeitsumfang in Punkten wider, den das Entwicklerteam bis zum Ende des Projekts bewältigen muss. Mit jedem umgesetzten *Backlog*-Eintrag sinkt dieser Arbeitsumfang um die entsprechende Anzahl an Punkten. Als ideales Ziel verfolgt *Scrum* die Umsetzung aller *Backlog*-Einträge. Der Fortschritt des Teams sollte daher gegen Ende des Projekts die horizontale Achse berühren. Ein idealer Teamfortschritt ergibt sich somit aus der linearen Verbindung zwischen anfänglichem Arbeitsaufwand und dem Schnittpunkt mit der horizontalen Achse zum Ende des Projekts (siehe gestrichelte Linie). Aus dieser Darstellung lässt sich die tatsächliche sowie die durchschnittliche Entwicklungsgeschwindigkeit des *Scrum*-Teams ableiten und überwachen. Das *Burndown Chart* weist dadurch auf mögliche Abweichungen der durchschnittlichen Effizienz hin und hilft dabei den Entwicklungsprozess zu optimieren [38].

Anstatt das *Burndown Chart* wie in Abbildung 4.3 gezeigt zur Visualisierung des gesamten Projektfortschritts in Bezug auf die Iterationsschritte zu verwenden, kann ebenfalls der Fortschritt einzelner Auslieferungszyklen oder Iterationsschritte dargestellt werden [38].

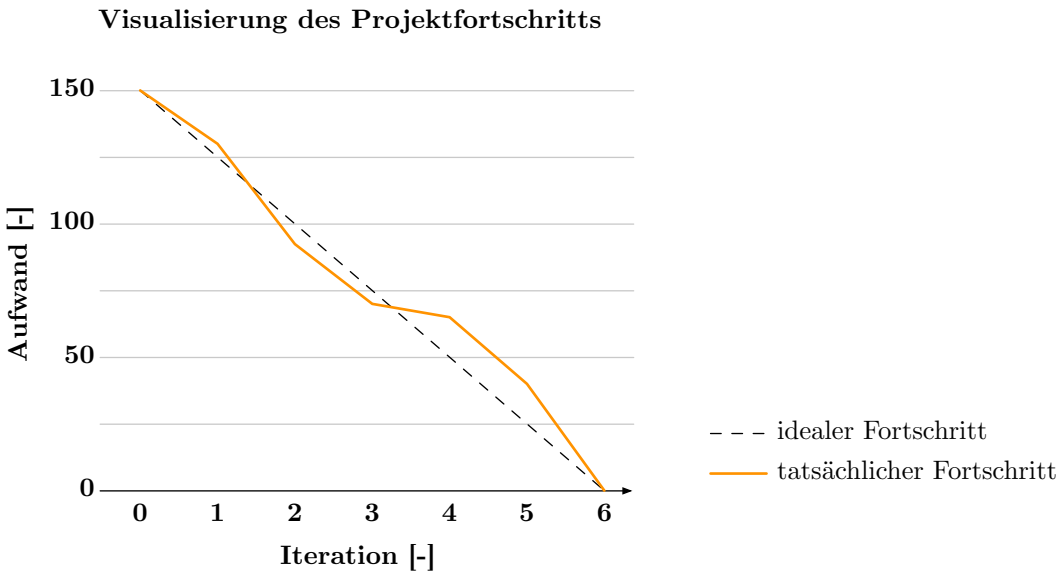

Abbildung 4.3: Beispielhafte und schematische Darstellung eines *Burndown Chart* zur Visualisierung des Projektfortschitts in Anlehnung an Sutherland [38].

Neben den genannten Werkzeugen stehen dem *Scrum*-Prozess weitere unterstützende Instrumente zur Verfügung. Da diese Instrumente im Rahmen der vorliegenden Arbeit nur von geringer Bedeutung sind, werden sie aufgrund des Umfangs im weiteren Verlauf vernachlässigt.

4.4.3 Workflow

Scrum basiert auf der iterativen Entwicklung von auslieferbaren Produktinkrementen, die gegen Ende eines Entwicklungszyklus evaluiert werden, um Verbesserungspotenzial aufzudecken. Gleichzeitig erfährt auch der Entwicklungsprozess mit fortschreitender Iteration kontinuierliche Optimierungen. Dazu folgen die Iterationsschritte des *Scrum*-Prozesses einem festen Zyklus, der sich aus verschiedenen Phasen zusammensetzt und fortwährend wiederholt. Die korrekte Ausführung des Entwicklungszyklus obliegt der Kontrolle des *Scrum Masters*. Abbildung 4.4 stellt eine grobe Übersicht über die Phasen sowie die Abfolge dar [38].

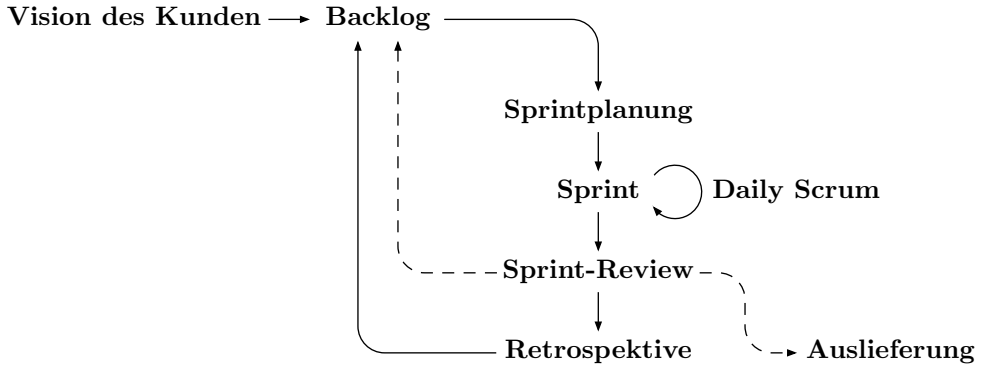

Abbildung 4.4: Workflow des *Scrum*-Prozesses in Anlehnung an Sutherland [38].

Den Ausgangspunkt des *Scrum*-Prozesses bildet der in Abschnitt 4.4.2 beschriebene *Backlog*, der sich aus der Vision des Kunden ergibt. Der *Backlog* enthält verschiedene priorisierte Einträge, die Teilaufgaben des Gesamtprojekts entsprechen. Dabei stehen diejenigen Aufgaben, deren Umsetzung den größten Wert für das Produkt liefern, oben an [38]. Die Umsetzung des Produkts geschieht in der Entwicklungsphase, die im Zusammenhang mit *Scrum* als Sprint bezeichnet wird und einen zeitlich festgelegten Rahmen

von ein bis zwei Wochen aufweist. In den Sprints setzt das Entwicklerteam ausgewählte *Backlog*-Einträge um, die zu einem auslieferbarem Produktinkrement führen. Tägliche Besprechungen, die stets zur gleichen Uhrzeit mit einer maximalen Dauer von 15 Minuten abgehalten werden, begleiten die Sprints – die sogenannten *Daily Scrums*. Da diese Besprechungen als Signal für ihre geringe Dauer im Stehen durchgeführt werden, bezeichnet man sie auch als *Daily Standups*. Die *Daily Scrums*, an denen Entwicklerteam und *Scrum Master* teilnehmen, verfolgen die Absicht, das gesamte Team über die Bearbeitungsstände sowie die Tätigkeiten einzelner Teammitglieder zu informieren. Zusätzlich erhalten die Entwickler dadurch regelmäßig die Möglichkeit, den *Scrum Master* auf Hindernisse hinzuweisen, die den Entwicklungsprozess behindern und die es zu beseitigen gilt [38].

Jedem Sprint geht eine Planungsphase voraus, die Sprintplanung, in der die umzusetzenden *Backlog*-Einträge ausgewählt werden. Für ein Inkrement, das eine möglichst hohe Wertsteigerung des Produkts hervorruft, gilt es dabei, die obersten *Backlog*-Einträge mit den höchsten Prioritäten zu bevorzugen. Die Auswahl der umzusetzenden Aufgaben orientiert sich an der vorherigen Geschwindigkeit des Teams, die sich beispielsweise durch das in Abschnitt 4.4.2 beschriebene *Burndown Chart* ermitteln lässt. Die Visualisierung des Projektfortschritts verdeutlicht den Arbeitsumfang (*Budget*), den das Entwicklerteam in einem Sprint umsetzen kann. Dieses *Budget* gibt ebenfalls darüber Aufschluss, wie viele und welche *Backlog*-Einträge das Entwicklerteam realistisch in einem nächsten Iterationsschritt fertigstellt. Aufgrund der stetigen Optimierung des Entwicklungsprozesses sollte sich die Geschwindigkeit bzw. das *Budget* des Teams kontinuierlich steigern. An der Sprintplanung nimmt das gesamte *Scrum*-Team teil. Dadurch erhalten sowohl *Product Owner* als auch das Entwicklerteam sowie der *Scrum Master* die Möglichkeit, die Richtung des Entwicklungsfortschritts zu beeinflussen [38].

Nach dem Sprint erfolgt ein Sprint-*Review* (dt: „Rückblick“), in dem das Entwicklerteam das entstandene Produktinkrement vorstellt. Als Teilnehmer sind neben dem gesamten *Scrum*-Team weitere Interessenten, wie beispielsweise die Geschäftsführung oder andere *Stakeholder* einzuladen. Das Entwicklerteam erhält Feedback und Kritik, die in den *Backlog* aufgenommen werden und dadurch in die weitere Entwicklung einfließen. Han-

delt es sich bei dem Produktinkrement um eine zufriedenstellende Lösung, erfolgt seine Auslieferung [38].

Im Anschluss an den Sprint-*Review* analysieren Entwicklerteam und *Scrum Master* in der Retrospektive den Entwicklungsprozess des zurückliegenden Sprints. Zur Optimierung der Entwicklung erarbeitet das Entwicklerteam zusammen mit dem *Scrum Master* Verbesserungsansätze, die in dem nächsten Sprint durchgeführt werden. Unter anderem adressieren die Verbesserungsansätze persönliche Schwächen einzelner Teammitglieder, unbefriedigende Kommunikationsstrukturen oder mangelnde Kollaboration [38]. Die Optimierung des Entwicklungsprozesses ergibt sich somit aus der Selbstorganisation des Teams. Als Startpunkt der nächsten Iteration wird der *Backlog* aktualisiert und auf die Auswahl neuer *Backlog*-Einträge für den nächsten Sprint vorbereitet [38].

4.5 Praktiken agiler Softwareentwicklung

Wie Abschnitt 4.4 beschreibt, stellt *Scrum* ein Prozessrahmenwerk bereit, das die Flexibilität sowie die Selbstorganisation während der Entwicklung unabhängig von der betrachteten Ingenieurdisziplin fördert. Es bestimmt nicht die während eines Sprints ausgeübten Aktivitäten zur Umsetzung des Produkts. In der Softwareentwicklung setzten sich in der Vergangenheit die agilen Praktiken durch, die wie *Scrum* die Produktivitätssteigerung des Entwicklerteams durch verbesserte Teamarbeit, erhöhte Transparenz und regelmäßiges Feedback fokussieren. Weiterhin verbessern die agilen Praktiken die Qualität der Software, die sich aus hoher Wartbarkeit und leichter Verständlichkeit des zugrunde liegenden Quelltexts ergibt. Die Umsetzung von Software basiert auf der Implementierung von Quelltext, der gegen Ende eines Sprints das auslieferbare Produktinkrement repräsentiert. Für einen Entwicklungsprozess, der hohe Anpassungsfähigkeit an die sich stetig ändernden Anforderungen aufweist, bedarf auch das Produktinkrement bzw. der zugrunde liegende Quelltext eine hohe Flexibilität. Dafür stehen in der Softwareentwicklung Entwurfsregeln wie beispielsweise die Heuristiken Kohäsion und Kopplung [45] sowie die SOLID-Prinzipien [24] des objektorientierten Designs zur Verfügung.

Die agilen Praktiken beinhalten eine konkrete Beschreibung zur Ausführung von Aktivitäten sowie Verhaltensregeln für die Entwicklung von Software [6, 30]. Sie liefern damit einen offiziellen Umgangsrahmen zwischen den Teammitgliedern und formalisieren den Entwicklungsprozess. Beck et al. [6] fassen die bekanntesten dieser Praktiken zusammen und beschreiben mit *Extreme Programming* eine Vorgehensweise, die auf deren Kombination und gemeinsamer Anwendung beruht. Der nachfolgende Abschnitt erläutert die im Rahmen dieser Arbeit verwendeten Praktiken der agilen Softwareentwicklung. Darunter befinden sich die testgetriebene Entwicklung, *Continuous Integration* und *Pair Programming*.

4.5.1 Testgetriebene Entwicklung

Als qualitätssichernde Maßnahme ist das Testen des Produkts auch in der Softwareentwicklung verankert. In der traditionellen Vorgehensweise handelt es sich dabei nach Meszaros [25] um eine Phase, die sich im Anschluss an die ganzheitliche Umsetzung ansiedelt (siehe auch Abbildung 4.1, Seite 39). Das nachträgliche Testen geht mit der Herausforderung einher, dass sich die Struktur bzw. das Design des zuvor umgesetzten Quelltext unter Umständen in einem wenig testbaren Zustand befindet. Das Testen der Software bedarf daher wahrscheinlich eine teilweise Neukonzeptionierung des entstandenen Systems und ist mit viel Aufwand verbunden. Darüber hinaus werden Fehler in der Umsetzung erst zu einem späten Zeitpunkt der Entwicklung aufgedeckt [17, 25].

Für kontinuierliches Feedback in Bezug auf die Fehlerfreiheit sollten die Tests und die zu implementierende Software parallel entstehen. Aus diesem Grund sieht die testgetriebene Entwicklung nach Beck [5] eine inkrementelle Erweiterung des Systems vor. Sie beinhaltet einen sich wiederholenden Ablauf von Aktivitäten, in dem einzelne Funktionalitäten umgesetzt und getestet werden. Um des Weiteren die Testbarkeit der Software zu erhöhen, basiert TDD zusätzlich zu der inkrementellen Entwicklung auf dem *Test-First*-Ansatz. Nach Beck [5] werden demnach die Tests vor der umzusetzenden Funktionalität implementiert. Die resultierende Software ergibt sich daher aus den Anforderungen zuvor formulierter Tests. TDD führt somit zu Quelltext, der eine inhärente Testbarkeit aufweist [25].

Die Vorgehensweise der testgetriebenen Entwicklung ergibt sich nach Beck [5] aus der strikten Anwendung der nachfolgenden Regeln:

- Implementiere nur neuen Produktionscode, wenn ein automatisierter Test existiert, der fehlschlägt.
- Entferne Redundanzen.

Aus diesen Regeln leitet sich der in Abbildung 4.5 dargestellte Entwicklungszyklus von TDD ab, der aus den dargestellten Phasen Rot, Grün und *Refactoring*[1] besteht. Rot steht dabei für die Implementierung eines Tests, der aufgrund der noch nicht existenten Funktionalität des Produktionscodes fehlschlägt. Mit der Forderung des *Test-First*-Ansatzes ergibt sich Phase Rot als Startpunkt für einen Durchlauf des Entwicklungszyklus. In Pha-

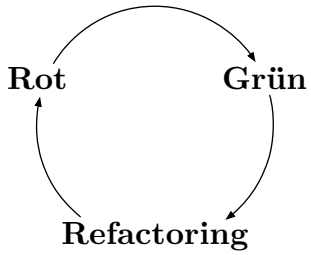

Abbildung 4.5: Phasen der testgetriebenen Entwicklung nach Beck [5] in Anlehnung an Freeman et al. [17].

se Grün wird die Software um die notwendige Funktionalität erweitert, die ein Bestehen des fehlschlagenden Tests erreicht. Dafür sollte eine möglichst minimalistische und naive Lösung angestrebt werden, die ausreicht, um den Test bestehen zu lassen. Ein anschließendes *Refactoring* transformiert die minimalistische und naive Lösung zu qualitativ hochwertigem Quelltext. Die Einteilung des Entwicklungszyklus in die drei dargestellten Phasen trennt die Verantwortlichkeiten des Entwicklers auf und führt deshalb zu einer kognitiven Entlastung. Anstatt die angestrebte Funktionalität sowie deren korrekte und qualitativ hochwertige Implementierung gemeinsam zu bewältigen, wird dem Entwickler die Möglichkeit geboten diese drei Herausforderungen nacheinander zu adressieren. Die-

[1] Nach Fowler [16] bezeichnet Refactoring eine Technik zur Modifikation von bereits umgesetztem Quelltext mit der Intention, sein Design zu verbessern. Die Funktionalität bzw. das Verhalten des Codes wird dabei nicht beeinflusst.

ses Vorgehen führt zumeist zu schlanken und fehlerfreien Implementierungen, die sich durch Redundanzfreiheit, Direktheit und leichte Verständlichkeit auszeichnen [5, 25].

Neben der kognitiven Entlastung des Entwicklers beinhaltet die Anwendung von TDD weitere positive Effekte. Für konsistente Testergebnisse und aussagekräftiges Feedback über die Korrektheit der Software müssen die formulierten Tests in Isolation betrachtet werden (vgl. *Unit Tests* [27]). Die Isolation der Testfälle führt zu klaren Systemstrukturen und der Entkopplung verschiedener Teilsysteme der Software. Martin et al. [24] bezeichnen diesen Effekt als *Serendipitous Decoupling* (dt: „glückliche Entkopplung“), durch die ebenfalls die Heuristiken Kohäsion und Kopplung sowie die Entwurfsprinzipien des objektorientierten Designs eingehalten werden. Software, die eine hohe Testbarkeit aufweist, trifft daher auch die Vorstellungen von flexiblem und wartbaren Quelltext. Zusätzlich dienen die implementierten Tests als Dokumentation und liefern eine sichere Quelle über die Funktionalität der Software.

4.5.2 Continuous Integration

Die Entwicklung im Team sollte nach Beck et al. [6] auf einer gemeinsamen Code-Basis beruhen, in die die einzelnen Teammitglieder ihre umgesetzten Funktionalitäten kontinuierlich einpflegen. Ein bekanntes Werkzeug dafür spiegeln die in Abschnitt 4.3 erwähnten Versionskontollsysteme wider, die es erlauben, einzelne Bearbeitungsstände der Entwicklung in einer chronologischen Reihenfolge (genannt *Branch*; dt: „Zweig“) zu speichern und sie zu einem späteren Zeitpunkt erneut abzurufen. Ein VCS bietet die Möglichkeit eines einheitlichen Produktionsstrangs (*Production Branch* oder *Master Branch*), der als gemeinsame Code-Basis dient. Für die Entwicklung einzelner *Features* (dt: „Funktionen“) zweigen die Entwickler sogenannte *Feature Branches* ab, die auf dem aktuellen Stand des Produktionsstrangs basieren. Im Anschluss an die Implementierung führen die Entwickler den *Production Branch* mit dem *Feature Branch* zusammen (*Merging*) und erweitern somit die einheitliche Code-Basis um die implementierte Funktionalität. Abbildung 4.6 stellt diesen Vorgang dar. Auf dem zweiten Entwicklungsstand des *Master Branch* startet Entwickler 1 die Implementierung einer neuen Funktionalität und richtet dazu einen *Feature Branch* ein, der nach erfolgreicher Umsetzung des *Features* wieder in

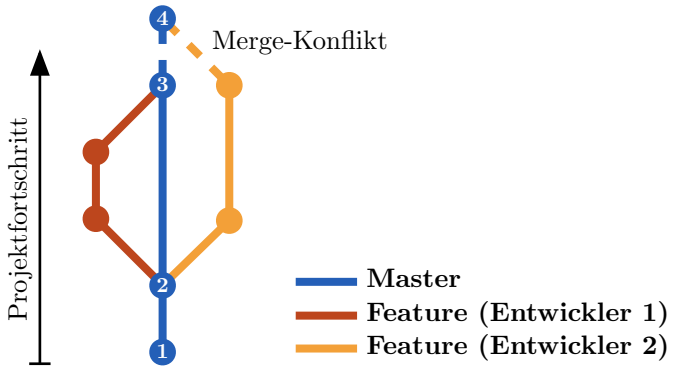

Abbildung 4.6: Verwendung eines VCS für die Entwicklung mit einheitlicher Code-Basis in Anlehnung an Beck et al. [6].

die einheitliche Code-Basis zurückfließt. Es ergibt sich der dritte Bearbeitungsstand des Produktionszweigs.

Bei dem Zusammenführen der unterschiedlichen Zweige können unter Umständen Schwierigkeiten auftreten – die sogenannen *Merge*-Konflikte. Bearbeiten mehrere Entwickler gleichzeitig verschiedene *Features*, existieren ebenfalls mehrere *Feature Branches* zugleich. In Abbildung 4.6 startet auch Entwickler 2 ein *Feature* auf dem zweiten Stand des Produktionszweigs. Die Entwicklung nimmt jedoch mehr Zeit in Anspruch als die Implementierung des *Features* von Entwickler 1. Mit dem Zusammenführen der Zweige durch Entwickler 1 verändert sich der Bearbeitungsstand des Produktionszweig. Es treten somit Änderungen in der einheitlichen Code-Basis auf, die Entwickler 2 in seinem *Feature* nicht bekannt sind. Da diese Änderungen das System betreffen, das Entwickler 2 für die Implementierung seines *Features* voraussetzt (Bearbeitungsstand 2), bedarf das Integrieren seiner Funktionalität weitere Anpassungen.

Um diesen Anpassungsbedarf und den damit verbundenen zeitlichen Aufwand zu reduzieren, fordert die agile Praktik CI die regelmäßige Integration der durch die unterschiedlichen Entwickler umgesetzten Teilsysteme in die gemeinsame Code-Basis [6, 24]. Weiterhin kann die Modifikation der bestehenden Code-Basis zu Regressionen führen. Dabei handelt es sich um bereits umgesetzte und funktionsfähige *Features*, die durch nachträgliche Veränderungen versehentlich beeinträchtigt werden [27]. Um die Regressionen unverzüglich aufzudecken und zu beseitigen, sieht CI die Ausführung aller vor-

handenen Tests bei jeder Integration eines neuen *Features* vor. Neben der Effizienz des Projekts trägt CI deswegen auch zur Qualität und Fehlerfreiheit der Software bei [6, 24].

4.5.3 Pair Programming

Eine weitere Praktik zur Verbesserung der Softwarequalität liegt in der paarweisen Programmierung, bei der sich Entwickler in Paaren zusammenfinden und gemeinsam programmieren. Bei PP kontrolliert einer der beiden Entwickler die Eingabehilfen des Computers und setzt den Quelltext technisch um (*Driver*; dt: „Fahrer“), während der andere dabei zusieht und Verbesserungsvorschläge einbringt (*Navigator*; dt: „Beifahrer“). Diese Rollenverteilung des PP ist flexibel und sollte mehrmals in der Stunde gewechselt werden. Insbesondere wenn sich der *Driver* „festfährt“ und der Projektfortschritt stagniert, sollte ein Rollentausch stattfinden. Der *Navigator* löst seinen Partner als *Driver* ab, übernimmt die Kontrolle der Eingabehilfen und setzt die Programmierung fort. Zusätzlich sieht PP nach Martin et al. [24] regelmäßig variierende Paarungen vor, sodass jeder der Entwickler mit unterschiedlichen Partnern zusammenarbeitet. Durch das interaktive Arbeiten der Entwickler fördert die paarweise Programmierung neben der Kollaboration auch die teaminterne Wissensausbreitung. Des Weiteren begünstigt PP die Fehlerfreiheit der Software und trägt daher maßgeblich zur Qualität des Produkts bei [6, 24].

KAPITEL 5

Agile Methodik des zukünftigen Lehrkonzepts

Wie die Analyse des bestehenden Lehrkonzepts in Kapitel 3 darstellt, beabsichtigt die derzeitige Methodik in der zentralen Übung die aktive, kooperative und selbstorganisierte Bearbeitung von Übungsaufgaben, die den Kompetenzerwerb der Studierenden sicherstellen soll. Fehlannahmen in der Konzeption verhindern jedoch die Ausführung dieser Methodik. Als Resultat ergibt sich eine darbietende Lehrweise, die den Erwerb der gewünschten Kompetenz unterbindet. Zur Verbesserung dieser Methodik werden nun die in Kapitel 4 erläuterten agilen Methoden auf die Lehre übertragen und in das Lehrkonzept eingearbeitet. Darüber hinaus wird gezeigt, wie die agilen Methoden neben dem Adressieren der in Kapitel 4 erarbeiteten Verbesserungsansätzen zusätzlich eine qualitativ hochwertige Lehre gemäß Kapitel 2 hervorrufen.

Das nachfolgende Kapitel beschreibt die agile Methodik des zukünftigen Lehrkonzepts. Es verdeutlicht zunächst die Anwendbarkeit der agilen Methoden im Sinne der „Guten Lehre“ und zeigt, wie sich die grundlegenden Werte agiler Vorgehensweisen auf die Lehre übertragen lassen. In einem nächsten Abschnitt werden die in Kapitel 4 beschriebenen agilen Methoden auf die Anwendung in der Lehre transferiert und ihre Effekte auf die

Qualität der Lehrveranstaltung dargelegt. Abschließend gibt das Kapitel einen Überblick über den zeitlichen Ablauf der Lehrveranstaltung.

Bei der in diesem Kapitel beschriebenen Methodik handelt es sich um eine innovative Konzeption der Lehrveranstaltung „Algorithmen und Programmieren“ des iRMB der Technischen Universität Braunschweig. Dabei werden die Ansätze „Guter Lehre“ mit der derzeitigen Praxis der Softwareentwicklung kombiniert. Zwar können die erwarteten positiven Effekte dieser Methodik auf die Lehre und den Lernprozess plausibel aus den in Kapitel 2, 3 und 4 dargestellten Zusammenhängen hergeleitet werden, dennoch gilt es das Lehrkonzept im Anschluss an die Umsetzung durchzuführen und zu evaluieren, inwieweit sich die erwarteten Effekte einstellen.

5.1 Agilität in der Lehre

Wie die „Gute Lehre“ (siehe Kapitel 2) fokussieren die agilen Methoden der Softwareentwicklung (siehe Kapitel 4) das kooperative und selbstorganisierte Arbeiten und unterstützen durch ihren iterativen und inkrementellen Ansatz den Austausch von Feedback. Sie stellen die beteiligten Individuen und deren Interaktion bzw. Kollaboration in den Mittelpunkt und weisen flexible Strukturen auf, um den Prozess kontinuierlich zu reflektieren und zu optimieren. Die geförderte Kommunikation erhöht weiterhin die Transparenz, da beteiligte Personen in engem Austausch stehen und die Handlungen sowie die Fortschritte anderer Personen beobachten und verfolgen können. Betrachtet man die agilen Methoden aus der Sicht der „Guten Lehre“, ergibt sich mit der angestrebten Selbstorganisation, Transparenz, Kommunikation, Reklexion und Kollaboration eine hohe Schnittmenge verfolgter Ziele (vgl. Definition 2.1, Seite 13). Die agilen Methoden gliedern sich daher nahtlos in die Forderungen der „Guten Lehre“ ein, unterstützen das aktive, kooperative und selbstorganisierte Lernen der Studierenden und verbessern den Lernprozess. Gleichzeitig bereitet die Anwendung der agilen Methoden in der Lehre die Studierenden auf den späteren Berufsalltag als Softwareingenieur vor und liefern ein realistisches Bild über die derzeitige Praxis in der Softwareentwicklung.

Aufgrund ihrer positiven Wirkung auf den Lernprozess – besonders im Bereich der Programmierung – fanden die agilen Methoden begleitet von zahlreichen Studien ihren Weg in die Lehre [37]. Stewart et al. [37] transferierten dazu das „Agile Manifest“ aus Abschnitt 4.3 (siehe Abbildung 4.2, Seite 42) und definierten die nachfolgenden vier Werte, an denen sich die agile Lehre orientiert:

- Studierende über traditionelle Lehrprozesse und Werkzeuge
- Funktionierende Projekte über umfassende Dokumentation
- Kollaboration von Studierenden und Lehrenden über unflexible Lehrpläne
- Reagieren auf Feedback über Befolgen des Lehrplans

Nachfolgend werden die von Stewart et al. [37] übertragenen Werte sowie ihre Bedeutung für die Lehre näher erläutert.

Studierende über traditionelle Lehrprozesse und Werkzeuge

In der traditionellen Lehre beruht die Vermittlung der Lehrinhalte zumeist auf festgelegten Lehrprozessen, die sich unter anderem aus individuellen Präferenzen der Lehrperson oder Ressourcenknappheit ergeben. Diese Art der Lehre orientiert sich nicht an den individuellen Merkmalen der Studierenden und erzeugt meistens nur wenig ertragreiche Lernerfolge. Ähnlich wie in der agilen Softwareentwicklung, in der sich die Prozesse und Werkzeuge aus der teaminternen Kommunikation und Kollaboration ergeben, sollte sich die Lehre ebenfalls auf individuelle Begebenheiten der Studierenden anpassen können. Anstatt sich an festgelegte Lehrprozesse und Werkzeuge zu halten, sollten die Studierenden im Mittelpunkt bei der Gestaltung der Lernumgebung stehen. Die Lehre sollte vollständig auf den individuellen Lernprozess der Studierenden ausgerichtet sein [37].

Funktionierende Projekte über umfassende Dokumentation

In Lernangeboten, in denen Studierende über den Verlauf der Veranstaltungsreihe ein ganzheitliches Projekt umsetzen, findet die Vorstellung der Ergebnisse oft erst in der letzten Lehrveranstaltung statt. Das führt dazu, dass Studierende häufig erst spät mit

der Umsetzung beginnen und deswegen das Projekt nicht fertigstellen. Der iterative und inkrementelle Ansatz der agilen Methoden unterstützt das kontinuierliche Bearbeiten des Projekts und erzeugt nach jeder Iteration ein auslieferbares Produktinkrement. Dabei steigert jede Iteration das „Eintauchen“ der Studierenden in die Projektarbeit und intensiviert den Lernprozess. Gleichzeitig erhöht sich ebenfalls die Qualität des umgesetzten Ergebnisses. Anstatt eine umfassende Dokumentation des Projekts zu priorisieren, sollte die Lehre eine hohe Qualität des Ergebnisses anvisieren, die mit einer hohen Anzahl an Iterationen und damit verbundenen ertragreichen Lernerfolgen einhergeht [37].

Kollaboration von Studierenden und Lehrenden über unflexible Lehrpläne

Betrachtet man eine Lehrveranstaltung als umzusetzendes Produkt, dessen Aufgabe in der Vermittlung von Kompetenzen liegt, nehmen die Studierenden die Rolle des Kunden ein. Die Lehrperson agiert als Entwickler, der die Lehrveranstaltung plant und durchführt. In dieser Metapher spiegelt der häufig durch Institutionen festgelegte Lehrplan den Vertrag zwischen den beteiligten Personen wider. Wie in der Softwareentwicklung sollte dieser Vertrag nicht das Umsetzen einer zufriedenstellenden Lösung verhindern. Deswegen sollte die Lehrperson auf die Bedürfnisse der Studierenden eingehen und gegebenenfalls den Lehrplan modifizieren bzw. flexibel gestalten.

Reagieren auf Feedback über Befolgen des Lehrplans

Lehrveranstaltungen folgen oftmals einem festgelegten Zeitplan, in dem für jedes zu vermittelnde Thema eine gewisse Zeitspanne eingeplant ist. Auch orientieren sich Lehrpersonen häufig an ihren Standardmethoden, um bestimmte Themen zu vermitteln. Sie vernachlässigen dabei, dass sich Lerngruppen aufgrund ihrer unterschiedlichen Zusammensetzung stark voneinander unterscheiden können. Für jede Lerngruppe bedarf es deswegen unter Umständen angepasste Zeitpläne und Lehrmethoden bzw. Lernaktivitäten. Um die Lehrveranstaltung auf spezifische Lerngruppen auszurichten, sollte die Lehrperson ihren Zeitplan sowie ihre Lehrmethoden flexibel anpassen.

5.2 Transfer und Effekte der agilen Methoden

In dem zukünftigen Lehrkonzept sollen die agilen Methoden der Softwareentwicklung angewendet werden, um die Qualität der Lehrveranstaltung zu erhöhen. Aufgrund der in Abschnitt 5.1 beschriebenen Schnittmenge verfolgter Ziele zwischen der agilen Softwareentwicklung und der „Guten Lehre“ wird eine positive Wirkung auf die Lehrveranstaltung erwartet. Der nachfolgende Abschnitt zeigt den Transfer der in Kapitel 4 beschriebenen Vorgehensmodelle und Praktiken der Softwareentwicklung auf die Anwendung in der Lehre und beschreibt die zu erwartenden Effekte.

Zunächst wird dargelegt, wie die Kombination von TDD und PP die derzeitige darbietende Lehrweise der zentralen Übung in das aktive und kollaborative Bearbeiten der Aufgabenstellungen überführt und die Verbesserungsansätze aus Abschnitt 3.3 adressiert. In einem nächsten Schritt wird gezeigt, wie die Ansätze von CI in Kombination mit einem *Burndown Chart* verwendet werden, um die Lernfortschritte der Studierenden zu visualisieren und die Transparenz der Lehrveranstaltung zu erhöhen. Anschließend wird der iterative und inkrementelle *Scrum*-Prozess auf die Situation in der Lehre übertragen. Es wird gezeigt, dass *Scrum* sowohl zur kontinuierlichen Optimierung des Lernprozesses der Studierenden eingesetzt werden kann als auch eine stete Reflexion und Verbesserung der Lehrveranstaltung ermöglicht.

5.2.1 Aktives und Kollaboratives Bearbeiten der Aufgaben

Neben den Ansätzen „Guter Lehre“ unterstützen die agilen Methoden weiterhin dabei, die in Abschnitt 3.3 erläuterten Verbesserungsansätze in das Lehrkonzept einzuarbeiten. Als Verbesserungsansätze sind ein formeller Verhaltensrahmen zur Überwindung der Heterogenität der Lerngruppe und zur Förderung der Kollaboration sowie die Umgestaltung der Aufgabenstellung zur kognitiven Entlastung der Studierenden und zur Erhöhung des Feedbacks während der Bearbeitung vorgesehen.

Die agilen Methoden definieren nacheinander auszuführende Aktivitäten sowie den Umgang der Studierenden untereinander. Sie dienen daher als Leitfaden für die Ausführung der kollaborativen Programmierung und treiben den kooperativen Entwicklungspro-

zess bzw. Lernprozess voran. So verdeutlichen beispielsweise Rollenverteilungen wie bei *Scrum* (*Product Owner*, *Developer*, *Scrum Master*, siehe Abschnitt 4.4.1) oder bei PP (*Driver*, *Navigator*, siehe Abschnitt 4.5.3) die Erwartungen an die Studierenden in der Lehrveranstaltung und ermutigen sie dabei, sich aktiv an der Kollaboration und der Interaktion mit ihren Kommilitonen und der Lehrperson zu beteiligen. Mit dem Einnehmen einer Rolle überträgt sich auch die Verantwortung dieser Rolle auf die Studierenden und bindet sie vollständig in die Bearbeitung der Aufgaben ein. Die Verantwortungen geben den Studierenden das Bewusstsein dafür, unmittelbarer und unverzichtbarer Bestandteil der Lernaktivität zu sein. Infolgedessen steigert sich das Selbstwertgefühl und das *Commitment* der Teilnehmer. Angetrieben durch die Verantwortungen der eingenommenen Rollen rücken motivationale und kulturelle Disparitäten in den Hintergrund und die Kollaboration wird gefördert. Des Weiteren formalisieren die definierten Abläufe der agilen Methoden wie beispielsweise der Zyklus der testgetriebenen Entwicklung ein Gerüst an auszuführenden Aktivitäten und fördern dadurch den gemeinsamen Arbeitsablauf. Zur Förderung der Kollaboration fokussiert das zukünftige Lehrkonzept eine klare Kommunikation der Aufgaben- und Rollenverteilungen. Ähnlich wie das derzeitige Lehrkonzept beruht die zukünftige Methodik auf der kollaborativen Bearbeitung der Übungsaufgaben in Gruppenverbänden. Im Sinne des PP werden dabei jedoch die Gruppen auf zwei Mitglieder begrenzt und die entsprechenden Rollen besetzt.

Wie Abschnitt 3.2 schildert, verhindert das derzeitige Lehrkonzept weiterhin die aktive und selbstorganisierte Bearbeitung der Übungsaufgaben. Grund dafür sind zu anspruchsvolle Aufgabenstellungen, die die Studierenden demotivieren und einschüchtern. Ein weiterer Verbesserungsansatz sieht daher die Umgestaltung der Aufgabenstellung vor, die die Gesamtaufgabe in kleine Teilaufgaben zergliedert. Infolgedessen können Studierende Schritt für Schritt durch die Aufgabe geleitet und kognitiv entlastet werden. (siehe Abschnitt 3.3). Auch bietet die Unterteilung der Aufgabenstellung die Möglichkeit, den Studierenden regelmäßiges Feedback über ihren Bearbeitungsfortschritt bzw. Lernfortschritt zu geben. Die Umgestaltung der Aufgabenstellung beruht auf dem Paradigma der testgetriebenen Entwicklung nach Beck [5] (siehe Abschnitt 4.5.1). Eine Zergliederung erfolgt mit Hilfe von einzelnen Tests, die in Summe das zu lösende Gesamtproblem spezifizieren. Somit besteht die ausgehändigte Aufgabenstellung aus mehreren ausführbaren

Tests, an denen sich die Studierenden bis zum Ende der Aufgabe orientieren können. Kapitel 6 stellt einen Leitfaden für die Spezifikation der Aufgabenstellungen bereit. Jeder Test beschreibt eine kleine zu lösende Teilaufgabe und lenkt die Studierenden in die Richtung der korrekten Gesamtlösung. Das geforderte Feedback über den Bearbeitungsstand bzw. den Bearbeitungsfortschritt erhalten die Studierenden durch das Ausführen der Tests. Sie sind dadurch in der Lage, selbstständig und zu jedem Zeitpunkt ihren eigenen Lernfortschritt zu evaluieren.

Auch wenn die Studierenden die Tests nicht selbst implementieren, wird der inkrementelle und iterative Entwicklungszyklus von TDD in der Bearbeitung der Aufgabe beachtet. Anstatt mehrere oder alle Tests gleichzeitig zu bearbeiten, konzentrieren sich die Studierenden immer nur auf das Bestehen eines einzelnen Tests. Phase Rot fokussiert das Verständnis des aktuell zu bearbeitenden Tests. Die Studierenden lesen und interpretieren ihn und erarbeiten seine Bedeutung im Kontext der Aufgabenstellung. Anschließend erfolgt in Phase Grün die Implementierung des Produktionscodes, der wie von TDD gefordert eine möglichst naive und minimalistische Lösung darstellt, um den Test bestehen zu lassen. Ein anschließendes *Refactoring* erhöht die Qualität des Produktionscodes. Die einzelnen Tests sind dabei so ausgelegt, dass es nur sehr wenig Quelltext bedarf, um ein Bestehen zu erreichen. Der nötige Code umfasst daher maximal zwei bis drei Zeilen. Es kommt zu sehr kurzen Mikrozyklen, die die Kontinuität des Feedbacks fördern. Neben einer sehr geringen momentan betrachteten Komplexität (siehe Abbildung 3.4, Seite 35) erlaubt die Bearbeitung der Aufgabe unter Anwendung des Zyklus von TDD wie in Abschnitt 4.5.1 beschrieben weiterhin die getrennte Bewältigung der korrekten und der qualitativ hochwertigen Implementierung. Infolgedessen nimmt die kognitive Belastung der Studierenden weiter ab und die Bearbeitung der Aufgaben erleichtert sich.

Durch die Kombination von PP und TDD während der Bearbeitung der Aufgabenstellungen lassen sich die in Abschnitt 3.3 genannten Verbesserungsansätze in das Lehrkonzept einarbeiten. Beide Praktiken fördern die aktive und kollaborative Bearbeitung der Aufgaben und sorgen somit für die Kompetenzorientierung der Lehrveranstaltung.

5.2.2 Lernfortschrittsvisualisierung und Transparenz

Begleitet wird die Lehrveranstaltung von einem technischen System, das die Lernfortschritte der Studierenden visualisiert und die Transparenz des Lernprozesses erhöht. Als Grundlage dafür dienen die Tests der Aufgabenstellungen, die für eine gute Vergleichbarkeit zwischen den Gruppen sorgen. Das technische System erfasst die Bearbeitungsstände der einzelnen Teams anhand der Anzahl der noch nicht bestehenden Tests einer Aufgabenstellung. Zu jedem Bearbeitungsstand zeichnet das technische System weiterhin den dazugehörigen Zeitpunkt auf. Aus diesen Informationen wird anschließend ein *Burndown Chart* erstellt (siehe Abschnitt 4.4.2), das den Fortschritt der Teams visualisiert.

Bearbeiten beispielsweise zwei Gruppen zur selben Zeit die gleiche Aufgabenstellung, könnte sich ein *Burndown Chart* nach Abbildung 5.1 ergeben. Enthält die Aufgabenstellung fünf einzelne Tests, startet auch das *Burndown Chart* für beide Teams mit fünf noch zu bearbeitenden Tests. Mit dem Bestehen jedes Tests nähert sich der Bearbeitungsfortschritt der horizontalen Achse und der sich inkrementell erweiternde Quelltext der Gesamtlösung an. Bestehen alle Tests, ist die Bearbeitung der Aufgabe abgeschlossen.

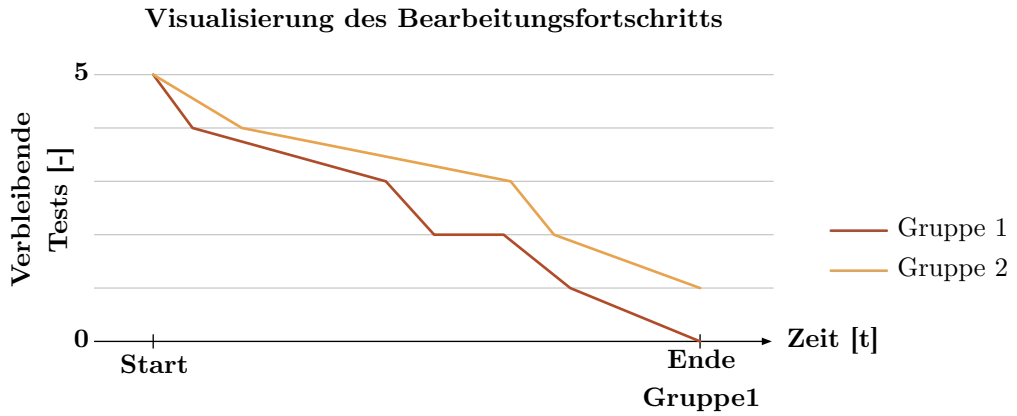

Abbildung 5.1: Schematische Visualisierung des Bearbeitungsfortschritts bzw. Lernfortschritts mittels *Burndown Chart*.

Das technische System beruht auf einem VCS, das auf einem zentralen Server ausgeführt wird und über das Internet erreichbar ist. Nach der Bearbeitung einer Teilaufgabe übertragen die Studierenden ihren neuen Bearbeitungsstand auf den Server.

Eine Kopplung des VCS mit einem Werkzeug für CI (siehe Abschnit 4.5.2) erlaubt die anschließende automatisierte Auswertung der Tests und Aktualisierung des *Burndown Chart*. Eine ausführliche Beschreibung der Funktionalität sowie der Umsetzung des technischen Systems erfolgt in Kapitel 7.

Wie zuvor beschrieben erhöht die Lernfortschrittsvisualisierung anhand des *Burndown Chart* die Transparenz des Lernprozesses, da sie der Lehrperson die Möglichkeit bietet, den Fortschritt der Studierenden zu verfolgen. Des Weiteren unterstützt das technische System die Lehrperson in der Evaluation der Aufgabenstellungen. So kann beispielsweise die aufgebrachte Bearbeitungszeit einzelner Tests Aufschluss darüber geben, ob die verschiedenen Teilaufgaben eine zu hohe Komplexität beinhalten. Gleichzeitig können sich Studierende in Bezug auf ihren Lernfortschritt in der Lerngruppe einordnen und ihre Effizienz im Vergleich zu ihren Kommilitonen evaluieren. Das Offenlegen der Bearbeitungsstände der einzelnen Teams regt die Studierenden zusätzlich dazu an, Unterstützung von anderen Gruppen zu erfragen, die bereits einen fortschrittlicheren Bearbeitungsstand aufweisen. Dadurch entsteht die Situation des *Peer-Tutorings* (siehe Abschnitt 2.1.2) und der Lernprozess intensiviert sich. Durch die öffentliche Erreichbarkeit des technischen Systems über das Internet sind die Teilnehmer weiterhin nicht daran gebunden, die Aufgaben während der Präsenzzeit der Lehrveranstaltung zu bearbeiten. Das System ermöglicht den Gruppen deshalb einen flexiblen und individuellen Zeitplan und fördert daher ebenfalls die Selbstorganisation.

5.2.3 Kontinuierliche Verbesserung des Lernprozesses

Wie in Kapitel 4 beschrieben entstanden die agilen Methoden aus der Notwendigkeit, flexibel auf die sich stetig ändernden Anforderungen in der Softwareentwicklung zu reagieren, die sich häufig aus dem kontinuierlichen Lernprozess während der Umsetzung ergeben. Verwendet man *Scrum* in der Lehre für die Vermittlung von Kompetenzen der Programmierung und betrachtet die durch die Aufgabenstellungen spezifizierte Software als umzusetzendes Produkt, nehmen die Studierenden die Rolle der Entwickler ein. Infolgedessen fällt der kontinuierliche Lernprozess während der Umsetzung mit dem angestrebten Lernprozess der Lehre zusammen. In dieser Metapher nimmt die Lehrperson

die Rolle des *Product Owners* ein, da sie die Anforderungen an die zu implementierende Software festlegt.

Da *Scrum* den kontinuierlichen Lernprozess der Entwickler in den Fokus stellt (siehe Abschnitt 4.4), bietet seine Anwendung in der Lehre die Möglichkeit, sich vollständig auf den Lernprozess der Studierenden ausrichten. Damit verfolgt die Anwendung von *Scrum* eines der höchsten Ziele „Guter Lehre“ (siehe Definition 2.1). Durch den iterativen und inkrementellen Ansatz sowie die geförderte Kollaboration und Kommunikation stellt *Scrum* ausreichend Strukturen für Feedback bereit, um den Lernprozess sowie die *Learning Outcomes* regelmäßig zu reflektieren und kontinuierlich zu optimieren. Aus den zuvor erläuterten Gründen beruht das zukünftige Lehrkonzept auf der Anwendung von *Scrum* für die Bearbeitung der Aufgabenstellungen. Es ergibt sich ein zyklischer Ablauf nach Abbildung 5.2, der dem wöchentlichen Rhythmus der Lehrveranstaltung folgt.

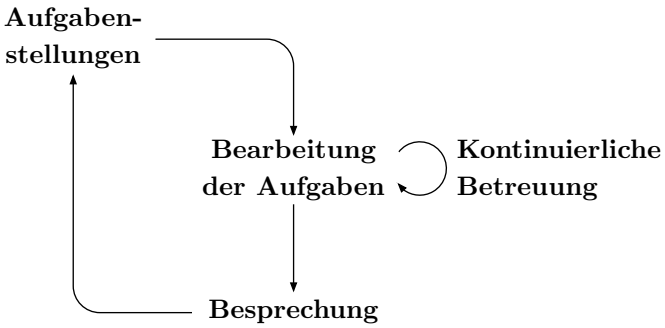

Abbildung 5.2: Die Anwendung von *Scrum* zur Bearbeitung der Aufgabenstellungen.

Wie in der Softwareentwicklung startet der Zyklus mit dem *Backlog*, in dem sich hier die im Rahmen der Lehrveranstaltung zu bearbeitenden Aufgabenstellungen befinden. In jedem Iterationsschritt findet wie in Abschnitt 5.2.1 geschildert die aktive und kollaborative Bearbeitung einer Aufgabenstellung in Gruppenverbänden statt. Die Bearbeitungszeit ist mit der Entwicklungsphase des *Scrum*-Zyklus zu vergleichen und spiegelt daher den Sprint wider. Da die Aufgaben durch ihre Zergliederung in mehrere aufeinander aufbauende Teilaufgaben (siehe Abschnitt 5.2.1) bereits eine ausreichende Planung aufweisen, entfällt der vorausgehende Schritt der Sprintplanung. Im Anschluss an den Sprint erfolgen Besprechungen, in der die Lehrperson mit jeder Gruppe einzeln auf die zuvor bearbeitete Aufgabe zurückblickt. Dadurch werden alle Teilnehmer der Lehrver-

anstaltung aktiviert. Die Studierenden erhalten die Möglichkeit, das gelernte Wissen in eigenen Worten auszudrücken und spezifische Fragen zu stellen. Gleichzeitig analysieren die Lehrperson und die Gruppenmitglieder den Entwicklungsprozess sowie die Kollaboration und erarbeiten individuelle Verbesserungsansätze, die die Teams in dem nächsten Iterationsschritt umsetzen. Da die Besprechungen sowohl die entstandene Software – respektive das Produkt – als auch den Entwicklungsprozess der Gruppen fokussieren, übernehmen sie die Aufgaben des Sprint-*Review* und der Retrospektive.

Während der Bearbeitung der Aufgaben in der zentralen Übung erfolgt eine kontinuierliche Betreuung durch die Lehrperson. Sie steht jederzeit für Fragen und Hilfestellungen zur Verfügung und kann von den Studierenden angesprochen werden. Basierend auf den Informationen der Lernfortschrittsvisualisierung aus Abschnitt 5.2.2 und dem Feedback aus den Besprechungen kann die Lehrperson zum richtigen Zeitpunkt unterstützend eingreifen und wirkungsvolle Impulse setzen. Die dauerhafte Anwesenheit der Lehrperson während der Bearbeitung der Aufgaben erlaubt die intensive Betreuung der Teams, ermöglicht unter anderem die regelmäßige Beseitigung von kleineren technischen Problemen oder Verständnisproblemen und erleichtert dadurch den Entwicklungsprozess der Gruppen. Die kontinuierliche Betreuung durch den Lehrenden ist daher mit dem *Daily Scrum* zu vergleichen.

Neben der Betreuung der Studierenden sowie der moderierenden Rolle in den Besprechungen stellt die Lehrperson des Weiteren den korrekten Ablauf des Zyklus sicher und übernimmt zusätzlich zu den Aufgaben des *Product Owner* ebenfalls die Verpflichtungen des *Scrum Master*.

5.2.4 Stete Reflexion der Lehrkonzepts

Aus den Ansätzen „Guter Lehre“ (siehe Abschnitt 2.1.4) sowie den transferierten Werten des „Agilen Manifests“ (siehe Abschnitt 5.1) geht hervor, dass qualitativ hochwertige und agile Lehrveranstaltungen eine stete Reflexion des Lehrkonzepts vorsehen. Neben dem Lernprozess der Studierenden, sollte die Lehrperson dazu ebenfalls ihr eigenes Handeln fortwährend verbessern. Sie sollte dazu ihre Lehrmethodik und Lehrinhalte auf die aktuelle Lerngruppe des Semesters anpassen und offen für Anregungen seitens der Studie-

renden sein. Dabei geht es in erster Instanz nicht darum, die grundsätzliche Ausrichtung sowie die intendierten Lernziele des Lernangebots zu modifizieren, sondern vielmehr um die Adaption der Lehrstrategie. Unter anderem betrifft diese Adaption die Veränderung des Vortragsstils in der Vorlesung, an die Lerngruppe angepasste Wiederholungen zu Beginn der Lehrveranstaltung oder Ideen für neue Aufgabeninhalte bzw. Aufgabenformate.

Auch für die stete Reflexion des Lehrkonzepts verwendet die zukünftige Lehrveranstaltung den iterativen und inkrementellen Ansatz von *Scrum*. Ein Iterationsschritt umfasst dabei die Durchführung einer Lehrveranstaltung, eine anschließende Retrospektive sowie eine nachfolgende Adaption der Lehrstrategie an das Feedback der Studierenden. Wie der *Scrum*-Zyklus zur Bearbeitung der Aufgabenstellungen ist damit der Zyklus zur Verbesserung des Lehrkonzepts an den wöchentlichen Rhythmus der Lehrveranstaltung gebunden. In Anlehnung an *Scrum* ergibt sich der in Abbildung 5.3 dargestellte Prozess. Im Rahmen dieser Vorgehensweise agiert die Lehrperson wie in Abschnitt 5.1 bereits erwähnt (siehe „Kollaboration von Studierenden und Lehrenden über unflexible Lehrpläne“) als Entwickler, der die Lehrveranstaltung iterativ an die Bedürfnisse der Studierenden anpasst. Die Studierenden nehmen daher die Rolle des *Product Owner* ein und liefern Feedback für die Verbesserung der Lehrveranstaltung. Die moderierende Rolle des *Scrum Master* verbleibt bei der Lehrperson. Zusammen mit dem Prozess der Aufgabenbearbeitung (siehe Abschnitt 5.2.3) ergeben sich somit zwei parallel ablaufende *Scrum*-Zyklen mit umgekehrten Rollenverteilungen.

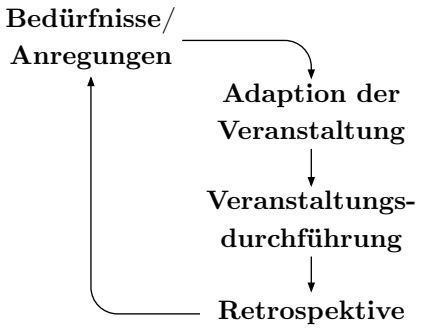

Abbildung 5.3: Die Anwendung von *Scrum* zur kontinuierlichen Adaption der Lehrveranstaltung.

Auch dieser Zyklus startet mit dem *Backlog*, der hier die Bedürfnisse und Anregungen der Studierenden zusammenfasst. Anschließend erfolgt die Anpassung der Lehrveranstaltung. Da in dieser Phase die Planung und die Umsetzung der nächsten Iteration geschieht, beinhaltet sie die Sprintplanung sowie den Sprint des herkömmlichen *Scrum*-Prozesses. Im Anschluss führt die Lehrperson die angepasste Lehrveranstaltung durch. Dieser Schritt ist vergleichbar mit der Präsentation des Produktinkrements und spiegelt daher den Sprint-*Review* wider. Nach der Durchführung jeder Lehrveranstaltung sieht der Zyklus eine Retrospektive vor, in der die Studierenden zusammen mit der Lehrperson die vergangene Veranstaltung analysieren und Verbesserungspotenzial erarbeiten. Das Feedback der Studierenden fließt erneut in den *Backlog* ein und bildet die Grundlage für die nächste Iteration.

Der zuvor geschilderte Prozess verfolgt nicht die Umsetzung eines neuen Produkts, sondern lediglich die Anpassung eines bestehenden an einen neuen bzw. erweiterten Kundenkreis. Der erste Zyklus beginnt daher mit der Vorstellung des bestehenden Lehrkonzepts, das sich an den intendierten Lernzielen orientiert. In einer ersten Lehrveranstaltung lernen die Studierenden die geplante Methodik des zukünftigen Lehrkonzepts kennen. Danach wird die Veranstaltung im Rahmen der Retrospektive analysiert und evaluiert. Das erste Feedback der Studierenden steht bereit, fließt in den *Backlog* ein und der zyklische Prozess zur Adaption der Lehrveranstaltung beginnt.

5.3 Zeitliche Konzeption der Lehrveranstaltung

Sowohl die kontinuierliche Optimierung des Lernprozesses (siehe Abschnitt 5.2.3) als auch die anhaltende Reflexion und Adaption der Lehrveranstaltung (siehe Abschnitt 5.2.4) sieht eine Zusammenkunft von Studierenden und der Lehrperson vor, in der gegenseitiges Feedback eingefordert und ausgetauscht wird. Findet der Austausch des Feedbacks beider Zyklen in einer gemeinsamen Besprechung statt, lassen sich die Prozesse aufgrund des gleichen Rhythmus (wöchentlich) analog zu Abbildung 5.4 synchronisieren. Während die Studierenden sich Woche für Woche mit neuen Aufgabenstellungen befassen und ihren kollaborativen Arbeitsprozess mit den Kommilitonen stetig verbes-

sern, passt die Lehrperson gleichzeitig Schritt für Schritt ihre Lehrstrategie an die aktuelle Lerngruppe des Semestern an. Für die Synchronisierung beider Zyklen wird in dem zukünftigen Lehrkonzept die Retrospektive aus dem Prozess zur Verbesserung der Lehrveranstaltung mit dem Gruppengespräch zur Analyse und Optimierung des Entwicklungsprozesses zusammengelegt.

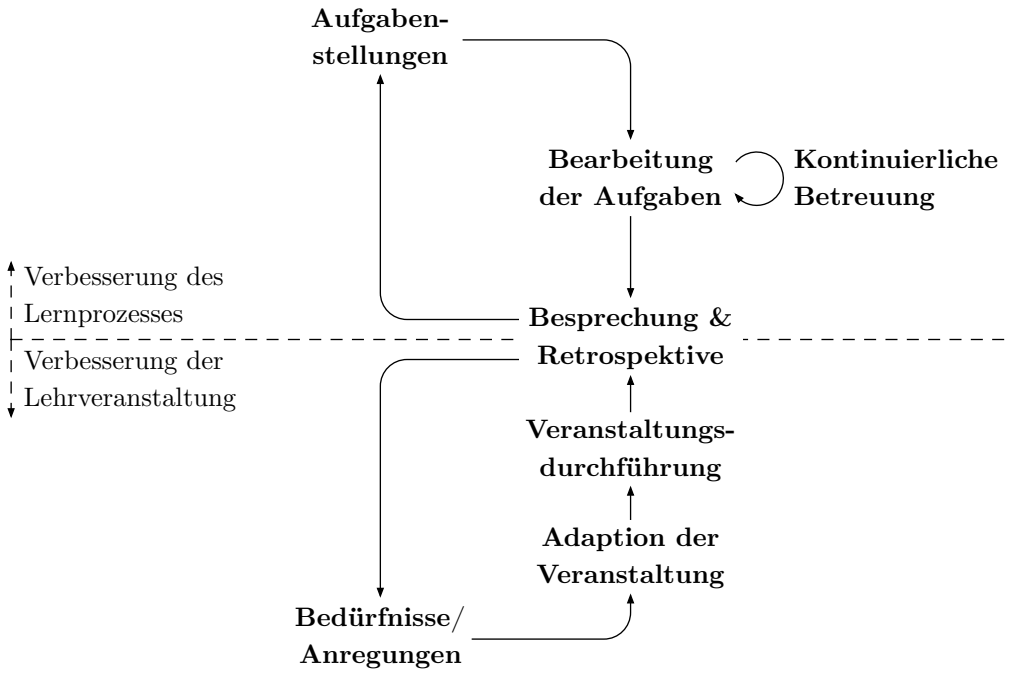

Abbildung 5.4: Parallele der *Scrum*-Zyklen des zukünftigen Lehrkonzepts.

Wie in Abschnitt 5.2.3 beschrieben, erfordert dieses Vorgehen ein isoliertes Gespräch mit jeder einzelnen Gruppe am Ende eines Zyklus. Daraus ergibt sich ein hoher zeitlicher Aufwand. Im Schnitt weist die Veranstaltung zwischen 30 bis 35 teilnehmenden Studierenden auf, die aufgrund der paarweisen Programmierung in ca. 15 bis 18 Gruppen eingeteilt werden. Zur Bewältigung des hohen Pensums an Gruppengesprächen würde ein korrekter Ablauf beider Zyklen, in denen das Feedback des vorherigen Durchgangs die Grundlage für den nächsten Iterationsschritt bildet, die Einführung einer weiteren wöchentlichen Veranstaltung bedeuten, die nur auf den Austausch des Feedbacks ausgerichtet ist. Da die Gespräche jedoch im Gruppenverband stattfinden, könnte die Lehrperson in dieser Veranstaltung nur eine Gruppe gleichzeitig betreuen. Für die an-

deren Gruppen – zumindest für diejenigen, die erst gegen Ende der Veranstaltung zum Gespräch gebeten werden – ergibt sich ein zeitlicher Leerlauf. Um sowohl die zeitlichen Ressourcen der Studierenden zu schonen als auch den erhöhten Zeitaufwand der Lehrperson infolge einer weiteren Veranstaltung zu verhindern, werden die Feedbackgespräche in der zentralen Übung des nachfolgenden Zyklus untergebracht. So können diejenigen Gruppen, die sich nicht in einem Gespräch mit der Lehrperson befinden, bereits mit dem Bearbeiten der nachfolgenden Aufgabe beginnen.

Ein Nachteil der zuvor beschriebenen Lösung liegt in dem verzögerten Feedback für die Studierenden, dass sie erst während der Bearbeitung der nächsten Aufgabe, also im Sprint des darauffolgenden Zyklus, erhalten. Das Feedback wird damit erst im Laufe des Folgezyklus eingearbeitet. Auch können die Kontextsprünge zwischen der zu besprechenden und der momentan zu bearbeitenden Aufgabe die Studierenden verunsichern und die Durchführung des Feedbackgesprächs erschweren. Zur zeitlichen Optimierung der Lehrveranstaltung werden die zuvor genannten Nachteile im Rahmen dieser Arbeit toleriert. Ergeben sich in der anschließenden Durchführung des Lehrkonzepts dadurch gravierende Auswirkungen auf die Qualität der Lehrveranstaltung, müssen Verbesserungsansätze erarbeitet werden. Eine Möglichkeit dafür wären beispielsweise festgelegte wöchentliche Termine für die einzelnen Gruppen mit der Lehrperson, die abseits von der Lehrveranstaltung stattfinden.

Für die Vorlesung und Übung sind wöchentlich insgesamt 150 Minuten vorgesehen, die sich analog zu Abbildung 5.5 in verschiedene Abschnitte einteilen. Die Veranstaltung beginnt mit einer ca. 50-minütigen Vorlesung, in der das notwendige Wissen für die Bearbeitung der nächsten Aufgabe vermittelt wird. In den darauffolgenden zehn Minuten stellt die Lehrperson die nächste Aufgabe vor und bespricht sie mit den Studierenden. Die verbleibenden 90 Minuten beinhalten die zentrale Übung, in der die Studierenden die Aufgaben gemäß Abschnitt 5.2.1 aktiv und kollaborativ bearbeiten. Währenddessen führt die Lehrperson die vorgesehenen Feedbackgespräche mit den einzelnen Gruppen. Zur kontinuierlichen Betreuung der Bearbeitungsphase der übrigen Studierenden wird die zentrale Übung von weiteren Lehrenden (wie. z.B. wissenschaftliche Hilfskräfte) unterstützt.

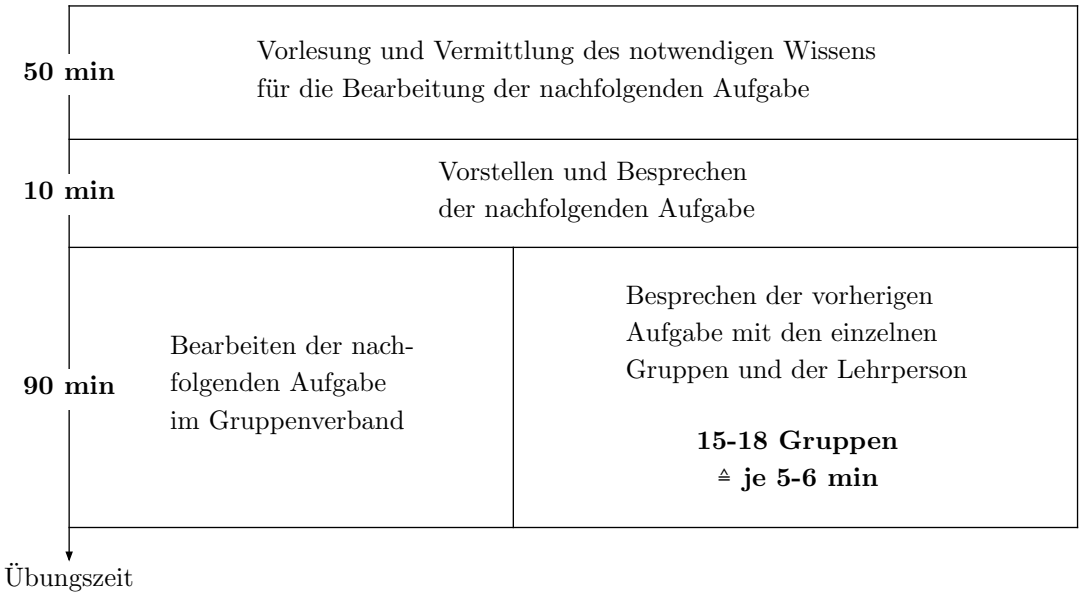

Abbildung 5.5: Zeitliche Konzeption der zukünftigen Lehrveranstaltung.

KAPITEL 6

Spezifikation von Programmieraufgaben

Um den Studierenden die schrittweise Bearbeitung einer komplexen Aufgabenstellung zu ermöglichen, beruht das zukünftige Lehrkonzept auf dem Paradigma der testgetriebenen Entwicklung. Wie in Abschnitt 5.2.1 beschrieben werden dazu Aufgabenstellungen mit vorgefertigten Tests ausgehändigt, die die Studierenden durch Implementierung von Produktionscode sukzessiv zum Bestehen bringen. Die Tests dienen als Spezifikationen, die das angestrebte Verhalten des Produktioncodes beschreiben. Sie enthalten konkrete Testfälle, die den Produktionscode einschränken und ihn in die Richtung der zu implementierenden Gesamtlösung lenken. Aufgrund der geringen Komplexität der einzelnen Testfälle soll diese Art der Aufgabenstellung die kognitive Entlastung der Studierenden hervorrufen und das eigenständige, aktive und kooperative Arbeiten fördern. Ein maßgeblicher Faktor für den Erfolg des Lehrkonzepts liegt in der Implementierungsweise der Tests. Sind diese beispielsweise nur schwer verständlich oder zu komplex, erschweren sie die Bearbeitung der Aufgaben und verfehlen die angestrebte selbstständige Programmierung der Studierenden.

Das nachfolgende Kapitel beschreibt, wie die Aufgabenstellungen zu spezifizieren bzw. die Tests zu implementieren sind, um die Erfolgschancen des zukünftigen Lehrkonzepts zu erhöhen. Zunächst werden dazu einige ausgewählte Übungsaufgaben vorgestellt. Anschließend wird die Funktionsweise des *Testing Frameworks* JUnit[1] erläutert, das in dem zukünftigen Lehrkonzept für die technische Umsetzung der Tests verwendet wird. In einem nächsten Schritt werden dann Ansätze und Eigenschaften beschrieben, die es bei der Umsetzung der Tests zu berücksichtigen gilt. Die Umsetzung dieser Eigenschaften wird beispielhaft anhand von Auszügen der erläuterten Übungsaufgaben verdeutlicht.

6.1 Vorstellung ausgewählter Übungsaufgaben

Die Vorlesung „Algorithmen und Programmieren“ vermittelt die Kompetenzen der Programmierung in der Programmiersprache Java. Dazu verfolgt sie die in Abschnitt 3.1.1 beschriebenen Lernziele, die sich auf die Vermittlung der selbstständigen Programmierung sowie die Implementierung von komplexen Algorithmen und Datenstrukturen beziehen. Um die Studierenden auf die Bearbeitung der komplexen Übungsaufgaben in der zweiten Semesterhälfte vorzubereiten, sollte ihnen ausreichend Gelegenheit geboten werden, die grundlegenden Programmierkenntnisse zu erwerben und zu verinnerlichen. Zu Beginn des Semesters sieht das zukünftige Lehrkonzept daher vor, einfache Aufgaben zu bearbeiten, die sich auf die Vermittlung dieser Grundkenntnisse konzentrieren. Eine kontinuierliche Steigerung in der Komplexität der Übungsaufgaben versetzt die Studierenden in die Lage, gegen Ende des Semesters die geforderten mäßig komplexen Ingenieurprobleme zu lösen sowie die ausgewählten Algorithmen und Datenstukturen zu implementieren.

Der nachfolgende Abschnitt stellt einige ausgewählte Übungsaufgaben vor. Dazu wird die zu implementierende Logik jeder Aufgabe kurz erklärt und dargelegt, welche Ziele die einzelnen Aufgabenstellungen verfolgen. Des Weiteren verdeutlichen die hier vorgestellten Übungsaufgaben die Komplexitätssteigerung des Semesters. Im einzelnen werden

[1] Bei JUnit handelt es sich um ein *Testing Framework*, das ursprünglich unter anderem von Kent Beck und Erich Gamma entwickelt wurde. Dokumentation verfügbar unter https://junit.org/junit5; Abgerufen am 18. Oktober 2018.

die Aufgaben *MathUtilities*, *LeapYear* und *BankAccount* beschrieben. Im Rahmen dieser Arbeit wurden weitere Übungsaufgaben umgesetzt, die aufgrund des Umfangs nicht weiter aufgeführt werden. Die ausführliche Spezifikation der Aufgabenstellung sowie die exemplarische Bearbeitung der Übungsaufgaben lassen sich dem beigelegten Datenträger entnehmen.

6.1.1 MathUtilities

Eine der wichtigsten Grundlagen der Programmierung liegt in dem Konzept, Quelltext durch Methoden zu strukturieren und aufzurufen. Die bisherige Lehrveranstaltung zeigte, dass insbesondere bei der Übergabe von Parametern anfänglich oft Verständnisprobleme der Studierenden auftreten. Zu Beginn des Semesters bedarf es deswegen Aufgaben, die es durch geringe Komplexität erlauben, die Studierenden langsam an die Implementierung und Verwendung von Methoden heranzuführen. Da für Novizen auch die Objektorientierung ein komplexes Paradigma darstellt, sollte sich hierbei zunächst auf statische Methoden beschränkt werden. Die Objektorientierung wird im Sinne der Komplexitätssteigerung erst zu einem späteren Zeitpunkt des Semesters thematisiert (siehe Aufgabe *BankAccount*, Abschnitt 6.1.3).

Ein häufiges Anwendungsgebiet der Programmierung liegt in der Automatisierung von Berechnungen, die in Programmen durch mathematische Ausdrücke repräsentiert werden. Ein mathematischer Ausdruck besteht dabei aus Zahlen, die durch Operatoren miteinander verknüpft sind. Um den Studierenden die Verwendung von mathematischen Ausdrücken zu vermitteln, fokussiert die Aufgabe *MathUtilities* den Umgang mit den geläufigsten numerischen Basisdatentypen `int` und `double` sowie deren Verknüpfung mit den von Java bereitgestellten mathematischen Operatoren. Dazu sieht die Aufgabe die Implementierung einfachster Methoden vor, die beispielsweise die vier Grundrechenarten und die Division mit Rest (*Modulo*) betreffen. Werden diese Methoden als statische Methoden umgesetzt, bietet *MathUtilities* durch die geringe Komplexität in der zu implementierenden Logik weiterhin eine gute Gelegenheit, das Konzept der Methoden zu verstehen und zu verinnerlichen. Im Rahmen der Aufgabe setzen die Studierenden die in Abbildung 6.1 gezeigte Klasse um. Darüber hinaus implementieren sie alle darge-

stellten Methoden zusätzlich für die Verwendung mit Fließkommazahlen, die für eine bessere Übersichtlichkeit in der Abbildung vernachlässigt werden. Anstatt des Basisdatentyps `int` nehmen diese Methoden dafür die Übergabeparameter als Zahlen des Typs `double` entgegen und geben diesen ebenfalls zurück. Die doppelte Belegung der Methodennamen thematisiert zugleich die Überladung von Methoden und die Bestandteile der Methodensignatur.

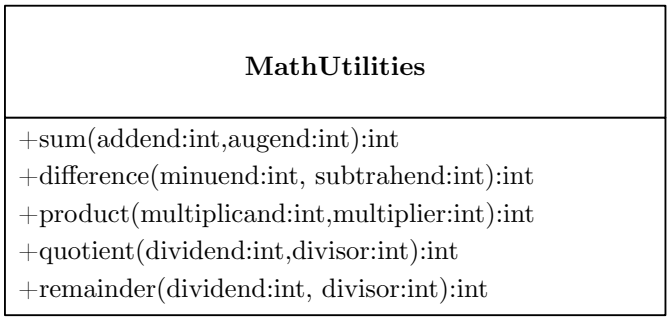

Abbildung 6.1: UML-Diagramm der Klasse `MathUtilities`.

6.1.2 LeapYear

Neben mathematischen Ausdrücken spielen auch boolsche Ausdrücke eine große Rolle in der Programmierung, da sie unter anderem auch das Entscheidungskriterium für Verzweigungen (`if-else`) in Algorithmen bilden. Die Auswertung eines boolschen Ausdrucks liefert entweder den Wert „wahr“ (`true`) oder „falsch“ (`false`), auf dessen Basis der Algorithmus entscheidet. Bei boolschen Ausdrücken handelt es sich meistens um Vergleiche zwischen unterschiedlichen Werten bzw. Variablen des gleichen Typs. Zu diesem Zweck stellt Java Vergleichsoperatoren bereit, die die zu vergleichenden Werte bzw. Variablen verknüpfen.

Entscheidungen eines Algorithmus, die von mehreren Kriterien gleichzeitig abhängen, ergeben sich beispielsweise durch ineinander verschachtelte oder sich wiederholende Verzweigungen, die die unterschiedlichen Kriterien nacheinander überprüfen. Es ergeben sich meistens komplexe Strukturen des Algorithmus, die die Verständlichkeit des Quelltexts herabsetzen. Eine weitere Möglichkeit, mehrere Entscheidungskriterien zu

berücksichtigen, bieten die logischen Operatoren, die die boolschen Ausdrücke miteinander verknüpfen und die gemeinsame Betrachtung der Kriterien ermöglichen. Die Verwendung von logischen Operatoren erlauben es deshalb, die verschachtelten und komplexen Verzweigungsstrukturen zu vereinfachen und erhöhen die Verständlichkeit des Quelltexts.

Damit die Studierenden erlernen, Algorithmen mit möglichst einfachen Verzweigungsstrukturen zu implementieren, fokussiert die Aufgabe *LeapYear* die Anwendung von logischen Operatoren und `if-else`-Anweisungen. Im Rahmen dieser Aufgabe setzen die Studierenden die in Listing 6.2 gezeigte Klasse `LeapYear` um, die die öffentliche Methode `isLeapYear` enthält. Diese Methode berechnet, ob es sich bei einer übergebenen Jahreszahl (`year`) um ein Schaltjahr handelt und gibt entsprechend den boolschen Wert `true` oder `false` zurück. Da die Berechnung eines Schaltjahres mehreren Bedingungen zugleich berücksichtigt, bietet diese Aufgabe die optimale Gelegenheit die logischen Operatoren im Zusammenhang mit Verzweigungen anzuwenden. Wie auch in der Aufgabe *MathUtilities* wird die Methode `isLeapYear` als statische Methode umgesetzt.

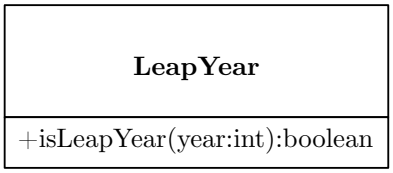

Abbildung 6.2: UML-Diagramm der Klasse `LeapYear`.

Neben den Aufgaben *MathUtilities* und *LeapYear* existieren weitere Übungsaufgaben, die die grundlegenden Programmierkenntnisse fokussieren. So vermittelt beispielsweise die Aufgabe *Comparison* die zuvor erwähnten mathematischen Vergleiche. Weiterhin sorgt die Aufgabe *Power*, in der eine Potenzfunktion implementiert wird, für die Anwendung von Schleifen.

6.1.3 BankAccount

Nach der erfolgreichen Vermittlung der grundlegenden Programmierkenntnisse steigern sich die Übungsaufgaben weiter, indem sie die Objektorientierung adressieren. Ein leich-

ten Einstieg dafür bietet die Aufgabe *BankAccount*, in der die Studierenden die Logik eines Kontos für das Abheben und Einzahlen von Geldbeträgen implementieren. Es wird die in Abbildung 6.3 dargestellte Klasse umgesetzt.

Um die Steigerung der zu vermittelnden Inhalte möglichst gering zu halten, weist die Klasse `BankAccount` keine Beziehungen zu anderen Klassen auf. Gleichzeitig kennzeichnet die zu implementierende Logik eine geringe Komplexität, sodass sich die Studierenden vollständig auf die Verwendung von Attributen, das Instanziieren von Klassen, das Aufrufen von Instanzmethoden sowie die Auswirkungen von Sichbarkeiten konzentrieren können.

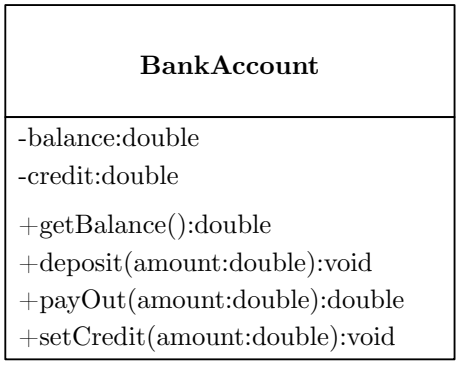

Abbildung 6.3: UML-Diagramm der Klasse `BankAccount`.

An die Aufgabe *BankAccount* schließen sich weitere Übungsaufgaben an, die die Vermittlung weiterer Aspekte der Objektorientierung fokussieren. Beispielsweise sehen die Aufgaben die Umsetzung gleichrangiger Aufruferbeziehungen (*Point*, *Interval*) oder Vererbungsbeziehungen (*Shape*) vor. In weiteren noch umzusetzenden Aufgaben ist auch die Vermittlung der Polymorphie geplant.

Nach den zuvor beschriebenen Aufgaben sollten die Studierenden die grundlegenden Kenntnisse der Programmierung sowie der Objektorientierung selbstständig anwenden können und in der Lage sein, die in den Lernzielen geforderten komplexen Algorithmen und Datenstrukturen zu implementieren. In der zweiten Semesterhälfte werden daher Aufgaben bearbeitet, die das zweite und dritte Lernziel verfolgen (siehe Abschnitt 3.1.1). Im Rahmen dieser Arbeit wurde bereits die Aufgabenstellungen *ArrayList* umgesetzt. Zukünftig sollten weitere Aufgabenstellungen entstehen, die die Implementierung

von einfach oder doppelt verketteten Listen (*Singly/Doubly LinkedList*), dünnbesetzte Matrizen (*Sparse Matrix*) sowie die in Abschnitt 3.1.1 erwähnten Sortieralgorithmen behandeln.

6.2 JUnit – Testing Framework

Wie Abschnitt 4.5.1 erklärt, basiert die testgetriebene Entwicklung auf der vorgezogenen Implementierung von Tests, die den Produktionscode überprüfen und das Systemverhalten kontinuierlich verifizieren. Auch die Übung des zukünftigen Lehrkonzepts sieht vor, dass sich Studierende während der Bearbeitung der Aufgaben an zuvor definierten Tests orientieren, die regelmäßiges Feedback über die Fehlerfreiheit des implementierten Produktionscodes liefern. Damit den Entwicklern bzw. den Studierenden dieses Feedback kontinuierlich zur Verfügung steht, sollten die Tests eine einfache und schnelle Ausführbarkeit aufweisen. Zu diesem Zweck existieren die sogenannten *Testing Frameworks*, die die Ausführung der Tests automatisieren und erleichtern. Da die betrachtete Lehrveranstaltung die Programmierung mit der Sprache Java vorsieht, bedarf es eines *Testing Frameworks*, das ebenfalls auf dieser Sprache beruht. Das zukünftige Lehrkonzept verwendet deshalb das *Testing Framework* JUnit, das das Verfassen und die Automatisierung von Tests in Java ermöglicht. Unter der Verwendung von JUnit werden die Tests als gewöhnliche Methoden umgesetzt, in denen die zu verifizierende Systemfunktionalität ausgeführt und überprüft wird. Listing 6.1 zeigt beispielhaft die Definition der Testmethode `testMethod`.

Für die automatisierte Ausführung stellt JUnit eine ausführbare Testkonfiguration bereit, die alle Testmethoden eigenständig aufspürt und aufruft. Die Testmethoden müssen dazu wie in Listing 6.1 dargestellt mit der *Annotation* (dt: „Anmerkung“) `@Test` versehen werden. Daraus ergibt sich ebenfalls die Notwendigkeit des `imports` in der ersten Zeile. Aufgrund der rein objektorientierten Ausrichtung von Java werden die Testmethoden in eine Testklasse (hier `TestClass`) eingebettet. Sie werden als Instanzmethoden implementiert, die üblicherweise den Rückgabetyp `void` aufweisen und keine Übergabeparameter entgegennehmen. Da JUnit die Testklassen und Testmethoden auch

```
1 import org.junit.jupiter.api.Test;
  class TestClass {
3   @Test
    void testMethod() {
5     // ...test body...
    }
7 }
```

Listing 6.1: Beispielhafte Definition einer Testklasse mit Testmethode unter Verwendung von JUnit.

mit der standardmäßigen Sichtbarkeit *Package-Private* von Java identifizieren kann, bedarf es keiner expliziten Angabe von Sichtbarkeiten. Zur Ausführung der Tests instanziiert JUnit die Testklassen und ruft die identifizierten Testmethoden auf.

Im Mittelpunkt eines Tests steht die Verifizierung der Systemfunktionalität. In diesem Zusammenhang gelten im Folgenden die von Osherove [27] definierten Bezeichnungen des *System Under Test* (SUT) und der *Unit of Work* (UoW). Das SUT spiegelt dabei das in einem Test betrachtete System wider, das einer Klasse zusammen mit ihren öffentlichen Abhängigkeiten entspricht. Die UoW bezeichnet hingegen die zu überprüfende Funktionalität, die durch eine öffentliche Methode des SUT ausgelöst wird und ein eindeutiges Endergebnis hervorruft. Das Endergebnis einer UoW äußert sich beispielsweise in dem Rückgabewert der aufgerufenen Methode.

Zur Verifizierung der UoW ruft eine Testmethode die entsprechende Methode des SUT auf und überprüft anschließend das Endergebnis. Dafür stellt JUnit die statischen *Assertion*-Methoden zur Verfügung, die es ermöglichen, das tatsächliche Endergebnis einem erwarteten Endergebnis gegenüberzustellen. Unter den geläufigsten dieser *Assertion*-Methoden befinden sich `assertEquals`, `assertTrue` und `assertFalse`. Während `assertTrue` und `assertFalse` die Werte von boolschen Ausdrücken verifizieren, prüft `assertEquals` zwei übergebene Parameter auf Gleichheit. Listing 6.2 zeigt beispielhaft die Verwendung von `assertEquals`. Als SUT dient die Klasse `MathUtilities`, die die statische Methode `sum` für das Summieren von zwei ganzen Zahlen beinhaltet (siehe Aufgabe *MathUtilities*, Abschnitt 6.1.1). Der Aufruf dieser Methode löst die UoW

aus und gibt das Endergebnis – in diesem Fall die Summe der Zahlen – zurück. Bei dem Rückgabewert handelt es sich um das tatsächliche Endergebnis der UoW, das die Methode `assertEquals` dem erwarteten Endergebnis gegenüberstellt. Da der Test das Summieren der Zahlen 1 und 2 vorsieht, wird ein Ergebnis von 3 erwartet. Der Zugriff auf die *Assertion*-Methoden wird durch den `import` in der zweiten Zeile ermöglicht. Die Implementierung eines Tests unter Verwendung von JUnit bedarf immer beider `imports`. Aus Gründen der Übersichtlichkeit werden diese in den nachfolgenden Listings vernachlässigt.

```
1 import org.junit.jupiter.api.Test;
  import static org.junit.jupiter.api.Assertions.*;
3
  class MathUtilitiesTest {
5   @Test
    void testSumOf1And2_ShouldBe3() {
7     assertEquals(3, MathUtilities.sum(1, 2));
    }
9 }
```

Listing 6.2: Anwendung der von JUnit bereitgestellten *Assertion*-Methoden zur Verifizierung des Endergebnisses am Beispiel der Aufgabe *MathUtilities*.

Basieren mehrere Tests auf den gleichen Daten oder beinhalten gleiche Logik, können diese zur Redundanzvermeidung als Attribute oder Methoden in der gemeinsamen Testklasse implementiert werden. Da jede Testmethode entsprechend der Objektorientierung sowohl auf die Attribute als auch auf die Methoden zugreifen kann, stellt die Testklasse jeder enthaltenen Testmethode eine gewisse Funktionalität bereit. Sie hat somit den Charakter einer „Testvorrichtung“, die Meszaros [25] als *Test Fixture* bezeichnet. Listing 6.3 zeigt beispielhaft die Umsetzung einer Testklasse mit Attributen und Methoden sowie deren Verwendung in einer Testmethode. Die Testklasse `TestClass` enthält das Attribut `data` sowie die Methode `helperMethod`, die in der Testmethode `testMethod` verwendet werden. Damit sich die unterschiedlichen Testfälle nicht gegenseitig beeinflussen, insbesondere durch die Manipulation der Attribute, führt JUnit jede Testmethode mit einer

neuen Instanz der Testklasse aus. Diese Art von *Test Fixtures* bezeichnet Meszaros [25] als *Transient Fresh Fixture*.

Das *Testing Framework* JUnit bietet weitere Funktionalitäten wie z.B. automatisierte Methoden die vor oder nach den Testmethoden ablaufen (siehe `@BeforeEach`, `@AfterEach`, etc.). Da diese Funktionalitäten im Rahmen der Arbeit nicht von Bedeutung sind, werden sie im weiteren Verlauf vernachlässigt.

```
class TestClass {
  private int data; //shared data across the tests

  private void helperMethod(int number) {
    //shared functionality across tests
  }

  @Test
  void testMethod() {
    // ...
    helperMethod(data);
    // ...
  }
}
```

Listing 6.3: Verwendung von Methoden und Attributen einer Testklasse in einer Testmethode.

6.3 Umzusetzende Eigenschaften der Tests

Aufgrund der Funktionsweise des *Testing Frameworks* JUnit unterliegen die Testmethoden einigen Rahmenbedingungen, die die Umsetzung der Tests beeinflussen. So sind die Testmethoden wie in Abschnitt 6.2 geschildert z.B. an die objektorientierte Programmierung, die Kennzeichnung durch *Annotations* und die Verwendung der bereitgestellten *Assertion*-Methoden gebunden. Dennoch hat die Lehrperson einen hohen Einfluss auf die Implementierungsweise der Testmethoden. Durch übersichtliche und leicht verständliche

Teststrukturen sowie Tests, die kontinuierlich aufeinander aufbauen, erstellt die Lehrperson beispielsweise Aufgabenstellungen, die unterbrechungsfreies Arbeiten der Studierenden fördern und sich in das zukünftige Lehrkonzept eingliedern. Um die Erfolgschancen des zukünftigen Lehrkonzepts zu steigern, sollten die Aufgabenstellungen bzw. die Tests einigen Anforderungen genügen. Zu den wichtigsten Eigenschaften, die die Tests für die Anwendung in der Lehre aufweisen müssen, gehören unter anderem

- Lesbarkeit und Verständlichkeit,
- eine konsistente Formulierung,
- die Kontinuität aufeinanderfolgender Tests sowie
- eine hohe Wartbarkeit.

Der nachfolgende Abschnitt erläutert die zuvor genannten Merkmale sowie ihre Bedeutung für den Erfolg des Lehrkonzepts. Die Umsetzung von Tests, die diesen Anforderungen entsprechen, wird anhand von einzelnen Testmethoden aus den in Abschnitt 6.1 vorgestellten Übungsaufgaben verdeutlicht.

6.3.1 Lesbarkeit und Verständlichkeit

Eine der grundlegenden Voraussetzung für das selbstständige Bearbeiten der Aufgabenstellungen liegt darin, dass Studierende die Tests lesen und verstehen können. Führt die Implementierungsweise der Testmethoden zu Verständnisproblemen, erschwert sich die in Abschnitt 5.2.1 beschriebene selbstständige Programmierung und das Lehrkonzept scheitert. Daher sollte die Lehrperson bei der Umsetzung der Aufgabenstellungen besonders darauf achten, ihre Intentionen in den Tests möglichst einfach und klar zu kommunizieren. Infolgedessen können die angestrebte Funktionsweise des SUT sowie die Funktionalität des zu implementierenden Produktionscodes schnell und problemlos verstanden werden.

Betrachtet man Listing 6.2 (siehe Seite 81) unter den Aspekten Lesbarkeit und Verständlichkeit, fällt auf, dass eine einzelne Zeile die gesamte Logik des Tests enthält. Dabei handelt es sich um eine schlanke und elegante Implementierung, die allerdings

einen sehr hohen Informationsgehalt aufweist und die Semantik der einzelnen Bestandteile verschleiert. Auch wenn der Kontext des Tests impliziert, dass die Methode `sum` die UoW, die Werte 1 und 2 die Summanden der Addition (*Addend* und *Augend*) und der Wert 3 das erwartete Endergebnis darstellen, ist das für einen Novizen der Programmierung nur schwerlich zu erkennen. Der verschachtelte Methodenaufruf der `sum`-Methode in der `assertEquals`-Methode wirkt sich weiterhin negativ auf die Verständlichkeit und die Lesbarkeit des Tests aus. Anstatt die Tests entsprechend Listing 6.2 in einer möglichst schlanken Form umzusetzen, sollte die Lehrperson zur kognitiven Entlastung der Studierenden eine explizite Implementierung wählen, die die Logik auf mehrere Zeilen aufteilt und die Intention der Lehrperson verdeutlicht. Jede Zeile sollte dabei nur einen geringen Informationsgehalt aufweisen und die Semantik durch entsprechende Variablen- oder Methodennamen erhöhen.

Listing 6.4 zeigt eine alternative Implementierung des Tests. Es werden zunächst die Variablen `addend` und `augend` mit den Werten 1 und 2 definiert. Erst in einer nachfolgenden Zeile übergibt der Test die Summanden an die `sum`-Methode und fängt den Rückgabewert – das tatsächliche Endergebnis der UoW – in der Variablen `actual` auf. Anschließend wird in einer eigenen Zeile die Variable `expected` definiert, die das erwartete Endergebnis repräsentiert und den Wert 3 enthält. Danach stellt die `assertEquals`-Methode die Variablen `expected` und `actual` gegenüber. Jede Zeile dieser Implementierung enthält nur

```
class MathUtilitiesTest {
  @Test
  void testSumOf1And2_ShouldBe3() {
    int addend = 1;
    int augend = 2;
    int actual = MathUtilities.sum(addend, augend);
    int expected = 3;
    assertEquals(expected, actual);
  }
}
```

Listing 6.4: Explizite Implementierung der Testmethode aus Listing 6.2 durch Aufteilung der Testlogik in mehrere Zeilen.

einen gewissen Anteil der gesamten Testlogik und gibt mit aussagekräftigen Variablen- und Methodennamen Aufschluss über die Semantik der einzelnen Bestandteile.

Simple Umstrukturierungen erhöhen weiterhin den semantischen Informationsgehalt der Testmethoden. So lassen sich die einzelnen Zeilen der Tests unterschiedlichen Verantwortungen zuordnen, die durch Leerzeilen optisch voneinander getrennt sind. Nach Beck [5] teilt sich die Struktur eines Tests in die drei Bereiche *Arrange*, *Act* und *Assert* auf. Der erste Bereich, das *Arrange*, dient dazu, das SUT zu instanziieren, es in einen definierten Ausgangszustand zu versetzen und alle Daten bzw. Objekte, die für die Ausführung der UoW benötigt werden, zu erstellen. In dem *Act* des Tests findet der Aufruf der UoW statt. Das anschließende *Assert* übernimmt die Verifizierung des Endergebnisses unter Verwendung der von JUnit bereitgestellten *Assertion*-Methoden.

Wendet man die von Beck [5] getroffene Einteilung auf die in Listing 6.4 dargestellte Testmethode an, ergibt sich eine Implementierung nach Listing 6.5. Da es sich bei der UoW in diesem Fall um eine statische Methode handelt, die nicht von dem Zustand eines Objekts abhängt, bedarf das *Arrange* der Testmethode weder die Instanziierung

```
class MathUtilitiesTest {
  @Test
  void IfAddendIs1AndAugendIs2__TheSum__ShouldBe3() {
    //Arrange
    int addend = 1;
    int augend = 2;

    //Act
    int actual = MathUtilities.sum(addend, augend);

    //Assert
    int expected = 3;
    assertEquals(expected, actual);
  }
}
```

Listing 6.5: Einteilung der Testlogik in *Arrange*, *Act* und *Assert* nach Beck [5].

der Klasse `MathUtilities`, noch stellt es einen definierten Ausgangszustand her. Das *Arrange* des Tests enthält hier lediglich die Definition der Variablen `addend` und `augend`, die für den Aufruf der UoW bzw. der `sum`-Methode in dem anschließenden *Act* benötigt werden. Die Definition des erwarteten Endergebnis und der Aufruf der `assertEquals`-Methode gehören dem *Assert* der Testmethode an.

Die optische Trennung der drei Verantwortungen führt zu einer übersichtlichen Struktur, die die Verständlichkeit der Tests erhöhen, sodass Studierende die Intention der Lehrperson sowie die angestrebte Funktionalität problemlos erfassen und den Test bearbeiten können. Ein angepasster Name der Testmethode unterstützt die Studierenden weiterhin, da er die verwendeten Variablen- und Methodennamen sowie das erwartete Ergebnis in einem gut lesbaren Satz ausdrückt, der die Teststruktur wiedergibt.

Aus den zuvor erläuterten Zusammenhängen ergibt sich, dass die Lehrperson bei der Implementierung der Testmethoden zugunsten der Lesbarkeit und Verständlichkeit die nachfolgenden Aspekte beachten sollte:

- Aufteilung der Testlogik in mehrere Zeilen mit geringem Informationsgehalt
- Verwenden von aussagekräftigen Variablen- und Methodennamen
- Einteilung der Testmethode in *Arrange*, *Act* und *Assert*

Des Weiteren sollten die Testmethoden keine Verzweigungen und Schleifen aufweisen, da diese die Komplexität erhöhen und für eine intransparente Teststruktur sorgen. Darüber hinaus sind gut auszusprechende Variablennamen zu verwenden, da schwierig auszusprechende Abkürzungen (wie z.B. `agd` statt `augend`) den Lesefluss stören und dadurch die Lesbarkeit sowie die Verständlichkeit der Tests herabsetzen. Ebenfalls sind verschachtelte Methodenaufrufe zu vermeiden [27].

6.3.2 Konsistente Formulierung

Wie Abschnitt 2.2.3 beschreibt, lässt sich der Lernerfolg durch die Konsistenz der Lernumgebung verbessern. So bietet beispielsweise die festgelegte Teststruktur mit den in Abschnitt 6.3.1 beschriebenen Bereichen nach Beck [5] einen hohen Wiedererkennungswert, der es den Studierenden ermöglicht, sich in jeder Aufgabe mühelos zurechtzufinden.

Auch sollten die verwendeten Vorlesungs- und Übungsunterlagen eine hohe Konsistenz aufweisen. Wie die didaktischen Elemente (LZ, LA und LÜ) sollte die Lehrperson daher ebenfalls die Aufgabenstellungen der Übungsaufgaben mit den Materialien der Vorlesung abstimmen.

Die Unterlagen des bisherigen Lehrkonzepts zeigen, dass bereits kleinste Diskrepanzen in Vorlesung und Übung zur Verwirrung der Studierenden führen und die selbstständige Bearbeitung der Aufgabenstellung erschweren. Aus diesem Grund sollte die Lehrperson bei der Formulierung der Tests darauf achten, dass sie eine konsistente Terminologie zwischen den Vorlesungsunterlagen und den Aufgabenstellungen bzw. den Tests verwendet. Basieren die Materialien der Vorlesung beispielsweise auf der zuvor verwendeten englischen Terminologie der Addition (*Sum* = *Addend* + *Augend*), sollten sich diese Begrifflichkeiten ebenfalls in der Implementierung der entsprechenden Tests wiederfinden (siehe Listing 6.4, Seite 84) und nicht etwa wie in Listing 6.6 gezeigt auf einer anderen Terminologie (wie z.B. Summe = Summand + Summand) beruhen.

```
class MathUtilitiesTest {
  @Test
  void IfSummand1Is1AndSummand2Is2__Summe__ShouldBe3() {
    int summand1 = 1;
    int summand2 = 2;

    int actual = MathUtilities.summe(summand1, summand2);

    int expected = 3;
    assertEquals(expected, actual);
  }
}
```

Listing 6.6: Implementierung des Tests aus Listung 6.2 unter Verwendung der deutschen Terminologie.

6.3.3 Kontinuität aufeinanderfolgender Tests

Insbesondere bei der Implementierung komplexer und generischer Algorithmen wirken die Tests unterstützend, da sie die Komplexität des Gesamtproblems in kleine, einfach zu erfassende Teilprobleme zergliedern und eine schrittweise Bearbeitung ermöglichen. Durch konkrete Testfälle schränken sie den Produktionscode sukzessiv ein und liefern eine immer spezifischer werdende Beschreibung des angestrebten Systemverhaltens. Das schrittweise Bestehen der einzelnen Testmethoden lenkt den Produktionscode in Richtung der Gesamtlösung und leitet somit die Studierenden durch die Aufgabe.

Für ein erfolgreiches unterbrechungsfreies Arbeiten der Studierenden, sollten die Aufgabenstellungen eine hohe Kontinuität aufeinanderfolgender Tests aufweisen. Dazu müssen nachfolgende Tests nahtlos auf den Spezifikationen der vorherigen Tests aufsetzen, um die Einschränkung des Produktionscodes stetig voranzutreiben. Um diesen Anforderungen gerecht zu werden, könnte die Lehrperson die Aufgabenstellungen selbst in testgetriebener Weise umsetzen. Durch die parallele und inkrementelle Entwicklung der Tests und des Produktionscodes kann die Lehrperson zu Beginn jedes Entwurfszyklus die aktuelle Funktionalität der UoW analysieren und einen entsprechenden Testfall formulieren, der den Produktionscode weiter einschränkt. Aufgrund der Forderung von TDD, dass der in Phase Rot des aktuellen Entwurfszyklus implementierte Test zunächst fehlschlagen muss, bevor der Produktionscode modifiziert wird, entstehen Testfälle, die das Systemverhalten automatisch weiter einschränken.

Anders als in der testgetriebenen Entwicklung entstehen in dem zukünftigen Lehrkonzept Produktionscode und Tests nicht parallel. Die Tests lassen sich somit nicht individuell auf die gewählte Implementierung des Produktionscodes anpassen. Da Studierende durchaus eine variierende Implementierung wählen können, um die Tests bestehen zu lassen, ist es möglich, dass nachfolgende Tests bereits bestehen, ohne den Produktionscode zu verändern. Dieser Fall tritt auf, wenn Studierende mehr Logik implementieren, als für das Bestehen des aktuell betrachteten Tests benötigt wird. Auch wenn Studierende sich an die möglichst minimalistische und naive Implementierungsweise in Phase Grün des Entwurfszyklus halten, kann diese Situation entstehen. Die Begründung dafür liegt beispielsweise in der Formulierung umgekehrter `if-else`-Anweisungen im Vergleich

zu der Implementierung der Lehrperson. Im Folgenden wird die zuvor geschilderte Situation am Beispiel der in Abschnitt 6.1.2 erklärten Aufgabe *LeapYear* verdeutlicht. Es wird zunächst die testgetriebene Entwicklung der ersten vier Testmethoden gezeigt, die anschließend exemplarisch bearbeitet werden.

In diesem Beispiel startet die Lehrperson die testgetriebene Entwicklung der `isLeapYear`-Methode mit dem in Listing 6.7 gezeigten Testfall. Der UoW wird als Jahreszahl 2000 übergeben. Da es sich dabei um ein Schaltjahr handelt, erwartet die Testmethode, dass die UoW als Endergebnis den boolschen Wert `true` liefert. Die einfachste Möglichkeit diesen Test bestehen zu lassen, bietet die in Listing 6.8 gezeigte Implementierung des Produktionscodes, in der die UoW den festen boolschen `true` zurückgibt.

```
@Test
void GivenTheYear2000__ShouldBeLeapYear() {
  int year = 2000;

  boolean actual = LeapYear.isLeapYear(year);

  assertTrue(actual);
}
```

Listing 6.7: TDD der Aufgabestellung *LeapYear*: Implementierung des ersten Tests.

```
public static boolean isLeapYear(int year) {
  return true;
}
```

Listing 6.8: TDD der Aufgabestellung *LeapYear*: Bestehen des ersten Tests.

Durch einen weiteren Testfall schränkt die Lehrperson das Verhalten des Systems weiter ein. Dazu übergibt sie in einem nächsten Test die Jahreszahl 1999 und erwartet, dass die UoW den boolschen Wert `false` zurückliefert. Die Implementierung der Testmethode entspricht Listing 6.9. Um ein Bestehen beider Tests zu erreichen, wählt die Lehrperson die in Listing 6.10 dargestellte Modifikation des Produktionscodes. Anstatt nur den festen Wert `true` zurückzugeben, überprüft die `isLeapYear`-Methode die übergebene Jahreszahl, sodass sie nur `true` zurückgibt, falls es sich um die Zahl 2000 handelt. In allen anderen Fällen gibt sie den festen Wert `false` zurück.

```
@Test
void GivenTheYear1999__ShouldNotBeLeapYear() {
  int year = 1999;

  boolean actual = LeapYear.isLeapYear(year);

  assertFalse(actual);
}
```

Listing 6.9: TDD der Aufgabestellung *LeapYear*: Implementierung des zweiten Tests.

```
public static boolean isLeapYear(int year) {
  if(year == 2000)
    return true;
  return false;
}
```

Listing 6.10: TDD der Aufgabestellung *LeapYear*: Bestehen des zweiten Tests.

Um den Produktionscode weiterhin stetig einzuschränken, formuliert die Lehrperson anschließend zwei weitere Tests, die überprüfen, ob die `isLeapYear`-Methode die Jahrezahlen 1996 und 1992 korrekterweise als Schaltjahre erkennt. Es ergibt sich eine Modifikation der `if`-Anweisung gemäß Listing 6.11. Basierend auf dieser Implementierung erkennt die Lehrperson die Regelmäßigkeit, dass sich alle Kriterien der `if`-Anweisungen auf Zahlen beziehen, die ganzzahlig durch vier teilbar sind und nimmt das in Listing 6.12 gezeigte *Refactoring* vor.

Werden diese Testmethoden anschließend schrittweise von einer anderen Person bearbeitet, ist nicht sichergestellt, dass sie nach den ersten vier Tests das gleiche Resultat

```
public static boolean isLeapYear(int year) {
  if(year == 2000 || year == 1996 || year == 1992)
    return true;
  return false;
}
```

Listing 6.11: TDD der Aufgabestellung *LeapYear*: Bestehen des vierten Tests

```
public static boolean isLeapYear(int year) {
  if(year % 4 == 0)
    return true;
  return false;
}
```

Listing 6.12: TDD der Aufgabestellung *LeapYear*: *Refactoring* nach dem Bestehen des vierten Tests.

erzielt. Von ausschlaggebender Bedeutung ist die Implementierung der `if`-Anweisung nach dem zweiten Test (Listing 6.9). Kehren die Studierenden diese gemäß Listing 6.13 um, bestehen die ersten beiden Tests mit einer minimalistischen Lösung. Die Studierenden halten sich demnach an die Forderung der testgetriebenen Entwicklung und weichen trotzdem von der Implementierung der Lehrperson ab. Betrachten die Studierenden nun die nachfolgenden Tests, ergeben sich keine weiteren Einschränkungen, da die `isLeapYear`-Methode sowohl bei 1996 als auch bei 1992 den Wert `true` zurückgibt. Beide Tests bestehen ohne Modifikation des Produktionscodes, die Entwicklung stagniert. Für weitere Einschränkungen wären in diesem Fall Tests notwendig, die Jahreszahlen überprüfen, bei denen es sich nicht um Schaltjahre handelt (z.B. 1998 und 1997). Basierend auf der Implementierung aus Listing 6.13 können die Studierenden nicht die zuvor erläuterte Regelmäßigkeit erkennen und den Produktionscode entsprechend Listing 6.12 in Richtung der generischen Lösung umformen. Die testgetriebene Entwicklung der Aufgabenstellung durch die Lehrperson stellt daher nicht sicher, dass alle Studierenden die generische Endlösung erreichen.

```
public static boolean isLeapYear(int year) {
  if(year == 1999)
    return false;
  return true;
}
```

Listing 6.13: TDD der Aufgabestellung *LeapYear*: Exemplarische Bearbeitung durch die Studierenden.

Dieser Problematik muss sich die Lehrperson bei der testgetriebenen Entwicklung der Aufgabenstellung bewusst sein. Um dennoch zu gewährleisten, dass die Gesamtheit aller Tests ausreicht, um alle Regelmäßigkeiten des Produktionscodes aufzudecken, muss sie gegebenenfalls unterschiedliche Implementierungen abtesten und die Tests zu einer Aufgabenstellung zusammenführen. Auch könnten die Studierenden dazu angehalten werden, sich an der gewählten Implementierung der Lehrperson zu orientieren. Besteht ein nachfolgender Tests ohne Modifikation des Produktionscodes, müssen die Studierenden dazu ihren Produktionscode so umformen, dass die vorherigen Tests weiterhin bestehen, der aktuelle Test jedoch fehlschlägt. Beide Möglichkeiten gilt es, bei der Umsetzung des zukünftigen Lehrkonzepts zu evaluieren.

6.3.4 Wartbarkeit

Entsprechend der Ansätze „Guter Lehre“ (siehe Abschnitt 2.1) basiert das zukünftige Lehrkonzept auf der steten Reflexion von Vorlesungs- und Übungsinhalten. Dazu wird die Übung unter anderem von einem technischen System begleitet (siehe Kapitel 7), das die Bearbeitungsstände der Studierenden kontinuierlich anhand eines *Burndown Chart* visualisiert (siehe 5.2.2). Mit Hilfe der aufgezeichneten Daten des Bearbeitungsfortschritts der Studierenden, kann die Lehrperson die Übungsaufgaben evaluieren. Ist beispielsweise ersichtlich, dass bestimmte Testfälle im Vergleich zu anderen eine überdurchschnittliche Zeitspanne für die Bearbeitung erfordern, lassen sich daraus mögliche Fehlkonzeptionen in der Aufgabenstellung ableiten. Auch fließt das Feedback aus den regelmäßigen Besprechungen der Lehrperson mit den Studierenden (siehe Abschnitt 5.3) in die Evaluation der Aufgabenstellungen ein. Ausgehend von diesem Feedback unterliegen die Aufgabenstellungen unter Umständen stetigem Anpassungsbedarf. Damit der Lehrperson die Möglichkeit gegeben wird, die Änderungen mit möglichst geringem Zeitaufwand einzuarbeiten, sollten die Aufgabenstellungen eine hohe Wartbarkeit und eine flexible Struktur aufweisen. Beide Eigenschaften werden in der Softwareentwicklung maßgeblich durch Redundanzen in dem Quelltext beeinträchtigt. Aus diesem Grund sollte die Lehrperson bei der Umsetzung der Aufgabenstellungen darauf achten, Redundanzen in den Testmethoden zu vermeiden.

In der Aufgabenstellung *BankAccount* bedarf beispielsweise jede Testmethode eine Instanz der Klasse `BankAccount`, da diese das SUT darstellt. Da sich Einzahlungen (Aufruf der `deposit`-Methode) und Auszahlungen (Aufruf der `payOut`-Methode) auf den Kontostand (*Balance*) auswirken, der daher in mehreren Testmethoden auf bestimmte Werte überprüft wird, kommt es auch in dem *Assert* der Tests zu Redundanzen. Zur Vermeidung dieser Redundanzen sollten wie in Abschnitt 6.2 beschrieben Attribute und Methoden der Testklasse verwendet werden. Listing 6.14 zeigt beispielhaft die Umsetzung eines Tests, der die zuvor genannten Redundanzen entsprechend vermeidet.

```
1  class BankAccountTest {
     private BankAccount sut = new BankAccount();
3
     private void assertBalanceIs(double expected) {
5      double actual = sut.getBalance();
       assertEquals(expected, actual, 0.001);
7    }

9    @Test
     void AfterADepositOf150Bugs__AnotherDepositOf50Bugs
11     __ShouldResultInABalanceOf200Bugs() {
       sut.deposit(150.0);
13
       sut.deposit(50.0);
15
       assertBalanceIs(200.0);
17   }
   }
```

Listing 6.14: Redundanzvermeidung durch Verwenden von Attributen und „Helfer“-Methoden in der Aufgabenstellung *BankAccount*.

Für das SUT enthält die Testklasse (hier `BankAccountTest`) das Attribut `sut`, das bei der Instanziierung der Testklasse mit einer neuen Instanz der Klasse `BankAccount` initialisiert wird. Da JUnit wie zuvor beschrieben *Transient Fresh Fixtures* verwendet, erhält jede Testmethode bei der Ausführung ein unbenutztes SUT. Weiterhin existiert

die „Helfer“-Methode `assertBalanceIs`, die die Logik für das Überprüfen des Kontostands enthält. Anstatt diese Zeilen in jeder Testmethode zu wiederholen, wird die `assertBalanceIs`-Methode aufgerufen.

Da die Ansätze zur Vermeidung der Redundanzen auf der Objektorientierung beruhen, können sie insbesondere zu Beginn des Semesters zu Verständnisproblemen bei den Studierenden führen. Daher sollten sie erst nach der Vermittlung der Objektorientierung eingesetzt werden. In Aufgaben wie z.B. *MathUtilities* oder *LeapYear* ordnet sich die Wartbarkeit der Testmethoden der angepassten Formulierung an den Wissensstand der Studierenden unter.

KAPITEL 7

Realisierung des technischen Systems

Zur Erhöhung von Transparenz und Feedback wird die Übungsveranstaltung wie in Abschnitt 5.2.2 beschrieben von einem technischen System begleitet, das den Bearbeitungsfortschritt der Studierenden automatisch auswertet und anhand eines *Burndown Chart* visualisiert. Zu diesem Zweck ist eine Softwareinfrastruktur umzusetzen, die die Bearbeitungsstände jeder Gruppe erfasst, die Tests automatisiert auswertet und die Lernfortschrittsvisualisierung anzeigt. Die Softwareinfrastruktur basiert dabei auf bestehenden und frei verfügbaren Lösungen der Softwareentwicklung. Sie setzt sich aus mehreren Komponenten zusammen, aus deren Interaktion die Gesamtfunktionalität des technischen Systems resultiert.

Das nachfolgende Kapitel beschreibt die Realisierung des technischen Systems. Dazu erläutert es zunächst kurz die Anforderungen sowie die Spezifikationen, denen das System genügen muss, um die Erfolgschancen des zukünftigen Lehrkonzepts zu erhöhen. Im Anschluss daran wird die Umsetzung sowie die Funktionsweise der Softwareinfrastruktur erläutert.

7.1 Anforderungen und Spezifikation

Neben der Evaluation der Aufgabenstellungen durch die Lehrperson am Ende eines Veranstaltungszyklus (siehe Abschnitt 5.2.4) sorgt die Visualisierung der Lernfortschritte ebenfalls für Feedback und Transparenz (siehe Abschnitt 5.2.2). Das *Burndown Chart* ermöglicht es den Studierenden beispielsweise, ihren eigenen Lernprozess einzuordnen und zu reflektieren. Weiterhin können Dozierende den Lernfortschritt der Studierenden während der Bearbeitung verfolgen und zielgerichtet eingreifen, um den Lernprozess der Studierenden zu lenken. Aus diesem Grund sollte die Lernfortschrittsvisualisierung stetig zur Verfügung stehen. Eine der wichtigsten Anforderungen des technischen Systems liegt deswegen darin, das *Burndown Chart* automatisiert zu aktualisieren und kontinuierlich auf den Rechnern der Studierenden bereitzustellen. Da das *Burndown Chart* die Lernfortschritte aller Gruppen zugleich visualisiert, müssen dem technischen System die Bearbeitungsstände der unterschiedlichen Gruppen bekannt sein. Eine Umsetzung des gesamten technischen Systems als lokale Applikation auf den Rechnern der Studierenden ist daher ohne Kommunikation zu anderen Rechnern nicht möglich. Um den Fortschritt aller Gruppen ohne hohen Kommunikationsaufwand gemeinsam in dem *Burndown Chart* anzeigen zu können, bedarf es wie in Abbildung 7.1 dargestellt eines zentralen Servers, auf dem die Studierenden ihre Bearbeitungsstände ablegen.

Für die Darstellung des *Burndown Chart* benötigt das technische System die Gesamtanzahl der vorhandenen Tests einer Aufgabenstellung sowie den Zeitpunkt und die Anzahl der nicht bestandenen bzw. der noch nicht bearbeiteten Tests der einzelnen Bearbeitungsstände (siehe Abbildung 5.1, Seite 64). Um diese Daten kontinuierliche zu erheben und das dargestellte *Burndown Chart* fortwährend zu aktualisieren, muss auf dem zentralen Server die automatisierte Ausführung der Tests zu den einzelnen Bearbeitungsständen der unterschiedlichen Gruppen stattfinden. Sobald eine Gruppe einen neuen Bearbeitungsstand registriert, sind die Tests auszuführen, um die Datengrundlage des *Burndown Chart* zu erneuern.

Die grafische Darstellung des *Burndown Chart* kann durch unterschiedliche Möglichkeiten realisiert werden. So könnte der Server beispielsweise über eine Webanwendung verfügen, die die Studierenden mit Hilfe eines Browsers aufrufen. Für ein unterbrechungs-

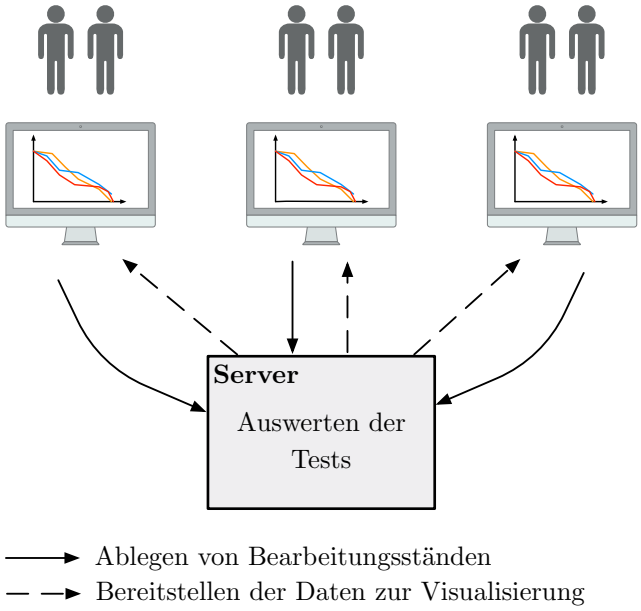

Abbildung 7.1: Prinzipielle Funktionsweise des technischen Systems.

freies Arbeiten sollten die Studierenden jedoch während der Bearbeitung der Übungsaufgaben nicht mit mehr als einer Applikation bzw. grafischen Oberfläche konfrontiert werden. Aus diesem Grund gilt es, die grafische Darstellung in die gewählte IDE zu integrieren. Dazu muss der Server die generierte Datengrundlage zur Verfügung stellen. Die IDE lädt die zur Darstellung aufbereiteten Daten von dem Server und zeigt das *Burndown Chart* an. Auch den Zugang zu dem Server, um neue Bearbeitungsstände zu registrieren, sollte die verwendete IDE unterstützen.

Das technische System unterteilt sich somit in serverseitige und clientseitige Komponenten. Während die serverseitigen Komponenten – die Registrierung neuer Bearbeitungsstände sowie die automatisierte Testauswertung – die Generierung der Datengrundlage übernehmen, dient die clientseitige Komponente zur Visualisierung des *Burndown Chart*.

Sollte sich das in dieser Arbeit entstandene agile Lehrkonzept bewähren und eine erhöhte Lerneffektivität sowie gesteigerte Lernerfolge der Studierenden hervorrufen, ist zukünftig ein Transfer auf weitere Lehrveranstaltungen denkbar. Dabei sollten die Daten der unterschiedlichen Veranstaltungen separiert werden. Aus diesem Grund bedarf jede Veranstaltung der Einrichtung eines eigenen Servers. Ebenfalls sollten die Server zu Beginn jedes

Semesters neu eingerichtet werden, um die Bearbeitungsstände und die Testauswertungen vorheriger Gruppen zu entfernen. Damit sich kein übermäßig erhöhter Zeitaufwand durch die sich wiederholende Einrichtung der serverseitigen Komponenten ergibt, sollte das umgesetzte technische System – insbesondere die serverseitigen Komponenten – eine leichte Reproduzierbarkeit aufweisen. Weiterhin sollte es das technische System ermöglichen, die serverseitigen Komponenten zum Zweck der Datensicherung auf andere Server zu migrieren und auszuführen. Dazu ist eine Entkopplung von der Hardwareplattform des Servers notwendig.

7.2 Umsetzung und Funktionsweise der Softwareinfrastruktur

Sowohl die Verwaltung von Bearbeitungsständen als auch die automatisierte Ausführung der Tests lassen sich üblichen Werkzeugen der Softwareentwicklung zuordnen. Die serverseitigen Komponenten können beispielsweise durch ein zentrales VCS in Kombination mit einem Werkzeug für CI realisiert werden. Die hier gezeigte Softwareinfrastruktur nutzt dafür Git[1] (VCS) und Jenkins[2] (CI). Um aus der automatisierten Testausführung ebenfalls die Datengrundlage des *Burndown Charts* zu gewinnen, gilt es, Jenkins durch ein Plugin zu erweitern, das die Daten der Testauswertung zur Visualisierung aufbereitet. Wie in Abschnitt 7.1 gefordert sollte die Visualisierung und der Zugang zu dem VCS weiterhin in die gewählte IDE integriert werden. Aufgrund der bereits bestehenden Anbindung zu Git und der leichten Erweiterbarkeit durch Plugins, wird als IDE IntelliJ IDEA[3] verwendet.

Der nachfolgende Abschnitt zeigt die Umsetzung und die Funktionsweise der Softwar-

[1] Git ist im Jahr 2005 erschienene, frei verfügbare und quelltextoffene Software, die ursprünglich von L. Torvalds, J. C. Hamano und S. O. Pearce entwickelt wurde. Derzeit wird Git von J. C. Hamano verwaltet. Verfügbar unter https://git-scm.com; Abgerufen am 18. Oktober 2018.

[2] Jenkins ist eine im Jahr 2011 erschienene, frei verfügbare und quelltextoffene Software, die ursprünglich von Kohsuke Kawaguchi entwickelte, von dem es zur Zeit verwaltet wird. Verfügbar unter https://jenkins.io; Abgerufen am 18. Oktober 2018.

[3] JetBrains s.r.o., Prag. Verfügbar unter https://www.jetbrains.com/idea; Abgerufen am 18. Oktober 2018.

einfrastruktur unter Verwendung der zuvor genannten Technologien und Werkzeuge. Zuerst wird dazu die Verwendung von Git zur Bereitstellung und Bearbeitung der Übungsaufgaben beschrieben. Anschließend wird die automatisierte Testausführung mit Jenkins erläutert. Danach wird ein umgesetztes Plugin gezeigt, das Jenkins um die Funktionalität erweitert, die Ergebnisse der Testausführung für die Darstellung in dem *Burndown Chart* aufzubereiten. Weiterhin wird die Einrichtung einer einheitlichen Schnittstelle mit NodeJS[4] erklärt, die den Zugriff auf die serverseitigen Komponenten erleichtert und dynamische Abfragen erlaubt. Ein nächster Abschnitt zeigt darüber hinaus die Implementierung eines weiteren Plugins, das die gewählte IDE IntelliJ IDEA um die Visualisierung der aufbereiteten Daten erweitert. Abschließend zeigt dieser Abschnitt die Entkopplung des technischen Systems von der verwendeten Hardwareplattform des Servers durch die Virtualisierung mit Docker.

7.2.1 Bereitstellen und Bearbeiten der Aufgaben mit Git

Zur Verwaltung und Registrierung der Bearbeitungsstände verwendet die Softwareinfrastruktur wie zuvor erwähnt Git als zentrales VCS. Wie in Abschnitt 4.5.2 erklärt, erlaubt es ein VCS, verschiedene Bearbeitungsstände (*Commits*) in chronologischer Reihenfolge in einem *Repository* zu speichern und sie zu einem späteren Zeitpunkt erneut abzurufen. Darüber hinaus bietet Git die Möglichkeit die *Repositories* auf einem Server abzulegen, um die Bearbeitungsstände zu sichern. Auch können die *Repositories* von dem Server auf lokale Rechner heruntergeladen, bearbeitet und anschließend auf dem Server aktualisiert werden.

Diese Funktionalität macht sich die Softwareinfrastruktur zu Nutze, um die Bearbeitungsstände der Studierenden auf dem zentralen Server zu sammeln und auszuwerten. Im Vorfeld richtet die Lehrperson *Repositories* auf dem Server ein und stellt sie den Studierenden zur Verfügung. Da der Lehrperson keine sicheren Informationen über die Anzahl der Teilnehmer – respektive die Anzahl und Zusammensetzung der Gruppen – vorliegen, ist eine Einrichtung von einzelnen *Repositories* für jede Gruppe nur bedingt möglich. Auch würde die Einrichtung von Gruppen-*Respositories* einen hohen zeitlichen

[4]Node.js Foundation. Verfügbar unter https://nodejs.org; Abgerufen am 18. Oktober 2018.

Aufwand für die Lehrperson bedeuten. Aus den zuvor genannten Gründen wird für jede Aufgabenstellung nur ein zentrales *Repository* geschaffen, das die unterschiedlichen Gruppen gemeinschaftlich nutzen. Diese Lösung ist unabhängig von der Anzahl der teilnehmenden Gruppen und minimiert so den zeitlichen Aufwand der Lehrperson.

Weiterhin werden die *Repositories* dafür verwendet, den Studierenden die Aufgabenstellungen bereitzustellen. Zu diesem Zweck enthält der *Master Branch* in jedem *Repository* ein ausführbares *Maven*[5]-Projekt mit den zu bearbeitenden Testmethoden. Um die Bearbeitung einer Aufgabenstellung zu beginnen, müssen die Studierenden daher nur das entsprechende *Repository* auf ihre lokalen Rechner laden und mit der Implementierung des Produktionscodes starten. Nach dem erfolgreichen Bearbeiten eines Tests erstellen die Studierenden einen neuen *Commit* in ihrem lokalen *Repository* und aktualisieren anschließend das VCS auf dem Server. Damit sich die chronologischen Reihenfolgen der unterschiedlichen Gruppen nicht stören, legen die Studierenden zu Beginn der Bearbeitung einen gruppenspezifischen *Branch* an.

Bearbeiten beispielsweise zwei unterschiedliche Gruppen gleichzeitig dieselbe Aufgabenstellung, ergibt sich eine Struktur nach Abbildung 7.2. Das zentrale *Repository* auf dem Server enthält zunächst lediglich die Aufgabenstellung in dem *Master Branch*. Die Studierenden laden das *Repository* auf ihre lokalen Rechner und richten gruppenspezifische Zweige ein. Ohne auch nur einen Test zu bearbeiten, aktualisieren sie den Server, respektive das zentrale *Repository*, um ihre gruppenspezifischen Zweige zu erweitern. Anschließend bearbeiten sie schrittweise die Tests der Aufgabenstellung. Mit jedem bestandenen Test erstellen die Studierenden einen *Commit* und aktualisieren das zentrale *Repository*.

Für die Einrichtung von Git auf dem zentralen Server wird GitLab[6] verwendet. Dabei handelt es sich um eine Webanwendung, die neben weiteren zahlreichen Funktionalitäten eine grafische Oberfläche für die Verwaltung von *Repositories* bietet. Infolgedessen erhöht sie den Bedienkomfort für die Vorbereitung und Bereitstellung der Übungsaufgaben.

[5] Apache Software Foundation, Wakefield MA. Verfügbar unter https://maven.apache.org; Abgerufen am 18. Oktober 2018.
[6] GitLab B.V. Verfügbar unter https://about.gitlab.com; Abgerufen am 18. Oktober 2018.

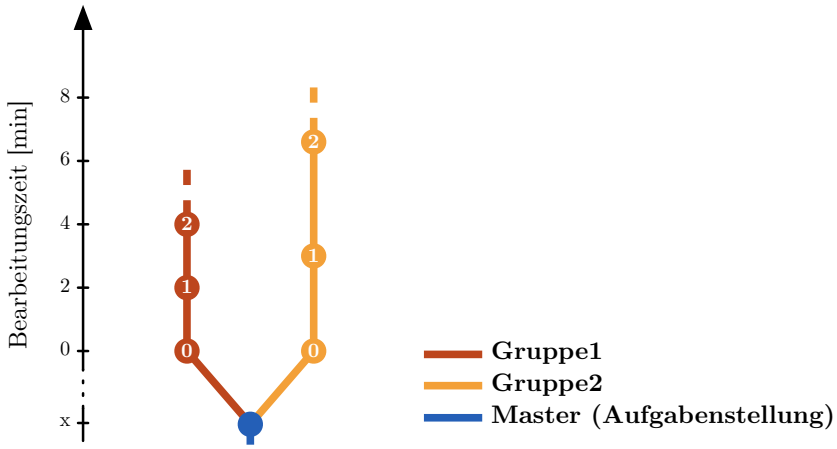

Abbildung 7.2: Zentrales *Repository* zur Bereistellung und Bearbeitung der Aufgaben unter Verwendung von Git.

7.2.2 Automatisierte Testausführung mit Jenkins

Sobald die Studierenden das zentrale *Repository* aktualisieren, gilt es, die Datengrundlage des *Burndown Chart* zu erneuern. Dazu müssen wie in Abschnitt 7.1 beschrieben zunächst die Tests ausgeführt werden. Die automatisierte Ausführung der Tests erfolgt mit Hilfe der Webanwendung Jenkins, die neben GitLab auf dem Server des technischen Systems eingerichtet ist.

Zu jeder Aufgabenstellung erstellt die Lehrperson zusätzlich einen Jenkins-Job, der nach dem Auslösen die Bearbeitungsstände der entsprechenden *Repositories* mit Hilfe des *Hypertext Transfer Protocol* (HTTP) von GitLab herunterlädt und die Tests ausführt. Die Automatisierung geschieht über die sogenannten *Git Hooks*.[7] Dabei handelt es sich um Skripte, die durch bestimmte Ereignisse in dem zentralen *Repository* ausgelöst werden. Damit die Jenkins-Jobs nach dem Registrieren neuer Bearbeitungsstände automatisiert ausgelöst werden, verwenden die *Repositories* hier die *Post-Receive Hook*, deren Auslöser die erfolgreiche Aktualisierung des zentralen *Repository* darstellt. Die ausgeführten Skripte benachrichtigen Jenkins über einen neuen Bearbeitungsstand und lösen den Jenkins-Job aus. Es ergibt sich das in Abbildung 7.3 dargstellte System.

[7] Dokumentation verfügbar unter https://git-scm.com/docs/githooks; Abgerufen am 18. Oktober 2018 (verwaltet von J. C. Hamano).

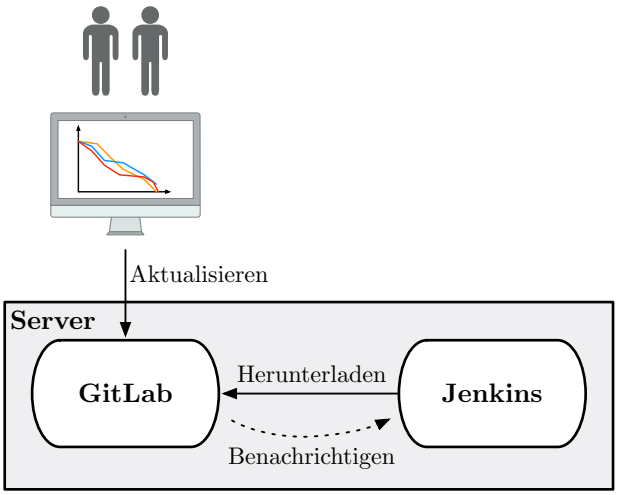

Abbildung 7.3: Kopplung von Git und Jenkins zur automatisierten Ausführung der Tests.

Aufgrund der unterschiedlichen Gruppen weist das *Repository* viele verschiedene *Branches* auf, die sich erst zur Übungsveranstaltung ergeben. Um die Lernfortschritte aller vorhandenen Gruppen zu visualisieren, muss der Jenkins-Job nach der Benachrichtigung neue *Branches* dynamisch erkennen und entsprechend die Tests ausführen. Für diesen Fall stellt Jenkins die *Multibranch Pipeline*[8] zur Verfügung, die das *Repository* stetig nach neuen *Branches* durchsucht. Mit Hilfe eines *Pipeline Script* erlaubt die *Multibranch Pipeline* weiterhin, eine Reihe an auszuführenden Schritten (*Stages*) zu definieren, die für jeden vorhandenen *Branch* ausgeführt werden.

Listing 7.1 zeigt ein exemplarisches *Pipeline Script* zur Ausführung der Tests. Zeile 2 des Listings sorgt zunächst für das Herunterladen des neuen Bearbeitungsstands. Unter Verwendung von *Maven* ruft die nachfolgende *Stage* („Test“) anschließend den Befehl für die Ausführung der Tests auf und erstellt dadurch für jeden Zweig eine Datei auf dem Server, die die Testergebnisse des aktuellen Bearbeitungsstand im Format der *Extensible Markup Language* (XML) enthält.

Mit jedem neuen *Commit* werden die Testergebnisse – die sogenannten *Test Reports* – aktualisiert und die Ergebnisse des vorherigen Bearbeistungsstands gehen verloren. Da-

[8] Version 2.20, Dokumentation verfügbar unter https://wiki.jenkins.io/display/JENKINS/Pipeline+Multibranch+Plugin; Abgerufen am 18. Oktober 2018(verwaltet von Kohsuke Kawaguchi)

```
node {
  checkout scm

  stage("Test") {
    sh 'mvn clean test'
  }
}
```

Listing 7.1: *Pipeline Script* zur automatisierten Ausführung der Testmethoden.

mit die Visualisierungskomponente in dem *Burndown Chart* jedoch die gesamte Historie einer Gruppe anzeigen kann, gilt es, die *Test Reports* der einzelnen Bearbeitungsstände zu sichern, sodass sich für jeden Bearbeitungsstand der unterschiedlichen Zweige ein *Test Report* ergibt. Dazu verwendet das *Pipeline Script* das bestehende JUnit Plugin[9], das wie in Listing 7.2 gezeigt in einer anschließenden *Stage* mit dem Pfad des zu sichernden *Test Reports* aufgerufen wird.

```
node {
  //...
  stage("Save Test Report History") {
    junit 'path/to/test/reports/*.xml'
  }
}
```

Listing 7.2: Erweiterung des *Pipeline Script* zur automatisierten Aufbereitung der *Repository*-Daten.

7.2.3 Burndown-Chart-Publisher-Plugin für Jenkins

Aus den zuvor gesicherten *Test Reports* (siehe Abschnitt 7.2.2) gilt es nun, das *Burndown Chart* in der clientseitigen Komponente darzustellen. Dazu muss die Visualsierungskomponente der Softwareinfrastruktur die Daten von dem Server herunterladen

[9] Version 1.26.1, Dokumentation verfügbar unter https://jenkins.io/doc/pipeline/steps/junit; Abgerufen am 18. Oktober 2018 (verwaltet von Kohsuke Kawaguchi).

und anzeigen. Da der Visualisierungskomponente keine Informationen über die Anzahl und Namen der unterschiedlichen *Branches* sowie deren Bearbeitungsstände vorliegen, stellt der dynamische Download sowie die Anzeige aller generierten *Test Reports* eine Herausforderung dar. Darüber hinaus sollte sich die clientseitige Komponente nicht an die Datei- und Ordnerstruktur des Servers anpassen. Für einen vereinfachten Datentransfer wird Jenkins um das *Burndown-Chart-Publisher*-Plugin erweitert, das die *Test Reports* auf dem Server in einer einzigen Datei zusammenfasst.

Zur besseren Verarbeitung der Testergebnisse in der Visualisierungskomponente erfolgt weiterhin eine Aufbereitung der Daten. Da sowohl das Jenkins-Plugin als auch die Visualisierungskomponente in Java implementiert ist (siehe Abschnitt 7.2.5), werden die Daten in eine objektorientierte Struktur überführt. Abbildung 7.4 zeigt exemplarisch eine Objektstruktur zur Repräsentation des *Repository* aus Abbildung 7.2. Da sich ein *Repository* aus mehreren *Branches* zusammensetzt, die wiederum mehrere *Test Reports* aufweisen, ergibt sich eine hierarchische Struktur. An der Stelle des Wurzelelements dieser Hierarchie befindet sich das *Repository* selbst, das einerseits den Namen der Aufgabenstellung (bspw. „Aufgabe1“) enthält und andererseits über Referenzen zu den einzelnen *Branches* verfügt. Als Name des *Repository* dient den Name des eingerichteten Jenkins-Jobs. Ebenso geben auch die *Branches* Aufschluss über ihren Namen und ihre zugehörigen *Test Reports*. Wie in Abschnitt 7.1 beschrieben, benötigt die clientseitige Komponen-

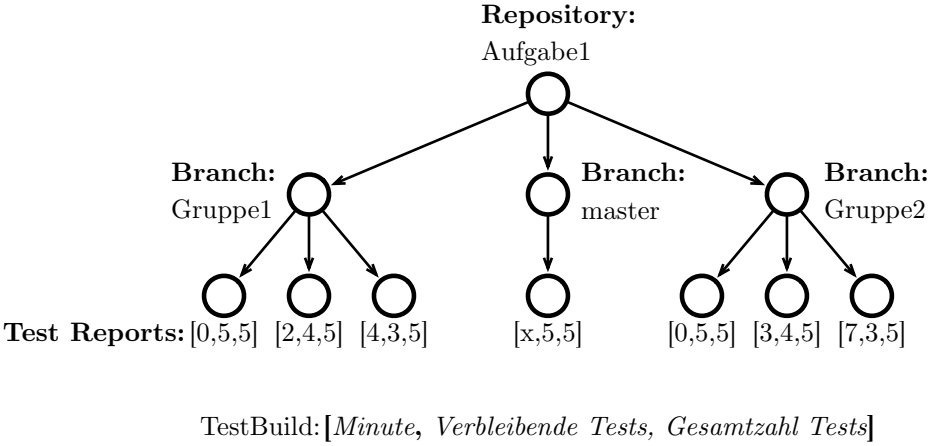

Abbildung 7.4: Hierarchische Objektstruktur zur Repräsentation des *Repository* aus Abbildung 7.2.

te die Gesamtanzahl der Tests sowie Zeitpunkt und Anzahl der noch zu bearbeitenden Tests der einzelnen Bearbeitungsstände. Diese Informationen sind Bestandteil der *Test Reports*. Geht man davon aus, dass die Aufgabenstellungen fünf zu bearbeitende Tests beinhaltet, ergeben sich die *Test Reports* wie in Abbildung 7.4 gezeigt.

Zur Realisierung dieser Objektstruktur verwendet das *Burndown-Chart-Publisher*-Plugin das in Abbildung 7.5 dargestellte Klassensystem. Mit `Repository`, `Branch` und `TestReport` entsteht zu jeder der drei unterschiedlichen Hierarchieebenen eine Klasse. Die sich aufgrund der Vererbungsbeziehung zu dem *Interface* `Serializable` durch den von Java bereitgestellen Mechanismus serialisieren lassen. Die von dem

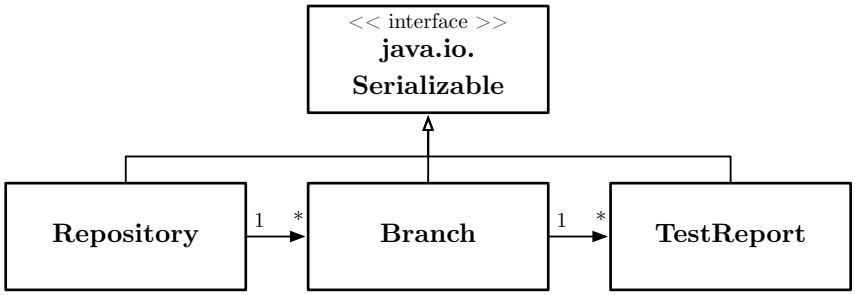

Abbildung 7.5: UML-Diagramm der Klassen zu objektorientierten Repräsentation der Daten.

Jenkins-Plugin abgelegte Datei, die die Ergebnisse aller *Test Reports* zusammenfasst, enthält somit die serialisierte Objektstruktur.

Das *Burndown-Chart-Publisher*-Plugin stellt einen ausführbaren *Pipeline*-Schritt dar, zu dessen Aufruf das *Pipeline Script* wie in Listing 7.3 dargestellt erweitert wird. Im Anschluss an den Schritt zur Sicherung der *Test Reports* (siehe Listing 7.2, siehe Seite 103) weist das *Pipeline Script* eine weitere *Stage* auf, die das Plugin durch den Befehl `publishBurndownChart` in Zeile 4 auslöst. Das Plugin liest anschließend wie zuvor beschrieben die einzelnen *Test Reports* ein, führt sie in der gezeigten Objektstruktur zusammen und legt sie in serialisierter Form in einer Datei auf dem Server ab, die anschließend von der clientseitigen Visualisierungskomponente heruntergeladen und angezeigt werden kann.

```
node {
  //...
  stage("Publish Burndown Chart Data") {
    publishBurndownChart()
  }
}
```

Listing 7.3: Erweiterung des *Pipeline Script* zur automatisierten Aufbereitung der *Repository*-Daten.

7.2.4 Einheitliche Schnittstelle durch NodeJS

Für das Herunterladen der Datengrundlage des *Burndown Charts* benötigt die clientseitige Visualisierungskomponente neben dem Zugriff auf GitLab ebenfalls Zugang zu Jenkins. Um die Kommunikationsschnittstelle der serverseitigen Komponenten zu vereinheitlichen, wird auf dem Server des Weiteren mit NodeJS ein *Reverse Proxy* eingerichtet. Dabei handelt es sich um eine Webanwendung, die Anfragen an andere Services weiterleitet und deren Antwort zurückliefert. Die eigentlich adressierten Services bleiben dabei vor dem Client verborgen. Es ergibt sich eine Softwareinfrastruktur nach Abbildung 7.6. Anstatt direkt mit GitLab und Jenkins zu kommunizieren stellt die clientseitige Komponente ihre Anfragen an den *Reverse Proxy*. Dieser leitet das Aktualisieren der Bearbeitungsstände an GitLab weiter, während die Downloads der zu visualisierenden Daten an Jenkins adressiert werden.

Jenkins kann durchaus mehrere Jobs für unterschiedliche Aufgabenstellungen aufweisen. Daher muss die clientseitige Komponente dem Benutzer die Möglichkeit bieten, das zu visualisierende *Burndown Chart* auszuwählen. Zu diesem Zweck müssen die Namen der vorhandenen Jobs dynamisch abgefragt werden können. Da Jenkins für jeden Job einen eigenen Ordner mit dem Namen des Jobs erstellt, muss dafür die Ordnerstruktur von Jenkins analysiert und eine Liste erstellt werden. Diesen Service übernimmt ebenfalls der *Reverse Proxy*.

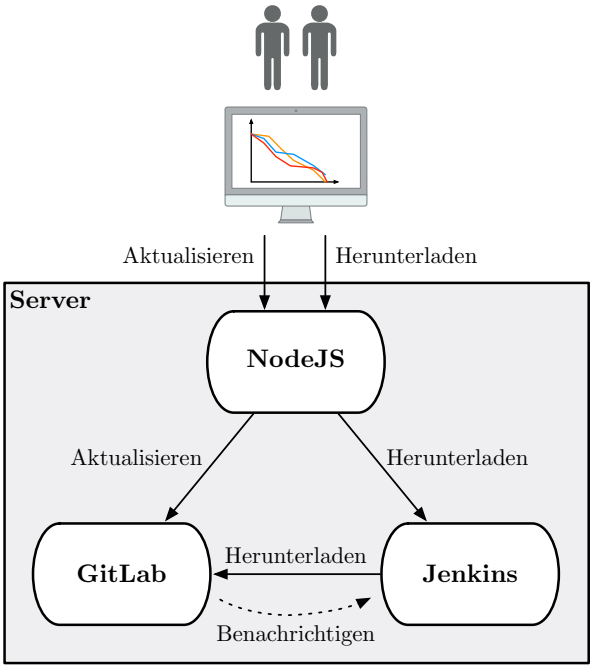

Abbildung 7.6: Einbindung eines Reverse-Proxy mit NodeJS zur Vereinheitlichung der Kommunikationsschnittstelle der serverseitigen Komponenten.

7.2.5 Burndown-Chart-View Plugin für IntelliJ IDEA

Die anschließende Visualisierung auf den Rechnern der Studierenden erfolgt wie in Abschnitt 7.1 gefordert in der grafischen Oberfläche der gewählten IDE. Dazu wird IntelliJ IDEA um ein Plugin erweitert, das das in Abbildung 7.7 gezeigte *Tool Window* in der rechten Seitenleiste der grafischen Oberfläche hervorruft. Betrachtet man das *Burndown Chart* aus Abbildung 7.7, fällt auf, dass die Bearbeitungsstände des *Master Branch* – respektive der Aufgabenstellung – nicht angezeigt werden. Dennoch ist der *Master Branch* für die Visualisierung der Lernfortschritte von hoher Bedeutung. Er kann in der Datengrundlage nicht vernachlässigt werden, da er die Konsistenz der Gesamtanzahl der zu bearbeitenden Tests gewährleistet. Zwar liefert jeder andere *Test Report* ebenfalls dieselbe Gesamtanzahl (siehe Abbildung 7.4, Seite 104), entfernen Studierende jedoch versehentlich einzelne Testfälle aus der Aufgabenstellung und erstellen anschließend Bearbeitungsstände, entstehen Diskrepanzen. Da keine Gruppe die Bearbeitungsstände des *Master Branch* modifiziert, dient seine Gesamtanzahl der zu bearbeitenden Tests als konsistente obere Grenze für das dargestellte *Burndown Chart*.

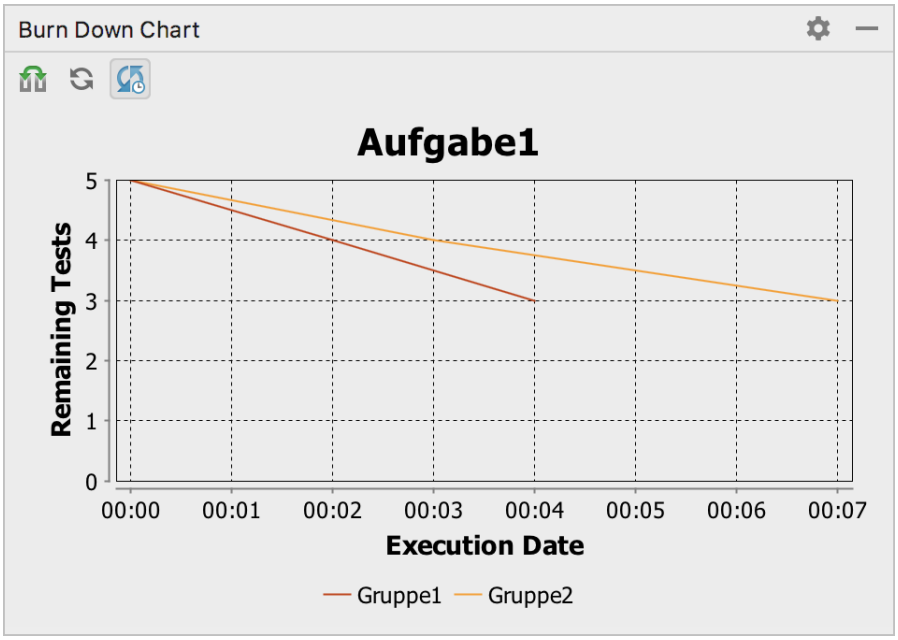

Abbildung 7.7: Visualisierung der Datenstruktur aus Abbildung 7.5 durch das *Tool Window* des *Burndown-Chart-View*-Plugins für IntelliJ IDEA.

Da die Übungsaufgaben durch die stetige Evaluation unter Umständen einigen Anpassungen für nachfolgende Semester unterliegen und sich beispielsweise die Anzahl der zu bearbeitenden Testfälle verändern kann, verwendet das Plugin als obere Grenze des Diagramms die Gesamtanzahl des letzten Bearbeitungsstands der Aufgabenstellung. Verändert die Lehrperson die Aufgabenstellung in dem *Master Branch*, ergibt sich daher durch Hinzufügen oder Entfernen von Testfällen automatisch eine andere obere Grenze des *Burndown Chart*.

Neben der Lernfortschrittsvisualisierung verfügt das *Tool Window* des Plugins weiterhin über eine *Toolbar*, mit deren Schaltflächen die Anzeige des *Burndown Chart* konfiguriert werden kann. Betätigt der Benutzer beispielsweise die erste der drei Schaltflächen in der *Toolbar* (links), öffnet sich der in Abbildung 7.8 dargestellte Konfigurationsdialog. Der Dialog fordert den Benutzer dazu auf, einen Server in das dafür vorgesehene Textfeld einzutragen. Handelt es sich bei der Eingabe des Benutzers um einen erreichbaren Server (beispielsweise den zuvor erläuterten *Reverse Proxy*; siehe Abschnitt 7.2.4), fragt das Plugin anschließend die Liste der vorhandenen Jobs ab und zeigt sie in der *Dropdown*-Liste an. Stellt der eingetragene Server diesen Service nicht bereit oder verfügt über kei-

ne Jobs, wird die *Dropdown*-Liste deaktiviert. Andernfalls wählt der Benutzer einen Job der Liste aus und schließt die Konfiguration mit dem Betätigen der „*Apply*“-Schaltfläche ab. Dadurch startet der Visualisierungsprozess des Plugins. Die auf dem Server abgelegte, serialisierte Objektstruktur, die die Datengrundlage des *Burndown Chart* bildet (siehe Abschnitt 7.2.3), wird heruntergeladen, deserialisiert und anschließend in das visualisierte *Burndown Chart* umgewandelt. Für die Darstellung des Diagramms verwendet das *Burndown-Chart-View*-Plugin für IntelliJ IDEA das frei verfügbare Framework JFreeChart.

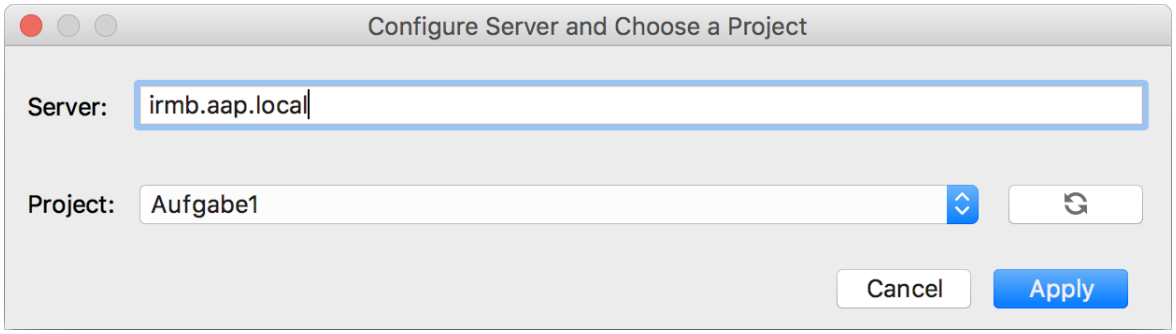

Abbildung 7.8: Darstellung des Dialogs zur Konfiguration des angezeigten *Burndown Chart* in dem *Burndown-Chart-View*-Plugin für IntelliJ IDEA.

Nach dem Einrichten der Konfiguration sollte sich das *Burndown Chart* automatisiert aktualisieren. Sobald neue Bearbeitungsstände der Gruppen vorliegen, gilt es deshalb, den Visualisierungsprozess des Plugins neu zu starten. Dazu muss erneut die Datengrundlage von dem Server heruntergeladen, deserialisiert und angezeigt werden. In der derzeitigen Umsetzung besteht jedoch keine Möglichkeit für die serverseitigen Komponenten, das *Burndown-Chart-View*-Plugin über eine Aktualisierung der Bearbeitungsstände zu informieren. Daher weist die *Toolbar* des *Tool Window* weitere Schaltflächen auf (siehe Abbildung 7.7, Seite 108). Betätigt der Benutzer beispielsweise die mittlere Schaltfläche der *Toolbar* löst er die Aktualisierung des *Burndown Charts* manuell aus. Bei der dritten Schaltfläche (rechts) handelt es sich um einen sogenannten *Toggle Button*, der als Schalter verwendet wird, um die automatisierte Aktualisierung an- bzw. auszuschalten. Die Betätigung dieser Schaltfläche startet eine Aktualisierungsschleife, die den Visualisierungsprozess in einem feststehenden Zeitintervall auslöst. Zur effizienteren Aktualisierung des *Burndown Chart* sollte zukünftig eine *Socket*-Verbindung zwi-

schen den verschiedenen Komponenten eingerichtet werden. Infolgedessen kann Jenkins die Aktualisierung des *Burndown Chart* auslösen, sobald der *Test Report* eines neuen Bearbeitungsstands vorliegt.

7.2.6 Virtualisierung mit Docker

Um die in Abschnitt 7.1 genannten Forderung nach leichter Reproduzierbarkeit, Portierbarkeit und Entkopplung des technischen Systems zu erreichen, werden die serverseitigen Komponenten mit Hilfe von Docker[10] virtualisiert. Dabei handelt es sich um eine Plattform für die Entwicklung, die Veröffentlichung und die Ausführung von Applikationen in Containern, die durch eigene Laufzeitumgebungen von der Hardwareplattform und dem Betriebssystem des ausführenden Servers entkoppelt sind. Im Gegensatz zu virtuellen Maschinen, die ein vollwertiges Betriebssystem mit einen virtuellem Zugriff auf die Ressourcen des Servers darstellen, teilen die Container unter Verwendung von Docker den Kernel des ausführenden Betriebssystems. Ihre Ausführung benötigt daher nicht mehr Ressourcen als jeder andere ausführbare Prozess. Infolgedessen bietet Docker durch die Virtualisierung mit Hilfe von Containern eine leichtgewichte Lösung für die Entkopplung von der Hardwareplattform des Servers.

Die Grundlage für die Ausführung von Containern bilden die sogenannten *Images*, die ein ausführbares Paket darstellen, das alles beinhaltet, was für die Ausführung einer Applikation notwendig ist. Darunter befinden sich beispielsweise der Quelltext der Applikation, benötigte Binärdateien, verwendete Bibliotheken, Umgebungsvariablen und Konfigurationsdateien. Um einen Container zu starten, werden die *Images* ausgeführt. Ein Container spiegelt somit eine laufende Instanz eines *Image* wider (vgl. Klassen und Objekte des objektorientierten Paradigmas). Das Erstellen der *Images* erfolgt mit Hilfe von *Dockerfiles*, die eine Reihe an Befehlen zur Einrichtung der *Images* enthalten. Des Weiteren bietet Docker bereits einige eingerichtete *Images* an, die heruntergeladen und gestartet werden können, ohne zuvor ein eigenes *Dockerfile* zu erstellen.

Durch mehrfaches Ausführen der *Images* lassen sich identische Container erzeugen. Wer-

[10] Docker, Inc. Verfügbar unter https://www.docker.com; Abgerufen am 18. Oktober 2018. Dokumentation verfügbar unter https://docs.docker.com; Abgerufen am 18. Oktober 2018.

den die serverseitigen Komponenten des technischen Systems mit Hilfe von Containern umgesetzt, beschränkt sich der Großteil des sich wiederholenden Einrichtungsaufwands auf das einmalige Erstellen der *Images*. Daher wird zur Virtualisierung des technischen Systems jede der zuvor beschriebenen serverseitigen Komponenten in einem eigenen Container ausgeführt. Während für den Reverse Proxy mit NodeJS sowie für Jenkins dazu entsprechende *Dockerfiles* umgesetzt und *Images* erstellt werden, basiert der Container für GitLab auf einem von Docker bereitgestellen *Image*.

Mit der Instanziierung eines *Image* bzw. der Erzeugung und Ausführung eines Containers entsteht ebenfalls ein *File System* (dt.: „Dateisystem“), das die während der Laufzeit des Containers erhobenen Daten vorhält. Dieses *File System* und somit auch die entsprechenden Daten existiert innerhalb des jeweiligen Containers und ist an dessen Lebensdauer gebunden. Durch Löschen eines Containers gehen daher ebenfalls seine Daten verloren. Für die angestrebte Portierbarkeit gilt es jedoch, den Zustand – respektive die Daten – der serverseitigen Komponenten unabhängig von der Lebensdauer der Container zu persistieren und zu einem späteren Zeitpunkt auf anderen Servern wiederherzustellen. Zu diesem Zweck stellt Docker die Funktionalität zur Verfügung, Verzeichnisse des Servers an Verzeichnisse des Containers zu binden (*Bind Mount*). Jede Veränderung, die ein Container an den gebundenen Verzeichnissen vornimmt, äußert sich somit ebenfalls in den Verzeichnissen des Servers (und umgekehrt). Dadurch bleiben die Daten auch über die Lebensdauer der Container hinaus bestehen. Durch erneutes Einbinden der Verzeichnisse in *File Systems* neuer Container lässt sich der vorherige Zustand der Komponenten wiederherstellen. Weiterhin bietet Docker für die Kommunikation zwischen Containern die Möglichkeit die einzelnen Services mit virtuellen Netzwerken zu vernetzen. Damit die einzelnen Komponenten wie zuvor beschrieben miteinander kommunizieren können, wird auf dem Server des Weiteren ein solches virtuelles Netzwerk eingerichtet, in dem sich alle Container befinden. Unter Verwendung von Docker ergibt sich das in Abbildung 7.9 dargestellte System. Sowohl an dem Container für Jenkins als für GitLab wird zur Sicherung der Daten je ein Ordner des lokalen Dateisystems des Servers gebunden. Infolge der Umsetzung der Virtualisierung mit Docker kann der *Reverse Proxy* nicht mehr auf die Ordnerstruktur von Jenkins zugreifen. Damit dynamische Abfrage der Jenkins-Jobs weiterhin möglich ist, wird sowohl an den Container für Jenkins als auch für den

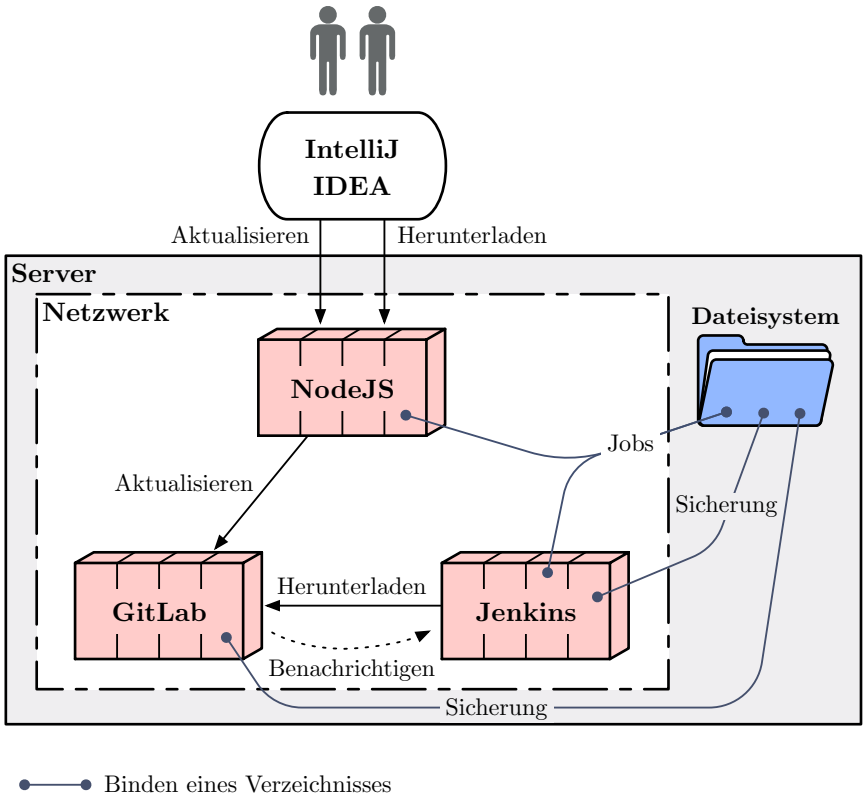

Abbildung 7.9: Entkopplung der serverseitigen Komponenten von der verwendeten Hardwareplattform mit Containern unter Verwendung von Docker.

Reverse Proxy ein Ordner gebunden, in dem Jenkins die Jobs verwaltet. Sobald Jenkins dann einen neuen Job erstellt, ist dieser auch in dem Dateisystem des *Reverse Proxy* verfügbar.

Neben der leichten Reproduzierbarkeit, Portierbarkeit und Entkopplung bietet die Virtualisierung mit Docker auch eine hohe Flexibilität sowie Skalierbarkeit des eingerichteten Systems. So erlaubt es Docker beispielsweise, die Ausführung der unterschiedlichen Container auf mehrere Servern zu verteilen. Des Weiteren können ebenfalls mehrerer identische Netzwerke für unterschiedliche Lehrveranstaltungen auf demselben Server eingerichtet werden. Durch die Möglichkeit das technische System zu Testzwecken ebenfalls auf lokalen Rechnern auszuführen, erleichtert sich weiterhin maßgeblich die Entwicklung.

KAPITEL 8

Zusammenfassung & Ausblick

Diese Arbeit zeigt die Entwicklung sowie die Umsetzung eines agilen Lehrkonzepts für die Vermittlung des Programmierens, das auf der Anwendung der agilen Methoden aus der Softwareentwicklung – insbesondere des Paradigmas der testgetriebenen Entwicklung – beruht. Dazu wird ein bestehendes Lehrkonzept untersucht und durch den Transfer der agilen Methoden auf die Lehre verbessert.

Zu Beginn dieser Arbeit werden die Aspekte der Unterrichtsplanung vorgestellt, die zur Erarbeitung des agilen Lehrkonzepts Anwendung finden. Aus der Betrachtung verschiedener Ansätze qualitativ hochwertiger Lehre wird eine Definition „Guter Lehre“ erarbeitet, die als Zielsetzung und Kontrollwerkzeug für die Ausarbeitung des zukünftigen Lehrkonzepts dient. Entsprechend dieser Definition ergibt sich „Gute Lehre“, wenn sie den Lernprozess der Studierenden in den Mittelpunkt stellt, diesen kontinuierlich verbessert, ausreichend Feedback zur Reflexion bietet, eine hohe Transparenz aufweist und neben fachlichen auch soziale Kompetenzen sowie Kollaboration und Interaktion adressiert.

Die zuvor beschriebene Definition wird anschließend verwendet, um das bestehende Lehrkonzept der Veranstaltung „Algorithmen und Programmieren“ zu analysieren, zu bewerten und um Ansätze für die Entwicklung des agilen Lehrkonzepts zu erarbeiten. Die bestehende Methodik sieht die aktive und kollaborative Bearbeitung von Programmieraufgaben vor und bietet infolge einer nachträglichen Besprechung der Lösung ebenfalls Strukturen für Feedback zur Reflexion. Die geplante Methodik des derzeitigen Lehrkonzepts erfüllt somit die wesentlichen Aspekte der zuvor genannten Definition. Dennoch zeigt die Praxis, dass ein Großteil der Studierenden die intendierten Lernziele nicht erreicht. Dies begründet sich in einigen Fehlannahmen, die der Konzeption der Methodik zugrunde liegen. Durch motivationale und kulturelle Diskrepanzen der internationalen Studierenden erschwert sich häufig die kollaborative Bearbeitung der Übungsaufgaben. Auch stellen sich die Aufgaben für die aktive und selbstständige Bearbeitung durch die Studierenden als zu komplex heraus.

Um das bestehende Lehrkonzept zu verbessern, gilt es daher, einerseits die Aufgabenstellungen umzugestalten und andererseits die Kollaboration durch formelle Verhaltensregeln zu unterstützen. Beide Verbesserungsansätze lassen sich durch die Anwendung der agilen Methoden der Softwareentwicklung in der Lehrveranstaltung adressieren. So kann beispielsweise eine Rollenverteilung wie bei *Scrum* oder *Pair Programming* dabei helfen, die kulturellen und motivationalen Barrieren zwischen den beteiligten Personen zu überwinden und die Kollaboration zu fördern. Weiterhin lassen sich die komplexen Programmieraufgaben durch Tests im Sinne der testgetriebenen Entwicklung in mehrere kleine, schrittweise zu bearbeitende Teilprobleme zergliedern. Infolgedessen werden die Studierenden kognitiv entlastet und die aktive, selbstständige Bearbeitung wird erleichtert.

Da die agile Softwareentwicklung mit der angestrebten Selbstorganisation, Kollaboration, Transparenz, Kommunikation und Reflexion eine hohe Schnittmenge verfolgter Ziele mit der qualitativ hochwertiger Lehre aufweisen, werden weitere positive Effekte erwartet. Durch den inkrementellen und iterativen Ansatz von TDD erhöht sich beispielsweise das Feedback während der Bearbeitung der Übungsaufgaben. Auch der zyklische *Scrum*-Prozess wirkt sich positiv auf die Lehrveranstaltung aus, da er regelmäßige Be-

sprechungen zwischen den beteiligten Personen vorsieht. Dadurch verstärkt sich neben der Transparenz der Lehrveranstaltung weiterhin das Feedback, das zur stetigen Evaluation und Verbesserung des Lehrkonzepts sowie der individuellen Lernprozesse genutzt werden kann. Aus den zuvor genannten Gründen basiert das Lehrkonzept auf einem wöchentlichen, *Scrum*-ähnlichen Zyklus, der die Vorlesungsveranstaltung, die testgetriebene Bearbeitung von Übungsaufgaben mit *Pair Programming* sowie eine anschließende Besprechung im Gruppenverband mit der Lehrperson vorsieht.

Zur weiteren Steigerung von Transparenz und Feedback wird die Veranstaltung von einem technischen System begleitet, das die Lernfortschritte der Gruppen visualisiert. Dazu erfasst es die Bearbeitungsstände sowie die zugehörigen Zeitpunkte und stellt sie in einem *Burndown Chart* dar. Das technische System basiert auf einem VCS, das auf einem zentralen Server eingerichtet ist. Des Weiteren verfügt es auf dem Server über ein Werkzeug für CI, das die Tests der Bearbeitungsstände automatisiert auswertet und die Datengrundlage des *Burndown Charts* erstellt. Dazu müssen die Studierenden nach dem Bearbeiten einer Testmethode einen neuen Bearbeitungsstand auf dem VCS ablegen. Damit die Bedienung des technischen Systems nicht den Workflow der Studierenden stört, sollten die Studierenden während der Bearbeitung nur mit einer Applikation konfrontiert werden. Aus diesem Grund sollte die gewählte IDE den Zugang zu dem VCS unterstützen. Auch die Visualisierung des *Burndown Charts* sollte in die IDE integriert werden. Dazu wird entsprechendes Plugin umgesetzt, das IDE um ein *Tool Window* erweitert und das *Burndown Chart* anzeigt. Aufgrund der leichten Erweiterbarkeit durch Plugins wird IntelliJ IDEA als IDE verwendet, die durch ein bereits bestehendes Plugin mit Git als VCS kompatibel ist. Als Werkzeug für CI verwendet das technische System Jenkins.

Das im Rahmen dieser Arbeit entwickelte und umgesetzte agile Lehrkonzept lässt entsprechend der betrachteten Ansätze der qualitativ hochwertigen Lehre ertragreiche Lernerfolge erwarten. Dabei handelt es sich jedoch nur um theoretische Erwartungen, die es durch die Anwendung des Konzepts im folgenden Semester zu belegen gilt. Das hier beschriebene agile Lehrkonzept spiegelt lediglich eine Idee der Konzeption wider, die die derzeitige Praxis der Softwareentwicklung zugunsten der Lehrqualität in die Lehrver-

anstaltung einbindet. Neben den theoretisch zu erwartenden positiven Effekten auf den Lernerfolg werden die Studierenden dadurch ebenfalls auf den zukünftigen Berufsalltag als Softwareingenieur vorbereitet.

Bringt die Anwendung des entwickelten Lehrkonzepts positives Feedback seitens der Studierenden hervor und sorgt für die erwartete Steigerung der Lernerträge, sollte zukünftig ein Transfer auf weitere Vorlesungen des iRMB (oder Programmierveranstaltungen anderer Institute) erfolgen. In diesem Fall sollte ebenfalls die Funktionalität des Plugins zur Visualisierung des *Burndown Charts* vorangetrieben werden. In der derzeitigen Umsetzung des technischen Systems basiert die automatisierte Aktualisierung des *Burndown Charts* auf einer Schleife, die nach einem festen Zeitintervall die neue Datengrundlage von dem Server herunterlädt und die Darstellung anpasst. Dabei handelt es sich um eine ineffiziente Lösung. Für eine effizientere Lösung könnte in Zukunft eine *Socket*-Verbindung zwischen Jenkins und der Visualisierung eingerichtet werden, die eine gezielte Aktualisierung des *Burndown Charts* nach dem Auswerten neuer Bearbeitungsstände erlaubt. Ein weiteres noch umzusetzenden *Feature* bildet die Einbindung des Git-Plugins von IntelliJ IDEA in das *Tool Window* des Visualisierungs-Plugins. Zwar erlaubt die IDE bereits durch Menüschaltflächen die Funktionalität des VCS aufzurufen, eine übersichtlichere Zusammenstellung der Schaltflächen würde jedoch den Workflow der Studierenden weiter unterstützen. Darüber hinaus sieht die bisherige Umsetzung des technischen Systems vor, dass die Aufgabenstellungen schon zu Beginn der Bearbeitung alle Testfälle enthält. Dadurch könnte die Aufmerksamkeit der Studierenden von dem aktuell zu bearbeitenden Test abgelenkt werden. Um diesem Ablenkunspotenzial zu entgehen, könnte das technische System dahingehend erweitert werden, dass es die *Test Suite* bearbeitet und die zu bearbeitenden Tests sukzessive in die Test-Datei einfügt. Den Studierenden liegen dadurch nur der aktuell zu bearbeitende sowie die bereits bestandenen Testfälle vor.

ANHANG A

Auswertung der Prüfungsaufgabe LinkedList

Im Rahmen der Prüfung wird üblicherweise überprüft, ob die Studierenden im Laufe des Semesters die in den Lernzielen angestrebten Kompetenzen erreichen. Die Lernziele der Vorlesung „Algorithmen und Programmieren“ beziehen sich auf die Vermittlung der Kompetenz des eigenständigen Programmierens. Im Einzelnen sollen die Studierenden erlernen, mäßig komplexe Simulationen sowie ausgewählte Algorithmen und Datenstrukturen eigenständig zu programmieren (siehe Abschnitt 3.1.1).

Zur Überprüfung dieser Kompetenzen enthält die schriftliche Klausur neben Aufgaben, die sich eher auf die Abfrage von konzeptionellem Wissen beziehen, ebenfalls auch eine Aufgaben, die es verlangt Quelltext zu implementieren (auf Papier). Dadurch soll überprüft werden, ob die Studierenden in der Lage sind eigenständig Quelltext umzusetzen. Die Auswertungen dieser Prüfungsaufgabe können auch zur Evaluation dienen und geben Aufschluss darüber, inwieweit die Studierenden die formulierten Lernziele erreichen. Im Vergleich zu den erreichten Gesamtnoten der Studierenden trifft die Auswertung dieser Prüfungsaufgabe ebenfalls eine Aussage über die konstruktive Abstimmung (*Constructive Alignment*; siehe Abschnitt 2.2.3) der der didaktischen Elemente.

Der nachfolgende Anhang zeigt die Auswertung der Prüfungsaufgabe *LinkedList*, die es von den Studierenden verlangt, vorgegebenen Quelltext zu vervollständigen, um das Anfügen von Elementen an eine einfach verkettete Liste zu ermöglichen. Diese Aufgabe war sowohl im Wintersemester 2017/18 als auch im Sommersemester 2018 Bestandteil der Klausur. Dazu werden zunächst die Kriterien für die Ermittlung der Noten gezeigt. Anschließend wird die Aufgabenstellung vorgestellt. Danach werden die erreichten Punktezahlen in der Aufgabe *LinkedList* sowie die Gesamtpunktzahl und die Klausurnoten dargelegt. Abschließend erfolgt Auswertung der gezeigten Prüfungsergebnisse.

A.1 Ermittlung der Klausurnoten

Bei der Klausur handelt es sich um eine schriftliche Klausur mit der Länge vom 60 Minuten. Es werden 60 Punkte vergeben. Die Notensprünge ergeben sich gemäß Tabelle A.1.

Tabelle A.1: Festgelegte Intervalle der erreichten Punktzahlen für die Ermittlung der Klausurnoten

Erreichte Punktzahl	**Klausurnote**
60 ... 58	1,0
57 ... 55	1,3
54 ... 52	1,7
51 ... 49	2,0
48 ... 46	2,3
45 ... 43	2,7
42 ... 40	3,0
39 ... 27	3,3
36 ... 34	3,7
33 ... 30	4,0
<30	5,0

A.2 Aufgabenstellung LinkedList

Abbildung A.1 zeigt die zu bearbeitende Aufgabenstellung der Prüfungsaufgabe *LinkedList* inklusive der Musterlösung (grün). Für die Bearbeitung dieser Aufgabenstellung sind sechs Minuten vorgesehen. Sie stellt daher mit 6 Punkten ein Zehntel der gesamten Prüfungsleistung dar.

Implementieren Sie die Methode `append`, die einen übergebenen Wert am Ende einer verketteten Liste anfügt. Beachten Sie, dass die Klasse `LinkedList` sowohl eine verkettete Liste als auch einen einzelnen Knoten einer verketteten Liste repräsentiert.

```
public class LinkedList {
    private Object value; // the actual data

    //next is the link to the next node otherwise null
    private LinkedList next;

    public LinkedList(Object value, LinkedList node) {
        this.value = value;
        this.next = next;
    }
    public void append(Object newValue) {
        // this is also the head node
        LinkedList current = this;
        while(current.next != null) {
            current = current.next;
        }
        current.next = new LinkedList(newValue, null);
    }
}
```

Abbildung A.1: Aufgabenstellung der Prüfungsaufgabe *LinkedList* aus dem Wintersemester 2017/18 und dem Sommersemester 2018.

A.3 Wintersemester 2017/18

Betrachtet man die Klausurergebnisse des Wintersemesters 2017/18 aus Tabelle A.2, fällt auf, dass 34 von 46 teilnehmenden Studierenden die Aufgabe *LinkedList* nicht oder nicht zufriedenstellend bearbeiteten ($\leq$ 2 Punkte; rot markiert). Lediglich 6 von 46 teilnehmenden Studierenden bearbeiteten die Prüfungsaufgabe zufriedenstellend und erhielten 5 bis 6 Punkte (grün markiert). Daraus ergibt sich eine durchschnittlich erreichte Punktzahl von 1,3 Punkten. In der Klausur fielen drei Studierende mit der Note 5,0 durch. Insgesamt ergibt sich eine durchschnittliche Note von 3,3.

Tabelle A.2: Tabellarische Darstellung der Prüfungsergebnisse sowie der erreichten Punktzahlen in der Aufgabe *LinkedList* aus dem Wintersemester 2017/18

Studierende	**Punkte LinkedList**	**Punke Gesamt**	**Note**
1	5,0	51,0	2,0
2	6,0	57,0	1,3
3	2,0	48,0	2,3
4	0,0	30,0	4,0
5	0,0	23,0	5,0
6	4,0	48,0	2,3
7	5,0	48,5	2,0
8	1,0	40,0	3,0
9	0,0	40,0	3,0
10	0,0	41,0	3,0
11	2,5	38,0	3,3
12	4,0	42,0	3,0
13	0,0	46,0	2,3
14	0,0	24,5	5,0
15	0,0	37,0	3,3
16	0,0	38,5	3,3
17	1,5	38,5	3,3
18	4,5	32,5	4,0

19	0,0	34,0	3,7
20	0,0	32,0	4,0
21	5,0	38,0	3,3
22	0,0	31,5	4,0
23	0,0	39,5	3,0
24	0,0	31,0	4,0
25	0,0	30,0	4,0
26	0,0	31,0	4,0
27	0,0	47,0	2,3
28	0,0	30,0	4,0
29	0,0	30,0	4,0
30	5,0	49,0	2,0
31	1,5	35,0	3,7
32	4,5	48,0	2,3
33	0,0	34,0	3,7
34	0,0	31,5	4,0
35	0,0	35,0	3,7
36	0,0	43,0	2,7
37	0,0	32,5	4,0
38	0,0	45,0	2,7
39	6,0	56,0	1,3
40	0,0	33,5	3,7
41	0,0	44,0	2,7
42	0,0	0,0	5,0
43	0,0	50,5	2,0
44	0,0	22,5	5,0
45	0,5	42,5	2,7
46	3,5	36,0	3,7
Durchschnitt:	1,3	37,7	3,3

A.4 Sommersemester 2018

Ein ähnliches Ergebnis liefert die tabellarische Darstellung der Prüfungsergebnisse aus dem Sommersemester 2018 (siehe Tabelle A.3). Hier erreichten ebenfalls nur 6 teilnehmende Studierende eine zufriedenstellende Lösung ($\geq$ 5 Punkte; grünb markiert) der Aufgabe *LinkedList*. Des Weiteren erhielten 34 von 50 teilnehmenden Studierenden nicht mehr als 2 Punkte (rot markiert). Für die Aufgabe *LinkedList* ergibt sich eine durchschnittlich erreichte Punktzahl von 1,5 Punkten. Im Sommersemester 2018 fielen zwei Studierende mit der Not 5,0 durch. Die durchschnittliche Note beträgt 2,5.

Tabelle A.3: Tabellarische Darstellung der Prüfungsergebnisse sowie der erreichten Punktzahlen in der Aufgabe *LinkedList* aus dem Sommersemester 2018

Studierende	**Punkte LinkedList**	**Punke Gesamt**	**Note**
1	0	50,0	2,0
2	2,0	46,5	2,3
3	0,0	25,5	5,0
4	0,0	35,5	3,7
5	0,0	33,5	3,7
6	0,0	42	3,0
7	5,0	55,5	1,3
8	3,0	52	1,7
9	0,0	49,5	2,0
10	0,0	42	3,0
11	0,0	44,5	2,7
12	0,0	34	3,7
13	3,5	51,5	1,7
14	5,5	56,5	1,3
15	0,0	22,5	5,0
16	5,0	49,5	2,0
17	0,0	53,0	1,7
18	0,0	47,0	2,3

19	1,0	40,5	3,0
20	2,0	52	1,7
21	0,5	46,5	2,3
22	3,0	55,0	1,3
23	0,0	46,5	2,3
24	0,0	51,0	2,0
25	1,5	44,0	2,7
26	5,0	54,5	1,3
27	2,0	52,0	1,7
28	0,0	34,0	3,7
29	2,0	49,5	2,0
30	0,0	50,0	2,0
31	3,0	49,5	2,0
32	5,0	57,5	1,0
33	0,0	40	3,0
34	3,0	44,5	2,7
35	0,0	38,5	3,3
36	0,0	42,5	2,7
37	0,0	38,0	3,3
38	0,0	47,0	2,3
39	3,0	46,0	2,3
40	3,0	40,0	3,0
41	6,0	55,5	1,3
42	3,0	48	2,3
43	0,0	39,5	3,0
44	2,0	51,5	1,7
45	3,0	52	1,7
46	0,0	45,5	2,3
47	0,0	37,5	3,3

48	0,0	39,0	3,3
49	0,0	46,5	2,3
50	3,0	33,0	4,0
Durchschnitt:	1,5	45,2	2,5

A.5 Auswertung

Die erreichten Punktzahlen in der Aufgabe *LinkedList* lassen darauf schließen, dass die Mehrheit der Studierenden nach Absolvieren der Lehrveranstaltung nicht in der Lage sind, eigenständig Logik in Quelltext zu übertragen und umzusetzen. Sie erreichen daher nicht vollständig die intendierten Lernziele. Da die Studierenden dennoch zum Großteil die Klausuren bestehen, lässt sich des Weiteren feststellen, dass keine vollständige Abstimmung der Klausur mit den intendierten Lernzielen vorliegt. Auch wenn die Studierenden mehrheitlich nicht in der Lage sind eigenständig zu programmieren, bestehen sie dennoch – sogar häufig mit befriedigenden bis guten Ergebnissen (Durschnittsnoten von 2,5 bzw. 3,3) – die Klausur.

Literaturverzeichnis

[1] R. Aichner et al. *Grundprinzipien und Erfolgsfaktoren guter Lehre*. Hrsg. von ProLehre Technische Universität München, Bundesministerium für Bildung und Forschung. Verfügbar unter https://www.prolehre.tum.de/fileadmin/w00btq/www/Angebote_Broschueren_Handreichungen/prolehre_erfolgsfaktoren.pdf, abgerufen am 18. Oktober 2018. München.

[2] L. W. Anderson et al. *A taxonomy for learning, teaching, and assessing: a revision of Bloom's taxonomy of educational objectives*. Longman, 2001.

[3] Arbeitsstelle für Hochschuldidaktik, Hrsg. *Taxonomie-Matrix zur Analyse und Selbstevaluation von Hochschullehre (TAMAS)*. 2017.

[4] B. Baumert et al. "Constructive Alignment als didaktisches Konzept". In: *Journal Hochschuldidaktik* Heft 1-2 (2013), S. 23–27.

[5] K. Beck. *Test-driven development: by example*. Boston: Addison-Wesley, 2003.

[6] K. Beck et al. *Extreme Programming Explained: Embrace Change*. 2. Aufl. Addison-Wesley Professional, 2004.

[7] K. Bielak. *Diskussionspapier "Gute Lehre an der TU Braunschweig"*. Hrsg. von Projektgruppe Lehre und Medienbildung, Technische Universität Braunschweig. Verfügbar unter https://www.tu-braunschweig.de/Medien-DB/teach4tu/

Diskussionspapier / 2017 - 01 - 19_diskussionspapier_gute_lehre . pdf, abgerufen am 18. Oktober 2018. 2017.

[8] J. Biggs. *Aligning Teaching and Assessment to Curriculum Objectives*. (Imaginative Curriculum Project, LTSN Generic Center), 2003.

[9] B. S. Bloom et al. *Taxonomy of Educational Objectives: The Classification of Educational Goals, by a commitee of college and university examiners. Handbook I: Cognitive Domain*. Taxonomy of Educational Objectives: The Classification of Educational Goals. Longmans, Green, 1956.

[10] G. Booch. *Object-oriented analysis and design with applications*. 3. ed., 1. printing. The Addison-Wesley object technology series. Upper Saddle River, NJ: Addison-Wesley, 2007.

[11] F. Brooks. "No Silver Bullet – Essence and Accidents of Software Engineering". In: *Computer* 20.4 (1987), S. 10–19. DOI: 10.1109/MC.1987.1663532.

[12] E. W. Dijkstra. "The humble programmer". In: *Commun. ACM* 15.10 (), S. 859–866.

[13] Z. Dörnyei. *Teaching and Researching Motivation*. Applied linguistics in action. Longman, 2001.

[14] Fakultät für Architektur, Bauingenieurwesen und Umweltingenieurwesen. Technische Universität Braunschweig, Hrsg. *Beschreibung des Studiengangs Bauingenieurwesen (PO WS2017/18), Master*. Verfügbar unter http : / / www . fk3 - documents . tu - bs . de / Bau / bau - msc - modulhandbuch_2018 - 04 - 04 . pdf, abgerufen am 18. Oktober 2018. 2018.

[15] Fakultät für Architektur, Bauingenieurwesen und Umweltingenieurwesen. Technische Universität Braunschweig, Hrsg. *Beschreibung des Studiengangs Computational Sciences in Engineering (CSE) (PO2013), Master*. Verfügbar unter https: / / www . tu - braunschweig . de / Medien - DB / cse / modulhandbuch15 . pdf, abgerufen am 18. Oktober 2018. 2015.

[16] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[17] S. Freeman et al. *Growing Object-Oriented Software, Guided by Tests*. 1st. Addison-Wesley Professional, 2009.

[18] J. Hattie. *Visible Learning: A Synthesis of Over 800 Meta-Analyses Relating to Achievement*. Taylor & Francis, 2008.

[19] C. Iller et al. "Prüfungen als Evaluation der Kompetenzentwicklung im Hochschulstudium". In: *Das Hochschulwesen*. Bielefeld: UniversitätsVerlagWebler, 2009, S. 195–201.

[20] B. Jorzik, Hrsg. *Charta guter Lehre. Grundsätze und Leitlinien für eine bessere Lehrkultur*. Essen: Stifteverband für die Deutsche Wissenschaft, 2013.

[31] A. Kaysser-Pyzalla, Hrsg. *Amtliches Bekanntmachung (Verkündungsblatt Nr. 1209), Allgemeiner Teil der Prüfungsordnung für die Bachelor-, Master-, Diplom- und Magisterstudiengänge an der Technischen Universität Braunschweig*. Verfügbar unter https://www.tu-braunschweig.de/Medien-DB/gb1/nr-1209.pdf, abgerufen am 18. Oktober 2018. 2018.

[21] C. Larman. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley, 2004.

[22] E. A. Locke et al. *A Theory of Goal Setting & Task Performance*. Prentice Hall, 1990.

[23] R. F. Mager. *Preparing instructional objectives*. Palo Alto: CA: Fearon Press, 1962.

[24] R. C. Martin et al. *Agile principles, patterns, and practices in C#*. 4. print. Robert C. Martin series. Upper Saddle River, NJ: Prentice-Hall, 2008.

[25] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[26] T. Ohno et al. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.

[27] R. Osherove. *The Art of Unit Testing: With Examples in C#*. 2nd. Greenwich, CT, USA: Manning Publications Co., 2014.

[28] T. Ottmann et al. *Algorithmen und Datenstrukturen, 5. Auflage.* Heidelberg: Spektrum Akademischer Verlag, 2012.

[29] M. Pittman. "Lessons Learned in Managing Object-Oriented Development". In: *IEEE Software* January (1993), S. 43–53.

[30] M. Poppendieck et al. *Lean Software Development: An Agile Toolkit.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[32] G. Reinmann. *Studientext Didaktisches Design.* München, 2011.

[33] C. Richter et al. *Vom Problem zur Evaluation - Folgerungen für die Entwicklung computergestützter Lernumgebungen.* Hannover, 2003.

[34] W. W. Royce. *Managing the Development of Large Software Systems.* 1970.

[35] N. Schaper et al. *Umsetzungshilfen für kompetenzorientiertes Prüfen, HRK-Zusatzgutachten.* 2013.

[36] K. Schwaber. *SCRUM Development Process. Advanced Development Methods.* Burlington, Massachusetts, 1995.

[37] J. C. Stewart et al. *Evaluating Agile Principles in Active and Cooperative Learning.* New York, NY, USA, 2009.

[38] J. Sutherland. *Scrum: The Art of Doing Twice the Work in Half the Time.* First Edition. New York: Crown Business, 2014.

[39] V. Szalvay. *An Introduction to Agile Software Development.* Bellevue, Washington, 2004.

[40] H. Takeuchi et al. *The New New Product Development Game.* Harvard, 1986.

[41] M. van Rahden. *Studienarbeit. Mechanismen und Methoden der agilen Softwareentwicklung.* Braunschweig, 2016.

[42] S. Walzik. *Kompetenzorientiert prüfen: Leistungsbewertung an der Hochschule in Theorie und Praxis.* Kompetent lehren. UTB GmbH, 2015.

[43] J. Wildt. *Vom Lehren zum Lernen – hochschuldidaktische Konsequenzen aus dem Bologna-Prozess für Lehre, Studium und Prüfung.* Berlin, 2005.

[44] J. Wildt et al. “Lernprozessorientiertes Prüfen im Constructive Alignment”. In: *Neues Handbuch Hochschullehre: Lehren und Lernen effizient gestalten*. Hrsg. von Behrendt et al. Stuttgart: Josef Raabe Verlag, 2011, Griffmarke H 6.1 (S. 1–46).

[45] E. Yourdon et al. *Structured design: Fundamentals of a discipline of computer program and systems design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[46] K. Zierer. *Kernbotschaften aus John Hatties Visible Learning*. Paderborn, 2014.

[47] J. Zisenis. *Studienarbeit. Anwendung und Evaluation der testgetriebenen Entwicklung am Beispiel eines plattformübergreifenden Frameworks zur Realisierung nativer Benutzeroberflächen*. Braunschweig, 2018.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorstehende Masterarbeit selbstständig angefertigt und die benutzten Hilfsmittel sowie die befragten Personen und Institutionen vollständig angegeben habe.

Braunschweig, den 18. Oktober 2018

_____

Janis Zisenis