Formale Methoden in der Softwaretechnik-Vorlesung

Bernd Westphal, Albert-Ludwigs-Universität Freiburg
westphal@informatik.uni-freiburg.de

Zusammenfassung

Dieser Artikel stellt die Ergänzung einer Einführung in die Softwaretechnik um Formale Methoden im Bereich Requirements Engineering, Software-Modellierung und Qualitätssicherung vor. Wir beschreiben die Konstruktion der Veranstaltung und berichten empirische Daten und Erfahrungen aus 4 Jahren der Durchführung. Ein Schwerpunkt ist die Beschreibung der verwendeten didaktischen Methoden zur Vermittlung der neuen Inhalte.

Abstract

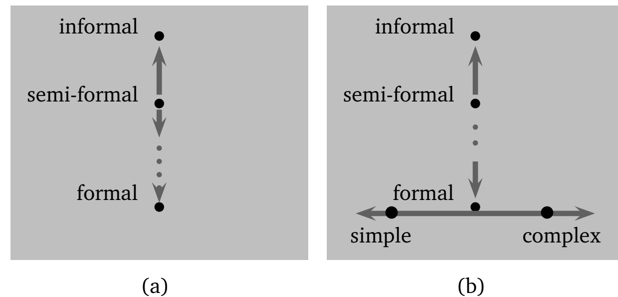
This article presents the extension of an introductory course on software engineering by formal methods in the topic areas requirements engineering, software modelling, and quality assurance. We describe the construction of the course and report empirical data and experience from 4 years of conducting the course. A focus of this article is the description of didactical methods employed for teaching the new content.

CEUR-WS.org/Vol-2358/paper-02.pdf

1 Einleitung

Im Studienplan des Bachelorstudiums der Informatik der Universität Freiburg ist eine einsemestrige Veranstaltung zur Einführung in die Disziplin der Softwaretechnik vorgesehen. Das Modulhandbuch fordert, wie in Deutschland nicht unüblich, eine Übersicht über die Problembereiche, Konzepte und Techniken der Softwaretechnik, wie sie von den bekannten Lehrbüchern, etwa (Sommerville, 2010; Balzert, 2009; Ludewig u. Lichter, 2013), abgedeckt werden. Im Jahre 2015 stand eine Überarbeitung der Vorlesung ‚Softwaretechnik' an, über deren Resultat wir in diesem Artikel berichten.

Zwei Ziele standen bei der Überarbeitung der Materialien im Fokus. Erstens ein „Zurechtrücken" der Proportionen der verschiedenen Bereiche der Softwaretechnik in den Materialien (die über mehrere Jahre von verschiedenen Veranstaltern verwendet und bearbeitet wurden), und zweitens eine neue Ergänzung der Materialien um eine verständliche und nachvollziehbare Einführung formaler Beschreibungssprachen und Analysetechniken (kurz Formale Methoden), also Beschreibungssprachen mit formal definierter Syntax und Semantik und darauf aufbauenden (Semi-)Entscheidungsprozeduren. Das Ziel der Ergänzung um Formale Methoden ist motiviert durch unsere Erfahrung in zahlreichen Projekten mit Partnern aus der Industrie von kleinen Anbietern sicherheitskritischer Systeme (z.B. (Arenis u. a., 2016)) bis hin zu großen Automobilanbietern und Zulieferern (z.B. (Langenfeld u. a., 2016)). In der Projektarbeit nehmen wir eine klare Nachfrage nach und Anwendung (!) von Formalen Methoden im oben genannten Sinne in verschiedenen Bereichen der Softwaretechnik in den Firmen wahr. Hierbei steht üblicherweise nicht die vollständige und umfassende formale Beschreibung von Anforderungen und Entwurfsentscheidungen bzgl. Struktur und Verhalten im Vordergrund (wie es etwa im Luft- und Raumfahrtbereich teilweise angestrebt wird), sondern eine angemessene Verwendung von Formalen Methoden, um bestimmte, besonders kritische oder risikobehaftete Aspekte von Software zu beschreiben und zu analysieren. Dem entsprechend möchten wir *heute* beginnen, die in vielen Einführungen in die Softwaretechnik fest etablierte (semi-formale) Modellierung von Softwareaspekten in unserer Einführungsveranstaltung um eine formale Sicht zu erweitern und den Studierenden eine Übersicht über Konzepte, Möglichkeiten und Voraussetzungen Formaler Methoden für ihre zukünftigen Berufssituationen zu vermitteln.



(a) (b)

Abbildung 1: Einführung Formaler Methoden.

Mit diesem Ziel ergibt sich neben der üblichen Herausforderung, aus dem umfangreichen Angebot der Lehrbücher eine ausgewogene Vorlesung zusammenzustellen, die Herausforderung, Lehrmaterialien für den Bereich Formale Methoden auszuwählen bzw. herzustellen. Interessanterweise ist die zweite Her-

ausforderung keine Teilmenge der ersteren. In den etablierten Lehrbüchern (s.o.) und dem vorhandenen Material wird ganz überwiegend ein Ansatz verfolgt, den wir in Abbildung 1(a) visualisieren. Wir sehen die für z.B. die Beschreibung von Anforderungen verfügbaren Mittel auf einer Achse, die von informal (rein natürlichsprachlich), über semi-formal (mit formaler konkreter Syntax, aber ohne formale abstrakte Syntax oder Semantik) hin zu formal (mathematisch präzise definierte Syntax und Semantik) reichen. Die vorgenannten Lehrbücher und Materialien bewegen sich auf dieser Achse zwischen informal und semiformal und geben kurze Ausblicke auf Formale Methoden, indem Beispiele auf einer intuitiven Ebene diskutiert werden. Bei diesem Ansatz sehen wir ein inhärentes Problem, das bereits Lehman & Buth (Lehmann u. a., 2015) beschreiben: „Es darf nicht erwartet werden, dass die Studierenden UML Modelle erstellen können, wenn in der Vorlesung nur die einzelnen Symbole durchgearbeitet werden und dann auf dieser Basis in der Prüfung Analyseaufgaben auf einem gegebenen Diagramm gestellt werden." Streng genommen können wir schon nicht erwarten, daß Studierende in Bloom's Taxonomie echt leichtere Aufgaben, wie etwa *gegebene* Objektdiagramme daraufhin analysieren, zu welchem der (drei) Wahrheitswerte eine *gegebene* OCL-Formel ausgewertet wird, in Übungen so zu bearbeiten, daß gegenüber den Tutor::innen die Korrektheit der Lösung argumentiert (im besten Falle: bewiesen) werden kann bzw. von Seiten der Tutor::innen die Inkorrektheit eines Lösungsvorschlags bewiesen und systematisch erklärt werden kann. Unser Lösungsansatz besteht darin, in der Veranstaltung Endpunkte der Formalitätsachse zu behandeln. Wir diskutieren sowohl informale Ansätze (und im Themenbereich Requirements Engineering verschiedene semi-formale Ansätze) und springen dann zu vollständig definierten Beschreibungssprachen, die für den Zweck der Vorlesung (bis auf eine Ausnahme) auf einen essentiellen Kern reduziert sind (vgl. Abbildung 1(b)). Auf diese Weise geben wir eine konkrete Übersicht über die gesamte Achse. Für eine ausgewählte Beschreibungssprache diskutieren wir mehr als einen essentiellen Kern, um zu demonstrieren, daß es eine orthogonale Achse der Einfachheit formaler Beschreibungssprachen gibt (Einfachheit sowohl bzgl. der Ausdrucksstärke als auch bzgl. des Aufwands, zu einer vollständigen Definition (von abstrakter Syntax und Semantik) zu gelangen).

Für den Bereich der Formalen Methoden in der Softwaretechnik steht mit (Bjørner, 2006a,b,c) ein dreibändiges Lehrbuch zur Verfügung, das versucht, die Softwareentwicklung vollständig aus formaler Sicht zu vermitteln. Wir haben uns aus zwei Gründen dagegen entschieden, diesem Lehrbuch zu folgen. Zunächst scheint uns das Ziel des Materials von Bjørner zu sein, formale Beschreibungssprachen möglichst in vollem Umfang darzustellen, was schlicht den Rahmen unserer Veranstaltung sprengen würde und für unsere Lernziele nicht notwendig ist. Weiterhin ist das Material u.E. so weit von der (heutigen und mutmaßlich morgigen) Lebenswirklichkeit und den etablierten Lehrbüchern entfernt, daß wir annehmen müssen, daß es Studierenden unnötig schwer fallen würde, andere Lehrbücher oder Fortbildungen in der Arbeitssituation zu unserer Veranstaltung in Bezug zu setzen. Unser (zugegeben: hoher) Anspruch an unsere Veranstaltung ist, daß nach der Teilnahme an unserer Veranstaltung keines der etablierten Lehrbücher große konzeptionelle Überraschungen bereit hält, sondern nur von den Studierenden einordnebares, zusätzliches Hintergrundwissen sowie weitere Techniken liefert.

In diesem Artikel beschreiben wir die Konstruktion unserer Vorlesung zur Einführung in die Softwaretechnik im Grundstudium, in der die in etablierten Lehrbüchern zur Softwaretechnik (s.o.) vorgestellten Inhalte um Techniken zur Beschreibung von Anforderungen und Entwurfsaspekten vollständig formal, also mit konkreter Syntax und präziser abstrakter Syntax und Semantik, *ergänzt* werden, und berichten von Erfahrungen aus vier Sommersemestern mit der neu gestalteten Veranstaltung. Schwerpunkt dieses Artikels sind die Voraussetzungen der Veranstaltung (insbesondere die Einbettung in den Studienplan), Konsequenzen der ergänzten Inhalte für Übungen und Klausur, sowie Organisation und didaktische Maßnahmen (vor allem Bereich des Übungsbetriebs), mit denen wir die Vermittlung und Erarbeitung der neuen Inhalte unterstützen. Die neuen Inhalte als solche werden für den Bereich Requirements Engineering in (Westphal, 2018) vorgestellt und sind zudem auf der Homepage der Veranstaltung frei zugänglich.[1]

Der Artikel ist wie folgt strukturiert. In Abschnitt 2 beschreiben wir die Situation der Veranstaltung im Studienplan der Universität Freiburg sowie der Studierenden zu Beginn der Veranstaltung. Abschnitt 3 legt unsere Lernziele dar und beschreibt, wie wir die neuen Inhalte bzgl. Formaler Methoden durch didaktische Maßnahmen im Übungsbetrieb unterstützen. Einen Überblick über die neuen Inhalte, ihre Einbettung in die gesamte Vorlesung und unsere Erfahrungen bzgl. Lernerfolgen gibt Abschnitt 4. Wir schließen den Artikel mit einer Betrachtung von Evaluationsergebnissen und einer Zusammenfassung.

2 Kontext der Veranstaltung und Situation der Studierenden

Die Veranstaltung ‚Softwaretechnik' ist eine Pflichtvorlesung im B. Sc.-Studiengang Informatik der Universität Freiburg mit einem Umfang von 3+1 SWS für Vorlesung und Übung und 6 ECTS-Punkten. Zusätzlich besuchen Studierende des nicht-konsekutiven Masterstudiengangs, der Studiengänge ‚Embedded

[1] Die URL für das Sommersemester 2018 ist: `https://swt.informatik.uni-freiburg.de/teaching/SS2018/swtvl`

Tabelle 1: Selbsteinschätzung der Erfahrung (Minimum · 1., 2., 3. Quartil · Maximum).

	Proj. Mgmt.	Req. Eng.	Programm.	Modellierung	QA
2016 (77 Antw.)	–	0 · 1 · 1 · 3 · 9	1 · 2 · 3 · 5 · 10	0 · 1 · 1 · 2 · 7	0 · 1 · 2 · 3 · 9
2017 (87 Antw.)	0 · 0 · 1 · 3 · 10	0 · 1 · 1 · 4 · 10	0 · 2 · 3 · 5 · 10	0 · 1 · 1 · 3 · 10	0 · 1 · 1 · 4 · 10
2018 (64 Antw.)	0 · 0 · 1 · 3 · 8	0 · 0 · 1 · 2 · 8	1 · 1 · 3 · 5 · 10	0 · 1 · 1 · 3 · 9	0 · 1 · 2 · 4 · 10

Systems Engineering‘ (ESE) und ‚Mikrosystemtechnik‘ sowie des bivalenten Studiengangs Informatik die Veranstaltung. In den vier Jahren der Veranstaltung haben wir zwischen 80 und 150 angemeldete Teilnehmer::innen beobachtet. Unter den Teilnehmer::innen (der Evaluation[2]) studierten zwischen 73 und 96 % Informatik, der zu 100 % fehlende Teil wird stark von ESE-Studierenden dominiert. Außer in 2016 strebten jeweils mehr als knapp zwei Drittel der Teilnehmer::innen einen B. Sc.-Abschluss an.

Die ‚Softwaretechnik‘ ist im 4. Semester vorgesehen, d.h. wir können einen zweiteiligen Programmierkurs und Algorithmen & Datenstrukturen, Technische und Praktische Informatik (Betriebssysteme, Netzwerke, Datenbanken) und die Mathematik-Vorlesungen voraussetzen. Logik und die Einführung in die Theoretische Informatik sind in Freiburg im 3. Semester vorgesehen, liegen also auch zeitlich vor der ‚Softwaretechnik‘. Parallel zur ‚Softwaretechnik‘ findet für die B. Sc.-Studierenden der Informatik ein Softwarepraktikum statt (vgl. Abschnitt 3).

Entsprechend der Position der ‚Softwaretechnik‘ im Curriculum und aus der Erfahrung der Jahre vor 2015 gehen wir einerseits davon aus, daß der Großteil der Teilnehmer::innen vor unserer Veranstaltung bzgl. der meisten Themenbereiche der Softwaretechnik über wenig bis sehr wenig Erfahrung verfügt. Andererseits wissen wir aus persönlichen Gesprächen, daß es im nicht-konsekutiven M. Sc.-Programm durchaus einzelne Studierende gibt, die über mehrjährige Berufserfahrung verfügen.

Seit 2016 überprüfen wir diese Annahme durch eine Aufgabe auf dem ersten Übungsblatt, in der wir die Teilnehmer um eine Selbsteinschätzung der Vorkenntnisse in den Themenbereichen (die wir in der ersten Vorlesung jeweils grob umreißen) Projektmanagement (seit 2017), Requirements Engineering, Programmierung, Modellierung und Qualitätssicherung bitten. Wir schlagen in der Aufgabe die folgende, subjektiv zu interpretierende Skala vor (die Übungsblätter sind in Englisch verfasst):

0: I have no experience in that activity whatsoever. I have not taken any related subjects during my studies.

1: I have only performed the activity in the context of a lecture or programming course.

10: I have performed the activity in a project with a large user base (100+ users), a large work volume (36+ person-months) or a specific commercial purpose. I have been responsible for the planning and execution of the activity in a software development project within defined resource and time constraints.

Tabelle 1 zeigt die bisherigen Ergebnisse, die unsere Annahmen durchweg bestätigen. Etwas überraschend finden wir, daß sich das obere Quartil der Studierenden im Bereich Programmierung regelmäßig als relativ erfahren einschätzte, sich im Bereich Qualitätssicherung (der die Aktivität des Testens einschließt, von der man erwarten sollte, daß diese mit ernsthafter Programmierung einhergeht) als eine oder zwei Stufen geringer erfahren einordnet.

In der zweiten Vorlesung zeigen und diskutieren wir die Ergebnisse des jeweiligen Jahres. Mit der Präsentation der Ergebnisse verbinden wir die folgenden Botschaften: Erstens, daß wir, d.h. die Veranstalter und die Studierenden, es (auch dieses Jahr wieder) mit einem bzgl. Vorerfahrung eher heterogenen Auditorium zu tun haben. Zweitens, daß die Vorlesung explizit für Studierende angelegt ist, die sich zwischen den Werten 0 und 1 einordnen (und daß diese Studierenden in der klaren Mehrheit sind, man also nicht „allein“ ist, wenn man sich dort eingeordnet hat). Drittens, daß für die Studierenden, die sich bei Werten von 5 und höher einordnen, möglicherweise keine Neuigkeiten vermittelt werden, daß diese Studierenden jedoch explizit eingeladen sind, in Form von Fragen oder Kommentaren ihre Vorerfahrung in den Vorlesungs- und Übungsterminen einzubringen. In der Diskussion der Ergebnisse stellen wir heraus, daß die Inhalte der Vorlesung die Studierenden darauf vorbereiten möchten, in Softwarentwicklungsprojekten höherer Stufen mitzuarbeiten, also Projekte die einen größeren Umfang und eine größere Nutzerbasis haben als jedes Projekt, das man im Studium in Freiburg erlebt, sowie üblicherweise einen wirtschaftlichen Aspekt haben.

3 Ziele und Konstruktion der Veranstaltung

Ein Lernziel unserer Veranstaltung ist zunächst, wie in der Einleitung beschrieben, eine Übersicht über die verschiedenen Themenbereiche der Softwaretechnik entsprechend der etablierten Lehrbücher zu geben. Unser Schwerpunkt ist hierbei die Vermittlung der verschiedenen Probleme, die sich daraus ergeben, daß mehrere Menschen über einen längeren Zeitraum zusammen nichttriviale Softwaresysteme entwickeln, sowie bekannte Lösungsansätze und deren Vor- und Nachteile vorzustellen.

[2] Zwischen 26 und 40 Angaben; leider haben wir keinen direkten Zugriff auf Studiengang oder angestrebten Abschluss.

Der neue Aspekt unserer Veranstaltung ist die vollständig formale Definition von Beschreibungssprachen und Analysetechniken an den entsprechenden Stellen in der Vorlesung (vgl. Tabelle 2). Die Lernziele sind hier das sichere Verstehen und Anwenden der eingeführten Formalen Methoden, sowie die Interpretation von Analyseergebnissen. Die entsprechenden Inhalte und die Einbettung in die Vorlesung werden in Abschnitt 4 beschrieben.

Wir beschreiben die Ziele unserer Veranstaltung in der ersten Vorlesung, um Missverständnisse (und Enttäuschungen) zu vermeiden. In einer Aufgabe des ersten Übungsblattes bitten wir die Studierenden, ausgehend von der ersten Vorlesung ihre Erwartungen an die Veranstaltung in eigenen Worten zu beschreiben. In der zweiten Vorlesung stellen wir eine Auswahl der Antworten vor, und gehen jeweils darauf ein, inwiefern die Veranstaltung diese Erwartungen erfüllen kann. Ein über die Jahre wiederkehrender Punkt, ist der Wunsch, gutes Design zu lernen. Diese Erwartungen liefern einen guten Anlass, erneut auszuführen, daß ein Ziel der Vorlesung ist, fundamentale Designprobleme zu verstehen und Designentscheidungen präzise zu beschreiben und auf Aspekte guten Designs hin zu analysieren, wir jedoch keine konkreten Verfahren zur Erstellung guter Designs für spezifische Anwendungsfelder diskutieren. Wir plausibilisieren diese Zielsetzung durch eine grobe Beschreibung des weiten Spektrums von Software: von Onlineshops, über Standardanwendungen (Textverarbeitung, etc.), kommerzielle Spiele, Betriebswirtschaftliche Software, bis hin zu sicherheitskritischen, eingebetteten Systemen. Jeder dieser Bereiche hat eigene, bewährte Vorstellungen von gutem Design, die untereinander nicht austauschbar sein müssen.

3.1 Klausur

Ähnlich wie (Lehmann u. a., 2015), haben wir sehr früh in der Überarbeitung der Veranstaltung mit der Vorbereitung der Klausur begonnen. Laut Modulhandbuch ist eine schriftliche Prüfung vorgesehen und somit war klar, daß die Vorlesung notwendig schriftlich prüfbare Inhalte in hinreichendem Umfang und Tiefe präsentieren muß. Um das Risiko zu mindern, zum zentral festgelegten Klausurtermin keine geeignete Klausur stellen zu können, haben wir, ausgehend von einem ersten Entwurf des Zuschnitts der Vorlesung, Aufgabenskizzen und eine grobe Festlegung der Proportionen für die Themenbereiche erstellt. Vorlesungsinhalte, Klausuraufgaben und Übungsinhalte wurden dann im Sinne eines guten *Constructive Alignments* (Biggs u. Tang, 2011) aufeinander abgestimmt. Zum Ende der Vorlesungszeit 2015 stand für die Studierenden zur Klausurvorbereitung eine Beispielklausur zur Verfügung, seit 2016 ist die Beispielklausur mit dem ersten Vorlesungstag auf der Lernplattform verfügbar.

Bei der Punkteverteilung der Klausur orientieren wir uns grob an den Stufen der (überarbeiteten) Taxonomie der Lernziele (Bloom, 1956; Anderson u. a., 2001). Circa 50 % der Punkte entfallen auf „leichte“ Aufgaben (Vokabular, einfache Verfahren verstehen und wie in den Übungen (jedoch auf andere Probleme) anwenden), ca. 33 % der Punkte entfallen auf „mittlere“ Aufgaben (ähnlich wie Übungsaufgaben, jedoch Analyse bzw. Transfer notwendig) und ca. 17 % der Punkte sind für „schwere“ Aufgaben (nicht in den Übungen besprochen, neue Sichten auf diskutierte Konzepte, offenere kreative Aufgaben) vorgesehen. Unsere Absicht ist, Kompetenzen über alle kognitiven Kategorien hinweg zu prüfen; in der o.g. Punkteverteilung reicht es z.B. zum Bestehen nicht aus, alle „schweren“ Aufgaben (ggf. intuitiv oder zufällig) zu lösen, sondern es sind Punkte aus den Bereichen Verstehen und Anwenden zum Bestehen notwendig.

3.2 Vorlesungen

Teil unserer Veranstaltung ‚Softwaretechnik‘ ist eine klassische Vorlesung im Umfang von 3 SWS. Um nicht an einem Tag der Woche auf eine Stunde Tutorium umschalten zu müssen, interpretieren wir das 3+1-Format als drei Vorlesungen (à 90 Minuten) und ein Tutorium (à 90 Minuten) in zwei Wochen.

Über die vorhandene Infrastruktur zeichnen wir das gesprochene Wort und den Bildschirminhalt auf und stellen die Aufzeichnungen zeitnah zur Verfügung. Auf diese Weise konnten wir die Vorlesungen ohne Skript bestreiten, da das gesprochene Wort jederzeit zum Nachhören verfügbar ist. Weiterhin können die Teilnehmer::innen, denen unser Vortrag zu langsam oder zu ausführlich ist, die Geschwindigkeit beim Anhören der Aufzeichnung selbst bestimmen. Entsprechend der Selbsteinschätzung der Teilnehmer::innen (vgl. Abschnitt 1) sprechen wir in der Vorlesung lieber zu langsam als zu schnell und betrachten neue Inhalte ggf. aus mehreren Perspektiven, was für einige Teilnehmer::innen zu langsam sein kann. In der Evaluation geben, über die Jahre 2016 bis 2018 aggregiert,[3] von 99 Rückmeldungen knapp 25 % an, sich die Inhalte hauptsächlich durch Anwesenheit anzueignen, und 18 % bzw. 33 % geben ‚teils/teils‘ bzw. ‚hauptsächlich über die Aufzeichnungen‘ an. Entsprechend haben wir eher geringe Besucherzahlen in der eigentlichen Vorlesung, dafür jedoch Besucher::innen, die die Präsenz für Fragen oder Kommentare nutzen möchten, denen wir, so gut es die Zeit erlaubt, Raum geben.

Das Lernziel, eine Übersicht entsprechend der etablierten Lehrbücher zu geben, hat für uns zwei Teilaspekte. Einerseits den Aspekt der Vermittlung von Techniken und Verfahren und andererseits den Aspekt „der Mensch steht im Mittelpunkt“ (Ludewig u. Lichter, 2013). Der erste Aspekt ist relativ gut prüf-

[3] Im Jahr 2015 wurde in der Evaluation zum Vortragsbesuch eine andere, nicht gut vergleichbare Frage gestellt.

bar (und kann als Vorbereitung auf die Klausur gesehen werden). Zum zweiten Aspekt ist unser Ziel, das (gesprochene und schriftliche) Kommunizieren von Softwaretechnik-Problemen und Lösungsideen und die Analyse und Bewertung von Lösungsideen einzuüben. Dieser zweite Aspekt ist ungleich schwerer prüfbar (und kann als Vorbereitung auf „das echte Leben“ gesehen werden) und findet in unserer Veranstaltung schwerpunktmäßig im Übungsbetrieb statt (siehe folgender Abschnitt).

3.3 Übungsaufgaben und Tutorate

Die Vorlesung wird von Übungen begleitet. Vor der ersten Vorlesung jedes 3er Blocks steht ein Übungsblatt zur Verfügung, das direkt nach dieser Vorlesung teilweise und nach der zweiten Vorlesung des Blocks vollständig bearbeitet werden kann, sodaß für jede Aufgabe mindestens eine Woche zur Bearbeitung zur Verfügung steht. Studierende bearbeiten die Aufgaben in Teams aus 2 bis 3 Personen und können ihre Bearbeitungen bis zur Minute vor Beginn des Tutoriums auf der Lernplattform einreichen.

In den Tutorien legen wir Wert auf Interaktion (i.S.v. (Krusche u. a., 2017)). Das heißt, es sind nicht die Tutor::innen, die eine Beispiellösung vorstellen, die „passiv konsumiert“ werden kann („man trifft sich nicht mehr nur, um ‚gemeinsam zuzuhören‘“ (Siegeris, 2017)),[4] sondern der Modus der Tutorien ist „wir erarbeiten eine gute Lösung zusammen“ bzw. „wir analysieren und bewerten Lösungsvorschläge und diskutieren weiterführende Fragen“. Die Tutor::innen sehen wir im Tutorium vor allem als Moderator::in, nicht als Vortragende::r. Hierbei sollen die Lösungswege und -überlegungen transparent werden. Zu diesem Zweck sind die Tutor::innen angehalten, für die technischen Aufgaben (im Sinne des Retrieval-Based-Learning) die Teilnehmer::innen zunächst nach den relevanten Definitionen zu fragen. Bei kleinen Aufgaben schreiben die Tutor::innen Lösungsvorschläge aus dem Auditorium auf den Bildschirm, bei größeren Aufgaben bringen die Tutor::innen bemerkenswerte Lösungen mit ins Tutorium und stellen sie zur Diskussion. Die bemerkenswert guten oder eigenartigen Lösungen werden aus sogenannten *Early Submissions* ausgewählt. Wir bieten denjenigen Teams, die 24 Stunden vor dem Tutorium Lösungen einreichen, einen 10 %-Bonus auf die erzielten Zulassungspunkte. Beispiele für weiterführende Fragen ergeben sich z.B. im Bereich des Requirements Engineering wie folgt. Eine technische Aufgabe kann darin bestehen, eine gegebene Formalisierung einer Anforderung auf (formale) Vollständigkeit zu untersuchen. Ausgehend von der (technischen) Lösung ergibt sich die Frage, wie das Resultat in der Rolle Requirements Engineer zu interpretieren ist und welche Konsequenzen für die Kommunikation mit der Kundenseite ggf. zu ziehen sind. Weiterführende technische Fragen können konstruiert werden, indem man die gezeigte Problemstellung leicht modifiziert und eine erneute Analyse erfragt. Bei der Diskussion von Lösungen und weiterführenden Fragen achten die Tutor::innen auf präzise Sprache, insbesondere die korrekte Verwendung von Softwaretechnik-Fachsprache.

Für die schriftliche Darstellung in den Einreichungen fordern wir regelmäßig die Form „Problem in eigenen Worten darstellen – eigenen Lösungsvorschlag formulieren – begründen, argumentieren, im besten Falle: beweisen, daß der Lösungsvorschlag das definierte Problem löst“ und halten die Tutor::innen dazu an, in Kommentaren zu den Einreichungen kontinuierlich diese Form nachzufragen.[5] Entsprechend sind die (ausgewogen vorkommenden) offenen Modellierungsaufgaben absichtlich unpräzise formuliert. Hierbei ist unserer Erfahrung nach der angemessene Grad an Unschärfe in der Aufgabenstellung im B. Sc.-Programm signifikant geringer als im M. Sc.-Programm. Diese absichtliche Unschärfe und die didaktische Absicht kommunizieren wir ganz explizit: wir gehen (stark) davon aus, daß die allerwenigsten Absolventen in ihrer beruflichen Laufbahn präzise Aufgabenstellungen bekommen werden — und unter dieser Annahme können wir ruhig im 4. Semester langsam mit Aufgaben dieser Art beginnen. Im ersten Jahr der Veranstaltung gab es in der Evaluation vereinzelt Wünsche nach ganz klaren Aufgaben. Unsere Hypothese ist, daß in den meisten vorherigen Veranstaltungen präzise Aufgabenstellungen vorherrschen (und den Lernzielen angemessen sind, vgl. auch Abschnitt 4.2) und die ‚Softwaretechnik‘ sich in diesem Aspekt für die Studierenden ungewohnt anfühlt.

Am Tag vor den Tutorien führen wir jeweils ein Tutorium für die Tutor::innen durch, vor dem die Tutor::innen das Übungsblatt selbst skizzenhaft lösen. In diesen *Tutors' Tutorials* gehen wir (in der Art der Tutorien für die Studierenden) die Aufgaben durch, diskutieren weiterführende Fragen, legen die Lernziele der jeweiligen Aufgaben und Fragen dar und geben Tips zur Moderation. Der Ablauf der Tutorien steht danach inklusive Vorschlägen für weiterführende Fragen und Lernziele als eine Art „Drehbuch“ zur Verfügung. In der ersten Durchführung hatten wir leichte Sorge, daß sich die Tutor::innen in ihrer gestalterischen Freiheit beschränkt fühlen könnten. Hier waren die Rückmeldungen bisher durchweg positiv: ganz im Gegenteil würden die *Tutors' Tutorials* Sicherheit (z.B. im Zeitmanagement) geben und kognitive Kapazitäten für tiefer gehende Fragen und Kommentare von Seiten der Studierenden freihalten. Weiterhin erreichen wir durch diesen Ansatz eine überwiegend gleichbleibende Qualität der Tutorien, unabhängig von dem/der konkreten Tutor::in.

[4] Auch wenn einige Studierende sich dies lt. Evaluation wünschen würden.

[5] Nicht zuletzt im Interesse der Studierenden: Wir gehen davon aus, daß gut präsentierte Lösungsvorschläge bei der Klausurvorbereitung zumindest nicht hinderlich sind.

Tabelle 2: Kursinhalte und Struktur.

Vorlesungen	Themenbereich	Inhaltsübersicht
1	Einleitung	Software, Engineering, Software Engineering; Erfolgreiche Softwareentwicklung, Empirische Daten zum Erfolg von Softwareprojekten; Aufbau des Kurses, Bezug zu anderen Lehrveranstaltungen, Organisatorisches.
2 – 5	Projektmanagement	Softwaremetriken, Skalen; Kostenschätzung (Experten- und algorithmische Schätzung); Projekt, Prozess, Prozess Modellierung; Prozedurmodelle; Prozessmodelle (Agil, V-Modell (*semi-formal*)).
6 – 10	Requirements Engineering	Vokabular: Anforderungen, Anforderungsanalyse; Anforderungen im Entwicklungsprozess; Eigenschaften von Anforderungsspezifikationen (Vollständigkeit, Konsistenz, . . .), Arten von Anforderungen, Analysetechniken; Dictionary; Spezifikationssprachen für Anforderungen: natürlichsprachliche Pattern, Entscheidungstabellen (*formal*), Use Cases und -Diagramme (*semi-formal*), LSCs (*formal*).
11 – 14	Design & Architektur	Modell; Sichten; Strukturmodellierung für Software (Klassen- und Objektdiagramme (*formal*), Proto-OCL (*formal*)); Designprinzipien; Design Patterns; Verhaltensmodellierung für Software (Communicating Finite Automata (*formal*), Query Language (*formal*)); ein Ausblick zu UML State-Machines.
15 – 18	Qualitätssicherung	Testfall, Testausführung, falsch/richtig positive/negative Ergebnisse (*formal*); Grenzen des Softwaretestens; Glas-Box-Testen (Anweisungs-, Verzeigungs-, Term-Überdeckung); Modell-basiertes Testen, Runtime-Verifikation; Programmverifikation (*formal*), Code Review.

In der auf das Tutorium folgenden Woche geben die Tutor::innen individuelle (für die Klausurzulassung relevante) Bewertungen und Kommentare zu den Einreichungen. Hierbei werden die Einreichungen auf zwei verschiedenen Skalen bewertet. Die für die Klausurzulassung relevante *good-will*-Skala bewertet „sinnvolle Bearbeitung“ relativ zum Wissen der Studierenden *vor* dem Tutorium. Hier interpretieren die Tutor::innen unklare Formulierungen im Zweifel zugunsten der Studierenden. Um einen Eindruck von der Bewertung in der Klausur zu geben, bewerten wir außerdem jede Aufgabe auf der *evil*-Skala, die sich am Punkteschema der Klausur und am Wissen *nach* dem Tutorium orientiert, insbesondere darüber, wie bestimmte Aufgabenstellungen im Kontext der Veranstaltung im Zweifel zu interpretieren sind. Hier werden unklare Formulierungen oder Fehler in der Syntax zuungunsten der Studierenden gewertet.

4 Formale Methoden in der Softwaretechnik-Vorlesung

Für die Darstellung von Vokabular, Problembereichen und nicht formalen Inhalten folgt unsere Vorlesung im Wesentlichen dem Lehrbuch (Ludewig u. Lichter, 2013). Wir schätzen an diesem Lehrbuch die explizit adressierte Sicht, „dass sich Software-Engineering vor allem mit den Menschen befasst, die Software in Auftrag geben, entwickeln und ändern oder benutzen“ und das u.E. sehr gelungene „[B]emühen [...] um eine rationale Wertung“ und darum, „alle Quellen [...] vollständig anzugeben“.

Im folgenden beschreiben wir unsere Erweiterung der Inhalte um *formale* Modelle in der Softwareentwicklung, die sich unserer Wahrnehmung nach nahtlos an die Diskussion der Begriffe der Modellierung und die Bedeutung von Modellen in der Softwareentwicklung bereits im ersten Kapitel von (Ludewig u. Lichter, 2013) anfügt. In der Diskussion der (gegenüber (Ludewig u. Lichter, 2013)) neuen Inhalte bemühen wir uns, dem evidenzbasierten Stil des Lehrbuchs zu folgen, also rationale Wertungen zu ermöglichen und soweit möglich die (ursprünglichen) Quellen anzugeben. Entsprechend zeigen wir anhand von Beispielen, was Formale Methoden konkret leisten können, was ihre Fürsprecher::innen versprechen und welche Kritik vorgebracht wird, sodaß die Studierenden Vor- und Nachteile verantwortlich und kontextabhängig abwägen können.

Tabelle 2 gibt einen Überblick über die Inhalte der Vorlesung. Wir beginnen mit dem Themenbereich Projektmanagement, da die Inhalte dieses Bereichs für die Teilnehmer::innen des parallel im 4. Semester stattfindenden Software-Praktikums (SoPra) für B. Sc.-Studierende nützlich sein können. Es gibt von Seiten der Teilnehmer::innen durchaus den Wunsch nach einer Abstimmung zwischen dem SoPra und der ‚Softwaretechnik‘, der aus verschiedenen Gründen nicht praktikabel (und, da das SoPra eine in sich abgeschlossene Veranstaltung mit eigenen Vorlesungen ist, auch nicht notwendig) ist. Ein Grund ist, daß im SoPra bereits in der dritten Woche die Spezifikation und ein Architekturentwurf abzugeben sind. Ein weiterer Grund ist, daß in jedem Jahr über 30 % der ‚Softwaretechnik‘-Teilnehmer::innen das SoPra nicht in ihrem Studienplan haben und Versuche einer engeren Abstimmung bei diesen Teilnehmer::innen zu signifikanten Verwirrungen geführt haben. Als Kompromiss haben einige Übungsaufgaben einen Bezug

zum SoPra, der sich den SoPra-Teilnehmer::innen erschließt, und für die anderen Studierenden einfach „ein Beispielprojekt“ ist.

In der Präsentation und Diskussion der nicht-formalen Inhalte folgen wir (Ludewig u. Lichter, 2013), wobei wir im Zweifel direkt auf die ursprünglichen Quellen zurückgreifen, z.B. die Normtexte IEEE 601.12 und ISO 24765 für Vokabular und IEEE 830 zur Struktur von Anforderungsdokumenten. In den folgenden Abschnitten geben wir einen Überblick über die von uns ergänzten formalen und semi-formalen Inhalte.

Aufgrund unserer in der Einleitung ausgeführten Beobachtung eines Mangels an geeigneten Lehrbüchern, haben wir die formalen Inhalte selbst ausgewählt bzw. konstruiert. Der Überarbeitung der Veranstaltung lag die Hypothese zugrunde, daß es durch eine angemessene Reduktion des Sprachumfangs möglich ist z.B. die formale Spezifikationssprache für Strukturaspekte OCL vollständig definiert einzuführen und die Idee Formaler Methoden und Analysen zu vermitteln, und gleichzeitig der Vermittlung der etablierten nicht-formalen Inhalte einen angemessenen Raum zu geben. Bei der Auswahl der Teilsprachen von z.B. OCL stand das Ziel im Vordergrund, die formalen Sprachen nicht artifiziell zu vereinfachen, sondern echte Teilsprachen auszuwählen, die in Spezialvorlesungen konservativ erweitert werden. Wir konnten hierbei auf eine langjährige Erfahrung aus Forschungs- und Projektarbeit sowie aus unseren Spezialvorlesungen ‚Software, Design, Modelling, and Analysis in UML‘ (UML) und ‚Real-Time Systems‘ im M. Sc. Informatik zurückgreifen, in denen für die Modellierungssprachen jeweils eine vollständige konkrete und abstrakte Syntax sowie eine formale Semantik eingeführt wird. In der UML-Vorlesung sind dies Klassen- und Objektdiagramme, OCL, State-Machines und Sequenzdiagramme.

Bei der Auswahl der Beispiele in der Vorlesung und der Konstruktion der Übungsaufgaben greifen wir soweit möglich auf reale, industrielle Softwareprojekte (bestenfalls aus unserer eigenen Erfahrung) zurück, da so sichergestellt ist, daß wir Nachfragen in beliebiger Detailtiefe zufriedenstellend beantworten können.[6] Dabei verwendet die Vorlesung für die verschiedenen Formalen Methoden verschiedene Beispiele an denen sich die spezifischen Stärken zeigen. Wir möchten (und können redlicherweise) nicht den Eindruck erwecken, daß der Stand der Technik der Formalen Methoden ist, ein Softwaresystem vollständig und durchgängig konsistent formal zu beschreiben. Formale Methoden sind dann effektiv, wenn geeignete Formalismen zur Beschreibung und Analyse spezifischer, besonders kritischer oder komplexer Aspekte von Software mit Bedacht eingesetzt werden.

[6] Dies ist bei artifiziell konstruierten Beispielen ungleich schwerer herzustellen und kann, wenn es nicht gelingt, unnötige Unzufriedenheit auf Seiten der Studierenden hervorrufen.

Wir nutzen bei der Auswahl der Inhalte die relative späte Lage der Veranstaltung im Studienplan aus (vgl. Abschnitt 2). So bauen wir auf ‚Grundlagen der Theoretischen Informatik‘, die Mathematik-Vorlesungen und ‚Rechnerarchitektur‘ auf, in denen die Studierenden bereits mit verschiedenen Formalen Methoden konfrontiert wurden (wenn auch vielleicht nicht unter diesem Namen). Unsere Veranstaltung zeigt gewissermaßen, wie ein großer Teil des bisher Gelernten in der Softwarekonstruktion angewandt werden kann.

4.1 Themenbereich Softwareprojektmanagement

Im Themenbereich Softwareprojektmanagement überwiegen nicht-formale Inhalte (vgl. Tabelle 2). Ein Schwerpunkt sind Softwaremetriken als ein Aspekt der ingenieurmäßigen, objektivierten Softwareentwicklung (inklusive des Bewusstseins für Pseudometriken (vgl. (Ludewig u. Lichter, 2013))). Ein weiterer Schwerpunkt ist die Prozessmodellierung. Hier führen wir zunächst eine semi-formale[7] graphische Beschreibungssprache für Prozesse ein, bestehend aus Rollen, Artefakten, Aktivitäten und den entsprechenden Relationen. Wie zeigen, wie ein konkreter Ablauf (oder Prozess) einer Softwareentwicklung modelliert werden kann (deskriptiv, Abbild), um dann zu zeigen, wie ein graphisches Prozessmodell präskriptiv (Vorbild) verwendet werden kann, um Prozessabläufe vorzugeben. Wir verwenden unsere graphische Beschreibungssprache, um konkrete Prozedur- und Prozessmodelle (Wasserfall, V-Modell, XP, Scrum) vorzustellen bzw. setzen die Notation z.B. der V-Modell-Beschreibung zu unserer Teilsprache in Beziehung.

In den Übungsaufgaben ist anhand von vorgegebenen Prozessbausteinen ein kleines Entwicklungsprojekt zu planen inklusive Besetzung von Rollen mit Personen. Um die Tutor::innen insbesondere auf Fragen und Diskussionen zur Verwendung und zum Nutzen von Prozessmodellen vorzubereiten, haben wir entsprechend des Prinzips „lehren, was wir praktizieren“ in 2018 den Prozess des Übungsbetrieb vollständig modelliert. Die Erläuterung des Prozesses ist aufwendig, dafür weiß über das gesamte Semester jede::r Tutor::in, wer wann welches Artefakt wie zu bearbeiten hat und wir können unsere ganze Aufmerksamkeit den eingereichten Lösungsvorschlägen widmen.

4.2 Themenbereich Requirements Engineering

Der Themenbereich Requirements Engineering (RE) ist in der Veranstaltung etwas stärker gewichtet als die weiteren Themenbereiche (vgl. Tabelle 2). Dies ist vor allem drei Überlegungen geschuldet. Erstens stimmen wir zu (etwa (Sedelmaier u. Landes, 2017)),

[7] d.h. präzise konkrete Syntax, informelle Beschreibung der Bedeutung.

daß dem Themenbereich RE in der Softwareentwicklung eine besondere Relevanz zukommt. Die Relevanz wird dabei in den späteren Themenbereichen sehr natürlich aufgegriffen: Der Begriff der Korrektheit von Softwareentwürfen (insbesondere bzgl. des Verhaltens) und von Software ist *relativ* zu einer Anforderung. Etwa ein Algorithmus kann nur als korrekt (bzgl. einer Anforderung) bewiesen werden, wenn es eine formale Beschreibung der Anforderung gibt. Zweitens beobachten wir, daß die Vermittlung von Problemen und Techniken des RE für Studierende mit wenig oder keiner Vorerfahrung eine besondere Herausforderung darstellt (vgl. Selbsteinschätzung der Studierenden, Abschnitt 2). Drittens führen wir in unserer Veranstaltung im Themenbereich RE zum ersten Mal Formale Methoden ein.

Wir sprechen bei unseren Diskussionen i.d.R. über die u.E. besonders herausfordernde Situation eines Softwareentwicklungsvertrages zwischen getrennten Kunden- und Entwicklungsunternehmen. Die Softwareentwicklung in Start-Ups, innerhalb einer Abteilung oder zwischen Abteilungen desselben Unternehmens hat ggf. andere Rahmenbedingungen.

Bei der Vermittlung von Problemen und Techniken des RE arbeiten wir mit zwei (bisher leider nicht empirisch belegten) Hypothesen zu Ursachen von Schwierigkeiten. Eine Ursache sehen wir darin, daß der weitaus überwiegende Teil der Übungsaufgaben, die die Studierenden bis zum 4. Semester bearbeiten, von Fachleuten in Fachsprache *präzise* formuliert sind (nicht zuletzt, um die Korrektur zu vereinfachen). Entsprechend stellen wir Übungsaufgaben zu unserer Veranstaltung absichtlich unpräzise und halten dazu an, in den Einreichungen zunächst die Aufgabenstellung in eigenen Worten wiederzugeben (vgl. Abschnitt 3.3). Als zweite Ursache vermuten wir, daß der Umgang mit Sprache im Alltag von dem in der RE-Arbeit verschieden ist. Im Alltag sind Menschen üblicherweise gern bereit, einen Satz wie:

> Spielfiguren müssen indirekt gesteuert werden.

direkt als „klar“ zu akzeptieren und keinen Bedarf für Nachfragen zu sehen (fehlende Information wird plausibel aufgefüllt; man sieht nur Information). In der RE-Arbeit geht es unseres Erachtens beim Blick auf den obigen Satz vor allem darum, zu sehen, welche Information der Satz *nicht enthält*.

Um diesen Sichtwechsel anzuregen, führen wir eine Übung durch, in der die Studierenden eine textuell gegebene Anforderung analysieren. Diese Übung wird in verschiedenen Ausprägungen im RE-Unterricht eingesetzt, etwa mit Paaren von studentischen Teams, die jeweils die Kunden- bzw. Entwicklerrolle übernehmen oder mit nicht technisch vorgebildeten (echten) Kunden. Wir verwenden eine Variante mit einem technisch vorgebildeten Kunden, da wir in den anderen Szenarien die Gefahr sehen, daß die Parteien aneinander vorbeireden und es aufwendig ist, eine für alle Beteiligten gute Beispiellösung zu erarbeiten. Unser Kunde ist ein Organisator des SoPra, unsere Anforderung war in 2018 der oben genannte Satz. Die Aufgabe besteht darin, durch Kommunikation mit dem Kunden ein Begriffslexikon zu erarbeiten und die nicht im Satz enthaltenen Informationen „herauszukitzeln“. Um die Aufgabe skalierbar zu gestalten, führen zunächst die Organisatoren unserer Veranstaltung eine Anforderungsanalyse durch und notieren die Findungen in einem Wiki. Die Studierenden kommunizieren mit dem (für sie anonymen) Kunden per Mail, vermittelt durch die Tutor::innen. Mit Hilfe des Wiki können die Tutor::innen die meisten Fragen ihrer Tutanden schon im Stile eines Kunden beantworten, noch unklare Aspekte werden an den SoPra-Organisator weitergeleitet und im Wiki nachgeführt; weiterhin sind die Tutor::innen angehalten, die Kommunikation in der Art eines Coaches zu beobachten und Tips für weitere Fragen zu geben. Für das Tutorium bereiten wir eine Liste von bekannten Beispielspielen vor, die die Anforderung lt. Kunde erfüllen bzw. nicht erfüllen. Im Tutorium sollen die Teilnehmer::innen durch Handzeichen pro Beispiel anzeigen, ob nach ihrer konkreten Anforderungsanalyse das Beispiel vom Kunden akzeptiert oder abgelehnt wird bzw. ob sie sich nicht sicher sind. In 4 Jahren Durchführung dieser Übung hat noch nicht ein Team alle Beispiele korrekt klassifiziert; dies wird durch das Verfahren mit den Handzeichen im Tutorium unmittelbar für die Studierenden erfahrbar.

In der Einführung der Problemstellung der Anforderungsanalyse und nicht-formalen Beschreibung von Anforderungen folgen wir (Rupp u. a., 2014). Wir beginnen die Diskussion von Beschreibungssprachen am nicht-formalen Ende der in Abbildung 1(b) dargestellten Achse mit der natürlichsprachlichen Beschreibung und Sprach-Patterns (Rupp u. a., 2014) und diskutieren Eigenschaften guter Anforderungsdokumente wie Vollständigkeit. Die Präsentation wechselt dann ans formale Ende der Achse und dort zu einem möglichst einfachen Formalismus (vgl. Abschnitt 4.2.1), um dann im semi-formalen Bereich z.B. User Stories und Use Cases einzuführen. Ein komplexer Formalismus (vgl. Abschnitt 4.2.2) und ein Rückblick schließen den Themenbereich ab.

4.2.1 Entscheidungstabellen

Als Einstieg in die formale Beschreibung von Anforderungen haben wir als eine möglichst einfache Sprache die in vielen Lehrbüchern semi-formal eingeführten Entscheidungstabellen (ET) ausgewählt. ET sind eine der einfachsten Beschreibungssprachen, die mit formaler Semantik und Analysen versehen werden können, sind jedoch weder *trivial* noch eine artifizielle Lehrsprache. Die schlichten Sprachmittel von ET reichen vollständig aus, um Anforderungen zu beschreiben, die in textueller Form sehr schnell nicht mehr überblickbar, wartbar oder analysierbar sind.

T: decision table		r_1	$\cdots$	r_n
c_1	description of condition c_1	$v_{1,1}$	$\cdots$	$v_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_m	description of condition c_m	$v_{m,1}$	$\cdots$	$v_{m,n}$
a_1	description of action a_1	$w_{1,1}$	$\cdots$	$w_{1,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
a_k	description of action a_k	$w_{k,1}$	$\cdots$	$w_{k,n}$

Abbildung 2: Konkrete Syntax von ET.

Formal ist eine ET T über disjunkte Mengen von *Bedingungen* C und *Aktionen* A eine $n \times m$-Matrix (siehe Abbildung 2) wobei $c_1, \dots, c_m \in C$, $a_1, \dots, a_k \in A$, $v_{1,1}, \dots, v_{m,n} \in \{-, \times, *\}$, und $w_{1,1}, \dots, w_{k,n} \in \{-, \times\}$. Eine wohlgeformte ET hat einen Namen. Spalten $(v_{1,i}, \dots, v_{m,i}, w_{1,i}, \dots, w_{k,i})$, $1 \leq i \leq n$, werden *Regeln* genannt und haben einen eindeutigen Namen (hier: $r_1, \dots, r_n$). Die Vektoren $(v_{1,i}, \dots, v_{m,i})$ und $(w_{1,i}, \dots, w_{k,i})$ heißen *Prämisse* und *Effekt* von Regel r_i. Die Semantik einer ET T ist durch die Funktion $\mathcal{F}$ gegeben, die jeder Regel r in T eine Formel der propositionalen Logik über C und A wie folgt zuordnet. Seien $(v_1, \dots, v_m)$ und $(w_1, \dots, w_k)$ Prämisse und Effekt von r. Dann ist

$$\mathcal{F}(r) := \underbrace{\bigwedge_{1 \leq i \leq m} F(v_i, c_i)}_{=:\mathcal{F}_{pre}(r)} \wedge \underbrace{\bigwedge_{1 \leq j \leq k} F(w_j, a_j)}_{=:\mathcal{F}_{eff}(r)} \quad (1)$$

mit $F := \{(\times, x) \mapsto x, (-, x) \mapsto \neg x, (*, x) \mapsto \mathit{true}\}$. Die Teilformeln $\mathcal{F}_{pre}(r)$ und $\mathcal{F}_{eff}(r)$ heißen Prämissen- bzw. Effektformel.

Eine ET kann wie folgt zur Formalisierung einer Anforderung der Art, daß Systemverhalten in akzeptabel (oder erwünscht) und unerwünscht klassifiziert wird, verwendet werden. Ein System *erfüllt* die durch ET T gegebene Anforderung genau dann, wenn jedes beobachtbare Systemverhalten durch eine Regel in T erlaubt wird. Formal kann eine Beobachtung eines Systems bzgl. der Bedingungen und Aktionen in C und A, also welche Bedingungen aus C erfüllt sind und welche Aktionen aus A ausgeführt werden, als Boole'sche Belegung $\sigma : C \cup A \to \{0, 1\}$ der logischen Variablen in $C \cup A$ notiert werden. Die Beobachtung σ erfüllt T genau dann, wenn es eine Regel r in T gibt, sodaß $\sigma \models \mathcal{F}(r)$.

Begriffe wie Vollständigkeit können für ET präzise definiert werden, z.B. nennen wir eine ET T (formal) vollständig, wenn die Disjunktion der Prämissenformeln von T eine Tautologie ist. Das Problem der Analyse auf (formale) Vollständigkeit kann auf das Erfüllbarkeitsproblem propositionaler Logik reduziert werden, für das in Form von sogenannten SAT-Solvern Entscheidungswerkzeuge zur Verfügung stehen.

In der Vorlesung ‚Softwaretechnik' diskutieren wir nun *nicht* eine Reduktionsprozedur oder die Konstruktion von SAT-Solvern, sondern nehmen eine Nutzersicht ein, nehmen also zur Kenntnis, daß Werkzeuge dieser Art existieren. Wir führen am Beispiel der ET die wichtige Diskussion, wie sich z.B. die formale Vollständigkeit einer ET zur vollständigen Erfassung einer Anforderung im Sinne von (Rupp u. a., 2014) verhält. Eine Analyse einer formalen Beschreibung bzgl. einer Eigenschaft wie Vollständigkeit kann positiv (Eigenschaft nicht gegeben) oder negativ ausfallen und dieses Ergebnis kann (wie Testresultate) falsch oder richtig sein. Ein Grund für ein falsch-positives Resultat kann z.B. ein simpler Schreibfehler beim Verfassen der formalen Beschreibung sein. Ein positives Resultat liefert jedoch in jedem Fall ein Indiz dafür, daß uns z.B. in der Anforderungserfassung ein Fehler unterlaufen ist und dieses Indiz ist im Zweifel zusammen mit den Kunden zu klären. Für ET liefern die Werkzeuge im positiv-Fall sogar alle Beobachtungen, die in T nicht berücksichtigt werden, und die sich üblicherweise gut in die Begriffs- und Erfahrungswelt der Kunden übersetzen und als solche klären lassen. Die Formalisierung einer Anforderung und die formale Analyse kann also sicherstellen, daß alle Probleme, die zu positiven Resultaten führen, erkannt und ggf. ausgeräumt werden. Die formale Analyse kann nicht sicherstellen, daß die Anforderungsanalyse als solche vollständig ist, da falsch-negative Resultate möglich sind. Ob der Aufwand der Formalisierung und Analyse von Anforderungen oder Designideen das verringerte Fehlerrisiko rechtfertigt, ist je nach Projektkontext zu entscheiden.

Die Übungsaufgaben zu ET umfassen verschiedene formale Analysen von gegebenen ET sowie eine Aufgabe, in der ein Transkript eines Kundeninterviews (mit absichtlichen Redundanzen und Mehrdeutigkeiten) im Umfang von ca. einer DIN-A4-Seite in eine ET mit ca. 10 Regeln über jeweils 5 Bedingungen und Aktionen überführt werden soll. Im Tutorium stellen wir die (provokante) Frage: Nehmen wir an, man müsste als Implementierer::in des Kundenwunsches zwischen Text und ET als Spezifikation wählen, was wäre die Wahl? Und warum? Meiner Kenntnis nach hat sich in den 4 Jahren der Veranstaltung noch niemand ausdrücklich den Text gewünscht. Hier versuchen wir, einen Aspekt sichtbar zu machen, der unserer Meinung nach im Kontext formaler Beschreibungen noch vor der Anwendung von formalen Analysen steht. Eine formale Beschreibung einer Anforderung hat den Effekt, daß (mit hoher Wahrscheinlichkeit) alle im verwendeten Formalismus ausgebildeten Softwaretechniker::innen zu jeder Zeit dieselbe Information sehen.[8] So sehen wir formale Beschreibungen zuvorderst als Werkzeug, um auf Seiten der Softwaretechniker::innen das Risiko von Missverständnissen zu verringern. Als Analogie verwenden wir z.B. von Jurist::innen konstruierte Individualverträge, deren Konsequenzen man als Nicht-Jurist üblicherweise nicht überblicken kann. Man hat als Auftraggeber des Individualvertrags jedoch den Anspruch, von

[8] Gestandene Praktiker::innen berichten, daß es vorkommt, daß dieselbe Person am selben Tag einen Text verschieden interpretiert.

der Fachperson eine „Übersetzung“ zu erhalten bzw. darauf, wichtige Szenarien gemeinsam durchzuspielen. Ebenso hat man als Softwarekunde einen Anspruch, von den Entwickler::innen deren z.B. in formaler Form notiertes Verständnis der Anforderungen erklärt zu bekommen. Dies kann insbesondere durch Szenarien gelingen (vgl. (Arenis u. a., 2016)). Für jedes von dem/der Kund::in formulierte Szenario können Entwickler::innen auf Basis der formalen Beschreibung Auskunft geben, ob ihrem Verständnis nach dieses Szenario erwünscht oder unerwünscht ist. Bei Verwendung formaler Beschreibungen ist diese Auskunft objektiv und beliebig reproduzierbar.

Die Klausur umfasst Aufgaben verschiedener Taxonomie-Stufen zu ET, vom Beweis z.B. der Vollständigkeit einer ET, über die Analyse einer gegebenen ET auf Konsistenz bis hin zur Erstellung einer kleinen ET zu einem Text. Die Ergebnisse (vgl. (Westphal, 2018)) sind sehr erfreulich. Den gesamten Aufgabenbereich lösten in 2015 bis 2017 mehr als 50 % der Teilnehmer::innen mit 14 bzw. 13 von 15 Punkten, liegen also im guten oder sehr guten Bereich, obwohl wir die Analyseaufgabe klar den „schweren“ Aufgaben der Klausur zuordnen. Die Grenze zum unteren Quartil liegt bei erfreulichen 12,5 bzw. 12 Punkten.

4.2.2 Live Sequence Charts

Als komplexen Formalismus (mit höherer Ausdrucksstärke und Informationsdichte, und höherem Aufwand zur Definition von (abstrakter) Syntax und Semantik) haben wir Live Sequence Charts (Damm u. Harel, 2001) (LSCs) ausgewählt, eine formale Variante der weit verbreiteten Sequenzdiagramme. Wir führen die Automatensemantik für LSCs nach (Klose u. Wittke, 2001) ein, inklusive Chart-Modus, Cold Conditions, und Pre-Charts. Ein System erfüllt eine (universelle) LSC genau dann, wenn alle Berechnungen des Systems von dem aus der LSC konstruierten Automaten akzeptiert werden. Weiterhin führen wir aus, wie LSCs (bzw. ihre Automaten) in der automatischen Testdurchführung verwendet werden können (Klose u. Lettrari, 2001).

Die Klausur umfasst Aufgaben zur Konstruktion der Automaten sowie erfragt Beispiele für akzeptierte bzw. nicht akzeptierte Systemberechnungen. Nicht zuletzt da die Einführung von LSCs den technisch anspruchsvollsten Teil unserer Vorlesung darstellt, sehen wir diese Aufgaben im „mittleren“ und „schweren“ Bereich, der am Ende die guten und sehr guten Gesamtnoten entscheidet. Mit den Resultaten sind wir auch hier zufrieden, das untere Quartil endet bei 8,5 bzw. 7 von 15 bzw. 16 Punkten, das obere Quartil beginnt bei 13 bzw. 12 Punkten.

Im ersten Jahr waren wir von den guten Ergebnissen bei aller Zuversicht positiv überrascht. Wir hatten uns vereinzelt vorgetragenen Sorgen, daß die LSC-Semantik zu schwer sein könnte, nicht völlig entziehen können und entsprechend Vorsorge getragen, die Aufgaben zu LSCs ggf. aus der Wertung zu nehmen.

4.3 Themenbereich Entwurf

Im Themenbereich Softwarearchitektur und Entwurf liegt der Schwerpunkt auf der Modellierung der Struktur und des Verhaltens von Softwaresystemen als Baupläne im Sinne von (Lamport, 2015). Zur Strukturmodellierung führen wir eine übersichtliche Teilsprache der Klassendiagramme von UML ein, die im wesentlichen Klassen mit Attributen von Basistypen und gerichteten Assoziationen der Multiplizitäten 0,1 und $0,*$ mit Ownership umfasst.Neben der bekannten Sicht, Klassendiagramme als Visualisierung von (Klassen in) Programmen zu betrachten, betonen wir die Sicht eines abstrakten (von der Programmierung zunächst unabhängigen) Strukturmodells. Die abstrakte Syntax eines Klassendiagramms ist eine Signatur mit Klassen, Basis- und abgeleiteten Typen, und je Klasse der Menge der Attribute dieser Klasse. Die Semantik eines Klassendiagramms ist die Menge der über dieser Signatur bildbaren Systemzustände (OMG, 2006), d.h. partiellen Funktionen, die (einige) Objektidentitäten auf jeweils eine Belegung der Attribute mit Werten abbilden. Systemzustände können graphisch (ohne Informationsverlust) durch vollständige Objektdiagramme dargestellt werden. Hier stellen wir die Anwendung von Objektdiagrammen in der Dokumentation und Spezifikation heraus. Wenn der Aufwand, bestimmte Annahmen einer Datenstruktur z.B. mit OCL zu formalisieren, in einem Projekt als zu hoch bewertet wird, kann eine geeignete Auswahl von Objektdiagrammen hervorragend zur Spezifikation dieser Annahmen „durch Beispiel“ verwendet werden. Wenn etwa eine Listenstruktur nur unter der Annahme verwendet werden darf, daß keine Zyklen konstruiert werden.

Eigenschaften von Strukturen können mit Hilfe der Object Constraint Language (OCL) formalisiert werden. Hier führen wir Proto-OCL ein, eine Teilsprache der OCL, deren konkrete Syntax sich an der den Studierenden vertrauten Form der Logik erster Stufe orientiert. Die Semantik von Proto-OCL ist eine Interpretationsfunktion, die eine gegebene Proto-OCL-Formel und einen gegebenen Systemzustand (oder ein vollständiges Objektdiagramm) auf einen der drei (!) Werte true, false und $\bot$ abbildet. Wir stellen die Besonderheiten heraus, daß OCL Assoziationen verschiedener Multiplizität verschieden behandelt sowie die Bedingungen, unter denen man eine Auswertung zum dritten Wahrheitswert beobachten kann.

Für die konstruktive Modellierung von Verhalten führen wir eine einfache Teilsprache von State-Machines ein. Unsere *Communicating Finite Automata* (CFA) sind im Wesentlichen nicht-hierarchische State-Machines mit Rendezvous-Synchronisation, d.h. zwei Kanten in verschiedenen Automaten können eine Transition begründen, wenn je eine der Kanten bzgl. eines Events sende- bzw. empfangsbereit ist. Dieses Synchronisationsparadigma ist mit durch State-Machines implementierten

Behavioural Features in UML vergleichbar jedoch nicht (ohne weitere Annahmen) identisch. Zu dieser Abweichung von der UML-Semantik haben wir uns vorrangig aus zwei Gründen entschieden. Einerseits erfordert die formale Einführung des abstrakten Event-Speichers (in der die UML-Semantik bekanntlich parametrisiert ist) sowie einer konkreten Ausprägung, etwa einer empfängerseitigen FIFO-Queue, wie sie IBM Rhapsody verwendet, in unserer UML-Veranstaltung ca. 2 Vorlesungen. Der Erkenntnisgewinn im Vergleich zum Aufwand erscheint uns für eine Softwaretechnik-Veranstaltung zu gering. Andererseits steht mit Uppaal (Larsen u. a., 1997) ein Werkzeug zur Erstellung, Simulation und Verifikation von (insbesondere)[9] CFAs mit einer reichen Programmiersprache für Guards und Aktionen zur Verfügung, das nicht zuletzt aufgrund seines sehr gut auf das Wesentliche reduzierten User-Interfaces[10] hervorragend für unsere Ziele geeignet ist.[11]

Die Übungsblätter zu CFA umfassen Aufgaben zur Erstellung eines Modells eines verteilten Algorithmus für Mutual-Exclusion (das den Teilnehmer::innen als Problem aus der Betriebssysteme-Vorlesung bekannt sein müsste), zur Simulation verschiedener Abläufe und zur Verwendung von Uppaal's Temporallogik (Query Language) zur Spezifikation und Verifikation der Mutual-Exclusion-Eigenschaft. Mit den Übungen verfolgen wir insbesondere das Ziel, das mächtige Modellierungskonzept des Nichtdeterminismus und die durch nebenläufiges Verhalten induzierten Schwierigkeiten sichtbar zu machen.

In der Klausur ist üblicherweise der Transitionsgraph eines gegebenen Systems von mehreren CFAs zu erstellen. Mit den Ergebnissen sind wir durchweg zufrieden, der Median liegt zwischen 10 und 12,25 von 13 Punkten, das obere Quartil beginnt zwischen 12 und 13 Punkten, liegt also im sehr guten Bereich.

4.4 Themenbereich Qualitätssicherung

Im Themenbereich Qualitätssicherung betrachten wir speziell die Prüfung von Programmcode. Hier diskutieren wir zunächst die Disziplin des Softwaretestens. Dem formalen Charakter der gesamten Veranstaltung folgend, führen wir Testfälle als Paare $(In, Soll)$ aus einer (Menge von) Eingabe(n) und Soll-Wert(en) ein. Eine Ausführung eines Testfalls ist eine Beobachtung des Systems, die Eingaben entsprechend In erhält, was ein einzelner Wert oder eine Sequenz von z.B. Events sein kann. Ein Test ist negativ (*Test passed*) genau dann, wenn die betrachtete Beobachtung ein Element von *Soll* ist. Hier betonen wir ((Ludewig u. Lichter, 2013) folgend) besonders den Aspekt, daß die Begriffe des positiven und negativen Tests relativ zu Soll-Werten (die sich ggf. aus (formalen) Anforderungen ableiten lassen) definiert sind. Die Resultate der Klausuraufgaben zum Testen sind durchweg ordentlich, jedoch sind wir jedes Jahr wieder überrascht, daß eine signifikante Anzahl der Studierenden keine Soll-Werte angibt, obwohl explizit nach einer *erfolglosen* Test-Suite gefragt wird.

Als gegenüber den etablierten Lehrbüchern neuen Inhalt präsentieren wir Programmverifikation nach (Hoare, 1969) in der Form des Lehrbuchs (Apt u. a., 2009). Für die Übungen verwenden wir das Web-Interface des Verifying C Compilers (Cohen u. a., 2009) (VCC) und diskutieren als weiterführende Fragen den Unterschied zwischen der Verifikation eines Algorithmus und der Verifikation eines C-Programms unter Annahmen über die Ausführungsplattform. Die Klausuraufgabe besteht üblicherweise darin, eine gegebene Beweisskizze mit den verwendeten Axiomen und Beweisregeln zu annotieren. Hier erreicht das obere Quartil zwischen 5,5 und 7 von 7 Punkten, das untere Quartil endet bei 3 bis 5 Punkten.

5 Lehrevaluation

Da die Veranstaltung neu konstruiert wurde, beobachten wir seit der ersten Durchführung verschiedene Metriken (vgl. Abschnitt 4.1) auf Indizien für Fehlentwicklungen. Daten beziehen wir aus dem Übungsbetrieb, den Klausuren und der zentralen Evaluation.

In der Evaluation beachten wir besonders die Aspekte „Workload“ und „Niveau“ (Uttl u. a., 2017) und die Freitexte. Den Workload sehen über alle 4 Jahre gesehen rund 62 % der Studierenden als ‚mittel‘ und rund 30 % als ‚hoch‘, das Niveau sehen rund 62 % als ‚angemessen‘ und rund 29 % als ‚hoch‘. Dies entspricht vollständig unserem Ziel für eine universitäre Lehrveranstaltung.

Den Lernerfolg bzgl. des Lernziels der Vermittlung von Methoden und Techniken messen wir anhand der Klausurergebnisse, mit denen wir über alle 4 Jahre sehr zufrieden sind. Das viel interessantere Lernziel der Vorbereitung auf das „echte Leben“ läßt sich leider notorisch nicht gut messen. Als ein positives Indiz sehen wir unsere und von Kolleg::innen berichtete Erfahrung aus Vorgesprächen zu individuellen Projekten wie etwa B. Sc.-Arbeiten. Hier nehmen wir die Inhalte der Vorlesung als (angestrebten) soliden Bezugspunkt wahr, um über spezialisierte Aufgabenstellungen zu sprechen. Wir hoffen, daß sich die Veranstaltung ebensogut als Bezugspunkt für den Einstieg in die berufliche Karriere der Studierenden eignet.

6 Zusammenfassung

Formale Methoden werden zunehmend in vielen Bereichen der Softwaretechnik angewandt. Unserer Überzeugung nach ist es hilfreich, wenn die Bedeu-

[9] Daß Uppaal für CFAs mit Uhrenvariablen, also Timed Automata, entwickelt wurde, verschweigen wir schlankerhand.

[10] Aus denselben Überlegungen führen wir kein Werkzeug zur Erstellung von Klassendiagrammen ein. Die für die Einarbeitung in die Bedienung der meisten dieser Werkzeuge notwendige Zeit steht unserer Meinung nach in keinem akzeptablen Verhältnis zu den Absichten z.B. unserer Übungen (vgl. (Glinz, 1996)).

[11] Einige Teilnehmer::innen unserer UML-Veranstaltung, in der wir IBM Rhapsody verwenden, vermissen die mächtige und leicht zu bedienende Simulationsfunktion von Uppaal.

tung formaler Beschreibungen vollständig durchdrungen wird und die aus formalen Analysen zu ziehenden Schlüsse verstanden werden.

Unsere Erfahrung zeigt, daß es möglich ist, eine Veranstaltung zur Einführung in die Softwaretechnik im Grundstudium um genau diese Aspekte zu ergänzen ohne hierbei die Inhalte der etablierten Lehrbücher zu vernachlässigen.

Danksagungen. Wir danken Sergio Feo Arenis und Christian Schilling für ihre Unterstützung als Assistenten der ersten beiden bzw. des letzten Jahres. Ohne die Gespräche mit Sergio und Christian und ihre Ausarbeitung und Weiterentwicklung der Übungsaufgaben wäre die Veranstaltung in ihrer heutigen Form nur schwer vorstellbar. Weiterhin danken wir unseren stud. Hilfskräften für ihre engagierte Mitarbeit an Aufgaben und Tutorials.

Literatur

[Anderson u. a. 2001] ANDERSON, L. W. (Hrsg.) u. a.: *A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, 2001

[Apt u. a. 2009] APT, K. R. ; BOER, F. S. ; OLDEROG, E.-R.: *Verification of Sequential and Concurrent Programs*. Springer, 2009

[Arenis u. a. 2016] ARENIS, S. F. ; WESTPHAL, B. u. a.: Ready for testing: ensuring conformance to industrial standards through formal verification. In: *FAoC* 28 (2016), Nr. 3, S. 499–527

[Balzert 2009] BALZERT, H.: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3rd. Spektrum, 2009

[Biggs u. Tang 2011] BIGGS, J. ; TANG, C.: *Teaching for Quality Learning at University*. 4th. Open University Press, 2011

[Bjørner 2006a] BJØRNER, .: *SWE, Vol. 1: Abstraction and Modelling*. Springer, 2006

[Bjørner 2006b] BJØRNER, D.: *SWE, Vol. 2: Specification of Systems and Languages*. Springer, 2006

[Bjørner 2006c] BJØRNER, D.: *SWE, Vol. 3: Domains, Req. and Software Design*. Springer, 2006

[Bloom 1956] BLOOM, B. S. (Hrsg.): *Taxonomy of Educat. Objectives: Cognitive Domain*. Longman, 1956

[Cohen u. a. 2009] COHEN, E. u. a.: VCC: A Practical System for Verifying Concurrent C. In: BERGHOFER, S. (Hrsg.) u. a.: *TPHOLs* Bd. 5674, Springer, 2009 (LNCS), S. 23–42

[Damm u. Harel 2001] DAMM, W. ; HAREL, D.: LSCs: Breathing Life into Message Sequence Charts. In: *FMSD* 19 (2001), Juli, Nr. 1, S. 45–80

[Glinz 1996] GLINZ, M.: The Teacher: "Concepts!" The Student: "Tools!". In: *Softwaret.-Trends* 16 (1996), Nr. 1

[Hoare 1969] HOARE, C. A. R.: An axiomatic basis for computer programming. In: *CACM* 12 (1969), Nr. 10, S. 576–580

[IEEE 1990] IEEE: *Standard Glossary of Software Engineering Terminology*, 1990. – Std 610.12-1990

[IEEE 1998] IEEE: *Recommended Practice for Software Req. Specifications*, 1998. – Std 830-1998

[ISO/IEC/IEEE 2010] ISO/IEC/IEEE: *Systems and software eng. – Vocabulary*, 2010. – 24765:2010(E)

[Klose u. Lettrari 2001] KLOSE, J. ; LETTRARI, M.: Scenario-based Monitoring and Testing of Real-time UML models. In: GOGOLLA, M. (Hrsg.) u. a.: *UML* Bd. 2185, Springer, 2001 (LNCS)

[Klose u. Wittke 2001] KLOSE, J. ; WITTKE, H.: An Automata Based Representation of Live Sequence Charts. In: MARGARIA, T. (Hrsg.) u. a.: *TACAS*, Springer, 2001 (LNCS 2031)

[Krusche u. a. 2017] KRUSCHE, S. ; FRANKENBERG, N. von ; AFIFI, S.: Experiences of a Software Engineering Course based on Interactive Learning. In: *SEUH*, 2017, S. 32–40

[Lamport 2015] LAMPORT, L.: Who builds a house without drawing blueprints? In: *CACM* 58 (2015), Nr. 4, S. 38–41

[Langenfeld u. a. 2016] LANGENFELD, V. ; POST, A. u. a.: Requirements Defects over a Project Lifetime. In: DANEVA, M. (Hrsg.) u. a.: *REFSQ* Bd. 9619, Springer, 2016 (LNCS), S. 145–160

[Larsen u. a. 1997] LARSEN, K. G. ; PETTERSSON, P. ; YI, W.: UPPAAL in a Nutshell. In: *STTT* 1 (1997), Dezember, Nr. 1, S. 134–152

[Lehmann u. a. 2015] LEHMANN, T. u. a.: Lecture Engineering. In: SCHMOLITZKY, A. (Hrsg.) u. a.: *SEUH* Bd. 1332, CEUR-WS, 2015, S. 103–109

[Ludewig u. Lichter 2013] LUDEWIG, J. ; LICHTER, H.: *Software Engineering*. 3rd. dpunkt, 2013

[OMG 2006] OMG: *Object Constraint Language, Version 2.0*. formal/06-05-01, 2006

[Rupp u. a. 2014] RUPP, C. u. a.: *Requirements-Engineering und -Management*. 6th. Hanser, 2014

[Sedelmaier u. Landes 2017] SEDELMAIER, Y. ; LANDES, D.: Experiences in Teaching and Learning Req. Engineering on a Sound Didactical Basis. In: DAVOLI, R. (Hrsg.) u. a.: *ITiCSE*, ACM, 2017, S. 116–121

[Siegeris 2017] SIEGERIS, J.: LearnTeamPlenum. In: *SEUH*, 2017, S. 1–7

[Sommerville 2010] SOMMERVILLE, I.: *Software Engineering*. 9th. Pearson, 2010

[Uttl u. a. 2017] UTTL, B. u. a.: Meta-analysis of faculty's teaching effectiveness: Student evaluation of teaching ratings and student learning are not related. In: *Stud. Educat. Eval.* 54 (2017), S. 22 – 42

[Westphal 2018] WESTPHAL, B.: An Undergraduate Requirements Engineering Curriculum with Formal Methods. In: *REET*, IEEE, 2018, S. 1–11