Divekit - Digitalisierung und Individualisierung als Schlüssel für eine moderne Softwaretechnik-Ausbildung

Stefan Bente[1], Jann Intveen[2], Fabian Krampe[3]

Abstract: Um Kompetenzen in modernen Softwaretechnik-Ansätzen wie Domain Driven Design oder Clean Code zu erwerben, müssen diese von Lernenden im Rahmen von Praktika auch selbst angewendet werden. Erst dann können sie in ihrer Komplexität und ihren Konsequenzen vollständig verstanden werden. Digitale E-Assessment-Tools eröffnen die Möglichkeit, dies mit vertretbarem Betreuungsaufwand zu tun, indem große Teile der studentischen Lösung automatisiert getestet werden. Somit erhalten die Studierenden direktes formatives Feedback, und die Betreuer:innen können sich auf Beratung in komplexen Fragestellungen konzentrieren. Dieses Paper formuliert 12 Prinzipien für E-Assessment-Tools im Bereich Softwaretechnik und Coding, darunter Individualisierung und Praxisnähe der gestellten Aufgaben. Anschließend wird das Open-Source-Framework Divekit vorgestellt, das diese Prinzipien erfüllt. Möglichkeiten und Grenzen von Divekit werden anhand einer Fallstudie kritisch bewertet.

Keywords: E-Assessment; Digitalisierung; Softwaretechnik; Coding; Individualisierung; DDD

1 Einleitung

Ein modern ausgerichtetes Softwaretechnik-Praktikum hat zwei primäre Aufgaben. Auf der einen Seite soll es den *Lernenden* ermöglichen, vorab vermittelte Inhalte selbst praktisch anzuwenden - und zwar in einer Arbeitsumgebung, die möglichst nahe an der berufspraktischen Realität ist. Dies gilt in besonderem Maß für die Vermittlung von komplexen Konzepten wie Domain Driven Design, Test Driven Development oder Clean Code. Hier ist zum echten Kompetenzerwerb tatsächliche Programmierung nötig - Lückentexte und Multiple-Choice-Fragen genügen nicht.

Auf der anderen Seite benötigen *Lehrende* eine effektive Rückmeldung zum Lernerfolg des gesamten Kurses (zur Überprüfung der Lernziele) und jedes einzelnen Mitglieds der Lerngruppe (zur Erteilung von Praktikumstestaten, oder der Benotung einer Klausur). Dies muss einher gehen mit einem möglichst hohen Grad an Automatisierung, um die Ressource "Betreuungspersonal" an den Stellen einsetzen zu können, wo ein echter Erkenntnis-Mehrwert für Studierende entsteht (beispielsweise in komplexen Fachdiskussionen zwischen Betreuer:in und Studierenden).

Diese Ziele lassen sich nur durch eine verstärkte Digitalisierung von studentischen Softwaretechnik-Praktika erreichen. Dieses Paper stellt zunächst die Anforderungen daran

[1] TH Köln, Cologne Institute for Digital Ecosystems (CIDE), Steinmüllerallee 1, 51643 Gummersbach, stefan.bente@th-koeln.de
[2] s.o., JannIntveen@gmail.com
[3] s.o., fabian.krampe@th-koeln.de

doi:10.18420/SEUH2022_04

auf, und begründet diese ausgehend vom Wissensstand der Software-Community sowie der hochschuldidaktischen Forschung. Dann wird das Werkzeug **Divekit** (Toolkit for In**div**idual **E**xercises) beschrieben, das an der TH Köln entwickelt wurde und seit mehreren Semestern erfolgreich in verschiedenen Lehrveranstaltungen eingesetzt wird. Diese Erfahrungen werden abschließend in einer Fallstudie bewertet.

2 Zwölf Prinzipien für eine praxisnahe digitale Softwaretechnik-Lehre

Eine praxisnahe digitale Softwaretechnik-Lehre muss einerseits didaktische Anforderungen erfüllen, andererseits aber auch die zurzeit gebräuchlichen softwaretechnischen Paradigmen unterstützen. Im Rahmen einer Masterarbeit [In21] wurde hierzu neben einer umfassenden Recherche auch eine Anzahl von Experteninterviews durchgeführt, um die Erfahrungen von anderen Lehrenden und Expert:innen auf diesem Gebiet einzubeziehen. Daraus ergaben sich 12 Prinzipien, wie eine erfolgreiche digitale Softwaretechnik-Lehre gestaltet sein sollte. Diese stellen den Orientierungsrahmen für Entwicklung und Einsatz von Divekit dar.

(P1) Realitätsnahe Programmieraufgaben gemäß *Active Learning* Moderne Programmierkonzepte werden am besten anhand von realistischen Problemstellungen und mit echten Tools (IDE, Git, Build-Pipelines) erlernt. Mittels *Active Learning* [FB09] können sich Lernende in Einzel- oder Gruppenarbeit damit beschäftigen. Lehrende tragen in bestimmten Intervallen Erkenntnisse zusammen und geben Inhaltsimpulse.

(P2) Mindestens Stufe 3 (*Applying*) der Revised Bloom's Taxonomy Klassische E-Learning-Werkzeuge bleiben oft auf den Stufen 1 (*Remembering*) oder 2 (*Understanding*) der Revised Bloom's Taxonomy [Ar16] stehen. Ein praxisrelevantes Lernen setzt aber mindestens voraus, die erlernten Konzepte auch selbstständig anwenden zu können (*Applying*, Stufe 3).

(P3) Intrinsische über extrinsische Motivation Intrinsische Lernprozesse gelten als besonders effektiv [RD00, 56f] und sollten daher gefördert werden, ergänzt durch extrinsische Faktoren wie etwa eine Erfolgskontrolle bei Praktikumsaufgaben.

(P4) Stetiges formatives Feedback für Lernende durch E-Assessment Ein E-Assessment-System muss effektives, kontinuierliches formatives Feedback [RMP04, 17ff] während des gesamten Lehr- und Lernprozesses bieten, auch wenn ein Teil des Feedbacks automatisch erfolgt.

(P5) Stetiger Zugriff auf individuelles formatives und summatives Assessment für Lehrende Lehrende benötigen Einsicht in das formative Assessment der Studierenden, um Stoff und Lehrmethoden dynamisch an den Lernfortschritt anpassen zu können. Iteratives summatives Assessment (z.B. in Form von Praktikums-Meilensteinen) ermöglicht es zusätzlich, Themen und Konzepte erneut aufzugreifen, die noch nicht gänzlich verstanden wurden [HRL20, S. 281].

(P6) Förderung von Zusammenarbeit im Team, aber *individuelle* Abgabe von Lösungen In informellen Studiengruppen spezialisieren sich oft einzelne Studierende auf eine bestimmte Veranstaltung und bearbeiten diese stellvertretend für die ganze Gruppe. In diesem Fall beschäftigen sich nicht alle Gruppenmitglieder in derselben wünschenswerten Tiefe mit den Studieninhalten. Um dem vorzubeugen, sollten Aufgaben in einem Maß individualisierbar sein, das zwar Diskussionen und Zusammenarbeit zwischen Lernenden ermöglicht, aber das einfache Kopieren von Lösungen anderer erschwert.

(P7) Keine Software-Architektur ohne Coding Wer Architektur nicht in Code umsetzen kann, wird im berufspraktischen Kontext selten ernst genommen[4]. Eine praxisnahe digitale Softwaretechnik-Lehre muss daher komplexe Architekturstile und Patterns auch in Code abbilden und die Studierenden praktisch umsetzen lassen.

(P8) Pragmatische Vermittlung von Modellierungs-Fähigkeiten Eine hauptsächliche Fokussierung auf UML-Kenntnisse ist in der Softwaretechnik-Lehre nicht mehr zeitgemäß (siehe P7). Wie in [AB19] ausgeführt, gibt es aber einige UML-Diagramme, die auch bei einer agilen, wenig dokumentenzentrierten Vorgehensweise sinnvoll nutzbar sind. Dazu gehören insbesondere Klassen- und Use-Case-Diagramme, und für besondere Aspekte auch Komponenten-, Deployment-, Status-, Aktivitäts- und Sequenzdiagramme, die bei einer Digitalisierung von Softwaretechnik-Lehre berücksichtigt werden sollten.

(P9) Realistische Größe von Aufgabenstellungen Wenn Studierende immer nur kleine "Aufgabenschnipsel" bearbeiten, dann wird sich kein Gefühl für den Umgang mit Komplexität einstellen. Daher muss es digitale Lehre ermöglichen, komplexe Aufgabenstellungen zu formulieren, die sich über mehrere Iterationen (wie etwa verschiedene Praktikums-Meilensteine) weiterentwickeln lassen.

(P10) Stetige Entwicklung gegen Unit Tests Der Ansatz des *Test Driven Development (TDD)* sieht vor, dass Tests geschrieben werden, bevor die dazugehörige Funktionalität implementiert wird [Be02]. Dies hilft Studierenden, Komplexität durch Aufteilung in kleine Code-Fragmente zu beherrschen, die mithilfe vorab geschriebener Tests überprüft werden.

(P11) Fokus auf Prinzipien des Domain Driven Designs (DDD) Domain Driven Design (DDD) [Ev04] ist in der IT-Community als Basis für moderne, lose gekoppelte Architekturen weit verbreitet[5]. Eine praxisnahe digital unterstützte Programmierausbildung muss es daher ermöglichen, den Studierenden Aufgaben zum Tactical Design (Entities, Repositories, Value Objects, Aggregates, Services [Ev04, S. 89–158]) zu stellen[6].

[4] Wie Martin Fowler schreibt: "Like many in the software world, I've long been wary of the term 'architecture' as it often suggests a separation from programming and an unhealthy dose of pomposity." [Fo19]

[5] Als - zugegebenermaßen recht anekdotisches - Indiz mag gelten, dass auf einer größten IT-Community-Konferenzen in Deutschland, der OOP in München, im Jahr 2020 [DA21] in einem Drittel der Vorträge mit Architekturbezug Teilaspekte von DDD thematisiert wurden (16 von 48 Vorträgen, gezählt nach Tags).

[6] Auch eine Hinführung zum Strategic Design, z.B. dem Begriff des "Bounded Context" [Ev04, S. 335], ist wünschenswert, kann aber eher im Kontext eines Praxis- oder Lehrforschungsprojekts geleistet werden.

(P12) Fokus auf Clean-Code-Regeln und SOLID-Prinzipien *Clean Code* betont die Relevanz und die Bedeutsamkeit von sauber geschriebenem, selbsterklärendem Programmcode. Eng verbunden damit sind die SOLID-Prinzipien [Ma00]. Beide Ansätze sind in der IT-Community durchgehend akzeptiert und sollten in einer digital unterstützten Softwaretechnik-Lehre thematisierbar sein.

2.1 Abdeckung der genannten Prinzipien durch existierende E-Assessment- und E-Learning-Systeme

Keunig et al. [KJH18] haben 2018 systematisch über 100 digitale Programmier-Lernsysteme bezüglich ihres Feedbacks für Lernende untersucht. Eine verwandte Untersuchtung haben Wasik et al. [Wa18] durchgeführt, die sogar noch mehr Softwaresysteme zur Onlinebewertung vergleichen. Dabei finden sich einige Ansätze, die eine ähnliche Richtung wie die genannten 12 Prinzipien verfolgen. Hier wäre etwa der *Praktomat* [BHS17] zu nennen, ein E-Assessment-Werkzeug, mittels dessen den Studierenden Programmieraufgaben gestellt und deren Lösungen überprüft werden können. Zusätzlich werden Funktionen zur automatischen Erstellung von Aufgabenvarianten bereitgestellt. Einen stark auf automatisiertes Feedback ausgerichteten Ansatz verfolgt *Code FREAK* [WK21] der FH Kiel. Auch im Bereich der Mathematik existieren viele E-Assessment-Systeme, die Aufgaben individualisieren, beispielsweise *ActiveMath* [Go10, S. 77–83] oder *MATEX* [He18].

Tools, die sämtliche oben geforderten Prinzipien umsetzen, finden sich allerdings nach unseren Recherchen nicht. Die Realität der digitalen Lehre an deutschen Hochschulen scheint eher aus in Moodle oder ILIAS umgesetzten Single-/Multiple-Choice-Fragen und Lückentexten zu bestehen, die aus einem Fragenpool zufällig ausgewählt werden. Ein Praxisbezug (P1-P3, P7-P12) kann damit nicht zufriedenstellend abgebildet werden. Daher fiel hier die Entscheidung, aufbauend auf Gitlab die eigene Lösung **Divekit** als Open-Source-System zu konzipieren und entwickeln.

3 Konzept und didaktischer Rahmen von Divekit

Eine Übungsaufgabe mit Divekit, etwa im Softwaretechnik-Praktikum, durchläuft die fünf Phasen *(1) Konzipieren*, *(2) Individualisieren und Verteilen*, *(3) Bearbeiten*, *(4) Feedback* und *(5) Korrigieren und Auswerten*.

3.1 Phase 1: Konzipieren

Der oder die Lehrende erstellt eine Aufgabe mitsamt einer Musterlösung für die Betreuer:innen, die diese in Beratung und Korrektur (falls manuelle Korrektur nötig ist) einsetzen können. Die Aufgabenstellung besteht aus Code, Diagrammen im UMLet-Format[7], Konfigurationsdateien sowie Markdown-Dateien zur Beschreibung. All das wird in einem Gitlab-Repository (dem “Origin-Repo”) zusammengefasst.

[7] UMLet ist ein freier UML-Editor (`https://www.umlet.com/`), der einen großen Teil der UML-Spezifikation in einem einfachen Scripting-Ansatz umsetzt. Dadurch ist er sehr intuitiv zu bedienen, und kann mittels Batch-Processing leicht in ein Framework wie Divekit eingegliedert werden.

In allen genannten Artefakten (Text, Code, Diagramme) können Platzhalter-Begriffe verwendet werden, erkennbar an der Schreibweise `...`. Auf diese Weise sind auch komplexe natürlichsprachliche Aufgabenstellungen umsetzbar, die dann von der fachlichen Modellierung bis hin zu Umsetzung in automatisiert getesteten Code von den Studierenden durchgängig bearbeitet werden können.

List. 1: Ausschnitt aus einem Aufgabentext mit Platzhaltern

```
$Robots$ $robot_purpose$. They can be moved across multiple
  $robot_grids$ with $walls$, which cannot be passed. A $robot_grid$ is
  rectangular and consists of $rasters$. One $raster$ can contain only
  one $robot$. Therefore, $robots$ are like $walls$ for each other.
```

Für diese Platzhalter werden in einer Konfigurationsdatei im JSON-Format Variationen spezifiziert. Für einen Platzhalter wie `$robot$` und seine Attribute können dabei Variationen wie "Maintenance Droid" oder "Mining Machine" angeben werden. Die Kombination der Variationen kann flexibel festgelegt werden. Spezialformen wie etwa getter/setter (`getMaintenanceDroid(...)`) werden von DiveKit automatisch bereitgestellt. Ebenso erkennt Divekit Groß- und Kleinschreibung am Satzanfang (`$Robot$` - `$robot$`) und reguläre Pluralformen (`$robot$` - `$robots$`)[8].

3.2 Phase 2: Individualisieren und Verteilen

Die generisch verfasste Aufgabenstellung wird so individualisiert, so dass jede:r Studierende:r ein individuelles Gitlab-Repository erhält. Die obige Aufgabenstellung aus dem Origin-Repo (siehe List. 1) kann sich für zwei unterschiedliche Studierende dann so darstellen:

List. 2: Zwei individualisierte Instanzen der Aufgabenstellung aus List. 1

Maintenance droids fix minor problems, like loose plugs, etc. They can be moved across multiple spaceship decks with obstacles, which cannot be passed. A spaceship deck is rectangular and consists of cells. One cell can contain only one maintenance droid. Therefore, maintenance droids are like obstacles for each other.

Mining machines collect minerals like cobalt, lithium, etc. They can be moved across multiple mining fields with barriers, which cannot be passed. A mining field is rectangular and consists of squares. One square can contain only one mining machine. Therefore, mining machines are like barriers for each other.

Diese Individualisierung geschieht in einem automatisierten Batchlauf, in dem die Platzhalter zu zufälligen Varianten expandiert werden. Durch die Kombination unabhängiger Platzhalter entstehen dann schnell so viele Varianten, dass es selbst in einer großen Kohorte keine zwei gleichen Aufgabenstellungen gibt.

Diese Expansion funktioniert nicht nur in Aufgabentexten, wie schon gesehen. Auch der Code (Tests, gegen die Studierende implementieren müssen, aber auch als Hilfestellung

[8] Unregelmäßige Pluralformen (women, children, mice etc.) können explizit in der Konfigurationsdatei überschrieben werden.

vorgegebene Stub-Klassen) kann entsprechend individualisiert werden. Das gleiche gilt für Markdowntabellen, in denen Studierende schon Teillösungen angeben müssen (beispielsweise für ein fachliches Glossar aus Basis des Domain Models), und auch für UML-Diagramme. Dadurch kann die Aufgabenstellung beispielsweise durch ein individualisiertes Klassen-, Zustands-, Sequenz- oder Aktivitätsdiagramm besser erklärt werden (siehe Abb. 1).

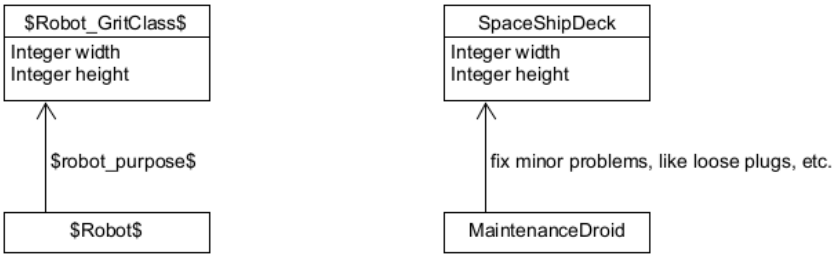

Abb. 1: UML-Diagramm in “Origin-Repo” (links), individualisierte Variante (rechts)

Divekit bringt eine vollständige Test-Bibliothek mit, die speziell auf mittels *Spring Data JPA*[VM] umgesetzte DDD-Konzepte zugeschnitten ist. Andere Teile sind auf REST-APIs ausgelegt. Diese Bibliothek kann leicht für weitere Anwendungsbereiche erweitert werden.

Für alle Studierenden speichert Divekit auf Wunsch die jeweilige Individualisierungs-Konfiguration und wendet sie bei nachfolgenden Aufgabenstellungen erneut an. Dadurch sind auf einander aufbauende Aufgabenstellungen möglich, die die originale Aufgabe erweitern. Jede:r Studierende findet dann wieder die vertraute Konfiguration vor.

Tests können in Divekit vor den Studierenden versteckt werden, da intern immer zwei Repositories pro Studierendem erzeugt werden, eins davon für den Studierenden unsichtbar. Dort können “Hidden Tests” abgelegt werden, deren Code die Lösung direkt verraten würde.

3.3 Phase 3: Bearbeiten

Die Studierenden bearbeiten die gestellte Aufgabe. Jede:r hat ein individuelles Repo mit eigener Aufgabenstellung. Die Studierenden reichen ihre Lösungen durch ein `git commit` / `git push` ein. In den Lehrveranstaltungen, in denen Divekit bislang eingesetzt wird, werden die Studierenden dennoch ausdrücklich zur Teamarbeit ermuntert. Flankierende ganztägige Übungen in Kleingruppen fördern dies. Das Feedback und die Rückfragen der Studierenden legen nahe, dass die gemeinsame Basisstruktur der Aufgaben erkennbar bleibt, was eine Diskussion ermöglicht. Ein simples Kopieren fremder Lösungen führt aber im Allgemeinen nicht zum Ziel.

3.4 Phase 4: Feedback

Über eine personalisierte Webseite erhalten die Studierenden Feedback. Dort sind die Rückmeldungen aller Tests zu sehen (rot/grün und Fehlermeldung, falls rot). Im Fall von

komplett automatisierten Tests erfolgt das Feedback innerhalb von ca. zwei Minuten nach dem `git push`.

Zusätzlich hat es sich in der Praxis bewährt, ein Diskussionsforum für Fragen zu eröffnen. *Discord* scheint aufgrund seiner große Verbreitung in der Gaming-Community eine gute Wahl zu sein, weil es dadurch Studierenden, Tutoren und Betreuer:innen gleichermaßen vertraut ist und niederschwellig genutzt wird. Es ist sinnvoll, die Studierenden zu Fragen zu ermuntern, wenn das Divekit-Feedback für sie unverständlich ist. Andernfalls droht ein "Brute-Force"-Vorgehen seitens einiger Studierenden, bei dem ohne tieferes Verständnis der Aufgabenstellung zahlreiche Varianten nacheinander durchprobiert werden.

3.5 Phase 5: Korrigieren und Auswerten

Nicht alle Teile einer Lösung lassen sich sinnvoll automatisiert testen. Divekit erlaubt auch manuelles Feedback, das die Betreuer:innen in einer dedizierten Markdown-Datei hinterlegen. Hier bietet Divekit Tools, um die Warteschlange der eingereichten Lösungen zu sehen und die jeweils am längsten wartende Lösung korrigieren zu können. Dieses manuelle Feedback wird dann auf der Testseite neben dem automatisierten Feedback angezeigt.

Als Strategie hat es sich bewährt, basale Aspekte der Lösung automatisiert zu prüfen und das menschliche Feedback erst dann zu geben, wenn die automatisierten Tests "grün" sind. Auf diese Weise wird die knappe Resource "Kontaktzeit" für semantisch höherwertige, komplexe Fragestellung reserviert. Als Beispiel für so ein Vorgehen kann in einem Domain Model (als UML-Klassendiagramm) das Vorhandensein der wesentlichen Entities und deren Haupt-Beziehungen automatisiert geprüft werden (Divekit unterstützt das mit einer eigenen Testbibliothek). Ist dieser Test bestanden, können Betreuer:innen Assoziationstypen und Multiplizitäten manuell prüfen - was für das breite Spektrum richtiger Modellierungen angemessener ist als eine automatisierte Prüfung, die dann unrealistischerweise von "der einen richtigen" Lösung ausgehen würde.

Ein weiteres praktisches Beispiel bei der Umsetzung in Code wäre es, die Clean-Code-Regel "Small!" [Ma09, S. 34] automatisiert prüfen zu lassen (für Divekit wurde dafür im Rahmen einer Bachelorarbeit eine auf PMD [Pr21] basierende Clean-Code-Testbibliothek entwickelt). Die semantisch weitaus komplexere "Stepdown Rule" ("We want the code to read like a top-down narrative" [Ma09, S. 37]) wird dann hingegen manuell durch Betreuer:innen geprüft.

3.6 Umsetzung der 12 Prinzipien durch Divekit

In der nachfolgenden Tabelle ist zusammengefasst, wie Divekit die oben eingeführten 12 Prinzipien für eine praxisnahe digitale Softwaretechnik-Lehre unterstützt.

Tab. 1: Unterstützung der 12 Prinzipien durch Divekit

Prinzip	**Unterstützende Features / Good Practices in Divekit**
(P1) Realitätsnahe Programmieraufgaben gemäß *Active Learning*	Große, zusammenhängende Aufgabenstellung; Individualisierung erzwingt Beschäftigung mit dem Stoff; flankierender Flipped-Classroom-Ansatz für Inhalte
(P2) Mindestens Stufe 3 (*Applying*) der Revised Bloom's Taxonomy	Studierenden müssen die Aufgabenstellung selbstständig in Modelle und Code umsetzen
(P3) Intrinsische über extrinsische Motivation	Praxisnähe, Komplexität der Aufgabenstellung und realistische Tools (Git, IDE) wirken intrinsisch motivierend; zusätzliche extrinsische Leistungskontrolle
(P4) Stetiges formatives Feedback für Lernende durch E-Assessment	Testseite von Divekit gibt formatives Feedback; Unverständliches wird im Dialog mit den Lehrenden geklärt, die über Discord erreichbar sind
(P5) Stetiger Zugriff auf individuelles formatives und summatives Assessment für Lehrende	Divekit-Werkzeuge erlauben das kontinuierliche Monitoring der Gesamtkohorte; für einen individuellen Eindruck kann das Studierenden-Repo geclont werden
(P6) Förderung von Zusammenarbeit im Team, aber *individuelle* Abgabe von Lösungen	Jede:r gibt eine individuelle Lösung ab (auch für große Kohorten aufwandsneutral skalierbar); flankierende Übungen in Kleingruppen zur Förderung der Teamarbeit
(P7) Keine Software-Architektur ohne Coding	Komplexe Aufgabenstellungen mit Modellierung und Coding; Speicherung der Individualisierungs-Konstellationen für auf einander aufbauende, immer komplexer werdende Praktikums-Meilensteine
(P8) Pragmatische Vermittlung von Modellierungs-Fähigkeiten	Individualisierung von UML-Diagrammen in der Aufgabenstellung; Test-Library zur automatischen Prüfung von UML-Klassendiagrammen
(P9) Realistische Größe von Aufgabenstellungen	Siehe Kommentar zu P1-P3.
(P10) Stetige Entwicklung gegen Unit Tests	"Test müssen grün werden", um das Praktikum zu bestehen; Divekit kann auch prüfen, ob selbst Tests geschrieben wurden
(P11) Fokus auf Prinzipien des Domain Driven Designs (DDD)	Native Test-Library für Entities, Value Objects und Aggregates; Strategic Design (z.B. "Bounded Contexts") allerdings nur indirekt über Tabellentests prüfbar

Prinzip	Unterstützende Features / Good Practices in Divekit
(P12) Fokus auf Clean-Code-Regeln und SOLID-Prinzipien	Dedizierte Test-Library für ausgewählte Clean-Code-Regeln und SOLID-Prinzipien

4 Anwendung von Divekit am Beispiel der Veranstaltung Softwaretechnik 2 (ST2)

Die Veranstaltung *Softwaretechnik 2 (ST2)* im Studiengang Informatik der TH Köln beschäftigt sich mit Architektur und Umsetzung von komplexen Softwaresystemen. Durchschnittlich besuchen den Kurs etwa 120 Studierende. Im Sommersemester 2020 wurde erstmals Divekit eingesetzt, um DDD-Konzepte als Programmieraufgaben, die automatisch getestet wurden, in das Praktikum zu integrieren. Aufgaben wurden auf Gitlab hochgeladen, mithilfe von JUnit-Tests überprüft und die Ergebnisse auf einer Übersichtsseite dargestellt. Alle Tests auf der Übersichtsseite mussten bis zum Ende einer Deadline erfolgreich durchlaufen sein, damit das Praktikum als bestanden galt.

Aufgrund der beginnenden Corona-Pandemie waren Präsenztermine nicht mehr möglich. Vorlesungen fanden online über Zoom statt. Da sämtliche Praktikumsaufgaben von zu Hause bearbeitet werden mussten, wurden diese individualisiert und automatisch auf Gitlab verteilt. Auch wenn die JUnit-Tests am Ende bestimmt haben, ob jemand bestanden hat, durften Betreuer über Discord nach Belieben angeschrieben werden, um Ratschläge oder Hilfestellungen zu erhalten. Wenn nötig, fanden sich Lernende und Betreuer in einem Discord-Channel ein, um das Problem zu diskutieren oder gegebenenfalls auch eine Bildschirmübertragung zu starten, damit Betreuer einen tieferen Einblick in die Aufgabenbearbeitung hatten.

Nicht nur die Praktika, sondern auch die Klausur musste digital von Zuhause stattfinden. Folglich wurde auch die Klausur ebenfalls mit Hilfe von Divekit individualisiert und verteilt. Lernende hatten jeweils ein eigenes Repo mit Klausuraufgaben. Für Fragen während der Klausur wurde ein eigener Fragen-Channel in Discord eingerichtet, passende Antworten wurden in einen anderen Antworten-Channel eingestellt. Natürlich waren auch Fragen außerhalb von Discord über E-Mails oder Telefonate mit einem Betreuer erlaubt.

4.1 Feedback der Studierenden

Nach dem Praktikum und der Klausur wurde auf Discord Feedback dazu gesammelt. Außerdem wurde eine anonyme Feedback-Umfrage gestartet, an der sich 17 Studierende beteiligten. Das Gesamtkonzept des Praktikums kam bei den Lernenden gut an. Die Kommunikation über Discord wurde positiv wahrgenommen. Hier fiel insbesondere auf, dass sich Lernende auch gegenseitig über Discord halfen. Auch die automatischen Tests wurden geschätzt, weil sie ein schnelles Feedback boten.

Bezüglich der Klausur wurde das Konzept mit viel Programmierung während der Klausur gut angenommen. Schwierigkeitsgrad und Komplexitätsgrad der Klausuraufgaben wurden

als angemessen empfunden. Allerdings berichteten mehrere Studie von einem Zeitproblem. Für die Zukunft wünschten sich die Studierenden klarere Aufgabenstellungen mit weniger Kontextwechseln. Diese scheinen ein schnelles Verstehen der Aufgaben zu behindern. Leider sind sie ein Nebenprodukt der Individualisierung - eine durchgehende Aufgabenstellung für die ganze Klausur wäre sehr komplex für die Ersteller.

Ein Großteil der Studierenden gab an, dass die Praktikumsabgabe über automatisierte Tests besser sei als herkömmliche mündliche Abgaben. Meistens waren Fehlermeldungen von Unit-Tests aussagekräftig und nachvollziehbar und erlaubten somit ein schnelles Debugging. Dies wurde allerdings erheblich erschwert, wenn Tests versteckt waren, der Programmcode des Tests also nicht einsehbar war.

Wenn Lernende bei der Bearbeitung der Aufgaben keinen Fortschritt machen konnten, weil das Feedback der automatischen Tests nicht ausreichte, wurde Feedback über Discord ermöglicht. Fast alle Studierenden gaben an, dass sie mit dem Kommunikationskanal Discord zur Betreuung viel zufriedener sind als mit herkömmlichen Kanälen wie beispielsweise Sprechstunden, E-Mails oder Foren. Dies könnte unter anderem durch das schnelle Feedback begründet sein, das durch Discord ermöglicht wird. Zwei Drittel der Studierenden erklärten, dass sie im Schnitt innerhalb von einer Stunde eine Hilfestellung erhielten.

Auch aus Sicht der Lehrenden war die Veranstaltung ein Erfolg. In Bezug auf DDD konnte der Bereich des *Tactical Designs* gut in Praktikumsaufgaben thematisiert werden, auch wenn diese von der Komplexität nicht an zu lösende Probleme aus der Praxis heranreichten. Mithilfe der Technologien Java und Spring konnte aufgezeigt werden, inwiefern sich bestimmte Konzepte aus dem DDD in Programmcode überführen lassen.

Das obige Feedback floss in die Weiterentwicklung von Divekit sowie der Aufgabenstellungen für ST2 im Sommersemester 2021 ein. Statt der Vorlesungen wurden zusätzlich der gesamte Stoff als Videos bereitgestellt, um die Veranstaltung im Flipped-Classroom-Format stattfinden zu lassen. Das Feedback wurde in ähnlicher Weise wie im SS2020 erhoben und war durchgehend ähnlich positiv. Das Zitat eines Studierenden fasst (in für die Lehrenden ermutigender Weise ...) viel positives Feedback der Studierenden zusammen: *"Meiner Meinung nach eines der, wenn nicht sogar das beste Modul mit dem meisten Realitätsbezug im gesamten Studium. Die Vorlesungsthemen sind endlich mal in moderner Form aufbereitet und auch die verwendeten Tools sind aktuell. Warum können sich andere Dozenten nicht ein Vorbild daran nehmen?"*

4.2 Vergleich der Studierenden-Ergebnisse zu früheren Klausuren

Vergleicht man die Klausurergebnisse aus den Jahren 2020 und 2021, wo Divekit eingesetzt wurden, mit früheren Jahren (2017 - 2019), so liegt der Notenschnitt mit Divekit bei 2,5 (drei Klausuren), in der Zeit vor 2020 ohne Divekit bei 3,0 (sechs Klausuren). Die Klausuren sind vor und nach Einführung von Divekit inhaltlich vergleichbar[9].

[9] Die Klausuren sind nach der Umstellung auf Divekit eher schwerer geworden, da Code jetzt tatsächlich während der Klausur geschrieben und in einem Repo gepusht werden muss, anstatt wie vorher eher kurze "Pseudocode"-Fragmente auf Papier zu schreiben.

Die Durchfallquote bleibt gleich (jeweils ca. 14,5%), wobei zu beachten ist, dass es in den betrachteten Klausuren ab 2020 aufgrund der Pandemie eine unbegrenzte Freiversuchsregelung gibt; man kann also spekulieren, dass Studierende seit dieser eventuell eher auch ohne Vorbereitung einen Versuch wagen, so dass es als Erfolg zu werten ist, dass sich die Durchfallquote nicht verschlechtert hat.

4.3 Weitere Veranstaltungen

Über Softwaretechnik 2 hinaus wird Divekit an der TH Köln noch in mehreren anderen Veranstaltungen eingesetzt (Softwaretechnik 1 und Algorithmik in Informatik (BA), Coding Essentials 1, Application Design und Microservice Architectures in Code & Context (BA)). Aus Platzgründen kann hier nicht im Detail darauf eingegangen werden, das Feedback der Studierenden ist aber ähnlich wie bei Softwaretechnik 2.

5 Bewertung und Ausblick

Die mit dem Einsatz von Divekit einhergehende Digitalisierung der Lehre hat in mehreren Softwaretechnik-nahen Modulen entscheidend dazu beigetragen, die Herausforderungen der Corona-Pandemie gut zu bewältigen. Nach übereinstimmendem Feedback von Studierenden und Lehrenden führt Divekit zu mehr Praxisnähe, Motivation und Effizienz in der Lehre. Daher wird das Framework in jedem Fall weiter entwickelt und eingesetzt werden, auch wenn irgendwann wieder "normale" Präsenzlehre durchgehend möglich sein wird.

Die Erfahrungen zeigen aber auch, dass eine technische Lösung allein nicht zu besserer Lehre führt. Ebenso wichtig sind ein ganzheitliches Konzept (wie etwa das konsequente Umsetzen von *Active Learning* mit Elementen des Flipped Classroom und Übungs-Workshops), kontinuierliche Ansprechbarkeit der Lehrenden über eine Messaging-Plattform wie etwa Discord, und Good Practices im Einsatz von Divekit (wie etwa ein geschicktes Zusammenspiel von automatisiertem und manuellem Feedback).

Viele Herausforderungen bleiben noch offen und bieten Raum für weitere Forschung und Entwicklung. Dazu zählt in erster Linie die Usability des Frameworks, sowohl für Studierende wie auch für Lehrende. Für Studierende ist das Feedback häufig noch zu kurz und kryptisch, da es ursprünglich aus den Messages fehlgeschlagener Unit-Tests stammt. Lehrende wiederum brauchen bessere Unterstützung beim Verfassen komplexer Aufgabenstellungen - die sprachliche und semantische Variationsvielfalt bei vielen Platzhaltern ist schwer zu beherrschen. Große zusammenhängende Aufgabenstellungen sind aber insbesondere bei Klausuren für Studierende hilfreich, weil sie den Stressfaktor von fachlichen Kontextwechseln minimieren.

Weitere Entwicklungsziele sind der Einsatz von Gamification zur weiteren Steigerung der intrinsischen Motivation, Nutzung von KI für komplexe semantische Fragen, und vieles mehr. Ein kontinuierliches Monitoring des Lernerfolgs bleibt ein zentrales Anliegen, um - gerade in hybriden Lehrformaten - wirklich zu wissen, ob die Studierenden erreicht werden und wo deren Schwierigkeiten liegen.

Divekit soll weiter in Richtung eines Ökosystems mit zahlreichen Nutzer:innen entwickelt werden. Das Framework ist als Open Source unter `https://github.com/divekit` veröffentlicht. Interessierte sind herzlich eingeladen, es zu nutzen und Verbesserungen, neue Features und Good Practices für den Einsatz beizusteuern.

Literatur

[AB19] Anke, J.; Bente, S.: UML in der Hochschullehre: Eine kritische Reflexion. In: SEUH 2019. Feb. 2019.

[Ar16] Armstrong, P.: Bloom's taxonomy, Vanderbilt University Center for Teaching, 2016.

[Be02] Beck, K.: Test-driven development: by example. Addison Wesley, 2002.

[BHS17] Breitner, J.; Hecker, M.; Snelting, G.: Der Grader Praktomat. In (Bott, O. J.; Fricke, P.; Priss, U.; Striewe, M., Hrsg.): Automatisierte Bewertung in der Programmierausbildung. Digitale Medien in der Hochschullehre 6, Waxmann Verlag GmbH, S. 159–172, 2017, URL: `https://www.waxmann.com/automatisiertebewertung/`.

[DA21] DATACOM, S.: Konferenzprogramm OOP 2020, 2021, URL: `https://www.oop-konferenz.de/oop-2021/programm/konferenzprogramm`, Stand: 23. 12. 2021.

[Ev04] Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004.

[FB09] Felder, R. M.; Brent, R.: Active learning: An introduction. ASQ higher education brief 2/4, S. 1–5, 2009.

[Fo19] Fowler, M.: Software Architecture Guide, Aug. 2019, URL: `https://martinfowler.com/architecture/`, Stand: 15. 10. 2021.

[Go10] Goguadze, G.: ActiveMath-generation and reuse of interactive exercises using domain reasoners and automated tutorial strategies, 2010.

[He18] Helfrich-Schkarbanenko, A.; Rapedius, K.; Rutka, V.; Sommer, A.: Mathematische Aufgaben und Lösungen Automatisch Generieren. Springer, 2018.

[HRL20] Hazzan, O.; Ragonis, N.; Lapidot, T.: Guide to Teaching Computer Science: An Activity-Based Approach. Springer Nature, 2020.

[In21] Intveen, J.: Digitalisierung und Individualisierung der praxisorientierten Lehre von modernen Coding-Ansätzen, Magisterarb., Technische Hochschule Köln, Cologne Institute for Digital Ecosystems (CIDE), 2021.

[KJH18] Keuning, H.; Jeuring, J.; Heeren, B.: A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. ACM Transactions on Computing Education 19/1, 3:1–3:43, Sep. 2018, URL: `https://doi.org/10.1145/3231711`, Stand: 25. 10. 2021.

[Ma00] Martin, R. C.: Design Principles and Patterns, 2000, URL: `https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`, Stand: 15. 10. 2021.

[Ma09] Martin, R. C.: Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.

[Pr21] Project, P. O. S.: PMD Source Code Analyzer, 2021, URL: https://pmd.github.io/pmd-6.37.0/, Stand: 25. 12. 2021.

[RD00] Ryan, R. M.; Deci, E. L.: Intrinsic and extrinsic motivations: Classic definitions and new directions. Contemporary educational psychology 25/1, S. 54–67, 2000.

[RMP04] Ridgway, J.; McCusker, S.; Pead, D.: Literature review of e-assessment. Futurelab, 2004.

[VM] VMware, I.: Spring Data JPA, en, URL: https://spring.io/projects/spring-data-jpa, Stand: 15. 10. 2021.

[Wa18] Wasik, S.; Antczak, M.; Badura, J.; Laskowski, A.; Sternal, T.: A Survey on Online Judge Systems and Their Applications. ACM Computing Surveys (CSUR) 51/1, S. 1–34, 2018, ISSN: 0360-0300; 1557-7341.

[WK21] Woelk, F.; Kasch, H.: Code FREAK: Automatisches Feedback für die Programmierausbildung. Die neue Hochschule/2021-5, S. 28–31, Okt. 2021.