Das Potenzial von Pair-Programming und testgetriebener Entwicklung für Berufseinsteiger

Christoph Hellmich
Hochschule Bonn-Rhein-Sieg
Fachbereich Informatik
Sankt Augustin
Email: Christoph.Hellmich@smail.inf.h-brs.de

Mirella Espinoza
Hochschule Bonn-Rhein-Sieg
Fachbereich Informatik
Sankt Augustin
Email: Mirella.Espinoza@smail.inf.h-brs.de

***Zusammenfassung*—*Kontext und Motivation*: Für Hochschulabsolventen/innen eines Informatikstudiengangs ändert sich die Situation häufig grundlegend, sobald sie beginnen im industriellen Umfeld zu arbeiten. Die praktische Anwendung des theoretischen Wissens gewinnt im Beruf immer mehr an Bedeutung. Die Abschluss inhabende Person muss sich unter anderen mit Dingen, wie technischen Schulden, einer neuen Domäne oder der Arbeit in größeren Teams auseinandersetzten. Kommunikation und viel Feedback helfen Berufsunerfahrenen sich in dieser neuen Situation zurechtzufinden. Feedback und Kommunikation sind gleichzeitig auch zwei elementare Werte des Extreme Programmings (XP). Die Umsetzung dieser Werte erfolgt im XP unter anderen durch die beiden Praktiken Pair-Programming und testgetriebene Entwicklung.**

***Zentrale Fragen*: In der vorliegenden Arbeit wird untersucht, welche Potenziale Pair-Programming und testgetriebene Entwicklung, einzeln und in Kombination, für Berufseinsteiger haben.**

***Ergebins*: Potenziale wurden vor allem in den Bereichen der Kommunikation, dem Übernehmen von Verantwortung, der Einarbeitung und der Stärkung des Selbstvertrauens identifiziert. Durch die Kombination von Pair-Programming und der testgetriebenen Entwicklung werden die meisten Herausforderungen angesprochen. Die Nutzung von nur einer der beiden Praktiken birgt ebenfalls viele Potenziale, kann in Einzelfall aber auch zu Problemen führen.**

I. Einleitung

Oft haben Studierende während ihrer Ausbildung nur eingeschränkt Zeit praktische Erfahrungen in der Programmierung zu sammeln. Für Hausarbeiten, die nach Abschluss im Schrank/Papierkorb landen, ist die Codequalität bei der Entwicklung oft kein wichtiges Kriterium. Dies ändert sich bei dem Berufseinstieg, wenn Projekte über mehrere Jahre laufen und gepflegt werden müssen. Technische Schulden, schwer zu wartender oder ungetesteter Code können langfristig zum Scheitern eines Projekts führen. Dies sorgt unter Umständen dafür, dass sich Absolventen[1] an ihrem neuen Arbeitsplatz unter Druck gesetzt fühlen, selbst wenn sie die theoretischen Grundlagen beherrschen. Für den Arbeitgeber bedeutet die mangelnde Praxis der Absolventen, dass teure Fachkräfte bezahlt werden, die sich nicht nur in eine neue Domäne, sondern auch mit der praktischen Programmierung auseinandersetzten müssen, bevor sie für das Unternehmen wirtschaftlich einsetzbar sind. Feedback hilft Unsicherheiten bzw. Unklarheiten zu klären. Häufiges Feedback ist ein zentraler Wert von *Extreme Programming* (XP). Erreicht wird dies im XP unter anderen durch die beiden Praktiken *Pair-Programming* (PP) und *testgetriebene Entwicklung* (TDD). Beim Pair-Programming wird die Software von zwei Entwicklern gemeinsam an einem Rechner implementiert. Die testgetriebene Entwicklung unterscheidet sich von der traditionellen Entwicklung (TLD) insofern, als dass die Tests nicht nach, sondern vor der Implementierung der Funktionalität geschrieben werden [1].

In dem nachfolgenden Abschnitt II sind die Vorgehensweisen beschrieben, nach denen die Potenziale von der TDD und dem PP untersucht wurden. Im Anschluss sind im Abschnitt III die Herausforderungen für Berufseinsteiger schematisch dargestellt. Darauf folgt im Abschnitt IV und im Abschnitt V die getrennte Untersuchung der Potenziale von TDD bzw. PP. Der nächste Abschnitt VI behandelt die Potenziale, wenn die Praktiken gemeinsam angewendet werden. Zum Schluss erfolgt in Abschnitt VII ein Fazit.

II. Methodik

A. Literatur Review zur testgetriebene Entwicklung

Für die Untersuchung der Potenziale der testgetriebenen Entwicklung für Programmieranfänger, wurde eine Literatur Review (LR) durchgeführt. In diesem Abschnitt wird das Vorgehen der LR beschrieben. Deren Auswertung steht in Abschnitt IV.

1) Suche: Für die Literatur Review wurden die digitalen Bibliotheken IEEE Explore, ACM Digital Library, SpringerLink und Science Direct - Elsevier durchsucht. Die Suchanfragen bezogen sich immer auf den Titel, der in der Bibliothek enthaltenden Dokumente. Als Suchbegriff wurden Variationen von „test-driven development“, „test-first“ und „tdd“ verwendet.

Alle Anfragen lieferten für die untersuchte Fragestellung relevante Ergebnisse. Nur die Suche nach Dokumenten die „tdd“ im Titel enthielten, listete vorrangig Dokumente ohne Bezug zur testgetriebenen Entwicklung auf. Daher wurde dieser Suche eine weitere Bedingung hinzugefügt, so dass nur Dokumente gesucht wurden, die „tdd“ im Titel *und* das Wort „test“ im Abstrakt enthielten.

[1] Aus Gründen der besseren Lesbarkeit wird hier und im Folgenden auf die gleichzeitige Verwendung männlicher und weiblicher Sprachformen verzichtet. Sämtliche Personenbezeichnungen gelten gleichwohl für beiderlei Geschlecht.

Die komplette Suche konnte für die Bibliothek IEEE Explorer in einer Suchabfrage zusammengefasst werden und ist nachfolgend repräsentativ für die Suche in den verschiedenen Bibliotheken dargestellt:

("Document Title":"test-driven development"
OR "Document Title":"test driven development"
OR "Document Title":"test-first"
OR "Document Title":"test first"
OR ("Document Title":"tdd" AND "Abstract":"test")).

Die Bibliotheken IEEE Explore, SpringerLink und Science Direct zeigten zu jedem Dokument Empfehlungen bzw. ähnliche Dokumente an. Diese Dokumente wurden ebenfalls mit ausgewertet. Insgesamt führte die Suche zu 38 Dokumente.

2) Beschreibung der Ausschlusskriterien: Die gefundenen 38 Dokumente wurden jeweils hinsichtlich drei Kriterien untersucht. Die Kriterien können wie folgt zusammengefasst werden:

i Empirisch Studie mit Fokus auf TDD vs. TLD
ii Teilnehmer haben theoretische Kenntnisse aber wenig praktische Erfahrung in der Software-Entwicklung
iii Durchführung der Studie in der Industrie

Nachfolgend sind die Kriterien beschrieben.

Erstens sollte es sich bei dem Dokument, um eine empirische Studie handeln. Auswirkung durch die testgetriebene Entwicklung sollten beispielsweise durch Versuche, Umfragen oder Interviews belegt oder widerlegt worden sein. In der Untersuchung im Abschnitt IV stehen die Potenziale der TDD im Vergleich zu denen der traditionellen Entwicklung im Fokus. Daher war es wichtig, dass die TDD oder das traditionelle Vorgehen, auch als test-last Development (TLD) bezeichnet, nicht mit anderen Praktiken kombiniert wurde. Falls zusätzlich, andere Praktiken, wie beispielsweise *Behaviour-Driven Development*, in einer Studie untersucht wurden, sollten die Aussagen der Studie über die TDD und TLD unabhängig davon gültig sein.

Zweitens sollten die Programmierfähigkeiten der Studienteilnehmer berücksichtig worden sein. Für die vorliegende Arbeit war insbesondere die Gruppe von Entwicklern interessant, die bereits einen Studienabschluss im Bereich des Software-Engineerings haben, aber noch keine oder nur wenig praktische Erfahrung in der Industrie vorweisen können.

Für das dritte Kriterium war das Umfeld in dem die Studie durchgeführt wurde entscheidend. Entsprechend der Fragestellung der vorliegenden Arbeit, wurden Studien bevorzugt, deren Untersuchung in einem industriellen Umfeld durchgeführt wurden.

3) Anwendung der Ausschlusskriterien: Nach der Anwendung des ersten Kriteriums wurden 13 Dokumente aussortiert.

Die übrigen 25 Studien boten wenig bis keine Informationen über die Programmierfähigkeiten und den TDD-Erfahrungen der Studienteilnehmer. Als Kompromiss für das zweite Kriterium, wurden Studien aussortiert, deren Teilnehmer aus „ungraduated Students“ oder aus professionellen Entwicklern bestanden. Dadurch wurden 22 Studien aussortiert. Übrig blieben fünf Studien, deren Teilnehmer ausschließlich aus „graduated Students“ bestanden. Aussagen über die Programmierfähigkeiten der Studenten konnten aus den Studien selten extrahiert werden.

Von den Studien deren Teilnehmer einen Hochschulabschluss haben, aber nicht in der Industrie tätig sind, wurde keine im industriellen Umfeld durchgeführt. Das dritte Kriterium wurde daher *nicht* angewandt.

Die übrigen gebliebenen drei Studien sind in Tabelle I beschrieben.

4) Snowballing: Aufgrund der geringen Anzahl an übrig gebliebenen Studien, wurde ein Snowballing mit Literatur Reviews (LR) durchgeführt, die während der oben beschrieben Suche ebenfalls gefunden wurden. Eine erneute Suche mit geänderten Suchabfragen blieb aus, da nach Mäkinen und Münch [5] selbst bei hunderten von Publikationen die TDD erwähnen, nur wenige verwertbare Ergebnisse liefern würden.

In Tabelle II sind die betrachteten LRs aufgelistet. Wurden in einer LR die Informationen über die Teilnehmer einer empirischen Studie extrahiert und angegeben, ob es sich bei ihnen ausschließlich um Studenten mit einen Abschluss handelt, befindet sich in der Spalte *Teilnehmer als Graduated identifiziert* ein ✓. Die Spalte *Auswertung pro Studie* ist mit einem Häkchen versehen, falls separat für jede in der LR analysierte empirische Studie eine Auswertung der Ergebnisse angegeben worden ist, und die LR nicht nur eine zusammenfassende Auswertungen über alle analysierten Studien enthält. Verweist eine LR auf eine andere LR in der Tabelle, ist dies in der letzten Spalte *Referenzen* kenntlich gemacht worden.

Mit Ausnahme von Rafique et al. [6] haben alle untersuchten LRs Aussagen über die interne Codequalität, externe Codequalität und Produktivität getroffen. Rafique et al. [6] verzichtete auf die Auswertung der internen Codequalität, da es dafür noch keine allgemein akzeptierte Metrik gäbe.

In Munir et al. [7] wurden sechs, [4], [16]–[20], und in Rafique et al. [6] zwei, [18], [19], Studien identifiziert, deren Teilnehmer ausschließlich aus Studenten bestanden, die einen Abschluss hatten. Zusätzlich wurde in Rafique et al. [6] eine Studie [17] erwähnt, in der 20 von den 22 Studienteilnehmern einen Abschluss hatten. Die Studie wurde ebenfalls ausgewertet. Abzüglich Überschneidungen zwischen den LRs kamen so weitere sieben Studien zusammen. Von den sieben Studien wurden drei, [19]–[21], aussortiert, da in ihren Untersuchungen TDD zusammen mit Pair-Programming oder einer anderen Praktik aus XP angewendet wurde. Insgesamt wurden durch das Snowballing vier weitere Studien ermittelt.

5) Ergebnis: Die drei gefundenen empirischen Studien aus der Suche in den digitalen Bibliotheken und die vier aus den untersuchten LRs wurden verglichen. Nur eine empirische Studie wurde sowohl bei der Suche in den digitalen Bibliotheken als auch bei der Untersuchung der LRs gefunden. Abzüglich dieser Überschneidung, standen für die Auswertung in Abschnitt IV sechs Studien zur Verfügung.

Tabelle I
ERGEBNISSE DER TDD LITERATUR REVIEW

Studie	Beschreibung	Teilnehmer	Umfeld	Ergebnis
[2]	Studenten wurden in zwei Gruppen eingeteilt. Eine Gruppe entwickelte nach TDD, die andere nach TLD. Der Quellcode, die Testfälle und ein Fragebogen, den die Studenten nach der Entwicklung ausfüllten, wurden analysiert.	22 Studenten mit einem Abschluss.	akademisch	Im Durchschnitt war die Qualität der Tests beider Gruppen gleich. Die Testabdeckung beider Gruppen war nahezu identisch. Die Summe der gefundene Fehler im Code, war bei der test-first-Gruppe geringer. Die test-first-Gruppe war im Durchschnitt eine Stunde vor der Kontrollgruppe mit der Aufgabe fertig.
[3]	Studenten haben in TDD und TLD entwickelt.	21 Studenten mit einem Abschluss	akademisch	Kein signifikanter Unterschied zwischen TDD und TLD in Testaufwand, externe Codequalität und Produktivität
[4]	Studenten wurden in test-first und test-last Gruppen eingeteilt.	Studenten mit einem Abschluss.	akademisch	Kein signifikanter Unterschied zwischen den beiden Gruppen hinsichtlich der Branch Coverage und Mutation Score Indicator.

B. Methodik Pair-Programming

1) Evaluierungsprotokoll: Für das Pair-Programming wurde ein Protokoll zur systematischen Evaluierung entwickelt. In diesem Protokoll werden Forschungsfragen bzgl. der Suchstrategien, Einschluss- und Ausschlusskriterien, Qualitätsbewertung, Datenextraktion und Synthesemethoden gestellt [22], [23].

2) Forschungsfragen: Die folgenden Fragen wurden im Rahmen dieser Arbeit gestellt und beantwortet:

- Was ist Pair-Programming (PP)?
- Welche Merkmale charakterisieren diese Methode?
- Was spricht für und gegen PP?
- Wo wird PP angewendet und warum?
- Welchen Einfluss hat die Zusammensetzung der Programmiererpaare?
- Nach welchen Kriterien werden die Gruppen gebildet?
- Welche Vorteile hat PP für Berufseinsteiger?
- Welche Herausforderungen stellt PP an Berufseinsteiger?

Tabelle II
ÜBERSICHT VON TDD LITERATUR REVIEWS

Studien	Studienteilnehmer als „graduate“ identifiziert	Auswertung pro Studie	Referenzen
Munir [7]	✓	✓	[6], [8]–[12]
Rafique [6]	✓	✓	[8], [9], [13]
Bissi [14]	-	✓	[6], [7], [10]–[12]
Kollanus [8]*	-	✓	-
Jeffries [11]	-	✓	-
Causevic [12]	-	✓	-
Sfetsos [10]	-	✓	-
Mäkinen [5]	-	✓	[6], [11], [13]
Turhan [13]	-	-	-
Shull [9]	-	-	[13]

Legende:

* In [8] werden die gleichen Studien wie in [15] untersucht.

3) Literaturquellen und Suchkriterien: Die Literaturquellen und Suchkriterien umfassten elektronische Datenbanken. Folgende elektronische Datenbanken wurden durchsucht:

- IEEE Explore
- ACM Digital Library
- ResearchGate
- Google Scholar

Folgende Suchbegriffe wurden verwendet:

- Paarprogrammierung
- Pair Programming OR Pair-Programming
- Advantages OR Benefits
- Expert-novice OR Expert AND novice

4) Datenextraktion: Es wurde mehr als 30 Paper gefunden und ausgewertet. Im ersten Schritt wurden die Papers gespeichert. Anschließend wurden die Studien nach ihrem Inhalt tabellarisch klassifiziert. Schließlich wurden hier jene Veröffentlichungen berücksichtigt, die relevante Informationen enthielten (Tabelle III).

C. Vorgehen Kombination beider Praktiken

Im Abschnitt VI lag der Fokus auf eine Interpretation der Ergebnissen der beiden vorhergehenden Abschnitten, in denen die Potentiale des TDDs und des PPs untersucht wurden. Infolgedessen wurde keine LR durchgeführt. Stattdessen wurden Studien verwendet, die aus den vorherigen beiden LRs aussortiert wurden, weil in diesen Studien die TDD und das PP gemeinsam untersucht wurden.

III. HERAUSFORDERUNGEN FÜR BERUFSEINSTEIGER

Für Absolventen, die ihr Informatik Studium beendet haben, stellen sich beim Berufseinstieg in dem Bereich der Software-Entwicklung einige Herausforderungen.

Das Umfeld ändert sich in der Regel. Der Berufseinsteiger muss das Unternehmen und die Mitarbeiter kennenlernen. Wahrscheinlich wird er sich in eine neue Domäne einarbeiten müssen, je nach dem im welchen Bereich das Unternehmen tätig ist. Anders als die Kommilitonen im Studium, werden die Mitarbeiter nicht mehr alle auf den selben Niveau wie er

Tabelle III
AUSGEWERTETE STUDIEN PAIR-PROGRAMMING

Studienkontext	Studientyp	Anzahl Studien	Anzahl Teilnehmer	Referenzen	Anmerkungen
Industrie	Umfrage	3	782	[24]–[26]	Vor-und Nachteile, Qualitätsverbesserung
Industrie	Fallstudie	2	k.A	[24]	Reduktion der Fehler
Industrie	Experiment	1	k.A	[26]	Überprüfung der Codequalität
Industrie	Studie	1	15	[24]	Überprüfung der Codequalität
Industrie	Bericht	1	k.A	[26]	Reduktion der Test und Korrekturarbeiten
Wissenschaft	Studie	3	1408	[24], [26]	Überprüfung der Kompatibilität der Paare
Wissenschaft	Experiment	1	41	[24]	Messung der Qualität des Codes
Industrie	Studie	1	15	[25], [27]	Gute Arbeitsatmosphäre; mehr Zuversicht im Team bzgl. der Produktqualität
Wissenschaft	Studie	1	41	[28], [29]	Vergleich SSolos vs. Paare"
Wissenschaft	Studie	1	k.A	[30]	Designqualität, Fehlerreduktion
Wissenschaft	Studie	1	40	[29], [31]	Messung der Produktivität Anfänger-Experten

sein. Einige Mitarbeiter sind beispielsweise für die Datenbank zuständig, andere für das Frontend oder für den Austausch mit dem Kunden. Jeder Mitarbeiter verfügt über Expertenwissen auf seinem eigenen Gebiet.

Vielleicht nutzt das Unternehmen neuere, ältere oder ganz andere Technologien als die, die der Berufseinsteiger aus seinem Studium kennt. Sehr wahrscheinlich ist, dass er sich in mindestens einer Technologie neu oder vertiefend auseinandersetzten muss. Wenn es sich bei seinem neuen Arbeitsplatz nicht, um ein Startup-Unternehmen handelt, wird vermutlich auch die Entwicklung an Altsystemen in seinen Aufgabenreich fallen. Es ist nicht unwahrscheinlich, dass er sich mit viel fremden Code auseinandersetzten muss. Zusätzlich wird das Unternehmen in der Regel bestimmte Werkzeuge für die Entwicklung, Versionsverwaltung, Fehlerberichte und ähnliches bereitstellen, mit denen er sich eventuell erst vertraut machen muss.

Die Arbeitsweise unterscheidet sich in einigen Punkten von der im Studium. Die Projekte werden meist längere Laufzeiten haben, als dies bei den Haus- oder Seminararbeiten im Studium der Fall war. Daher ist beispielsweise während der Entwicklung, vermehrt auf die Vermeidung von technischen Schulden zu achten. Es könnte sein, dass der Absolvent seine Arbeit nicht mehr ganz so frei, wie im Studium, einteilen kann. Arbeiten müssen evtl. plötzlich für einen kritischen Bugfix unterbrochen werden. Nachdem der Fehler behoben ist, muss die vorherige Arbeit ohne lange Einarbeitung fortgesetzt werden können. Anders als im Studium, wo Kommilitonen und Professoren meistens erreichbar sind und schnell Rückmeldung geben, ist der Austausch im Beruf mit anderen Projektbeteiligten, wie Mitarbeitern oder Kunden vielleicht nicht mehr ohne weiteres möglich. Die Mitarbeiter arbeiten vielleicht an einem anderen Standort oder haben Urlaub, der Kunde ist nicht erreichbar.

Baytiyeh und Naja [32] befragten 217 Ingenieure, darunter 78% männlich und 22% weiblich, aus den Bereichen Bauingenieurwesen (32%), Maschinenbau (23%), Elektrotechnik (27%), Technische Informatik (10%) und Wirtschaftsingenieurwesen (8%) über Herausforderungen die sie im Übergang vom Studium in den Beruf sehen. Die Teilnehmer arbeiteten im Libanon (43%), der Region um den Persischen Golf (33%), Europa und Nordamerika (14%) und Afrika (10%). Die größten Herausforderungen fallen in die drei Bereiche Kommunikation, Verantwortung und Selbstvertrauen. Die drei Bereiche wurden aus den Antworten der Teilnehmer, wie folgt abgeleitet:

- Kommunikation
  - Zusammenarbeit mit Menschen aus einem anderen Bereich
  - Umgang mit dem Vorgesetzten
- Verantwortung
  - Verantwortung übernehmen
  - Verantwortlich für Ergebnisse
  - Arbeitsdruck
  - Selbständig arbeiten
- Selbstvertrauen
  - Angst Fehler zu machen
  - Nicht genug zu Wissen
  - Selbständig lernen

Die Liste wurde mit den oben beschriebenen Herausforderungen kombiniert und dient in den nachfolgenden Abschnitten als Schema, um die Potenziale von Pair-Programming und der testgetriebenen Entwicklung zu bewerten.

- Kommunikation
  - Zusammenarbeit mit Menschen aus einem anderen Bereich, kein Expertenwissen, unterschiedliches Niveau
  - Kommunikation mit Kunden nur eingeschränkt möglich
  - Kommunikation mit Mitarbeitern nur eingeschränkt möglich
- Verantwortung
  - Verantwortung übernehmen, längere Projektlaufzeiten
  - Verantwortlich für Ergebnisse, technische Schulden

  - Arbeitsdruck, Unterbrechungen
  - Selbständig arbeiten
- Selbständig lernen, Einarbeitung
  - Neue Domäne
  - Neue Technologie
  - Altsysteme / fremder Code
  - Neue Werkzeuge
- Selbstvertrauen
  - Angst Fehler zu machen
  - Nicht genug zu Wissen

IV. Testgetriebene Entwicklung

Die testgetriebene Entwicklung (TDD) ist eine der zentralen Praktiken des XPs, kann aber auch unabhängig davon angewendet werden. Die Methode setzt auf kurze Iterationen, in denen sowohl die Software als auch die Software-Architektur schrittweise entsteht. Anders als der Name vermuten lässt, ist TDD keine Teststrategie [33].

Nichtsdestotrotz beginnt jede Iteration mit dem Schreiben eines Tests. Anschließend wird genauso viel Code geschrieben, wie der Test für eine erfolgreiche Prüfung benötigt. Sobald der Test mit einem positiven Ergebnis durchgelaufen ist, kann, falls nötig, die bestehende Codebasis einem *Refactoring* unterzogen werden [33].

Während des Refactoring wird die interne Struktur von existierenden und funktionierenden Programmcodes verbessert. Ziel ist es, den Code verständlicher zu machen, ohne neue Funktionalität hinzuzufügen [34].

Etwas formaler kann die testgetriebene Entwicklung anhand von zwei Regeln beschrieben werden [35]:

- Es soll nur neuer Code geschrieben werden, wenn ein Test fehlschlägt.
- Redundanzen sollen eliminiert werden.

Diese beiden Regeln geben die Reihenfolge der Arbeitsschritte bei der Entwicklung vor [35]:

1) Schreibe einen Test der kompiliert, aber fehlschlägt (Red).
2) Schreibe gerade so viel Code, dass der Test läuft (Green).
3) Überarbeite den Code, so dass keine Redundanzen mehr enthalten sind (Refactor).

Insgesamt kann der Ablauf von der TDD mit dem kurzen Mantra „Red/Green/Refactor“[2] beschrieben werden [35]. In Abbildung 1 sind die Unterschiede von TDD und TLD in Form zweier Aktivitätsdiagramme dargestellt.

Im Folgenden sind Vor- und Nachteile von der TDD aufgelistet, die häufig in der Literatur genannt werden. Im Anschluss sind positive und negative Auswirkung von der TDD beschrieben, die in Studien ermittelt wurden.

A. Vorteile der testgetriebenen Entwicklung

Nachfolgend sind Vorteile von der TDD beschrieben.

[2]Fehlgeschlagene Tests werden in der Praxis oft durch die Farbe Rot und erfolgreiche Tests durch die Farbe Grün signalisiert.

1) Testabdeckung/Flexibilität: In der TDD soll neue Funktionalität nur dann hinzugefügt werden, wenn zuvor der entsprechende Test geschrieben wurde. Dadurch wird sichergestellt, dass nahezu der gesamte Code der Anwendung getestet ist. Die so erreichte hohe Testabdeckung, ermöglicht es, den gesamten Programmcode auf Fehler zu prüfen. Dies hilft dem Entwickler festzustellen, ob beispielsweise eine vorgenommene Änderung im Code, an einer anderen Stelle versehentlich einem Fehler in Programmablauf hervorruft. Der Code bleibt durch die hohe Testabdeckung flexibler. [33], [36].

Die hohe Testabdeckung erlaubt es außerdem, eine Aussage über die Funktionsfähigkeit der Anwendung zu machen. Dies ist sowohl für die Planung als auch die Durchführung einer Softwareintegration wichtig [34].

2) Programmdesign: Eine Klasse lässt sich einfacher Testen, wenn sie keine oder wenig Abhängigkeit zu einer anderen Klasse besitzt. Durch die TDD werden die Entwickler ständig angehalten, auf eine lose Kopplung bei den Entwurf von Klassen zu achten [34], [36], [37].

Allerdings heißt es auch in Martin [36, S. 35]: „Good designs aren’t free, and TDD doesn’t guarantee good designs. However, TDD provides powerful impetus to decouple, forcing developers to think through their designs in ways that they otherwise might not.“ Die TDD kann den Entwickler also nur dazu bringen auf das Design zu achten, aber dies muss nicht unweigerlich zu einen guten Design führen.

3) Motivation: Bei der traditionellen Entwicklung, wird mit der Implementierung der Funktionalität begonnen und im Anschluss sollen die Tests geschrieben werden, um die Funktionalität zu prüfen. Häufig fehlt dem Entwickler jedoch die Motivation nach der Implementierung noch Tests zu schreiben. Er ist sich sicher, dass seine Implementierung fehlerfrei ist. Es ist für ihn dann oft interessanter, mit der Implementierung des nächsten Features zu beginnen. Die ohnehin geringe Motivation Tests zu schreiben, kann zusätzlich noch negativ beeinflusst werden, wenn die Abhängigkeiten zwischen den erstellten Klassen sehr groß sind, da dies das Schreiben von Tests erschwert. Beim TDD werden diese Probleme umgangen, da die Entwicklung mit dem Schreiben des Tests startet [34].

In Shull [9, S. 17] heißt es, dass TDD hilft Schludrigkeiten zu verhindern und Disziplin beim Programmieren fördert: „He [Grigori] described working with a young, energetic programmer whose work unfortunately included many mistakes. Following TDD rigorously helped the programmer become more intentional in his work, thinking through the functionality he wanted to add. TDD doesn’t just require skill and discipline; it also helps develop them.“

4) Dokumentation: Ein der Teil der Dokumentation des Codes erfolgt in Form von Tests. Die Tests geben Auskunft über die normale Verwendung von Klassen und deren Umgang mit Ausnahmen. Außerdem können die Tests als Kommunikationsgrundlage für den Austausch mit anderen Entwicklern dienen [34], [36].

Diese Dokumentation in Form von (Test-)Code wird von jedem Entwickler verstanden. Und dass die Tests bzw. die Dokumentation immer auf dem aktuellen Stand sind, ist durch

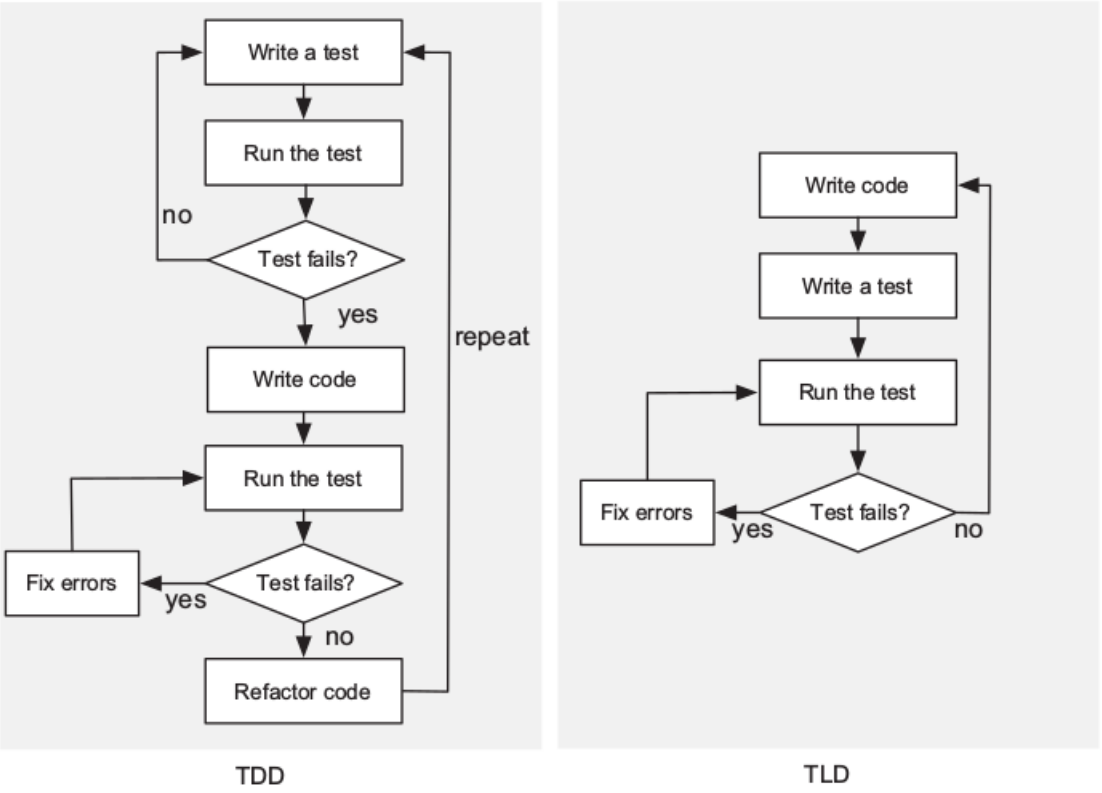

Abbildung 1. TDD vs. TLD, aus [7]

das Vorgehen der TDD festgelegt. Jedoch muss für eine gute Lesbarkeit der Tests, bei deren Erstellung mit entsprechender Sorgfalt vorgegangen werden [36].

5) Feedback/Unsicherheiten nehmen: TDD kann dabei helfen mit Unsicherheiten bei der Entwicklung umzugehen, die einen begegnet, wenn man auf ein schwer zu lösendes Problem trifft. Wenn Unsicherheiten einen zaghaft, weniger Kommunikativ und ängstlich gegenüber Fehlern machen, soll TDD den Entwickler, durch das kleinschrittige Vorgehen, unterstützen schnell zu lernen, klarer zu kommunizieren und konkretes Feedback zu finden [35].

Dies wird auf verschiedene Weise erreicht. Durch die vielen kurzen Iterationen erhält der Entwickler häufig Feedback. Eine fremde Schnittstelle kann mit der Hilfe von Tests schrittweise kennengelernt werden. Schwer zu lösende Probleme können in kleinere, einfacher zu lösende Probleme aufgeteilt und schrittweise bearbeitet werden. Dabei wird durch die Tests, für jeden Schritt ein Maß an Korrektheit sichergestellt, welches bei der Entwicklung ohne Tests fehlen würde. Durch die vielen Tests, erhält man ein sehr granulares Feedback, so dass weniger Zeit im Debugger verbracht werden muss, um die Ursache eines Fehlers zu finden [34]–[36].

B. Nachteile der testgetriebenen Entwicklung

Kollanus [15] listet einige potentielle Probleme von TDD auf.

1) Testabdeckung/Flexibilität: Typischerweise ist das Verhältnis zwischen Test- und Produktionscode eins zu eins. In großen Projekten kann eine große Menge von Testfällen zu verschiedenen Problemen führen. Zum einen kann die Entwicklungszeit durch die vielen Tests stark verzögert werden, wenn das Ausführen aller Tests beispielsweise mehrere Minuten in Anspruch nimmt. In solchen Fällen, kann nur noch eine Untermenge der Tests nach jeder Iteration durchlaufen werden. Zum anderen benötigt das Refactoring mehr Aufwand, wenn viele Testfällen von den Änderungen betroffen sind, und angepasst werden müssen [15].

2) Programmdesign: Das Fehlen einer vorab durchgeführten, detaillierten Designphase kann zu einem mangelhaften Design der Anwendung führen. Was wiederum ein umfangreiches Refacotring zur Folge haben kann [15].

3) Motivation: Es ist schwer die TDD im Unternehmen einzuführen, da es nicht leicht zu lernen ist und mehr Aufwand und Zeit benötig als zunächst von den Entwicklern erwartet wird. Außerdem ist die Grundeinstellung gegenüber TDD bei einigen negativ. Zum einen wird die Produktivität des Verfahrens hinterfragt. Zum anderen möchten einige Entwickler ungern ihre Art zu programmieren ändern. Und selbst, wenn die Entwickler von möglichen Vorteilen überzeugt sind, fehlt es an Motivation die TDD praktisch anzuwenden, da der Aufwand für den Einzelnen zu hoch ist [15].

4) Erforderliches Fähigkeitslevel: Obwohl der Ablauf der testgetriebenen Entwicklung überschaubar ist

(Red/Green/Refactore), gibt es einige Probleme, auf die Anfänger zu Beginn schnell stoßen. Der Entwickler benötigt ein Big-Picture des Programms, um Tests schreiben zu können. Erfahrene Entwickler können das Design im Kopf entwerfen, und daher funktioniert die TDD für sie. Auf Anfänger trifft dies jedoch nicht zu. Neben dem Problem, zu bestimmen *was* getestet werden soll, bereitet zusätzlich auch das *wie* ein Tests geschrieben wird, einem Anfänger Probleme. Direkt beim Schreiben des ersten Tests, stellt sich beispielsweise die Frage, wie groß ein Test bzw. eine Iteration sein sollte. Insgesamt erfordert es viel Übung, bis ein Entwickler fähig ist, gute Testfälle zu schreiben. Anfänger können damit schnell überfordert sein [15], [35].

Die Umfrage von Aniche und Gerosa [38] zeigt ebenfalls Schwierigkeiten, die die Entwickler mit der testgetriebenen Entwicklung haben. Sie befragten 218 Entwickler nach den Fehlern, die ihnen bei dem TDD-Prozess *häufig oder immer* unterlaufen:

- 26.61% schreiben zu komplexe Testszenarien.
- 23.85% führen ein Refactoring eines anderen Codeteils durch, anstatt den Teil zu korrigieren, der einen Tests fehlschlagen lässt.
- 19.72% vergessen den Refactoring-Schritt durchzuführen.
- 15.14% halten sich nicht an die Empfehlung, mit dem Schreiben des einfachsten Tests zu beginnen.
- 14.22% vergessen zu prüfen, ob der gerade geschriebene Test tatsächlich fehlschlägt.
- 11.01% verwenden schlechte Testname.
- 11,01% implementieren nicht nur den nötigsten Code, damit der Test erfolgreich ist.
- 8.72% vergessen den Testcode einem Refactoring zu unterziehen.
- 5.96% vergessen am Ende einer Iteration alle Tests ausführen.

5) Anwendbarkeit: Ungeachtet der Erfahrung des Entwicklers, kann es je nach Kontext sehr schwer sein, Tests zu schreiben. Dies gilt zum Beispiel für das Schreiben von Tests für Benutzeroberflächen. Ein weiteres Beispiel sind eingebettete Systeme, da es unter Umständen unmöglich sein kann, die Tests auf der Zielhardware auszuführen. In manchen Fällen ist die Anwendbarkeit der TDD abhängig von den zur Verfügungen stehenden Entwicklungswerkzeugen. Ohne die passenden Werkzeuge bzw. Werkzeugunterstützung, kann die TDD schnell zu Zeitaufwändig werden [15].

6) Positives Test-Bias: Bei der TDD werden vor allem positive Testfälle geschrieben, d.h. es wird getestet wie das System im Normalfall reagieren soll. Negative Testfälle, beispielsweise fehlerhafte Benutzereingaben, werden seltener überprüft. Jedoch werden durch negative Testfälle mehr defekte gefunden als durch positive Testfälle. Negative Testfälle sind daher für die Entwicklung wichtig, werden bei der TDD aber zu wenig beachtet [39].

C. Auswertung Studien

In diesem Abschnitt werden die Ergebnisse der gefundenen Studien aus Abschnitt II-A vorgestellt. Die Auswirkungen von der TDD werden in Experimenten, Befragungen, Fallstudien durch verschiedenen Metriken beschrieben. Munir et al. [7] listet acht Variablen auf, die in seinen untersuchten Studien häufig verwendet wurden: Produktivität, Aufwand/Zeit, Größe, externe Qualität, interne Codequaltät, Entwicklermeinung, Robustheit und Konformität. Für jede Variable nennt er wiederum verschiedene Metriken. Nachfolgend werden die Ergebnisse zur Produktivität, externe Qualität und interne Codequalität näher betrachtet. Diese wurden von den meisten der betrachteten Studien verwendet.

1) Produktivität: In der Studie [17] von Gupta und Jalote wurden 22 Studenten gleichmäßig in zwei Gruppen eingeteilt. Eine Gruppe entwickelte testgetrieben, die andere traditionell. Die Gruppe die testgetrieben entwickelte, benötigte im Durchschnitt weniger Zeit zum Lösen von zwei vorgegebenen Programmieraufgabe. Der unterschied zwischen beiden Gruppen war jedoch nicht statistisch signifikant. Berechnet wurde die Produktivität mit der Formel 1, wobei NCLOC für nicht auskommentierte Codezeilen steht und DE, die Zeit für die Entwicklung in Personenstunden beschreibt.

$$Produktivitaet = \frac{NCLOC}{DE} \tag{1}$$

Mülller und Hagner [18] teilte die 19 Teilnehmer seiner Studie ebenfalls in eine TDD-Gruppe und eine TLD-Gruppe ein, bestehend aus 10 bzw. 9 Studenten. Die Arbeit der Teilnehmer wurde in zwei Phasen eingeteilt. In der erste Phase wurde eine Anwendung implementiert. In der zweiten Phase führten die Studenten Akzeptanztest aus und behoben gegebenenfalls Fehler in ihrer Implementierung. Für die Evaluation der Produktivität wurden verschiedene Zeiten und Brüche verglichen. Dazu gehörte die Zeit, die für die die komplette Aufgabe benötigt wurde, die Zeit die für die Implementierungsphase benötigt wurde und das Verhältnis zwischen den beiden zuvor genannten Zeiten. Hinsichtlich der für die Aufgabe insgesamt benötigten Zeit, wurde kein Unterschied zwischen den beiden Gruppen festgestellt.

An der Studie von Causevic et al. [2] nahmen 14 Studenten teil. Sie wurden in zwei Gruppen eingeteilt. Die eine Gruppe nutzte die TDD, um eine vorgegebene Aufgabe zu lösen. Die andere Gruppe ging traditionell vor. Die TDD-Gruppe war im Durchschnitt eine Stunde vor der TLD-Gruppe mit der Entwicklung fertig. Dies war jedoch nicht statistisch signifikant.

In der Studie von Fucci et al. [3] entwickelten 21 Studenten einen Tag testgetrieben und einen Tag traditionell. Es wurden 40 Beobachtungen von 20 gültigen Teilnehmern analysiert. Dabei konnten keine Unterschiede hinsichtlich der Produktivität zwischen der TDD und der TLD festgestellt werden.

In zwei Studien neigten die Entwickler die testgetrieben entwickelten dazu, produktiver zu sein [2], [17]. Drei Studien konnten keinen Unterschied in der Produktivität zwischen TDD und TLD feststellen [3], [17], [18]. Werden jedoch nur die statistisch signifikanten Ergebnisse betrachtet, konnte weder die TDD noch die TLD als produktivere Praktik identifiziert werden.

2) Externe Codequalität: An der Studie von Madeyski [16] nahmen 188 Studenten teil. Die Studenten wurden in vier Gruppen eingeteilt. Für diesen Abschnitt sind zwei Gruppen interessant. Die eine Gruppe entwickelte testgetrieben und bestand aus 34 Studenten. Die andere Gruppe entwickelte traditionell und bestand aus 32 Studenten. Der von der TDD-Gruppe geschriebene Code bestand weniger Akzeptanztests als der Code der TLD-Gruppe. Die externe Codequalität war statistisch signifikant geringer.

Auch in Gupta und Jalote [17] wurde die externe Codequalität durch die Anzahl erfolgreich durchlaufener Akzeptanztests ermittelt. Die Studienteilnehmer mussten zwei Programmieraufgaben lösen. Für die eine Programmieraufgabe war die externe Codequalität der TDD-Gruppe statistisch signifikant besser. Bei der anderen Aufgabe wurde kein statistisch signifikanter unterschied hinsichtlich der externe Codequalität zwischen den beiden Gruppen festgestellt. Für diese Aufgabe war die Anzahl der bestandenen Akzeptanztest bei der TLD-Gruppe höher, was eine bessere externe Codequalität vermuten lässt.

In Müller und Hagner [18] wurde die Anzahl der Fehler ebenfalls durch Akzeptanztest gemessen. Die externe Codequalität der TDD-Gruppe war nach der Implementierungsphase statistisch signifikant geringer. Nachdem die Studienteilnehmer Zeit hatten, die durch die Akzeptanztest aufgedeckten Fehler zu beheben, war die externe Codeqaulität der TDD-Gruppe ein wenig, jedoch nicht statistisch signifikant, besser als die der TLD-Gruppe.

Am Ende des Versuchs in Causevic et al. [2] wurde auf die Lösung jedes Teilnehmers, die Tests aller anderen Teilnehmer angewändet. Die fehlgeschlagenen Assertions dieser Tests wurden jeweils für jeden Teilnehmer gezählt und ausgewertet. Der Code der Teilnehmer die testgetrieben entwickelten, hatten im Durchschnitt weniger Defekte als der Code der Teilnehmer die traditionell entwickelten. Der Unterschied war jedoch nicht statistisch signifikant.

Fucci et al. [3] kam in seiner Sudie zu dem Ergebnisse, dass es kein Unterschied zwischen der TDD und der TLD hinsichtlich der externe Codequalität gibt. Die Codequalität wurde mit Hilfe von Akzeptanztests gemessen.

Für die externe Codequalität zeigt sich ein gemischtes Bild. In einer Studie gibt es keinen Unterschied zwischen TDD und TLD [3]. Drei Studien erhalten keine statistisch signifikanten Ergebnisse. Davon tendieren in zwei der Studien die Teilnehmer, die TDD anwenden, zu besseren Ergebnissen [2], [18]. Die andere Studie kommt zu einem entgegengesetzten Resultat. In dieser liefert die TLD-Gruppe besseren Ergebnissen [17]. Zwei Studien kommen zu statistisch signifikanten Ergebnissen. In [17] schneidet TDD signifikant besser ab. Und in [16] ist wiederum TLD signifikant im Vorteil gegenüber TDD.

3) Interne Codequalität: In der Studie von Madeyski [4] wurden 19 Studenten in zwei Gruppen eingeteilt. Die TLD-Gruppe bestand aus zehn Studenten, die TDD-Gruppe aus neun. Zwischen den beiden Gruppen wurde kein statistisch signifikanter Unterschied hinsichtlich des Branch-Coverage gemessen. Die durchschnittliche Branch-Coverage der TDD-Gruppe war, um 8% Punkte höher.

Die Branch-Coverage wurde in Müller und Hagner [18] gemessen, nachdem die Aufgabe der Teilnehme vollständig abgeschlossen war. Die Ergebnisse der Messungen beider Gruppen waren ähnlich. Es konnte kein statistisch signifikanten Unterschied festgestellt werden.

In Causevic et al. [2] wurde die Code-Coverage der Teilnehmer untersucht. Im Durchschnitt war die Code-Covergage der TDL- und der TDD-Gruppe nahezu identisch.

Drei Studien haben die interne Codequalität untersucht. Zwei Studien konnten keinen Unterschied zwischen der TDD und der TLD feststellen [2], [18]. In [4] war die Testabdeckung der Teilnehmer, die testgetrieben entwickelten, geringfügig höher.

D. Potenziale für Berufseinsteiger

Die Auswertung der Studien nach den Metriken Produktivität, externe Codequalität und interne Codequalität in Abschnitt IV-C ist in Tabelle IV zusammengefasst. Die Auswertung hat gezeigt, dass viele Studien, einzeln und zusammengenommen, noch keine eindeutigen Ergebnisse liefern. Ob die TDD oder die TLD verwendet wird, scheint insgesamt keinen oder kaum Auswirkung auf die Produktivität, externe Codequalität und interne Codequalität zu haben. Nach Fucci et al. [40] besteht kein Unterschied zwischen der TDD und der TLD, und Vorteile ergeben sich allein durch das schrittweise Vorgehen.

Tabelle IV
AUSWERTUNG TDD-STUDIEN

Studie	Produktivität	Externe Codequalität	Interne Codequalität
[4]	k.A.	k.A.	$=^{+}$
[16]	k.A.	-	k.A.
[17]^{A1}	=	+	k.A.
[17]^{A2}	$=^{+}$	$=^{-}$	k.A.
[18]	=	$=^{+}$	=
[2]	$=^{+}$	$=^{+}$	=
[3]	=	=	k.A.

Legende:

= keine statistisch signifikanten Auswirkungen
+/- statistisch signifikant positive/negative Auswirkungen von TDD gegenüber TLD
$^{+/-}$ positive/negative Tendenzen von TDD gegenüber TLD
^{A1} Aufgabe 1
^{A2} Aufgabe 2

Nachfolgend sind die Vor- und Nachteile der TDD aus dem Abschnitt IV-A zusammengefasst. Manche Punkte, wie z.B. die Testabdeckung, konnten sowohl als Vorteil als auch als Nachteil ausgelegt werden.

- Vorteile
  1) Testabdeckung/Flexibilität
  2) Programmdesign
  3) Motivation
  4) Dokumentation

5) Feedback/Unsicherheiten nehmen

- Nachteile
  1) Testabdeckung/Flexibilität
  2) Programmdesign
  3) Motivation
  4) Erforderliches Fähigkeitslevel
  5) Anwendbarkeit
  6) Positives Test-Bias

Die Auswertung in Abschnitt IV-C hat gezeigt, wie schwer es ist, die Effekte von TDD eindeutig zu bestimmen. Selbst Untersuchungen von klar definierten Metriken wie die Anzahl fehlgeschlagener Akzeptanztests, lieferten unterschiedliche Ergebnisses. Im Folgenden werden die Potenziale von der TDD für Berufseinsteiger beschrieben.

Obwohl die TDD eine Praktik ist, die von einem Entwickler allein durchgeführt werden kann, bietet die TDD dennoch Potenziale für die Kommunikation im Beruf. Zum einen können die Tests als Grundlage für Fragen genutzt werden und so dem Berufseinsteiger helfen sein Problem einem anderen Mitarbeiter zu kommunizieren, unabhängig davon auf welchem Niveau er oder der Mitarbeiter ist. Des Weiteren können die Tests auch eine asynchrone Kommunikation unterstützten, wenn der Mitarbeiter beispielsweise nur per E-Mail erreichbar ist. Für die Kommunikation mit den Kunden bietet TDD hingegen keinen Mehrwert. Hierfür würde sich das Vorgehen nach Acceptancetest-driven Development (ATDD) [41] eigenen. Bei ATDD werden die Akzeptanzkriterien gemeinsam mit dem Kunden identifiziert und in Tests festgehalten. Diese Praktik fällt jedoch aus dem Rahmen der vorliegenden Arbeit.

Die TDD unterstützt den Entwickler das Programm und das Design schrittweise zu entwickeln. Umfangreiche Projekte können in kleine Probleme aufgeteilt und schrittweise angegangen werden. Das Gefühl zu wissen, dass der bisher geschriebene Code getestet ist, kann dem Entwickler mehr Sicherheit hinsichtlich seiner Arbeit geben. Ungeachtet dessen, dass der Code gleichwohl Fehler enthalten kann, ist die Lauffähigkeit zumindest für ein Teil der Anwendung sichergestellt. Der Refacotring-Schritt am Ende jeder Iteration im TDD-Prozess, erinnert den Entwickler auch bei längeren Projektlaufzeiten, technische Schulden kontinuierlich zu beheben. Der Abschluss einer Iteration ist ein guter Zeitpunkt, die Arbeit zu unterbrechen, falls dies erforderlich ist. Mit dem Schreiben eines neuen Tests, kann die Arbeit dann zu einem späteren Zeitpunkt fortgesetzt werden, ohne dass eine größere Einarbeitung nötig ist. Die hohe Testabdeckung hilft dem Entwickler, den Stand seiner Entwicklung besser einschätzen zu können, was ihm bei der Terminplanung hilft und den Arbeitsdruck senken kann. Dadurch, dass eine Iteration mit dem Schreiben eines Tests beginnt, ist der Entwickler gezwungen sich mit Problemstellung auseinanderzusetzen, bevor er mit der Programmierung beginnt. Die Fokussierung auf den Test, hilft das Problem nicht aus den Augen zu verlieren und selbständig zu arbeiten.

Der Entwickler kann seine Annahmen über ein Altsystem oder ein unbekanntes Framework mit Tests überprüft. Wurden für das Altsystem oder Framework bereits Tests geschrieben, kann er diese ebenfalls studieren, um sich über die Verwendung des Altsystems bzw. Frameworks klar zu werden. Sowohl das Schreiben als auch das Lesen von Tests können die Einarbeitung erleichtern. Für die Einarbeitung in eine neue Domäne oder Werkzeug eignet sich die TDD jedoch weniger bzw. gar nicht.

Die Tests bzw. das Aufteilen größerer Probleme in kleinere Testfälle kann helfen, mit Unsicherheiten umzugehen. Ist der Entwickler nicht geübt testgetrieben zu entwickeln, kann sich das Gefühl nicht genug zu Wissen noch verstärken.

Nachfolgend sind die Auswirkungen von der TDD auf die Herausforderungen von Berufseinsteigern abgebildet. Falls die TDD eine Herausforderung adressiert, steht am Ende ein ✓. Hat die TDD keinen Einfluss auf eine Herausforderung, ist diese durchgestrichen. Wirkt sich die TDD möglicherweise negativ auf die Herausforderung aus, ist dies durch ein Bkenntlich gemacht worden.

- Kommunikation
  - Zusammenarbeit mit Menschen aus einem anderen Bereich oder mit unterschiedlichen Niveau ✓
  - ~~Kommunikation mit Kunden nur eingeschränkt möglich~~
  - Kommunikation mit Mitarbeitern nur eingeschränkt möglich ✓
- Verantwortung
  - Verantwortung übernehmen ✓
  - Verantwortlich für Ergebnisse ✓
  - Arbeitsdruck ✓
  - Selbständig arbeiten ✓
- Selbständig lernen
  - ~~neue Domäne~~
  - Neue Technologie ✓
  - Altsysteme / fremder Code ✓
  - ~~neue Werkzeuge~~
- Selbstvertrauen, unterschiedliches Niveau
  - Angst Fehler zu machen ✓
  - Nicht genug zu Wissen B

Zusammenfassend bietet die TDD einige Potenziale für den Berufseinsteiger. Allerdings sollte er bereits Kenntnisse in der TDD besitzen oder zumindest motiviert sein, diese zu erlangen. Ansonsten könnte sich das Gefühl des Berufseinsteigers verstärken, nicht genug zu Wissen. Zusätzlich muss sich die Aufgabe dazu eigenen, testgetrieben entwickelt zu werden. Dies kann beispielsweise im Frontend-Bereich problematisch sein.

V. Pair-Programming

Pair-Programming (PP) bezieht sich auf eine Praxis, bei der zwei Programmierer an einem Rechner am gleichen Design, Algorithmus, Code oder Test zusammenarbeiten [42]. Das Paar besteht aus zwei Entwicklern, die zwischen den Rollen des *Piloten* und *Navigators* wechseln. Der Pilot nimmt alle Tastatur- und Maus-Eingaben innerhalb der Entwicklungsumgebung vor. Der Navigator beobachtet die Eingaben, vergleicht

die tatsächliche Funktionalität des neuen Codes mit der erwarteten Funktionalität und hält die aktive Kommunikation mit dem Pilot hinsichtlich der Korrektheit und Angemessenheit des Codes aufrecht.

PP unterstützt grundlegende Design- und Review-Phasen im Entwicklungsprozess. Die Programmierer müssen mögliche Ansätze diskutieren, über die beste Lösung entscheiden und diese anschließen umsetzten. Beide Mitglieder überprüfen die Argumente des anderen. Hierdurch wird zusätzlich die Fähigkeit der Entwickler erhöht, effektive Bewertungen durchzuführen [24], [26], [29].

A. Vorteile von Pair-Programming

Begel und Nagappan [43] haben 487 Programmierer, inklusive Softwareentwickler, Softwaretester und Manager von Microsoft über ihre Erfahrungen mit Pair-Programming befragt. Im Folgenden sind die von den Teilnehmern beschriebenen Vor- und Nachteile zusammengefasst.

1) Weniger Bugs: PP reduziert die Anzahl der Fehler erheblich. Außerdem werden Fehler in der Entwicklung früher entdeckt. So kann verhindert werden, dass sie sich tief in das Design einbetten [43].

2) Vertieftes Code Verständnis: PP hilft das Code-Verständnis des Paares zu vertiefen. PP führt zu geteilten, gleich guten Kenntnissen des Produkts und ist ein effizientes Mittel, um ein tieferes Verständnis für eine größere Codebasis im gesamten Team zu fördern. Hinsichtlich der Risikovermeidung vermindert PP die Auswirkung von Mitarbeiter-Abwanderungen, da niemals nur eine Person im Team einen bestimmten Teil des Codes kennt [26], [43].

3) Hohe Codequalität: PP verbessert die Software-Eigenschaften und die Codequalität in Bezug auf die Übereinstimmung mit vorgegebenen Richtlinien. Durch intensive Überprüfung und Zusammenarbeit verbessert PP die Qualität [26], [43].

4) Vom Partner lernen: PP ermöglicht es von dem Partner zu lernen. Damit ist PP ist ein exzellenter Weg, um schnell neue Mitarbeiter einzuarbeiten und neue Techniken schneller zu lernen, da Teilnehmer ihr Wissen mit ihren Mitarbeitern teilen. Mentoring von Kollegen, die mit dem Code nicht vertraut sind, ist ein wichtiger Vorteil von PP. Beide Partner profitieren vom PP, da sie voneinander lernen können [26], [30], [43].

5) Menschlicher Faktor: Die Einführung von Paar-Programmierpraktiken ist weder sofort umsetzbar noch einfach. Der Erfolg der Umsetzung dieser Praktiken basiert auf der positiven Erfahrung seitens der Programmierer. Bei überwiegend negativen Erfahrungen kann PP kaum erfolgreich in die Praxis umgesetzt werden. Im besten Fall wird sie nicht so effizient sein wie sie es sein könnte. Viele der in diesem Bereich gemachten Interviews zeigen, dass es häufig vorkommt, dass die Entwickler der Idee des PP eher skeptisch gegenüber sind. Auf der anderen Seite wächst für gewöhnlich die Zufriedenheit der Teilnehmer mit der Zeit – dank der sich einstellenden Erfolgserlebnisse. Darüber hinaus ermöglicht PP in der Regel ein Arbeiten in entspannter Umgebung, wodurch Stress, Langeweile und Frustrationen vermindert werden [30].

6) Projektmanagement: PP-basierte Projekte bergen weniger Risiken, dass sich der Verlust von erfahrenen Programmierern negativ auf ein Projekt auswirkt, da es mindestens zwei Personen gibt, die mit dem Zustand jeder Aufgabe und mit den erforderlichen Technologien vertraut sind. Dementsprechend wichtig ist es bei Ausfall eines Programmierers, seinen Partner nicht allein weiter arbeiten zu lassen. Stattdessen sollte ein anderer Programmierer der Aufgabe zugeordnet werden, um das Team wieder zu vervollständigen. Im gleichen Sinne wird auch die Rotation von Programmiererpaaren ermutigt, da diese Praxis, die Risiken von Programmiererabwanderungen deutlich verringert. Das rotierende Team profitiert von verbesserter Teambildung und effizienterem Lernen, da die Kommunikation und Kooperation gefördert wird. Die Erhöhung der Kommunikationsfrequenz verbessert die Effizienz und Geschwindigkeit des Teams indem es Lernprozesse beschleunigt und vereinfacht.

7) Zeitaufwand: Im Zusammenhang mit dem Zeitaufwand für die Entwicklung gibt es keine allgemeine Meinung ob PP vorteilhaft ist oder nicht. Während viele Studien darauf hinweisen, dass Paarprogrammierer eine Aufgabe in weniger Zeit als Solo-Programmierer bewältigen, wird in einigen Fälle das Gegenteil berichtet. Ein erhöhter Zeitaufwand scheint häufiger bei Teams mit unterschiedlichen Lernhintergründen und Expertisen vorzukommen [30].

Dagegen zeigen PP-Studien für gewöhnlich, dass Paar-Programmierer weniger Zeit mit Überprüfung und Testen der Codes verbringen, wohl aufgrund der geringeren Fehlerrate in den Codezeilen.

8) Besseres Design: PP resultiert in einer besseren Architektur und Umsetzung aufgrund der Einhaltung von gutem Design und Standards, was zu einer positiven Teamerfahrung führt. Um dieses Ziel zu erreichen wird Dissens nicht nur geduldet, sondern sogar ermutigt. Designskizzen und Arbeitsläufe werden von Anfang an kritisch hinterfragt. Entwürfe werden nur dann umgesetzt, wenn diese gut sind, oder sie werden verworfen und durch ein besseres Design ersetzt. Von besonderem Vorteil sind regelmäßige Code-Reviews und die Gewissheit, dass zwei Köpfe mehr leisten können als einer. Kreatives Brainstorming, solide Tests und effizientes Debugging der Software werden gefördert, was sich oft in einer verbesserten Arbeitsmoral wiederspiegelt [30], [43].

B. Nachteile von Pair-Programming

1) Kosteneffizienz: Zu den Nachteilen des PP zählen die erhöhten Kosten. Da PP doppelt so viele Entwickler als Solo-Programming erfordern, werden faktisch zwei Menschen bezahlt, um die Arbeit von einem zu machen. Damit sind die Kosten oft nur schwer zu rechtfertigen. Skeptiker zweifeln, ob die Lösung einer Aufgabe durch zwei Personen den Prinzipien guter Ressourcennutzung entspricht [43].

2) Zeitliche Abstimmung: Ein spezielles Problem der Teamarbeit ist die Zeitplanung, da die Partner ihre Zeitpläne eng aufeinander abstimmen müssen, um effektiv arbeiten zu

können. Die Abstimmung von zwei Terminkalendern kann schwierig sein, da PP die Auslastung der einzelnen Mitarbeitereee weiter erhöht [43].

3) Persönlichkeitskonflikte und Konsensfindung: Das meistzitierte Problem sind Persönlichkeitkonflikte, welche sich oft negativ auf die Produktivität auswirken. Missstimmungen resultieren potenziell auch in einer verringerten Produktqualität. Daher ist das Finden kompatibler Partnern ein entscheidender aber schwieriger Prozess. Grundsätzlich sollten Paar-Programmierer kompatible Persönlichkeiten, Wert-Systeme und Lebensstile haben. Viele Teams scheitern aufgrund von Persönlichkeitskonflikten, die auf mangelnde Konfliktlösungserfahrung, Egoismen und Geltungsdrang eines oder beider Teilnehmer zurückzuführen sind [43]. Eng damit verbunden sind Schwierigkeiten mancher Teams einen Konsens bei konkurrierenden Ideen zu finden [43].

4) Unterschiedliche Fähigkeiten: In einigen Fällen wurde von Entwicklern berichtet, die darüber besorgt waren mit einem Partner zusammenarbeiten zu müssen der weniger Erfahrungen, Kenntnisse und Fertigkeiten mitbrachte als sie selbst. Sie befürchteten einen Effektivitätsverlust, wenn Sie den unerfahreneren Mitarbeiter allgemeine Grundlagen erklären müssten. Von Ängsten wurde auch im Hinblick auf Unterschiede im Programmierstil sowie der Schwierigkeiten bei der Suche nach einem geeigneten Programmierpartner berichtet [26], [30], [43].

C. Ergebnisse Studien

1) Industriepraktiken Pair-Programming: Auch industrielle Teams haben ihre Erfahrungen mit der Nutzung des PP geteilt. Demnach zögern Praktiker oft mit dem Einstieg in PP. Die Entwickler benötigen oft mehrere Tage, um sich beim Wechsel von Solo-Programmierung zum PP [44] mit den neuen Praktiken und der Dynamik vertraut zu machen. Oft arbeiten Entwickler nicht den vollen Arbeitstag in Teams. Als eine angemessene Zeitspanne für die Zusammenarbeit werden zwischen 1,5 und 4 Stunden angesehen. Längere Paarprogrammiersitzungen können für Entwickler zur Herausforderung werden. PP ist aufgrund der höheren Geschwindigkeit mit der ein Team arbeiten kann sowie wegen des ständigen Fokus auf die jeweilige Aufgabe, mental sehr anstrengend [26], [30].

In Industrie-Teams ist die Paarrotation eine häufige Praxis, um die Paare dynamisch zu halten und Lernprozesse zu fördern. Viele Teams rotieren täglich manchmal sogar mehrere Male pro Tag. Häufige Paarrotation fördert nachweislich den Wissenstransfer zwischen Kollegen. Insbesondere unterstützt die Paarrotationen das Schulen und Training neuer Teammitglieder, wie Studien von Menlo Innovations, Microsoft, Motorola und Silver Platter Software empirisch belegt haben. Die Zusammensetzung der Programmiererpaare ist in der Regel zufällig und wird oft in einem kurzen Teammeeting entschieden [26].

Der Rollentausch zwischen Pilot und Navigator ist für den Motivationserhalt beider Teammitglieder wichtig. Auf der anderen Seite fanden Chong und Hurlbutt [45] in einer viermonatigen Studie bei zwei professionellen Software-Entwicklungsteams keine eindeutige Pilot / Navigator-Rollen. Stattdessen, engagierten sich Programmierer mit entsprechendem Fachwissen gemeinsam eean Brainstorming und Diskussionen. In diesem Fall repräsentierte der Pilot hauptsächlich die Rolle der Schreibkraft. Lediglich wenn die Programmierer unterschiedliche Kenntnisse hatten, dominierte der Software-Ingenieur mit mehr Fachwissen die Interaktion. Chong und Hurlbutt [45] stellten ebenfalls fest, dass die Kontrolle über die Tastatur eine raffinierte aber konsequente Wirkung auf die Entscheidungsfindung hatte, mit dem Piloten als endgültigen Entscheidungsträger. Sie beobachteten auch, dass die Tastatur wiederholt hin und her geschoben wurde und dass die Ingenieure am effektivsten waren, wenn sie gemeinsam die Rollen von Pilot und Navigator übernahmen. Die Entwickler zeigten ein höheres Engagement, wenn sie die Tastatur bedienten bzw. die Tastatursteuerung unmittelbar bevorstand. Infolgedessen empfehlen sie die Verwendung von Doppel-Tastaturen und -Mäusen.

Entwicklungsteams bei IBM und Guidant hatten die Wahl zwischen PP und der Inspektions- / Review-Methode. Dies hat die Verwendung von PP auf 5% bis 50% der Arbeitszeit bei IBM und auf fast 100% bei Guidant erhöht.

Häufig gaben Entwickler an, dass PP hauptsächlich für Spezifikation, Design und komplexere Programmieraufgaben geeignet ist. Nach Angaben von 295 Beratern gibt es mit PP Qualitätsverbesserungen bei der Lösung komplexer Aufgaben aber keine Qualitätsunterschiede im Vergleich zu Solo-Programmierung bei einfacheren Aufgaben [26].

Teams waren erfolgreicher, wenn sie einen strukturierten und organisierten Ansatz beim PP verfolgten, im Gegensatz zu Teams, welche ad hoc und unstrukturiert arbeiteten. Eine Umfrage zeigte eine positive Grundhaltung gegenüber PP sowie den Wunsch die Technik häufiger zu verwenden. Allerdings begründeten die befragten Entwickler die Nicht-Verwendung des PP mit logistischen Schwierigkeiten bei der Umsetzung, z.B. wegen eines Mangels an gemeinsamen Arbeitszeiten, unmotivierten Teamleitern und Nichtberücksichtigung des PP in den Projektplänen. Weiterhinsahen es Paarprogrammierer als vorteilhaft an, die Büros gezielt für das PP auszustatten, insbesondere mit großen Schreibtischen, großen Bildschirmen, drahtlosen Mäusen, Doppeltastaturen sowie Whiteboards an den Wänden [26].

2) Ergebnisse der Verwendung von Pair Programming in der Industrie: Viele PP-Teams berichten über eine verbesserte Produktqualität. Insbesondere eine große Telekommunikationsfirma in Finnland, deren Software-Ingenieure fast ausschließlich paarweise arbeiteten, hatten nur fünf Fehler in eineinhalb Jahren Produktentwicklung. Eine andere kontrollierte Fallstudie in Finnland demonstrierte die ähnlichen Fehlerraten für Paar- und Solo-Entwickler in einem Projekt und eine sechsmal niedrigere Fehlerrate für PP-Produkte in einem anderen Projekt [26].

Von einem Programmiererpaar geschriebener Code ist leichter verständlich, da der Code zunächst von einem Pilot geschrieben und anschließend vom Navigator überarbeitet wird, um dessen Kohärenz zu verbessern. Positiv wirkt sich aus, dass

der Pilot motiviert ist, den Code leicht verständlich zu halten, um häufige Rückfragen des Navigators zu vermeiden.

Teams finden Code, der von einem Paar geschrieben wurde leichter verständlich. Der Code wird von einem Pilot geschrieben und vom Navigator verständlicher gemacht. Der Pilot fühlt sich motiviert, den Code leicht verständlich zu halten, um häufige Rückfragen des Navigators zu vermeiden.

Teams berichten, dass die Verwendung von PP zu ihrer Arbeitsmoral beigetragen hat. Sie bestätigen, dass PP sich auch bei der Verwendung anderer Praktiken, wie z.B. testgetriebenen Entwicklung, Verwendung von Codierungsstandards und häufiger Integration positiv auf die Arbeitsdisziplin auswirkte.

Allerdings kann der Geräuschpegel eines diskutierenden Programmiererpaares andere Softwareentwickler bei der Arbeit stören. Ein Raum oder Bereich, der exklusiv für PP vorgesehen ist, kann dazu beitragen, dieses Problem zu lindern [26].

3) Praktiken für Bildung: Bei der Verwendung von PP in Bildungseinrichtungen bestimmt in der Regel das Lehrpersonal, wie man effektive Teams bildet. Die Lehrkräfte können den Studenten die Möglichkeit geben, ihre Partner frei zu wählen. Alternativ können die Lehrkräfte die am besten geeigneten Programmiererpaare proaktiv bilden. Eine Studie zeigt, dass heterogene Paare, die von einem Mann und einer Frau gebildet wurden, höhere Qualität und kreativere Lösungsansätze hervorbrachten als Teams, die nur aus Männern oder nur aus Frauen bestanden. Eine Studie mit 58 Bachelorstudenten zeigte, dass Teams am besten funktionierten, wenn sie sich die Teilnehmer ähnlich bewerteten, wenn sie nach der Offenheit und dem Verantwortungsgrad des jeweiligen Partners gefragt wurden. Williams et al. [25] befragten 1350 Studenten zu den Faktoren, welche das Lehrpersonal verwenden sollten, um proaktiv kompatibel Paare zu bilden. In 93% der Fälle berichteten die Studenten, mit ihren Partnern kompatibel zu sein. Aus den Ergebnissen lassen sich folgende Regeln zur Bildung hochkompatibler Paare ableiten [26]:

Teams sollten aus Studenten bestehen, die bezüglich ihrer Informatikkenntnisse und ihres Notendurchschnitts ein ähnliches Niveau an Kenntnissen und Fertigkeiten haben. Darüber hinaus sollten Studentenpaare aus einem Myers-Briggs-Sensor und einem Myers-Briggs-Intuitor geformt werden, die eine ähnliche Arbeitsmoral haben. Dafür werden die Studenten auf einer Skala von 1 bis 9 bewertet. Studenten, die sich nur so viel anstrengen, um den Kurs gerade noch zu bestehen werden der Kategorie 1 zugewiesen, während Studenten, die sich um die bestmögliche Note bemühen in die Kategorie 9 fallen. Die Paarrotation unter Studenten erfolgt weniger häufig als in der Industrie. Meistens bleiben die Paare für die Dauer einer Aufgabe erhalten, d.h. in der Regel für ein bis drei Wochen. Einige Lehrkräfte bevorzugen Studententeams während desganzen Semester beizubehalten [26].

4) Ergebnisse der Verwendung von Pair-Programming in Bildungsveranstaltungen: Verschiedene Studien [46]–[48] haben gezeigt, dass PP ein fortgeschrittenes Lernumfeld schafft, in dem aktives Lernen und soziale Interaktion gefördert wird. Studenten fühlen sich seltener frustriert, haben mehr Selbstvertrauen und mehr Interesse an der Lösung selbst komplexer Aufgaben. Die PP-Vorteile stehen im Kontrast zu den negativen Erfahrungen, die Studenten mit dem traditionellen Solo-Programmier-Ansatz erfahren. Bei letzterem fühlen sich die Studierenden oft isoliert, frustriert und unsicher in Bezug auf ihre Fähigkeiten. PP ermutigt die Studenten, mit Gleichgesinnten in ihrem Umfeld zu interagieren und dadurch eine gemeinsame und positive Lernatmosphäre zu schaffen. Studierende der aktuellen Millenniums-Generation legen besonderen Wert auf kollaborative Umgebungen [26].

Darüber hinaus stärkt die intensive Zusammenarbeit während des PP die Team- und Kommunikationsfähigkeit der Student, welche insbesondere in der Industrie erforderlich sind. Im Fachbereich Informatik helfen diese Vorteile insbesondere den Frauen ihr Studium fortzusetzen. Im Allgemeinen spiegelt der PP-Ansatz das „Labormodell“, welches in Naturwissenschaften wie Chemie oder Physik üblich ist.

Studierende, die zu zweit arbeiten, haben eine bessere Chance Projekte höherer Qualität zu produzieren. Weiterhin haben sie höhere Kursdurchlaufraten, selbst wenn das Projekt in einer verteilten Weise durchgeführt wird.

Leider gibt es auch zwei klare Nachteile der PP-Implementierung. Eine kleine Gruppe der Studenten (ca. 5%) wird sich wohl immer der Gruppenarbeit verweigern. Oft sind dies Top-Studenten, die sich von der Teamarbeit keine Vorteile erhoffen und sich nicht von anderen Studenten bremsen lassen wollen. Ein anderes Problem der Studenten ist die Notwendigkeit, Zeitpläne koordinieren zu müssen, wenn PP außerhalb eines regulären Kurses oder Labors erforderlich ist [26].

Durch die Nutzung von PP wird die Anzahl der Codezeilen deutlich verringert. Quantitative Studien zeigten auch, dass Studentenpaare Code mit höherer Qualität in rund 20% weniger Zeilen produziert haben als Solo-Entwickler. Außerdem zeigten die Ergebnisse, dass PP-Code, weniger Fehler hat sowie lesbarer und besser kommentiert ist [30]

5) Produktivität der Zeit: Unter Produktivität wird hier die höchste Qualität innerhalb der minimalen Zeit verstanden. Um die PP-Produktivität zu messen führten Kim und Keith [31] den Begriff „Relative Effort Afforded by Pairs“ (REAP) ein, der mit der Produktivität von Solo-Programmierung direkt verglichen werden kann. Die Formel 2 zeigt die Berechnung von REAP, wobei E die verstrichene Zeit („elapsed time“) repräsentiert, die für die Entwicklung von einem Programmierer-Paar (Pair) bzw. von einem Solo-Programmierer aufgewendet wurde.

$$REAP = \frac{E_{pair} * 2 - E_{individual}}{E_{individual}} * 100\% \qquad (2)$$

Wenn REAP Null ist, halbiert PP die für die Solo-Programmierung benötigte Zeit. Wenn REAP größer Null aber weniger als 100% ist, benötigen Paare mehr Gesamtstunden, aber erledigen die Aufgaben schneller. Dies kann vorteilhaft sein, wenn die Zeit bis zur Markteinführung entscheidend für den Erfolg eines Produkts ist. Durch rasche Markteinführung könnte sich das Pionierunternehmen einen größeren Maerktan-

teil sichern. Damit könnten sich höhere Kosten für eine kürzere Entwicklungszeit auszahlen.

Wenn rasche Markteinführung weniger wichtig ist, stellt sich die Frage, warum Firmen zwei Programmierer beschäftigen sollten, wenn die gleiche Arbeit auch von einem getan werden kann? In diesem Sinne stellt sich auch die Frage, warum erfahrene Programmierer beschäftigt werden sollten, wenn auch unerfahrene Universitätsabsolventen die Arbeit erledigen können? [31]

Experimente haben gezeigt, dass die Qualität der Software die von Paaren entwickelt wurde, in mehr als 80% der Testfälle einen REAP von etwa 15% aufwies [28]. Grundsätzlich wird PP empfohlen, wenn die Qualität einen Schwellenwert von 80% erreicht. Wird die erwartete Softwarequalität in nur 70% der Testfälle erreicht, dann wird PP unwirtschaftlich und sollte nicht angewendet werden [31].

6) Wirtschaftlichkeit: Die finanziellen Kosten des PP sind ein entscheidender Aspekte bei der Auswahl der geignesten Entwicklungsmethode. Ist eine Methode zu teuer, dann wird sie sehr wahrscheinlich nicht implementiert. Oft sehen es Projektmanager als Verschwendung von humanen Ressourcen an, wenn zwei Personen an der gleichen Aufgabe arbeiten. Dies gilt umso mehr, wenn auch noch die Arbeitskosten verdoppelt werden.

Da die Wirtschaftlichkeit eines Software-Projekts eng mit dem durch die Programmierung entstehenden Zeitaufwand verbunden ist, können die Kosten für PP in der Tat nahezu doppelt so hoch sein wie für die Solo-Entwicklung, trotz der potentiellen Reduktion der Gesamtprogrammierzeit. Dies liegt darin begründet, dass die Gesamtentwicklungszeit der Summe aller für das Programmieren aufgewendeten Zeit entspricht. Bei der Berechnung der Gesamtentwicklungszeit müssen aber die Prüf- und Reparaturkosten berücksichtig werden, da diese in der Regel viel größer sind als als die eigentlichen Programmierkosten.

In der Software-Entwicklungsbranche zeigten Berichte von IBM aus dem Jahr 1995 durchschnittliche Kosten von über 8000 USD für die Behebung der von Kunden gemeldeten Programmierfehler. Fallstudien belegen, dass nicht nur die Fehlerrate bei Verwendung der PP-Methodik abnehmen, sondern auch die damit verbundenen Test- und Korrekturarbeiten, weil der Code oft besser entwickelt und verständlicher ist als dies bei Solo-Programmierung der Fall wäre. Daher rechtfertigen die wirtschaftlichen Vorteile im Allgemeinen die Verwendung des PP-Ansatzes [30].

Unterschiede zwischen erfahrenen und unerfahrenen Entwicklerpaaren: Unterschiede in der Programmiererfahrung sollten sich auf den Zeitaufwand und die Softwarequalität auswirken. Bei einem Anfänger dauert das Programmieren bei gleicher Aufgabe durchschnittlich länger als bei einem erfahrenen Programmierer. Sobald der Anfänger einige Erfahrungen gesammelt hat, kann er besser und schneller schreiben und sich auch komplexeren Aufgaben zuwenden. Deshalb haben Kim und Keith [31] zwei Gruppe von Programmiererpaaren verglichen: Anfänger – Anfänger und Experte – Experte.

Beim Paar Anfänger – Anfänger arbeiten zwei unerfahrene Programmierer zusammen. Als Hypothese wird vermutet, dass die absolute Programmierzeit reduziert wird und die Aufgabe schneller durchgeführt werden kann als von einem Solo-Programmierer [31].

Beim Paar Experte – Experte arbeiten zwei erfahrende Entwickler zusammen. Hypothetisch sollte dieses Paar präziser und schneller arbeiten als das Anfängerpaar.

Grundsätze für Anfänger-Anfänger-Paare vs. Experten-Experte Paare: Lui und Chan [31] beschreiben zwei Grundsätze von PP:

i Ein Paar ist deutlich produktiver und erreicht höhere Qualität und besseres Fehlermanagement in weniger Zeit als zwei individuell arbeitende Einzelpersonen. Wenn das Programmiererpaar unerfahren ist, dann erhöht sich im Vergleich zu erfahreneren Paaren der Aufwand für das Design sowie das Schreiben des Codes und der Algorithmen [31].

ii Der Produktivitätsgewinn von PP im Vergleich zur Soloprogrammierung kann deutlich geringer ausfallen, wenn ein Paar bereits vorherige Erfahrung mit einer ähnlichen Aufgabe sammeln konnte und diese Erfahrung noch nicht vergessen hat [31].

Der Grundsatz (i) bedeutet, dass PP gut funktioniert, wenn ein Team anspruchsvolle Programmieraufgaben zu lösen hat. Damit sind Aufgaben gemeint die nur mit Hilfe komplexer Algorithmen gelöst werden können und nicht Programmierkenntnissen in einer bestimmten Computersprache voraussetzen.

Der Grundsatz (ii) bezieht sich nicht auf Änderungen der Softwarequalität. Er bezieht sich auf die Tatsache, dass Solo-Programmierung produktiver als PP sein kann, wenn Programmierer an Lösungen arbeiten, die sie bereits kennen. Sobald die Teammitglieder eine Programmierlösung gut genug kennen, um diese alleine auszuführen, dann ist es oft effektiver, wenn der Pilot das Schreiben nicht unterbricht, außer für Korrekturen kleinerer (Tipp-)Fehler. Gleichzeitig könnte sich sein Partner, der Navigator, unterfordert fühlen, weil er vom Piloten vor allem bereits bekannte Lösung vorgelegt bekommt.

Wenn beide Grundsätze kombiniert werden, wird deutlich, dass der Produktivitätsgewinn in Bezug auf Zeitaufwand und Softwarequalität bei Anfänger – Anfänger-Paaren im Vergleich zu Solo-Anfänger größer ausfällt als bei Experten – Experten-Paaren im Vergleich zu Solo-Experten [31].

Anfänger - Anfänger vs. Anfänger - Experte: Alshehri und Benedicenti [29] haben PP-Paare unterschiedlichen Gruppen zugeordnet, das Ergebnisse ist in Tabelle V dargestellt.

a) Anfänger–Anfänger: Anfänger-Programmiererpaaren sollten relativ einfache Codes zugeteilt werden. Dies ermöglicht eine gute Lernerfahrung für die Programmierer, vor allem wenn jeder Vorkenntnisse aus einem anderen Spezialgebiet mitbringt und diese mit dem Teammitglied teilt. Bei dieser Methode wird aber ein Betreuer benötigt [29].

b) Expert – Anfänger: Das Paar Experte – Anfänger ist die beste Wahl, um einen unerfahrenen Programmierer in ein neues Entwicklungsteam oder in ein ungewohntes Projekt

Tabelle V
KLASSIFIZIERUNG DER PAIR-PROGRAMMING PAARE

Gruppe	Lernen / Wissenstransfer	Einfache Aufgabe	Komplexe Aufgabe	Geschwindigkeit	Codequalität
Anfänger-Anfänger	+	+ +	- -	- - -	0
Fortgeschrittener-Anfänger	+ +	+	-	- -	+
Experte-Anfänger	+ + +	-	-	-	+ +
Fortgeschrittener-Fortgeschrittener	+	-	+	+	+ +
Fortgeschrittener-Experte	+ +	- -	+ +	+ +	+ +
Experte-Experte	+	- - -	+ + +	+ + +	+ + +

Legende:

- leichter Nachteil
- - großer Nachteil
- - - sehr großen Nachteil
0 neutral
+ leichter Vorteil
+ + großer Vorteil
+ + + sehr großen Vorteil

einzuführen, da dieser Ansatz eine gute und ständige Führung durch den Expertenprogrammierer ermöglicht. Der Anfänger kann auch dann angeleitet werden, wenn er in der Rolle des Navigators ist. Einer der wichtigsten Vorteile des PP ist die Zeitersparnis für das Einarbeiten neuer Mitarbeiter [29].

Kritiker könnten diesen Ansatz als Verschwendung von Expertenzeit ansehen. Auf der anderen Seite ist es vorteilhaft, wenn Anfänger die Aufgabe haben, die durch den Experten gemachten Fehler zu finden, da er so gezwungen wird im Zweifelsfall den Piloten nach potentiellen Programmierfehlern zu fragen. Dabei eröffnet der Anfänger unter Umständen auch eine andere Perspektive auf die Arbeit des Experten. Dies setzt voraus, dass der Experte geduldig genug ist dem Anfänger den Code genauer zu erklären. Andernfalls können die Vorteile dieser Paarkombination nicht zum Tragen kommen.

Auch ein übermäßig ausgeprägtes Ego des Experten-Programmierers ist problematisch, da er dem Anfänger das Gefühl geben könnte, für den Erfolg des Teams unbedeutend zu sein. In diesem Fall wird der Anfänger oft vom Experten ignoriert, was eine schlechte Arbeitsatmosphäre schafft. Auf der anderen Seite kann der Anfänger durch Missachtung der Schweigepflicht das Vertrauen des Experten verlieren. In einer solchen Situation kann die Zusammenarbeit selbst dann leiden, wenn die Partner sich wieder einer Solo-Aufgabe widmen.

c) Experte-Fortgeschrittener: Wenn ein Experte mit einem bereits fortgeschrittenen Entwickler ein Paar bildet, kann der fortgeschrittene Entwickler unter Umständen seine Fähigkeiten weiter verbessern. Voraussetzung ist, dass er sein Wissen erweitern möchte und gut mit dem Experten interagiert. Andernfalls kann die Situation leicht zu Spannungen oder Konflikten führen und die Effizienz des Paares gefährden [29].

d) Experte – Experte: Ein Experten-Programmiererpaar wird dann als die beste Wahl angesehen, wenn ein Team mit sehr komplexen Aufgaben bzw. der Lösung von kritischen Fehlfunktionen konfrontiert wird [29].

Die Entwicklungszeit ist in der Regel deutlich kürzer als dies mit anderen Paarkombinationen der Fall wäre, da die Programmierer sich ganz auf Ihre Aufgaben konzentrieren können und weniger Zeit für Erklärungen aufwenden müssen. Es ist auch weniger wahrscheinlich, dass ein Experten-Programmiererpaar, von Problemen blockiert wird. Eine gute Zusammenarbeit von beiden Programmierern ist auch hier entscheidend, da egoistisches Verhalten die größte Bedrohung für Experten–Paar ist.

Vorgeschlagene Kriterien für die Auswahl optimaler Paare: Um geeignete Paare zu bilden, werden bestimmte Kriterien definiert und angewendet, welche das Zusammenstellen der besten Paarkombinationen unterstützen [29].

Dabei werden die Kriterien auf Grundlage des Projektziels unterschiedlich gewichtet. Schließlich werden die Kriterien verwendet, um die Eignung der potentiellen Paare objektiv zu bewerten. Die Autoren unterscheiden zwischen:

a. Geschwindigkeit: Paare mit der höchsten Wahrscheinlichkeit, den Codierungsvorgang zu beschleunigen.
b. Austausch von Wissen: Paare mit der höchsten Wahrscheinlichkeit, Wissen auszutauschen.
c. Codequalität: Paare mit der besten Wahrscheinlichkeit, die Codequalität zu verbessern.
d. Lernen: Paare mit der höchsten Wahrscheinlichkeit, eine positive Trainings- und Lernatmosphäre zu schaffen [29].

Die Autoren [29] haben die Bedeutung dieser Kriterien in drei Industrieunternehmen untersucht, hier A, B und C genannt. Ziel war es, die Kriterien auf Grundlage der Expertenmeinung zu gewichten:

a. Die drei Unternehmen bewerteten das Team Experte – Experte am höchsten, wegen der erreichten Geschwindigkeit und Codequalität. Sie gaben an, dass gemeinsame soziale Aktivitäten der Teammitglieder, die Beziehung im Team stärken und sich positiv auf die Effektivität auswirken können.
b. in Bezug auf Wissenstransfer bewerteten B und C das Team Experte – Anfänger am höchsten, während A das Paar Experte – Experte bevorzugte.

c. Die Codequalität wurde von Unternehmen B und C als das wichtigste Kriterium angesehen, während von A Wissenstransfer eher als Risiko wahrgenommen wurde.
d. Hinsichtlich der Lernmethode beurteilten A, B und C das Team Experte – Anfänger am positivsten.

Richtlinien des Pair-Programmings: Im Folgenden werden einige Grundlagen von PP beschrieben, welche alle Pair-Programmierer und vor allem Berufseinsteiger berücksichtigen sollten, um die Methode möglichst effizient anwenden zu können.

e) Alles teilen: Beim PP werden zwei Programmierern gemeinsam mit der Entwicklung eines Programms (Design, Algorithmus, Code etc.) beauftragt. Die beiden Programmierer sind für jeden Aspekt des Programms verantwortlich. Eine Person, der Pilot, tippt oder schreibt, während die andere Person, der Navigator, die Arbeit des Piloten laufend überprüft. Beide sind aber grundsätzlich gleichberechtigt. Es sollte unbedingt vermieden werden, Dinge zu sagen oder zu denken, wie zum Beispiel: „DU hast einen Fehler in DEINEM Design gemacht“ oder „Das Problem ist in DEINEM Teil aufgetreten.“ Stattdessen ist die Devise „WIR haben das Design verpatzt“ oder besser noch „WIR finden keine Fehler im Code.“ Beide Partner sind für jeden Aspekt des Projekts verantwortlich [49].

f) Fair spielen: Beim PP hat der Pilot die Kontrolle über die Tastatur bzw. den Entwurf der Designideen, während der Navigator zeitgleich die Arbeit überprüft. Auch wenn einer der beiden Programmierer deutlich erfahrener ist als der andere, ist es wichtig die Aufgabe des Piloten zwischen den Partnern zu alternieren, damit jeder die Perspektive beider Rollen kennenlernen kann. Diese Strategie ermöglicht es sich in die Rolle des Partners hinein zu versetzen und Lernprozesse zu optimieren. Die Person, die keinen Code schreibt sollte niemals nur passiver Beobachter sein, sondern immer aktiv und engagiert die Funktionalität der Software prüfen. In einer Umfrage gaben etwa 90% der PP-erfahrenen Programmierer an, dass die wichtigste Aufgabe des Navigators in einer kontinuierlichen Analyse und Bewertung des Designs und Codes besteht. Während einer der Partner mit dem Design bzw. der Entwicklung des Codes beschäftigt ist, beschäftigt sich der andere mit der strategischen Ebene der Aufgabe; d.h. mit Fragen wie „Wo sind die Entwicklungslinien im Code? Könnte die bisherige Strategie in eine Sackgasse führen? Gibt es womöglich eine effizientere Gesamtstrategie? [49]

g) Verletzen Sie Ihren Partner nicht: Dennoch muss sichergestellt sein, dass der Partner fokussiert und On-Task bleibt. Zweifellos ist ein wichtiger Vorteil des PP, dass es weit weniger wahrscheinlich ist, dass Zeit mit Email-lesen oder mit Surfen im Internet vergeudet wird, weil die Partner auf den kontinuierlichen Beitrag des jeweils anderen angewiesen sind und auf dessen Input warten. Darüber hinaus erwarten beide vom Partner, dass dieser den etablierten Entwicklungspraktiken folgt. Zusammenfassend lässt sich festhalten, dass die Entwicklung der Software effizienter ist als bei der klassischen (Solo-)Programmierung, weil ein gewisses Tempo von der anderen Person vorgegeben wird. Da die Dynamik des PP-Prozesses beide Partner dazu drängt, auf die technische Aufgaben fokussiert und konzentriert zu bleiben, werden enorme Produktivitätsgewinne und Qualitätsverbesserungen erzielt [49].

h) Kontrollieren Sie negative Gedanken: Die Wahrnehmung ist eine heikle Angelegenheit im menschlichen Dasein. Die gilt natürlich auch für die Arbeitswelt. Wenn du lange genug an etwas glaubst, dann wird dein Gehirn es als wahr betrachten. Wenn du dir etwas Negatives einredest, wie „Ich bin ein schlechter Programmierer“, dann wirst Du schließlich fest davon überzeugt sein, dass dies wirklich zutrifft. Daher ist es so wichtig negativen Gedanken zu kontrollieren. Eine Strategie ist negative Gedanken zu unterdrücken sobald diese einsetzen. Eine andere Strategie ist negative Gedanken mit positivem Denken „zu überschreiben“, was nicht in jeder Situation gelingt. Daher betonen befragte Paarprogrammierer, wie schwierig es sei mit jemandem zusammenzuarbeiten, der bzgl. seiner Programmierkenntnisse unsicher ist. Solche Personen befürchten, dass den Partner ihre Schwächen herausfinden und gegen sie verwenden könnten. Bei verunsicherten Programmierern sollte PP als Mittel zur Verbesserung ihrer Fähigkeiten eingesetzt werden, z.B. indem sie durch ständiges Beobachten und das Feedback von einem erfahreneren Programmierer lernen.

Einer der Befragten gab an, dass das Beste an PP die kontinuierliche Diskussion über Design und Programmierstrategien war, was ihn zu einem besseren Softwareentwickler gemacht hätte. In diesem Sinne befragten zwei Forscher 750 Programmierer über Kommunikationsstrategien in der Softwareentwicklung (Kraut 1995). Die Kommunikationstechniken mit der höchsten Nutzung und der höchsten Bewertung waren: „Hilfsbereitschaft und Bescheidenheit“. Bei Auftreten eines Problems, welches nicht alleine gelöst werden kann, wendet man sich ist sich an einen Kollegen (Kraut 1995). Beim PP steht der „Kollege in der Nähe“ immer zur Verfügung. Zusammen kann das Team Probleme lösen, die kaum alleine gelöst werden könnten.

Negative Gedanken wie „Ich bin ein genialer Programmierer, muss aber mit einem totalen Versager zusammenarbeiten“ sollten nicht zugelassen werden, damit sie nicht das Arbeitsklima belasten. Kein Mensch, egal wie geschickt er ist, ist unfehlbar und sollte sich nicht über andere erhaben fühlen. John von Neumann, der geniale Mathematiker und Schöpfer der Neumann Computerarchitektur, erkannte seine eigenen Unzulänglichkeiten und forderte fortwährend andere auf, seine Arbeit zu überprüfen. [49].

i) Nehmen Sie die Dinge nicht zu ernst: Egoismus-freie Programmierung ist für ein effektives PP unerlässlich, wie bereits vor einem Vierteljahrhundert von Weinberg (1998) in der „Psychologie der Computer-Programmierung“ formuliert. Gemäß der Umfragen zum PP, kann sich ein übermäßiges Ego auf zwei Arten manifestieren, die sich beide negativ auf die Zusammenarbeit auswirken. Erstens, die Haltung „mein Weg oder keiner“ kann andere Ideen unterdrücken und damit verhindern, dass alternative Lösungsansätze berücksichtigt werden. Zweitens, kann ein zu starkes Ego eines Programmierers dazu führen, dass der Partner eine defensive Haltung

einnimmt, was der Effizienz schadet.

Umgekehrt kann eine Person, die nicht immer mit ihrem Teampartner einverstanden ist, auch vorteilhaft für die Zusammenarbeit sein. Für einen effektiven Ideenaustausch sind gesunde Meinungsverschiedenheiten wichtig. Daher sollte es eine ausgewogene Balance zwischen zu viel und zu wenig Ego geben. Effektive Paarprogrammierer entwickeln diese Balance während einer Einarbeitungsphase. Laut Cunningham, einem der XP-Gründer und erfahrenen Paarprogrammierer, dauert diese Einarbeitungsphase Stunden oder Tage, je nach den Charaktereigenschaften und der Vorerfahrung der Personen mit PP [49].

j) Einrichtung des Arbeitsplatzes: In der Umfrage gaben 96% der Programmierer an, dass ein angemessener Arbeitsplatzaufbau für den Erfolg der Projekte ganz entscheidend war. Die Programmierer müssen in der Lage sein, nebeneinander zu sitzen und gleichzeitig zu programmieren, den Computerbildschirm zu betrachten und die Tastatur und die Maus zu bedienen [49].

Eine effektive Kommunikation ist von herausragender Bedeutung, sowohl innerhalb des zusammenarbeitenden Paares als auch mit anderen Programmiererpaaren. Die Programmierer müssen einander ohne viel Aufwand sehen, sich gegenseitig Fragen stellen und Probleme effektiv lösen können. Eine Situation endloser Diskussionen sollte aber vermieden werden. Programmierer profitieren durchaus auch vom Informationsaustauch durch „versehentliches“ Mithören von Gesprächen anderer Programmiererpaare, durch die sie wichtige alternative Problemlösungsansätze lernen können. Separate Büros und Arbeitskabinen können diesen notwendigen Wissensaustausch hemmen [49].

k) Legen Sie die Skepsis ab, bevor Sie beginnen: Viele Programmierer nehmen bei ihrer ersten PP-Aufgabe eine skeptisch Haltung hinsichtlich des Wertes der Zusammenarbeit ein, insbesondere weil sie nicht erwarten von der Erfahrung profitieren zu können. Treffen zwei skeptische Programmierer aufeinander, besteht das Risiko einer sich selbstverwirklichenden Prophezeiung. In einer Programmiererumfrage stimmten 91% zu, dass ein „sich auf den Partner einlassen“ ganz entscheidend für eine erfolgreiche Programmierung war [49].

PP-Beziehungen können informell von einem Programmierer gestartet werden, indem er einen anderen bittet sich neben ihn zu setzen, um ihm beim Lösen einer Aufgabe zu helfen. Ist diese anfängliche Erfahrung positiv, dann bietet es sich an die PP fortzusetzen. Die Erfahrung zeigt, dass mit nur einer als positiv erlebten PP-Erfahrung, das Paar zu einem erfolgreichen Team werden kann [49].

l) Unabhängige von anderen geschriebenen Code unbedingt überprüfen: Es ist unvermeidlich, dass Programmierer ab und an unabhängig von anderen Aufgabe bearbeiten. Von den befragten Programmierern, gab über die Hälfte an, dass sie unabhängig entwickelten Code erst dann in ein Projekt integrierten, nach dem dieser von ihrem Programmierpartner überprüft wurden war. Die Mehrheit der Fehler, die bei der Anwendung der XP-Methode gefunden wurden, ließe sich auf eine Phase zurückverfolgen, in der ein Programmierer selbstständig arbeitete.

Die Entscheidung, die Arbeit selbst zu erledigen und zu überprüfen, kann von einem Programmierer getroffen werden, oder die Wahl kann aktiv gefördert werden, wie es bei der Extremen Programmierung der Fall ist. Allerdings ist es wichtig zu beachten, dass keiner der befragten Programmierer die Arbeit selbständig übernommen hat, ohne sie nochmals zu überprüfen [49].

m) Ein ausgewogenes Leben erhält die Produktivität im Arbeitsalltag: Die regelmäßige Kommunikation mit anderen ist der Schlüssel für ein ausgewogenes Leben. „Die meisten Programmierer würden vermutlich sagen, dass sie es vorziehen an einem Ort zu arbeiten, wo sie nicht von anderen Menschen gestört werden“ (Weinberg 1998). Es sollte aber berücksichtigt werden, dass informelle Gespräche mit anderen Programmierern einen effektiven Ideenaustausch und einen effizienten Transfer von Informationen ermöglichen [49]. Ein Einzelkämpferdasein kann daher schnell in eine intellektuelle Sackgasse führen.

n) Legen Sie regelmäßige Pausen ein: Paarprogrammierer motivieren sich gegenseitig auf die Arbeit fokussiert zu bleiben. Die kontinuierliche Konzentration kann sehr intensive und mental anstrengend sein. Daher sind Pausen in regelmäßigen Abständen sehr wichtig. Nur so kann die Ausdauer für eine weitere produktive PP-Runde aufrechterhalten werden. Während der Pausen ist es am besten ganz von dem Programmieraufgabe abzulassen damit ein frischer Neustart möglich ist. Empfohlene Pausenaktivitäten umfassen: E-Mails checken, telefonieren, im Internet surfen, einen Snack essen und etwas trinken [49].

o) Zusammenhalt ist der Schlüssel zum Erfolg: Beim PP sollte der Verstand der beiden Programmierer verschmelzen. Es sollte keine Konkurrenz zwischen den beiden geben. Beide müssen auf dasselbe Ziel hin arbeiten, so als ob das Endprodukt von einem einzigen Verstand produziert würde. Probleme oder Fehler sollten niemals auf einen der Partner geschoben werden. Paarprogrammierer müssen sich auf die Einschätzung und die Loyalität des anderen verlassen können.

p) Zwei Gehirne sind leistungsfähiger als eins: Das Erinnerungsvermögen und die Lernfähigkeit des Menschen sind begrenzt. Um diese Einschränkungen überwinden zu können ist es wichtig mit anderen zusammenzuarbeiten. Wenn zwei Programmierer zusammenarbeiten, bringt jeder seine eigenen Kenntnisse und Fähigkeiten mit. Eine gewisse Menge an Wissen und Fähigkeiten werden beide teilen, so dass sie effektiv interagieren können. Die einzigartigen Fähigkeiten jedes einzelnen werden es ihnen jedoch ermöglichen, selbst komplexe Aufgaben effektiv zu lösen.

Die Erfahrung zeigt, dass Paarprogrammierer gemeinsam mehr als doppelt so viele Lösungen finden, als wenn die beiden alleine arbeiten würden. Zusammen finden sie schneller die „beste“ Lösung und setzen diese mit höherer Qualität um. Ein Umfrageteilnehmer sagte: „Es ist eine mächtige Technik, da sich zwei Gehirne auf das gleiche Problem konzentrieren.

Es zwingt einen sich voll und ganz auf das Problem zu konzentrieren“ [49].

7) Herausforderungen für Berufseinsteiger: Nachfolgend werden einige Herausforderungen beschrieben, mit denen PP-Anfänger konfrontiert werden.

a) Kommunikationsdefizite: Problematisch ist, wenn es keinen effektiven Ideenaustausch bzw. Wissenstransfer gibt.

b) Kenntnisunterschiede: Erfahrene Programmierer können auf umfangreiche Gedächtnisinformationen bzgl. alternativer Lösungsansätze zurückgreifen. Anfänger hingegen können für gewöhnlich nicht auf vorheriges Wissen zurückgreifen, sondern müssen sich Lösungsansätze erst herleiten bzw. von anderen erlernen. Darüber hinaus nutzen Experten auch übergeordnetes Wissen, um neue Probleme zu verstehen und zu lösen, während Anfänger dazu neigen, sich auf die spezifischen Lösungen zu konzentrieren, die laut Lehrbuch normalerweise für ein bestimmtes Problem verwendet werden [50].

c) Einarbeitung: Die Phase der Einarbeitung kann stressig sein. Anfänger müssen sich oft mehr Stunden mit PP beschäftigen als seine Kollegen, um neue Techniken und fremden Code zu lernen.

Die oben genannten Herausforderungen können von Anfängern kurzfristig überwunden werden, weil

i einige Berufseinsteiger während des Studiums Erfahrungen mit der Nutzung von PP gemacht haben;
ii Studenten durch den Einsatz von PP ihre Zusammenarbeit sowie Team-und Kommunikationsfähigkeit deutlich verbessern können, was beim Berufseinstieg sehr hilfreich ist.

8) Einarbeitungsphasen für Berufsanfänger: Laut Fronza, Sillitti und Succi [51] sollten Anfänger während der Einarbeitung in ein Team vier Phasen durchlaufen:

i Einführung: Während des ersten Monats seiner Tätigkeit wird ein Anfänger sich die meiste Zeit mit PP beschäftigt, ungefähr 47% seiner Zeit, von der er über 70% mit bereits etablierten Teammitgliedern verbringen sollte. Diese Startphase stellt eine wichtige Phase der Wissensvermittlung dar.
ii Unabhängigkeit: In den darauffolgenden zwei Monaten wird vom Anfänger weniger PP (zwischen 3 und 5% der Zeit) geübt. Davon sollte er die Hälfte der Zeit mit Experten arbeitet. In dieser Zeit versucht der Einsteiger eine gewisse Unabhängigkeit und Selbstständigkeit im Team zu erlangen.
iii Reife: Die dritte Phase findet etwa in den Monaten vier bis acht statt. Der Anfänger beschäftigt sich wieder mehr als 50% mit PP. In dieser PP-Zeit wird er überwiegend ohne Anleitung von Experten arbeiten und seine Eigenständigkeit innerhalb des Teams festigen.
iv Integration: Schließlich beginnen neue PP-Teammitglieder regelmäßig mit den erfahreneren Teammitgliedern zusammen zu arbeiten. In dieser letzten Phase gibt es keine signifikanten Unterschiede mehr zwischen den Anfänger / Experten-Paaren und den PP-Paaren, die nur aus Experten bestehen.

[i]

9) Potenziale für Berufseinsteiger: Studien belegen wie PP viele Fähigkeiten eines Anfängers unterstützen und verbessern kann. Nachfolgend werden die wichtigsten Potenziale von PP für Berufsanfänger zusammengefasst.

i Teamfähigkeit: PP fördert die Gruppenarbeit. Der Einsteiger lernt mit unterschiedlichen Programmierergruppen zu arbeiten und Kontakte zu knüpfen. Dadurch wird Teamfähigkeit trainiert und weiterentwickelt.
ii Kommunikationsfähigkeit: Die regelmäßige Zusammenarbeit im Team verstärkt die Kommunikationsfähigkeit. Probleme werden gemeinsam analysiert und Lösungsansätze von anderen erlernt. Eine besondere Rolle in diesem Zusammenhang spielt die Paarrotation, bei der Einsteiger sich immer wieder auf neue Teamkollegen einstellen müssen.
iii Einarbeitung: PP bietet dem Einsteiger die exzellente Möglichkeit von den Programmierpartnern neue Techniken und alternative Codes innerhalb kürzester Zeit zu lernen.
iv Verringerte Fehlerrate: Neueinsteiger lernen, dass Programmierfehler aufgrund der ständigen Überprüfung direkt entdeckt und behoben werden. Dadurch verringert sich der Aufwand für separate Codereviews und steigt die Codequalität erheblich.
v Verantwortung: Das PP-Team entwickelt ein gemeineinsames Verantwortungsbewusstsein für ein Projekt. Der Berufsanfänger lernt rasch, dass nicht Individuen für den Erfolg oder Misserfolg verantwortlich sind, sondern immer das Team gemeinsam. Große und kleine Entscheidungen werden zusammen getroffen und verantwortet.

Zusammenfassend lässt sich feststellen, dass PP das Lernen unterstützt, was insbesondere Berufsanfängern zugutekommt, indem sie sich rasch einarbeiten können. Bei Fragen steht immer ein erfahreneres Teammitglied zur Verfügung. Daher macht das Entwickeln in einer Gruppe dem Anfänger mehr Spaß, als wenn er Aufgabe alleine bearbeiten würde PP ermöglicht einen optimalen Wissenstransfer zwischen neuen und erfahrenen Mitarbeitern und verhindert so die Konzentration von Spezialwissen auf nur wenige Personen. Alle Teammitglieder lernen den Code zu verstehen und beim Ausfall eines Mitarbeiters kann der verbleibende Partner die Aufgabe ohne große Verzögerungen weiterentwickeln.

VI. Kombination beider Praktiken

In diesem Abschnitt wird die Kombination von TDD und PP untersucht. Der Abschnitt besteht aus zwei Teilen. Im ersten Teil werden verschiedene Studien vorgestellt, die in verschiedenen Konstellationen die testgetriebene Entwicklung und das Pair-Programming untersuchen. Im darauffolgenden Teil wird beschrieben, welche Potenziale die Kombination beider Praktiken für den Berufseinsteiger bietet.

A. Auswertung einzelner Studien

Flohr und Schneider [19], [20] bezieht sich auf das selbe Experiment, untersucht die Auswirkungen von der TDD auf die Produktivität und interne Codequalität. Dazu wurde die Zeit gemessen die Studienteilnehmer für die Umsetzung einer festgelegten Anzahl von Story-Cards benötigten sowie die

Code-Coverage. An der Studie nahmen 18 Studenten teil, die in 9 Paaren arbeiteten. Fünf Paare entwickelten testgetrieben und vier traditionell. Die Paare die testgetrieben Entwickelten benötigten weniger Zeit bzw. waren 26.5% produktiver. Dies war aber nicht statistisch signifikant. Hinsichtlich der Code-Coverage wurde kein statistisch signifikanter Unterschied zwischen den beiden Gruppen festgestellt.

An der Studie von Madeyski [52] nahmen 98 Studenten teil. Davon arbeiteten 70 in Paaren und 28 allein. Alle Teilnehmer entwickelten testgetrieben. Die Branch-Coverage war bei den 35 Paaren ähnlich wie bei den 28 Solo-Entwicklern. Es konnte kein statistisch signifikanter Unterschied festgestellt werden.

In Madeyski [16] wurden verschiedenen Kombinationen von TDD, TLD, Pair-Programming und Solo-Entwicklung (SP) verglichen. An der Studie nahmen 188 Master-Studenten teil. Die Studenten wurden in vier Gruppen eingeteilt. 28 entwickelten traditionell und alleine (TLD+SP), 28 entwickelten testgetrieben und alleine (TDD+SP), 62 entwickelten traditionell und in Paaren (TLD+PP), und 70 testgetrieben und in Paaren (TDD+PP). Die externe Codequalität war statistisch signifikant geringer, wenn testgetrieben Entwickelt wurde. Es konnte kein Unterschied hinsichtlich der externen Codequalität festgestellt werden, wenn Pair-Programming angewendet wurde.

In der Tabellen VI und Tabelle VII sind die Ergebnisse der Auswertung zusammengefasst.

Tabelle VI
AUSWERTUNG TDD+PP VS. TLD+PP

Studie	Produktivität	Externe Codequalität	Interne Codequalität
[19]	$=^{+}$	k.A.	=
[16]	k.A.	-	k.A.

Legende:

= keine statistisch signifikanten Auswirkungen
- statistisch signifikant negative Auswirkungen von TDD gegenüber TLD
$^{+}$ positive Tendenzen von TDD+PP gegenüber TLD+PP

Tabelle VII
AUSWERTUNG TDD+PP VS. TDD+SP

Studie	Produktivität	Externe Codequalität	Interne Codequalität
[52]	k.A.	k.A.	=
[16]	k.A.	=	k.A.

Legende:

= keine statistisch signifikanten Auswirkungen

B. Potenziale für Berufseinsteiger

Die gemeinsame Anwendung von Pair-Programming und der testgetriebenen Entwicklung deckt viele der in Abschnitt III beschriebenen Herausforderungen ab. Dies betrifft auch Punkte, die bei der einzelnen Anwendung einer der beiden Praktiken, ansonsten nicht betroffen wären, vergleiche die Unterabschnitte IV-D und V-C7.

Für den Berufseinsteiger der mit der testgetriebenen Entwicklung nicht vertraut ist, kann es hilfreich sein, sich während des Pair-Programmings mit der TDD auseinanderzusetzen. Bestenfalls kennt sich sein Partner mit der Praktik aus, und kann ihn anleiten. Dieses Vorgehen wird auch von dem Experten in [9] empfohlen. Andernfalls, wenn beide Entwickler wenig Erfahrung mit der TDD haben, kann der Entwickler in der Rolle des Navigators, zusätzlich ein Augenmerk auf die korrekte Einhaltung der TDD-Regeln legen. Dass es für einen Entwickler sehr schwer sein kann TDD zu lernen, wenn er gleichzeitig mit einer anspruchsvollen Programmieraufgabe beschäftig ist, war auch ein Ergebnis der Studie von Kollanus und Isomöttönen [53]. An der Studie nahmen Studenten teil, die keine oder wenig Kenntnisse von der TDD hatten. Nachdem sie eine Programmieraufgabe testgetrieben gelöst hatten, wurden sie nach ihren Erfahrungen befragt.

Die Herausforderungen von Berufseinsteigern sind schematisch in der letzten Spalte *PP+TDD* in Tabelle VIII aufgeführt. Falls die Kombination von der TDD und dem PP eine Herausforderung adressiert, ist dies durch ein ✓gekennzeichnet. Hat die Kombination beider Praktiken keinen Einfluss auf eine Herausforderung, ist diese mit einem - dargestellt. Wirkt sich die TDD und PP möglicherweise negativ auf die Herausforderung aus, ist dies durch ein Bkenntlich gemacht worden.

Tabelle VIII
POTENZIAL TDD UND PP

Herausforderung	PP	TDD	PP + TDD
Kommunikation			
Zusammenarbeit mit Menschen aus einem anderen Bereich oder mit unterschiedlichen Niveau	✓	✓	✓
Kommunikation mit Kunden nur eingeschränkt möglich	-	-	-
Kommunikation mit Mitarbeitern nur eingeschränkt möglich	✓	✓	✓
Verantwortung			
Verantwortung übernehmen	✓	✓	✓
Verantwortlich für Ergebnisse	✓	✓	✓
Arbeitsdruck	B	✓	✓
Selbständig arbeiten	✓	✓	✓
Selbständig lernen			
Neue Domäne	✓	-	✓
Neue Technologie	✓	✓	✓
Altsysteme / fremder Code	✓	✓	✓
Neue Werkzeuge	✓	-	✓
Selbstvertrauen			
Angst Fehler zu machen	✓	✓	✓
Nicht genug zu Wissen	✓	B	✓

Legende:

✓Herausforderung wird angesprochen
BHerausforderung wird verschlimmert
- keine Auswirkung

VII. Fazit

Die Auswertungen der Studien ergab, dass hinsichtlich der Produktivität, internen Codequalität und externen Codequalität kaum Unterschiede zwischen der testgetriebenen und der traditionellen Entwicklung bestehen bzw. nicht eindeutig ist, welches Verfahren vorzuzuziehen ist. Hier sind weitere Studien mit mehr Teilnehmern, längeren Laufzeiten und genauerer Beobachtung, ob TDD korrekt angewendet wurde, abzuwarten. Dennoch kann die TDD den Berufseinsteigern bei Herausforderungen in den Bereichen Kommunikation, Verantwortung, Einarbeitung und Selbstvertrauen helfen. Voraussetzung dafür ist jedoch, dass der Berufseinsteiger Kenntnisse von der TDD hat oder motiviert ist diese zu erlangen. Sonst kann das Erlernen bzw. Anwenden der TDD schnell frustrierend werden.

Die betrachteten Studien wiesen nach, dass die Anwendung von PP hilft Programmfehler früher zu finden sowie das Codeverständnis das Design und die Codequalität zu verbessern. Studenten die PP anwenden sind seltener frustriert, haben mehr Selbstvertrauen und fühlen sich weniger isoliert. PP hilft den Berufseinsteigern mit den Herausforderungen in den Bereichen Kommunikation, Verantwortung, Einarbeitung und Selbstvertrauen umzugehen. Zusätzlich unterstützt PP den Berufseinsteiger in das Team zu integrieren. Neben Organisatorischen Schwierigkeiten, kann PP zu Problemen führen, wenn die beiden Entwickler im zwischenmenschlichen Bereich nicht harmonieren. Das könnte zum Beispiel der Fall sein, wenn der Berufseinsteiger wenig kommunikativ ist oder der Experte, mit mehrjähriger Erfahrung, sich von dem Berufseinsteiger gestört fühlt.

Beide Praktiken unterstützen den Berufseinsteiger mit seinen Mitarbeitern zu kommunizieren, mit den neuen Verantwortungen zurechtzukommen, bei der Einarbeitung bzw. mit dem selbständigen Lernen und sein Selbstvertrauen zu stärken. Die Schwächen der einen Praktik können teilweise durch Anwendung der anderen Praktik kompensiert werden. So kann es beispielsweise weniger frustrierend sein, wenn TDD als Paar gelernt wird. Allerdings gilt für beide Praktiken, dass die erfolgreiche Anwendung von der Persönlichkeit der beteiligten Entwickler abhängig ist. Wenig Probleme sollte die Kombination beider Praktiken machen, wenn der Berufseinsteiger bereits Grundkenntnisse in der testgetriebenen Entwicklung hat und über die für das PP nötigen Soft Skills verfügt.

Literatur

[1] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, 2004.

[2] A. Causevic, D. Sundmark, and S. Punnekkat, "Test case quality in test driven development: A study design and a pilot experiment," in *16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012)*. IET, 2012, pp. 223–227. [Online]. Available: http://digital-library.theiet.org/content/conferences/10.1049/ic.2012.0029

[3] D. Fucci, G. Scanniello, S. Romano, M. Shepperd, B. Sigweni, F. Uyaguari, B. Turhan, N. Juristo, and M. Oivo, "An External Replication on the Effects of Test-driven Development Using a Multi-site Blind Analysis Approach," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '16. New York, NY, USA: ACM, 2016, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/2961111.2962592

[4] L. Madeyski, "The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment," *Information and Software Technology*, vol. 52, no. 2, pp. 169–184, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584909001487

[5] S. Mäkinen and J. Münch, "Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies," in *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering: 6th International Conference, SWQD 2014, Vienna, Austria, January 14-16, 2014. Proceedings*, D. Winkler, S. Biffl, and J. Bergsmann, Eds. Cham: Springer International Publishing, 2014, pp. 155–169. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-03602-1_10

[6] Y. Rafique and V. Mĭsíc, "The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 835–856, 2013.

[7] H. Munir, M. Moayyed, and K. Petersen, "Considering rigor and relevance when evaluating test driven development: A systematic review," *Information and Software Technology*, vol. 56, no. 4, pp. 375–394, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2014.01.002

[8] S. Kollanus, "Test-Driven Development - Still a Promising Approach?" in *2010 Seventh International Conference on the Quality of Information and Communications Technology*, 9 2010, pp. 403–408.

[9] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus, "What Do We Know about Test-Driven Development?" *IEEE Software*, vol. 27, no. 6, pp. 16–19, 11 2010.

[10] P. Sfetsos and I. Stamelos, "Empirical Studies on Quality in Agile Practices: A Systematic Literature Review," in *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology*, ser. QUATIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 44–53. [Online]. Available: http://dx.doi.org/10.1109/QUATIC.2010.17

[11] R. Jeffries and G. Melnik, "The Art of Fearless Programming," *IEEE Software*, vol. 24, no. 3, pp. 24–30, 2007.

[12] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 337–346.

[13] B. Turhan, L. Layman, M. Diep, H. Erdogmus, and F. Shull, "How effective is test-Driven Development," in *Making Software: What Really Works, and Why We Believe It*. OReilly Press, 2010, ch. 12, pp. 207–217.

[14] W. Bissi, A. G. Serra Seca Neto, and M. C. F. P. Emer, "The effects of test driven development on internal quality, external quality and productivity: A systematic review," *Information and Software Technology*, vol. 74, pp. 45–54, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2016.02.004

[15] S. Kollanus, "Critical Issues on Test-Driven Development," in *Product-Focused Software Process Improvement: 12th International Conference, PROFES 2011, Torre Canne, Italy, June 20-22, 2011. Proceedings*, D. Caivano, M. Oivo, M. T. Baldassarre, and G. Visaggio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 44, no. 4, pp. 322–336. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21843-9_25

[16] L. Madeyski, "Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality," in *Proceedings of the 2005 Conference on Software Engineering: Evolution and Emerging Technologies*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2005, pp. 113–123. [Online]. Available: http://dl.acm.org/citation.cfm?id=1565142.1565156

[17] A. Gupta and P. Jalote, "An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 9 2007, pp. 285–294.

[18] M. Müller and O. Hagner, "Experiment about test-first programming," *IEE Proceedings - Software*, vol. 149, no. 5, pp. 131–136, 2002.

[19] T. Flohr and T. Schneider, "Lessons Learned from an XP Experiment with Students: Test-First Needs More Teachings," in *Product-Focused Software Process Improvement: 7th International Conference, PROFES 2006, Amsterdam, The Netherlands, June 12-14, 2006. Proceedings*, J. Münch and M. Vierimaa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 305–318. [Online]. Available: http://link.springer.com/10.1007/11767718_26

[20] ——, "An XP Experiment with Students – Setup and Problems," in *Product Focused Software Process Improvement: 6th International Conference, PROFES 2005, Oulu, Finland, June 13-15, 2005. Proceedings*, F. Bomarius and S. Komi-Sirviö, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 474–486. [Online]. Available: http://dx.doi.org/10.1007/11497455_37

[21] M. Müller and W. Tichy, "Case Study: Extreme Programming in a University Environment," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 2001, pp. 537–544. [Online]. Available: https://ps.ipd.kit.edu/downloads/ka_2001_extreme_programming_university_environment.pdf

[22] B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering," *EBSE*, 2007.

[23] J. Higgins and S. Green, "Cochrane Handbook for Systematic Reviews of Interventions Cochrane Book Series THE COCHRANE COLLABORATION ®," *John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England*, 2008.

[24] J. Bevan, L. Werner, and C. Mcdowell, "Guidelines for the Use of Pair Programming in a Freshman Programming Class," in *Proceedings of the 15th Conference on Software Engineering Education and Training*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 100 – 103. [Online]. Available: http://dl.acm.org/citation.cfm?id=872751.873501

[25] L. Williams, L. Layman, O. Jason, N. Katira, J. Osborne, and N. Katira, "Examining the compatibility of student pair programmers," in *AGILE 2006 (AGILE'06)*, 2006, pp. 411–420.

[26] L. Williams, "Pair Programming," in *Making Software: What Really Works, and Why We Believe It*, 1st ed. O'Reilly Media, Inc., 2010, ch. 17.

[27] J. Nosek, "The Case for Collaborative Programming," *ACM*, vol. 41, no. 3, pp. 105 – 108, 1998.

[28] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the Case for Pair-Programming," *IEEE Software*, vol. 17, no. 4, pp. 19–25, 2000.

[29] S. Alshehri and L. Benedicenti, "Ranking and Rules for Selecting Two Persons in Pair Programming," *JOURNAL OF SOFTWARE, NO. 9*, vol. 9, no. 9, pp. 2467 – 2473, 2014.

[30] A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," in *Extreme Programming examined*. Boston: Addison-Wesley Longman Publishing Co., Inc, 2001, pp. 223–243.

[31] K. M. Lui and K. Chan, "Pair programming productivity: Novice–novice vs. expert–expert," *International Journal of Human-Computer Studies*, vol. 64, no. 9, pp. 915–925, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1071581906000644

[32] H. Baytiyeh and M. Naja, "Challenges Facing Graduating Engineers in their Transition from College to Career," in *2011 ASEE Annual Conference \& Exposition*. ASEE Conferences, 2011, pp. 22.317.1 – 22.317.10. [Online]. Available: https://peer.asee.org/17598

[33] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, 9 2005.

[34] J. Link, *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*, 2nd ed. dpunkt. verlag, 2005.

[35] K. Beck, *Test Driven Development: By Example*. AddisonWesley Professional, 2003.

[36] R. Martin, "Professionalism and Test-Driven Development," *IEEE Software*, vol. 24, no. 3, pp. 32–36, 2007.

[37] N. Yahya and N. Bakar, "The analysis of programming competency in test driven development," in *2015 9th Malaysian Software Engineering Conference (MySEC)*, 2015, pp. 290–295.

[38] M. F. Aniche and M. A. Gerosa, "Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 4 2010, pp. 469–478.

[39] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, "Effects of Negative Testing on TDD: An Industrial Experiment," in *Agile Processes in Software Engineering and Extreme Programming: 14th International Conference, XP 2013, Vienna, Austria, June 3-7, 2013. Proceedings*, H. Baumeister and B. Weber, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 91–105. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38314-4_7

[40] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A Dissection of Test-Driven Development: Does It Really Matter to Test-First or to Test-Last?" *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1 – 21, 2016.

[41] K. Pugh, *Lean-agile acceptance test driven development : better software through collaboration*. Upper Saddle River, NJ: Addison-Wesley, 2011.

[42] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner, "Building Pair Programming Knowledge Through a Family of Experiments," in *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, ser. ISESE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 143 – 153. [Online]. Available: http://dl.acm.org/citation.cfm?id=942801.943642

[43] A. Begel and N. Nagappan, "Pair Programming: What's in It for Me?" in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: ACM, 2008, pp. 120–128. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414026

[44] Jari Vanhanen; Harri Korpi, "Experiences of Using Pair Programming," *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International*, pp. 274b – 274b, 2007.

[45] J. Chong and T. Hurlbutt, "The Social Dynamics of Pair Programming," *IEEE Computer Society Washington, DC, USA*, pp. 354–363, 2007.

[46] L. Layman, L. Williams, J. Osborne, S. Berenson, K. Slaten, and M. Vouk, "How and Why Collaborative Software Development Impacts the Software Engineering Course," in *Proceedings Frontiers in Education 35th Annual Conference*, 2005.

[47] L. Layman, "Changing students' perceptions: An analysis of the supplementary benefits of collaborative software development," in *Software Engineering Education Conference, Proceedings*, 2006.

[48] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik, "Improving the CS1 experience with pair programming," *ACM SIGCSE Bulletin*, 2003.

[49] L. Williams and R. Kessler, "All I Really Need to Know about Pair Programming I Learned In Kindergarten," *Communications of the ACM*, vol. 43, pp. 108–114, 2000.

[50] A. Bateson, R. Alexander, M. Murphy, and A. Bateson, "Cognitive processing differences between novice and expert computer programmers," *Int. J. Man-Machine Studies*, vol. 26, pp. 649–660, 1987.

[51] I. Fronza, A. Sillitti, and G. Succi, "An interpretation of the results of the analysis of pair programming during novices integration in a team," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, 2009.

[52] L. Madeyski, "On the Effects of Pair Programming on Thoroughness and Fault-Finding Effectiveness of Unit Tests," in *Product-Focused Software Process Improvement: 8th International Conference, PROFES 2007, Riga, Latvia, July 2-4, 2007. Proceedings*, J. Münch and P. Abrahamsson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4589, pp. 207–221. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73460-4_20

[53] S. Kollanus and V. Isomöttönen, "Test-driven Development in Education: Experiences with Critical Viewpoints," in *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '08. New York, NY, USA: ACM, 2008, pp. 124–127. [Online]. Available: http://doi.acm.org/10.1145/1384271.1384306