

**Ausarbeitung im Rahmen des Seminars
Ausgewählte Themen zu agilen Softwareprozessen**

Wintersemester 2006/07

Pair Programming

Miao Wang

mwang@inf.fu-berlin.de

Betreuer: Prof. Lutz Prechelt, Stephan Salinger

Pair Programming ist eine Arbeitstechnik im Rahmen der agilen Softwareprozesse, die in jüngster Zeit mehr und mehr Interesse geweckt hat. Im Vergleich zum traditionellen Programmierstil bietet Pair Programming enorme qualitative und prozessspezifische Verbesserungen, die es durchaus attraktiv erscheinen lassen. Diese Ausarbeitung beleuchtet die verschiedenen positiven Faktoren, die den Einsatz befürworten. Wissenschaftliche Ergebnisse dazu werden anhand von drei empirischen Untersuchungen dargestellt. Weiterhin werden kritische Stellen des Pair Programmings gesondert beleuchtet.

Inhaltsverzeichnis

1	Einleitung	2
2	Faktoren	3
2.1	Pair Pressure	3
2.2	Pair Quality	4
2.3	Pair Learning	5
2.4	Pair Time	5
2.5	Pair Think	6
2.6	Pair Relaying	6
2.7	Pair Satisfaction	7
2.8	Teambildung und Kommunikation	7
2.9	Projektrisiko	7
3	Empirische Untersuchungen	8
3.1	Nosek 1998: The Case for Collaborative Programming	8
3.2	Williams 2000: The Collaborative Software Process	9
3.3	Katira, Williams et al. 2004: On Understanding Compatibility of Student Pair Programmers	11
4	Kritik	13
4.1	Pair Management	13
4.2	Angsterzeugende Umgebung	13
4.3	Wissensaustausch	13
4.4	Widerstände und Grundeinstellungen	14
4.5	Ungeeignete Situationen für Pair Programming	14
5	Zusammenfassung	14

"Knowledge is commonly socially constructed through collaborative efforts toward shared objectives or by dialogues and challenges brought about by differences in persons' perspectives"- Salomon [1]

1 Einleitung

Pair Programming (zu Deutsch: Paarprogrammierung oder paarweises Programmieren) ist eine Arbeitstechnik im Rahmen der agilen Softwareprozesse, die in jüngster Zeit mehr und mehr Interesse geweckt hat. Im Vergleich zum traditionellen Programmierstil bietet Pair Programming enorme qualitative und prozessspezifische Verbesserungen, die es durchaus attraktiv erscheinen lassen [2, 3, 4]. Obwohl paarweises Programmieren bereits lange Zeit sporadisch angewandt wurde [5], ist es erst Mitte der Neunziger Jahre im Kontext von Extreme Programming (XP) als zentrale Praktik zum weitverbreiteten Einsatz gekommen [6]. Heutzutage wird Pair Programming in Verbindung mit vielen anderen agilen Softwareprozessen wie Scrum und Crystal empfohlen und angewandt.

Beim Pair Programming arbeiten zwei Programmierer gemeinsam an demselben Softwareartefakt und teilen sich dafür im ständigen Wechsel einen Rechner mit einer Tastatur und einem Monitor. Während die eine Person als sogenannter *Driver* aktiv programmiert und seine Vorgehensweise erläutert, schlüpft die zweite Person in die Rolle des *Navigators* (oder auch als *Observer* bezeichnet), die zuständig dafür ist die Arbeit des Drivers kontinuierlich zu überwachen und Verbesserungsvorschläge zu entwickeln. In regelmäßigen Abständen werden diese beiden Rollen getauscht um das Wissen innerhalb des Teams besser zu verteilen. Zusätzlich wird auch die Zusammensetzung der Paare häufig geändert.

Bei der Erstellung des Artefakts agieren die beiden Entwickler wie ein zusammenhängender, intelligenter Organismus, der mit einem Verstand arbeitet und verantwortlich für jeden Aspekt des Artefakts ist [7]. Der Driver hat die Verantwortung seine kognitiven Prozesse akustisch zu schildern und die gemeinsam getroffenen Entwicklungsentscheidungen im Code festzuhalten. Der Navigator hat etwas mehr Abstand zur aktuellen Arbeit und kann die gegenwärtige Strategie beurteilen und abwägen. Bei Ungereimtheiten oder Entdeckungen von Defekten werden die Probleme unmittelbar im Gespräch gelöst. Auf diese Weise werden Fehler frühzeitig erkannt und problematische Lösungen vermieden, was zu einer gesteigerten Softwarequalität führen soll. Genauso wichtig sind Lerneffekte, das gegenseitige Kennenlernen und die geförderte Teamarbeit. Durch sogenanntes *Collective Code Ownership* wird das Wissen über die gesamte Codebasis auf verschiedene Entwickler verteilt, sodass ein Ausfall eines einzelnen Mitarbeiters weniger ins

Gewicht fällt. Dabei muss sich Pair Programming nicht nur auf das konkrete Programmieren beschränken, sondern kann auch in der Analyse-, Entwurfs- und Testphase zum Einsatz kommen [7].

Trotzdem begegnen viele Projektleiter und erfahrene Programmierer dieser Praktik eher mit Skepsis. Insbesondere auf Managementseite sorgt man sich über verdoppelte Personenstunden und Kosten, während aus dem Lager der Programmierer Abneigungen gegenüber disziplinierte Kommunikation und Teamarbeit entstammen, da man gewohnt ist alleine zu programmieren. Selbst für die Projektleiter, die bereit sind auf Pair Programming zu setzen, ergeben sich Fragen, wie die Teamfindung am gescheitesten zu verwalten und einzuplanen wäre.

Diese Ausarbeitung konzentriert sich auf die wesentlichen Faktoren von Pair Programming, die in Abschnitt 2 beschrieben werden. Im 3. Abschnitt werden empirischen Untersuchungen anhand der wissenschaftlichen Studien von John T. Nosek [2] und Laurie Williams et al. [3, 4] beleuchtet. Hierbei geht es primär darum die Effekte von Pair Programming abzuwägen und die Methodik zu optimieren. Andere Studien wiesen kritische bis negative Ergebnisse in Bezug auf Pair Programming auf, die in Abschnitt 4 näher erläutert werden.

2 Faktoren

Eine vermehrte Anzahl an wissenschaftliche Studien zeigte auf, dass Pair Programming in der Regel förderlich ist. Insbesondere Extreme Programming baut auf Pair Programming auf und praktiziert dessen Einsatz in allen Bereichen, wie auch im Prototyping [8]. Es konnte gezeigt werden, dass die entstandene Software eine erhöhte Qualität aufwies und die Produkteinführungszeit beschleunigt wurde. Zusätzlich attestierten viele Programmierer eine erhöhte Leidenschaft zu Pair Programming, selbst wenn sie zuvor demgegenüber abgeneigt waren. Aus den wissenschaftlichen Ausarbeitungen geht eine Reihe von positiven Faktoren hervor, die dem Pair Programming zugeschrieben werden und zu solchen Effekten führen [7, 3, 9].

2.1 Pair Pressure

Entwickler berichten von einem gewissen positiven Druck, der auf sie lastete, wenn sie in Paaren programmierten. Sie gaben zu, als Driver den Drang zu empfinden eine gute Arbeit zu hinterlegen und den Partner möglichst zufrieden zu stimmen. Währenddessen verspürte der Navigator eine Verantwortung gegenüber dem Partner die Überprüfung gewissenhaft zu führen und Fehler möglichst frühzeitig zu erkennen. Insgesamt führt das dazu, dass beide konzentriert und verant-

wortungsbewusst zu Werke gehen um die andere Person nicht zu enttäuschen. Dies erhöht die Motivation und Geschwindigkeit der Arbeit, da auch der Partner stets ein gewisses Tempo erwartet. In einem Experiment der Universität von Utah äußert sich ein Student über dieses *Pair Pressure* wie folgt:

"When I worked on the machine as the driver, I concentrated highly on my work. I wanted to show my talent and quality work to my partner. When I was doing it, I felt more confident. In addition, when I had a person observing my work, I felt that I could depend on him, since this person cared about my work and I could trust him. If I made any mistakes, he would notice them, and we could have a high quality product. When I was the non-driver, I proofread everything my partner typed. I felt I had a strong responsibility to prevent any errors in our work. I examined each line of code very carefully, thinking that, if there were any defects in our work, it would be my fault. Preventing defects is the most important contribution to the team, and it put a bit of pressure on me."[7]

Zudem wird durch die gegenseitige Verantwortung die gemeinsam verbrachte Zeit effektiver genutzt, da beide Programmierer sich seltener durch Unterbrechungen, wie Telefonaten, Email oder Surfen im Internet, ablenken lassen. Gewisse Vorgehensweisen und Standards werden ebenso eingehalten, da dies prinzipiell von der anderen Person erwartet wird [3].

2.2 Pair Quality

In dem Experiment von Utah zeigte sich auch, dass die Programme von Paaren in den letztendlichen Defekttests meist besser abschnitten. Sie enthielten alle weniger Defekte und bestanden mehr Testläufe. Diese *Pair Quality* erklärt sich durch die ständige manuelle Durchsicht des Navigators. Durchsichten haben sich als kostenwirksame Methoden gegenüber Defekttest in der analytischen Qualitätssicherung etabliert. Wie Gerald M. Weinberg schon festgestellt hatte, haben Programmierer, die alleine arbeiten, eine unbegrenzte Kapazität ihre eigenen Fehler zu übersehen:

"The human eye has an almost infinite capacity for not seeing what it does not want to see . . . Programmers, if left to their own devices, will ignore the most glaring errors in their output - errors that anyone else can see in an instant."[10]

Jedoch zeigen sich Entwickler meist unerfreulich Durchsichten durchzuführen. Beim Pair Programming wird eine kontinuierliche Durchsicht verpflichtend und verantwortungsbewusst gegenüber dem Partner durchgeführt. Dadurch wird die Motivation zu Durchsichten erhöht und die Defekte relativ früh im Entwicklungsprozess eliminiert. Eine Durchsicht um Defekte während der Programmerstellung ausfindig zu machen ist deutlich preiswerter als die Defekte in einem nachträglichen Defekttest zu ermitteln, da die Defektlokalisierung direkt vor Ort geschieht.

Durch die ständige Kommunikation zwischen den Teilnehmern entsteht ein weiterer Defektfindungsprozess, den man mit *Debugging by Explaining* bezeichnet, indem man alleinig durch das Erläutern seines Problems zu einer Lösung kommt. Ebenso erkennt man möglicherweise neue Probleme durch die Präsentation und Verarbeiten eines alten Problems [3].

2.3 Pair Learning

Pair Programming zeigt auch einen einzigartigen Bildungsaspekt, indem Kenntnisse ständig weitergereicht werden zwischen den Teammitgliedern. *Pair Learning* geht in Hinblick auf die kontinuierlichen Durchsichten sogar einen Schritt weiter zur Defektbekämpfung, denn aus Fehlern lernt man und begeht sie seltener in der Zukunft. In dieser Hinsicht kann das Programmieren in Paaren auch zur konstruktiven Qualitätssicherung beitragen.

Zusätzlich wird zwischen den Partnern der Gebrauch von Werkzeugen vermittelt, die von speziellen Sprachspezifika bis zu allgemeinen Entwurfs- und Programmierstrategien reichen. Auch werden unerwähnte Kenntnisse, wie unterbewusste Handlungen und Angewohnheiten weitergegeben. In einer Art Lehrverhältnis kann ein Anfänger von einem Experten durch bloßes Beobachten lernen. Dies geschieht in einer *"line of sight"*-Umgebung zum Experten, in der Fachkenntnisse durch fortlaufende audiovisuelle Übertragung übermittelt werden. Als *"Expert In Earshot"* wird eine Methode bezeichnet, in der Anfänger einem Experten zugewiesen werden, wenn der Anfänger keine guten Lernprozesse erzielt und kein Experte als Vollzeit-Lehrer abgestellt werden kann. Der Anfänger lernt so durch zuschauen und zuhören, während der Experte seiner gewohnten Arbeit nachgeht. Pair Programming ist ein Beispiel für *"line of sight"* und *"Expert in Earshot"* zugleich [9].

2.4 Pair Time

Durch die gewonnenen Erkenntnisse in Pair Pressure, Pair Quality und Pair Learning kann man für Pair Programming eine verbesserte Entwicklungszeit in Softwareprojekten erwarten. Durch den gegenseitig positiven Druck der Entwickler werden weniger Unterbrechungen eingelegt,

während die Qualität steigt und der Bedarf an Defekttests reduziert wird. Überdies können durch den Lernprozess neue Mitarbeiter schnell in das Projekt integriert und fördernde Kenntnisse weiter verbreitet werden.

Zum Erreichen dieser Faktoren bedarf es jedoch eine gewisse Einarbeitungszeit für ein neues Paar, sich kennenzulernen und aufeinander abzustimmen. Diese Anpassungsphase wird auch als *Jelling Time* bezeichnet und kann je nach Paar und Umgebung Stunden bis Tage dauern [9]. Nichtsdestotrotz kommen viele Studien zu dem Schluss, dass Pair Programming die Entwicklungszeit absenkt anstatt erhöht [3, 9, 11]. Zudem wird behauptet, dass bei Pair Programming die Qualität der Software in Form von Anzahl Codezeilen und der Entwicklungszeit deutlich vorhersagbarer wird [12].

2.5 Pair Think

Viele glorreiche Leistungen sind aus gemeinschaftlicher Arbeit entstanden, in der viele Ideen oder verschiedene Ansichten in einen Topf geworfen wurden um eine differenzierte gute Lösung zu finden. Bei der Entwicklung in Paaren wird ebenso vorgegangen. In der Entwurfsphase kommt es besonders darauf an einen möglichst grundlegenden und guten Plan für den weiteren Verlauf aufzustellen. Dabei bringen verschiedene Akteure verschiedene Erfahrungen aus vergangenen Projekten mit, haben verschiedenen Einsichten zu problembezogenen relevanten Informationen und stehen in verschiedenen Beziehungen aus unterschiedlichen funktionalen Rollen zum zugrundeliegenden Problem. Dies hat zur Konsequenz, dass ein gewisser Konflikt entsteht, der zu einer Aushandlung einer großen Anzahl an Alternativen führt. Dies wiederum reduziert die Wahrscheinlichkeit einen schlechten Plan zu wählen [13]. Dieser Prozess des kollaborativen Brainstormings wird dem Faktor *Pair Think* zugeschrieben, den auch viele Studenten bestätigten:

"We often came up with different ideas about how the design should go and the result of arguing over which one was better often led to a truly superior hybrid design." [3]

2.6 Pair Relaying

Früher oder später tritt unweigerlich eine Situation ein, in der beide Entwickler verwirrt sind, warum eine Sache nicht funktioniert wie es soll oder wie weiter verfahren werden sollte. Hier setzt *Pair Relaying* ein. Im Gegensatz zum *Pair Think* wird versucht ein unbegreifliches Problem durch gemeinsames Wissen und Einsatz reihum zu lösen. Dadurch wird stets versucht einen kleinen Aspekt des Problems zu erschließen und somit langsam eine Lösung zu entwickeln.

Durch die Kombination von Pair Think und Pair Relaying wird die Qualität weiter verbessert und der Entwicklungsprozess vereinfacht, wie ein Student ebenfalls bemerkte:

"I have found that, after working with a partner, if I go back to working alone, it is like part of my mind is gone. I find myself getting confused about things." [3]

2.7 Pair Satisfaction

Diverse Studien haben gezeigt, dass Pair Programming eine Technik ist, die Programmierer erfreut und ein gewisses Maß an Zufriedenheit und Selbstvertrauen liefert [7, 2]. Die Paare genießen ihre Arbeit mehr, da sie überzeugter sind vor ihrem Werk, denn es gibt eine weitere Person, die einem hilft und mit der man sich beraten kann. Sie verbringen mehr Zeit in herausfordernden Lösungsansätzen als im ärgerlichen Debugging. Zudem kann eine Lösung gemeinsam gefeiert werden. Es sei jedoch angemerkt, dass nicht alle Paare solche positiven Facetten mitbringen, sondern dass Paare gezielt zusammengestellt werden müssen um unpassende Zusammenstellungen zu vermeiden. Eine Analyse zu einer Studie zu dieser Thematik folgt im späteren Verlauf 3.3.

2.8 Teambildung und Kommunikation

Pair Programming fördert grundsätzlich die *Teambildung* und die *Teamarbeit*. Dabei spielt die *Kommunikation*, genauso wie für agile Softwareprozesse im Allgemeinen, eine zentrale Rolle. Dies lässt sich leicht nachvollziehen, denn dadurch dass die Programmierer miteinander arbeiten müssen, lernen sie auch zweifellos voneinander. So werden Probleme und Lösungen öfter miteinander ausgetauscht und Hintergedanken zwischen einander treten seltener auf. Auf dieser Weise wird das Teamwork verbessert. Durch die Zusammenarbeit wird gelernt wie man miteinander kommuniziert und dies Kommunikation immer öfter praktiziert. Das erhöht die Kommunikationsbandbreite und -frequenz innerhalb des Projekts und den gesamten Informationsfluss innerhalb des Teams. All dies führt zur erhöhten Effizienz von Teams [9]. Diese Entwicklung eines kollektiven Zusammengehörigkeitsgefühls ist eines der Schlüsselemente von Pair Programming.

2.9 Projektrisiko

Durch den beschriebenen Informationsfluss innerhalb eines Paares und auch innerhalb des gesamten Personals (wenn man die Partner öfter wechselt), ist es nicht so schwerwiegend einen Ausfall zu erleiden. Da stets mehrere Personen mit dem System vertraut sind, kann der Verlust

von Schlüsselpersonen gut aufgefangen werden. Eine gebräuchliche informelle Metrik für das *Projektrisiko* beruft sich auf die "truck number" erfunden von Jim Coplien. Dabei geht es um die Antwort auf die Frage "Wie viele oder wenige Personen müssten von einem Laster überfahren werden, damit die Fertigstellung des Projekts verhindert wird?". Die schlimmste Antwort wäre "Eine". Durch die erwähnte Collective Code Ownership wird das Wissen über das Team verteilt und die "truck number" und somit die Projektsicherheit erhöht [3].

3 Empirische Untersuchungen

3.1 Nosek 1998: The Case for Collaborative Programming

John T. Nosek untersuchte 1998 die Vorteile vom Programmieren in Paaren in einem Experiment mit erfahrenen Programmierern aus der Industrie [2]. Da damals die Bezeichnung Pair Programming noch nicht weit verbreitet war, bezeichnete Nosek es als *Collaborative Programming*, wo zwei Personen gemeinsam am selben Algorithmus und Code arbeiten. Eine klare Trennung der Rollen in Driver und Navigator gab es noch nicht.

Es wurden 15 Systementwickler einer Softwarefirma gewählt, die ein für ihre Firma wichtiges Programm zur Konsistenzprüfung einer Datenbank entwickeln sollten. Dabei durften sie in ihrer gewohnten Arbeitsumgebung und Ausstattung arbeiten. Es erfolgte eine zufällige Einteilung in eine experimentelle Gruppe mit 5 Paaren und eine Kontrollgruppe mit den restlichen 5 als einzelne Programmierer. Die Bearbeitungszeit wurde auf maximal 45 Minuten beschränkt, wonach die Testpersonen zu ihrer Zuversicht (CONFID) und Zufriedenheit (ENJOY) befragt wurden. Zur Evaluation wurden die Programme mit den Kenngrößen READABILITY (Lesbarkeit des Quellcodes) von 0 bis 2 und FUNCTIONALITY (Funktionalität gemäß Spezifikation) von 0 bis 6 bewertet (Abb. 1).

Die Ergebnisse bestätigten die Hypothese, dass Paare in den Punkten Lesbarkeit und Funktionalität die Einzelprogrammierer übertroffen haben. Ebenso war die Bearbeitungszeit knapp 12 Minuten (41%) länger bei einzelnen Programmierern als in Paaren. Jedoch sei dieser Wert aufgrund von hoher Abweichung statistisch unzuverlässig. Ebenso zeigten die Paare deutlich höheres Vertrauen und Vergnügen an ihrer Arbeit als die, die einzeln programmierten. Zudem wurde mit hoher Korrelation noch bestätigt das erfahrenere Programmierer besser abschnitten als unerfahrenere. Dieses Experiment war, in Hinblick, dass es mit professionellen Entwicklern statt fand und nicht wie üblicherweise mit Studenten, einzigartig. Nosek führt diese Ergebnisse als Beleg dafür an, dass kollaboratives Programmieren die Entwicklungszeit beschleunigt und

die Softwarequalität verbessert. Außerdem wurde darauf hingewiesen, dass Programmierer, die sich zunächst skeptisch gegenüber dem Prozess zeigten, sich nachher an ihrem Einsatz erfreuten.

Variable	Control Group (Individuals) mean (st. dev.)	Experimental Group (Teams) mean (st. dev.)
Performance Scores:	n = 5	n = 5
READABILITY	1.40 (0.894)	2.00 (0.000)
FUNCTIONALITY SCORE	4.20 (1.788)	5.60 (0.547)*
SCORE	5.60 (2.607)	7.60 (0.547)*
TIME (minutes)	42.60 (3.361)	30.20 (1.923)
Satisfaction Measures:	n = 5	n = 10
CONFID	3.80 (2.049)	6.50 (0.500)*
ENJOY	4.00 (1.870)	6.60 (0.418)*

*less than 1 in 20 that results are due to chance

Abbildung 1: Ergebnisse aus der Studie von Nosek 1998 [2]

3.2 Williams 2000: The Collaborative Software Process

In der bereits erwähnten Studie an der Universität von Utah 1999 von Laurie A. Williams geht es ebenso um die Entwicklungszeit und Qualität beim Einsatz von Pair Programming [3]. In ihrer Dissertation vergleicht sie den *Collaborative Software Process* (CSP) in Paaren mit dem bis dato anerkannten *Personal Software Process* (PSP) von Watts S. Humphrey. Dabei bildet Pair Programming den grundsätzlichen Unterschied zwischen beiden Prozessen und wird daher auch öfters synonym zu CSP verwendet.

Das Experiment bezog sich auf 41 Studenten aus einem Software Engineering Kurs, von denen 20 bereits im Sommer 1999 mittels Webprogrammierung Erfahrungen in CSP gesammelt hatten. Alle Studenten wiesen Erfahrungen aus Praktika und/oder großen Projekten auf und hatten 2-3 Jahre Erfahrung in C++. Zu bewerkstelligen waren vier verschiedene Aufgaben innerhalb von sechs Wochen. Die Aufteilung geschah in ein Drittel, die alleine arbeitete, und den restlichen zwei Dritteln in Paaren. Um den Arbeitsaufwand gleichmäßig zu gestalten, erhielten die Paare stets eine Aufgabe zusätzlich.

Die Ergebnisse bestätigten im Allgemeinen die Studie von Nosek zuvor. Durch automatisierte Tests wurde eine 15% niedrigere Defektrate bei Paaren ermittelt als bei der Kontrollgruppe. Ebenso war die Konsistenz der Daten gegeben, da die Varianz bei Paaren deutlich geringer war. Dies bestätigt die vorher beschriebene These, dass die Qualität aus Pair Programming genauer vorhersagbar ist.

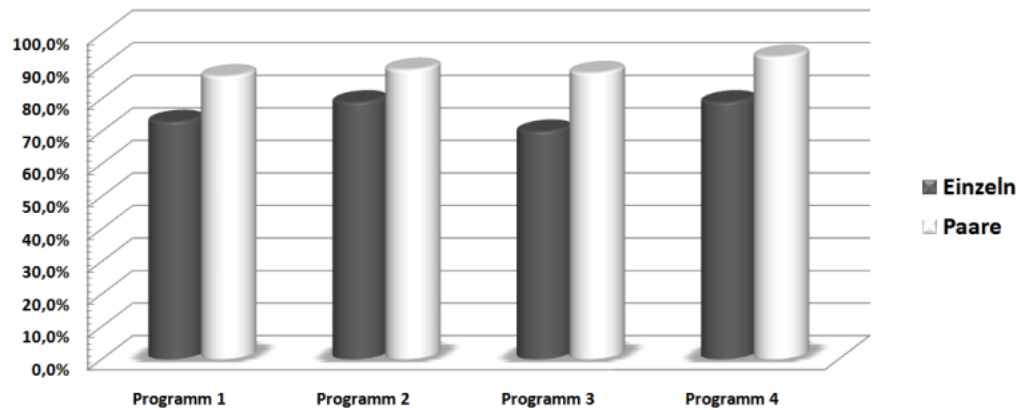


Abbildung 2: bestandene Testfälle [3]

Die Befürchtung, dass durch Pair Programming 100% mehr Personenstunden investiert werden, wurde entkräftet. Bei der benötigten Arbeitszeit wiesen die Paare im Schnitt lediglich 15% mehr Zeit für ein Programm auf als ein Einzelner. Da jedes Paar noch eine Zusatzaufgabe gestellt bekam, kann geschlussfolgert werden, dass eine Person innerhalb eines Paares deutlich effektiver war als ein Einzelprogrammierer. Ebenso ist die Jelling Time gut erkennbar, die von Programm 1 zu Programm 2 um 21% sank. Für die Bewertung ging die vierte Messung wegen starker Schwankungen aufgrund der Abgabe eines weiteren Projekts nicht ein.

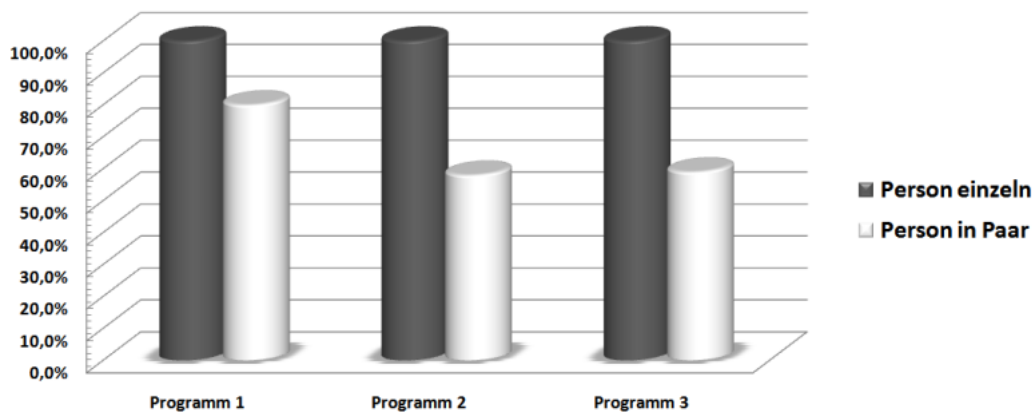


Abbildung 3: relative benötigte Arbeitszeit [3]

Eine nebenbei durchgeführte Online-Befragung bestätigte auch die Ergebnisse von Nosek bezüglich der Zuversicht zur eigenen Leistung und der Freude am Prozess.

3.3 Katira, Williams et al. 2004: On Understanding Compatibility of Student Pair Programmers

Durchaus wird in Williams Studie auch angemerkt, dass Pair Programming nicht generell funktioniert, sondern nur wenn sich die Beteiligten entsprechend verhalten. Für eine wohlgesonnene Kommunikation und reibungslosen Funktionsablauf müssen gewisse Regeln und Verhaltensweisen eingehalten werden. In einem weiteren Artikel wird dieser Verhaltenskodex zusammengefasst und veröffentlicht [8]. Eine weitere Frage, die sich auch stellt, ist in wie weit man Paare zusammensetzen soll und ob eine gewisse Korrelation zur *Kompatibilität* von Paaren existiert. In einem 2004 durchgeführten Experiment an der North Carolina State University (NCSU) wurde diese Fragestellung anhand von Studenten beleuchtet. Dabei ergab sich eine hohe Datenbasis von 1003 Erstsemestlern in einem Einführungskurs zu Java (CS1), weiteren 496 Studenten in einer Software Engineering Veranstaltung (SE) und 64 Absolventen in einer Veranstaltung zur objektorientierten Programmierung (OO). Vier Hypothesen wurden aufgestellt, die jeweils überprüft werden sollten:

Paare sind verträglicher, wenn Studenten

- H-1: mit unterschiedlichen **Persönlichkeitstypen** nach dem *Myers Briggs Personality Type* [14] gruppiert werden
- H-2: mit ähnlichen **tatsächlichen Leistungsniveaus** gruppiert werden
- H-3: mit ähnlicher **wahrgenommener technischer Kompetenz** gruppiert werden
- H-4: mit ähnlichem **Selbstwertgefühl** gruppiert werden

Zusätzlich war es für jeden Studenten verpflichtend eine webbasierte Evaluation über ihren Partner durchzuführen, um zu ermitteln ob die Leistung beiden oder nur einem Mitglied anzurechnen ist.

Durch die quantitativen Ergebnisse konnten zwei Schlussfolgerungen gezogen werden. Zum einen waren die Paare studentenübergreifend meist verträglich, was durch die Aussagen der Teilnehmer bestätigt wurde. Lediglich 10% der Paare zeigten Inkompatibilitäten auf, sodass man daraus ableiten kann, dass Paare in der Regel zufällig gruppiert werden können ohne Rücksicht auf Persönlichkeitstyp, Leistungsniveau oder Selbstwertgefühl.

Class	N	Very Compat.	OK	Not Compat.
CS1	1003	63% (633)	26% (264)	11% (106)
SE	496	65% (324)	27% (132)	8% (40)
OO	64	72% (46)	19% (12)	9% (6)

Abbildung 4: Kompatibilität der Studenten im Überblick [4]

Andererseits wollte man auch ermitteln, welche der oben genannten Faktoren die Verträglichkeit von Paaren verbessert. So zeigte sich nach Hypothese H-1 die Gruppe CS1 verträglicher bei unterschiedlichen Persönlichkeitstypen, was in der Gruppe SE nicht zum Vorschein kam. Durch eine Messung anhand des tatsächlichen Leistungsniveaus (H-2) zeigte sich die Gruppe OO verträglicher, wenn das Leistungsniveau in etwa identisch war. Dies konnte nicht für CS1 und SE bestätigt werden. Durch die Webbefragung konnte die wahrgenommene technische Kompetenz innerhalb der Paare ermittelt werden. So zeigten alle drei Testgruppen positive Ergebnisse, wenn die wahrgenommene Kompetenz des Partners besser eingeschätzt wurde als seine eigene. Zur Bewertung des Selbstwertgefühls konnte lediglich die Gruppe CS1 beitragen, wobei die aufgestellte Hypothese nicht belegt werden konnte. Abb. 5 zeigt die Ergebnisse der Studie nochmal in zusammengefasster Form.

	Hypothesis Pair are more compatible if students with ...	CS1	SE	OO
H-1	... different personality type are grouped together	Yes	No	
H-2	... similar actual skill level are grouped together	No	No	Yes
H-3	... similar perceived technical competence are grouped together	Yes	Yes	Yes
H-4	... similar self-esteem are grouped together	No		

Abbildung 5: Zusammenfassung der Ergebnisse [4]

4 Kritik

4.1 Pair Management

Obwohl die Studie von Katira, Williams et al. Ergebnisse zur Kompatibilität von Paaren hervor gebracht hat, bleibt die Frage der Zusammensetzung der Paare außerhalb von Studenten weiterhin ein großes Problem. Es gibt zudem noch eine Reihe weiterer Fragestellungen, um die sich das *Pair Management* zu kümmern hat: Wie oft und in welchen Zeitabständen sollte man innerhalb von Paaren rotieren? Wann sollen zwischen Paaren rotiert werden? Was macht man, wenn man eine ungerade Anzahl an Personen hat? Was sollte man unternehmen, falls Konflikte entstehen (z.B. eine extrem ordentliche Person arbeitet mit einem chaotischen Typ zusammen oder Personen, mit denen niemand gerne arbeiten will)? Wie soll die Logistik verwaltet werden, wann Paare zeitlich anfangen und aufhören zusammenzuarbeiten?

Diese und andere Fragen, wie man effektiv Paare zusammensetzt, stellen sich Ally, Darroch und Toleman in ihrer Delphi Studie zu Pair Programming [15]. Sie kommen zum Schluss, dass Paare nicht zufällig, sondern sorgfältig ausgewählt werden müssen, da viele Kombinationen nicht funktionieren. Während ein Anfänger einen Experten verlangsamen kann, können zwei Experten durch verschiedene Lösungsansätze ins Geklirre kommen ("*clashing of the minds*") und zwei Anfänger keine Vorteile aus Pair Programming ziehen ("*the blind leading the blind*").

4.2 Angsterzeugende Umgebung

Gelegentlich zeigen sich Programmierer auch beängstigt gegenüber der anderen Person ignorant oder inkompetent zu erscheinen. Man fürchtet sich, dass gewisse individuelle Schwächen im Entwicklungsprozess enthüllt werden [15]. Auf der anderen Seite, kann eine ständige Defektmeldung vom Navigator die Moral des Drivers absenken [15].

4.3 Wissensaustausch

Obwohl der kollektive Wissensaustausch bei Pair Programming förderlich ist, zeigen sich hier auch Kritikpunkte. So kommt es unweigerlich zum Problem, wenn Leute mit großem Ego nicht bereit sind ihr Wissen mit anderen zu teilen oder anfangen andere Leute abzustempeln.

Bezüglich der Lerneffekte kann es bei der Gruppierung von Anfängern mit Experten passieren, dass eine zu exzessive Beratung zu einer Ausbildung ausartet. Weist der Experte ständig an ohne dass der Anfänger Vorschläge einbringt, so wird aus dem Pair Programming eine praktische Schulung ("*Mentoring becomes Training*") [15].

Mit der zuvor erwähnten Collective Code Ownership wird die Verantwortung für eine Komponente auf mehrere Personen verteilt. Dies ist problematisch, wenn eine Verantwortung eindeutig vergeben werden muss oder wenn eine der beiden Personen aus der Entwicklung ausscheidet.

4.4 Widerstände und Grundeinstellungen

Als Vorurteil gegenüber Programmierer wird der einsame Hacker ins Spiel gebracht. In der Tat wurde Programmierung lange Zeit als alleinstehende Aktivität gesehen und viele erfahrene Programmierer sind heutzutage immer noch widerwillig ihre Arbeit mit anderen Personen zu teilen. Genauso zeigen sich auch Manager widerwillig Pair Programming einzusetzen, da sie davon ausgehen doppelte Personenstunden zu benötigen und somit Ressourcen unnötig zu verbrauchen.

Pair Programming hat in vielen Fällen gezeigt, dass diese Befürchtung widerlegt und die Einstellung umgedreht werden kann.

4.5 Ungeeignete Situationen für Pair Programming

An manchen Stellen ist Pair Programming tatsächlich eher ungeeignet. So lohnt es sich nicht während eines Prozesses mittels neuen Mitarbeitern auf Pair Programming umzusteigen, wenn ein kritischer Liefertermin einzuhalten ist. Nach Brook's Gesetz wirken sich die neuen Arbeitskräfte negativ aus bezüglich der Termineinhaltung. Pair Programming kann demgegenüber entgegenwirken, es aber nicht ausser Kraft setzen [16]. Ebenso kann Pair Programming nicht auf neue Mitarbeiter angewandt werden, wenn man nicht die Ressourcen und die Freiheit dazu hat.

Erfahrenere Paarprogrammierer priorisieren Pair Programming in der Praxis im Bezug auf Analyse und Design im Vergleich zur Implementierung. So wird z.B. für die Realisierung der GUI in der Analyse und im Entwurf im Paar gearbeitet, während für die Implementierung, dem Zeichnen der GUI, eine zweite Person wenig hilft [3]. Dies gilt auch für künstlerische oder andere Tätigkeiten, die sehr ins Detail gehen.

5 Zusammenfassung

Pair Programming hat sich in vielen Studien hinsichtlich Softwarequalität und Entwicklungszeit als geeignet erwiesen. Dies wurde durch etliche Experimente, die jedoch meist im universitären Umfeld stattfanden, belegt. Ein Entwicklungsprozess in Paaren weist verschiedene positive Faktoren auf, die zu diesen Ergebnissen führen. Überdies steigert Pair Programming auch die Zuversicht in die eigene Entwicklung und die Freude an der Arbeit. Des Weiteren bietet es einen

einfachen Weg neue Mitarbeiter einzuschulen und Teamwork in der Belegschaft zu animieren. Als Ergebnis tritt eine durchdachte Lösung mit niedriger Defektrate hervor. Die einzigartigen Lerneffekte bieten sich an, Pair Programming als generelle Praktik in der akademischer Lehre anzuwenden.

Zur erfolgreichen Durchführung bedarf es jedoch der Einhaltung bestimmter Regeln und die Beachtung einiger kritischer Punkte, wie die Zusammensetzung der Paare oder den Widerständen der verschiedenen Persönlichkeiten. Zum reibungslosen Ablauf spielt daher auch eine disziplinierte extensive Kommunikation eine wichtige Rolle. Erst im unablässigen Dialog zwischen Entwicklern kann Pair Programming seine positiven Auswirkungen entfalten. Insbesondere das Management der Paare im wirtschaftlichen Umfeld bietet offene Fragen für weitere Studien.

Nicht weiter eingegangen wurde auf die Thematik der verteilten Paarprogrammierung (*Distributed Pair Programming*), welches heutzutage aufgrund von Outsourcing stärker in den Blickpunkt wissenschaftlicher Arbeit gerückt ist. In dieser Form arbeiten die Entwickler in Paaren in räumlich getrennten Lokalisationen über diverse Werkzeuge zusammen. Diese Werkzeuge, wie Sangam [17] oder Saros [18], sind demnach zuständig für eine übertragbare visuelle Darstellungsform und verbale Kommunikation. Somit wird die räumliche Begrenzung des Pair Programming aufgehoben, was jedoch auch neue Problematiken mit sich bringt.

Literatur

- [1] Gavriel Salomon. *Distributed Cognitions: Psychological and educational considerations*. Cambridge University Press, 1993.
- [2] John T. Nosek. The case for collaborative programming. *Communications of the ACM*, Volume 41(3):105–108, 1998.
- [3] Laurie A. Williams. *The Collaborative Software Process*. PhD thesis, University of Utah, 2000.
- [4] Neha Katira, Laurie A. Williams, Eric Wiebe, Carol Miller, Suzanne Balik, and Ed Gehringer. On understanding compatibility of student pair programmers. 2004.
- [5] Laurie A. Williams and Robert R. Kessler. *Pair Programming Illuminated*. Addison-Wesley, 2002.
- [6] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
- [7] Laurie A. Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair-programming. *IEEE Software*, Volume 17, Number 4:19–25, 2000.
- [8] Laurie A. Williams and Robert R. Kessler. All i really need to know about pair programming i learned in kindergarten. *Communications of the ACM*, 2000.
- [9] Alistair Cockburn and Laurie A. Williams. The costs and benefits of pair programming. 2001.
- [10] Gerald M. Weinberg. *The Psychology of Computer Programming*. Dorset House Publishing, 1998.
- [11] Gerardo Canfora, Aniello Cimitile, and Corrado Aaron Visaggio. Empirical study on the productivity of the pair programming. *Extreme Programming and Agile Processes in Software Engineering: 6th International Conference*, pages 92–99, 2005.
- [12] Jerzy Nawrocki and Adam Wojciechowski. Experimental evaluation of pair programming. 2001.
- [13] N. V. Flor and E. L. Hutchins. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. *Empirical Studies of Programmers: Fourth Workshop*, 1991.

-
- [14] David Keirse. *Please Understand Me II*. Prometheus Nemesis Book Company, 1998.
- [15] Mustafa Ally, Fiona Darroch, and Mark Toleman. A framework for understanding the factors of influencing pair programming success. *Extreme Programming and Agile Processes in Software Engineering: 6th International Conference*, pages 82–91, 2005.
- [16] Laurie A. Williams, Anuja Shukla, and Annie I. Antón. An initial exploration of the relationship between pair programming and brooks' law. 2004.
- [17] Chih wei Ho, Somik Raha, Edward Gehringer, and Laurie A. Williams. Sangam - a distributed pair programming plug-in for eclipse. 2004.
- [18] Riad Djemili. *Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung*. PhD thesis, Freie Universität Berlin, 2006.