Promotionsvorhaben: Mikroprozessanalysen von Fehlerverhaltensmustern beim Programmieren

Sebastian Jekutsch

22. Dezember 2006

Dieses Exposé stellt meinen Forschungsplan zum Thema „Mikroprozess des Programmierens“ zur Vermeidung von Programmierfehlern vor. Es beschriebt damit mein Promotionsvorhaben und die angestrebte Form der Dissertation.

Nach einer kurzen Typisierung von Programmierfehlern wird dargelegt, warum ohne Kenntnis der Programmierabsicht nur Beanspruchungsfehler überhaupt analysierbar sind. Diese sind dann jedoch – das ist eine der vier zentralen Hypothesen – am Programmierverhalten, repräsentiert im Mikroprozess, als wiederkehrendes Muster erkennbar. Eine systematische Untersuchung wird einige Verhaltensmuster aufdecken, deren Einordnung als fehlerinduzierend, fehlerbegleitend oder irrelevant jedoch letztlich nur der Programmierer selbst in seiner konkreten Situation unternehmen kann. Es soll daher ein Werkzeug konzipiert werden, welches dem Programmierer die Möglichkeit gibt, seinen Mikroprozess von ihn interessierenden Phasen seiner Arbeit nachträglich reflektierend zu betrachten.

Diesen Hauptargumentationsstrang findet man im Kapitel 1. Kapitel 2 mit der Forschungsfrage ist eine konsequente Folgerung daraus. Nach der Vorstellung verwandter Arbeiten in Kapitel 3 wird in Kapitel 4 ein möglichst realistischer Plan entwickelt, dem eine Erwähnung der schon geleisteten Arbeiten (dort auch erste Ideen und Ansätze, inklusive Veröffentlichungen) vorangeht.

Inhaltsverzeichnis

1 Motivation

In meiner Promotion sollen Programmierfehler im Mittelpunkt stehen, genauer: Die Möglichkeiten zur Werkzeugunterstützung zwecks Vermeidung von Programmierfehlern. Das zentrale Mittel zur Erreichung dieses Ziels ist die kontinuierliche, semi-automatische Bewertung der persönlichen Programmiertätigkeiten. Der Ansatz setzt dabei auf eine umfassendere Theorie und Praxis von Tätigkeiten während der Softwareentwicklung.

Die Unvollständigkeit der Aufzeichnungen sowie schwer überwindbare Schwierigkeiten in der *Beurteilung* der Aufzeichnungen einerseits und die jeder Fehleranalyse inhärente Individualität und Situationalität andererseits führen zu einem speziellen Lösungsansatz: Die *persönliche* – aber ebenso systematische – Analyse *typischer, einzelner* Fehlerursachen, Fehlverhaltensweisen oder Fehlervermeidungsstrategien.

Dieses Kapitel eröffnet den Promotionsvorschlag mit einer Begründung der in Kapitel 2 schließlich festgelegten Forschungsfrage. Zwecks Lesbarkeit gibt es hier keine internen oder externen Querverweise und Referenzen. Es soll einfach lesbar sein, motivieren und hat deshalb keine wissenschaftliche Stringenz.

1.1 Ausgangspunkt

Qualitäts-sicherung

Ein wichtiges, wenn nicht das zentrale Thema der Softwaretechnik ist die Qualität: Man möchte unter gegebenen Bedingungen (Ressourcen an Zeit, Geld und Arbeitskraft) Software mit bestimmten Qualitätseigenschaften herstellen. Neben der Funktionalität ist der zentrale Qualitätsaspekt die Zuverlässigkeit und darin vor allem die Reife der Software, also die Frage, ob eine Software das tut, was ihr auch „beigebracht“ werden sollte.

analytische QS

Dabei konzentriert sich die Qualitätssicherung in der Softwareentwicklung vor allem auf reagierende Aktivitäten: Das Testen oder Durchsehen von Software passiert, *nachdem* ein Softwaremodul oder eines seiner Vorgängerdokumente erstellt wurde. So wertvoll diese *analytische* Qualitätssicherung vor allem in der Praxis ist, so offensichtlich ist auch, dass auf Dauer eine *konstruktive* Qualitätssicherung anzustreben ist, d.h. prozessuale Vorkehrungen zu treffen, damit Qualitätsmängel erst gar nicht entstehen.

konstruktive QS

Interessanterweise wurde die Theorie der konstruktiven Qualitätssicherung mit der Definition grundlegender qualitätssichernder Entwicklungsprozesse begonnen, vor allem diverser Phasenmodelle. Diese „großen“ Prozessmodelle nutzen als Einheit typischerweise ganze zu erstellende Prototypen oder Softwareversionen und organisieren dies in Zyklen von Wochen, meist Monaten, je nach Projektgröße.

Programmier-fehler

Die diesem Ansatz sich entgegen stellenden agilen Softwareprozesse brechen konstruktive Qualitätsverbesserungsvorschläge auf entwicklungsnahe Einheiten herunter, nämlich einzelne Anforderungen und kurze Zyklen. Sie bewegen sich damit in Richtung einer nicht zu unterschätzen Quelle von Qualitätsmängeln: Den Fehlern der einzelnen Akteure während der Durchführung ihrer Teilaufgaben, im speziellen Fall den Programmierfehlern, auf die ich mich in meiner Arbeit konzentrieren werde.

Programmier-tipps

Einige Tipps und Tricks zur Vermeidung von Programmierfehlern oder zur effizienteren Programmierung wurden schon vor langer Zeit entwickelt und – obschon selten empirisch belegt – allgemein als qualitätsverbessernd akzeptiert, z.B.:

- „Goto considered harmful“
- Geheimnisprinzip
- Design-by-Contract

kognitive Begründung

Letztlich wollen alle Tipps die kognitiven Schwächen des Programmierers überwinden, vor allem mittels Komplexitätsreduktion. Diese Komplexitätsreduktion ist produktbasiert, d.h. es werden das zu lösende Problem oder die daraus resultierenden Arbeitspakete beherschbar gemacht. Zudem sind die hier genannten Prinzipien überindividuell, d.h. nicht abhängig vom Programmierer sondern allgemein gültig.

Programmierprozesstipps

Neuere Prinzipien zeigen hingegen eine Tendenz zur Prozessorientierung, z.B.:

- Inkrementelles Vorgehen auch im Kleinen
- Peer Review und Paarprogrammierung
- Testgesteuerte Programmierung

Personal Software Process

Auch diese Vorschläge sind überindividuell und zudem situationsinvariant, d.h. in jedem Fall hilfreich und deshalb unspezifisch. Erst im Persönlichen Software Prozess (PSP^{SM}) wird auf individuelle Prozesse eingegangen. Die Grundidee des PSP^{SM} ist die persönliche Aufzeichnung bestimmter beim Programmieren auftretender Ereignisse oder Erkenntnisse wie Defekteinbauten, Defektbehebungen, Zeitverbrauch oder Umfang von Arbeiten, etc. Diese dienen der systematischen Ermittlung von Statistiken über die persönliche Leistung, d.h. Stärken und Schwächen unter gegebenen Umständen. Beim PSP^{SM} geht es also um individuelle, programmiernahe, konstruktive Qualitätssicherung, dem Hauptthema dieser Arbeit.

1.2 Fehlerursachen

Programmierfehlertypen

Eine Analyse in Art des PSP^{SM} ermöglicht es, über allgemeine Tipps hinausgehende Prozessverbesserungen zu motivieren. Denn die Ausprägungen und Ursachen der Fehler, die ein Programmierer macht, sind äußerst verschieden. Ein Programmierfehler ist auf jeden Fall ein Falschtun des Programmierers. Dieses kann aus kognitiver Sicht zwei genotypische Ursachen haben:

Irrtum

Irrtum Der Programmierer meint, das richtige Wissen zu haben, aber es ist falsch. In voller Überzeugung macht er einen Programmierfehler. Diese Fehler werden *Irrtumsfehler* genannt. Beispiel: Die Bedeutung eines Methodenparameters wird missverstanden.

Beanspruchung

Beanspruchung Der Programmierer ist während der Arbeit überfordert, was ihn zu Fehlern aller Art provoziert, obschon er – bessere Umstände und reichlich Zeit angenommen – durchaus den Fehler hätte verhindern können. Das Wissen ist also prinzipiell vorhanden, nur die Anwendung des Wissens oder die Durchführung dieser Anwendung war falsch. Eine solche Beanspruchung kann wiederum zwei Ursachen haben:

Überforderung Das Problem ist an sich zu schwer für ihn, vielleicht weil es neu ist oder zu kompliziert. Dies mag ihm bewusst sein, die entstehenden Fehler jedoch nicht. Diese Fehler werden *Überforderungsfehler* genannt. Beispiel: Der Programmierer sorgt inmitten eines sehr komplizierten Algorithmus für einen Feldüberlauf bei bestimmten Eingabeparametern. Überforderung

Belastung Die Umstände (Ablenkungen, Müdigkeit u.ä.) belasten den Programmierer, so dass seine Aufmerksamkeit für das Programmierproblem sinkt. Er bemerkt auch hier die Fehler nicht, nennt sie aber im Nachhinein oft „dumm“, weil er es eigentlich besser gewusst hätte. Diese Fehler werden *Belastungsfehler* genannt. Beispiel: Der Programmierer vergisst wegen einer plötzlichen Unterbrechung den Index in einer Zählschleife hochzuzählen. Belastung

Auf diese Weisen machen sich auch Ineffizienzen bei der Arbeit bemerkbar, also eine nur langsame Erkenntnis des „richtigen Weges“. Im Grunde ist das Fehlermachen und das später notwendige Beheben eines Defekts nur eine Spezialform von Ineffizienz, also nicht optimaler Zeitnutzung. Zeitverlust

Das Erkennen eines Irrtums ist nur dem Eingeweihten möglich, also jemandem, der das „Richtig“ kennt zu dem was falsch gedacht wurde. Dieser Jemand kann zum Zeitpunkt des Fehlermachens nicht der Programmierer selbst sein, sonst hätte er den Fehler nicht gemacht. Da ich aber eine Fehleranalyse ohne allwissendes Orakel anstrebe, geraten Beanspruchungsfehler in den Fokus meiner Arbeit: Diese können nur entstanden sein aufgrund mangelnder Ausführung. Fehlendes Orakel

Meine Behauptung ist, dass sich beim Programmieren Beanspruchungsfehlersituationen durch auffällige (evtl. persönliche) *Verhaltensweisen* bemerkbar machen. Dies ist die erste und zentrale Hypothese meiner Forschungsarbeit. Hypothese 1

Das Besondere ist, dass diese Verhaltensweisen unabhängig von dem zu lösenden Problem sind. So wie man z.B. einen unsicheren (also unfallträchtigen) Autofahrer an seinem Verhalten erkennen kann, ohne zu wissen wo er hinfahren will und soll, so gibt es nach Hypothese 1 typische Verhaltensweisen beanspruchter Programmierer. Unnötiges Orakel

1.3 Verhaltensmuster

Vier Arten von Verhaltensweisen sind im Hinblick auf Fehler zu unterscheiden:

Falschverhalten Ein Verhalten, das selbst schon der Fehler ist. Dies ist beim Programmieren praktisch nie verallgemeinernd beschreibbar. Wenn, dann dürfte meistens ein syntaktischer oder durch statische Analyse entdeckbarer Defekt vorliegen. Ein Beispiel ist das Tippen eines '=' statt eines '==' in dem Bedingungsteil einer if-Anweisung. Falschverhalten

Fehlverhalten Ein Verhalten, das belastend wirkt. Dies ist ein meist ein stereotypisches Verhalten, welches nur allzu leicht ein Falschtun provoziert, etwa ein stupides, großflächiges Ändern des Codes oder das Offenhalten vieler „Baustellen“ während der Programmierung. Fehlverhalten

Symptomverhalten Der Weg zu den Fehlern kann auch über die Symptome führen, d.h. man beobachtet nicht das Falsch- oder Fehlverhalten, sondern typische begleitende Symptome. Es gibt leider keine Untersuchung in diesem Bereich für das Programmieren; plausibel erscheinen z.B. häufiger Wechsel des aktuellen Arbeitsfokusses, viele Tippfehler, langsames Kodieren, Ausfluchttätigkeiten, etc. Symptomverhalten

Bewältigungsverhalten Die Symptome sind schwer von den „Therapien“ zu trennen, also Verhaltensweisen, die aktuelle Probleme bewältigen lassen, wie z.B. verstärktes Lesen in der Dokumentation, häufigeres Testen des Codes oder auch Trial-and-Error-Strategien. Dies wird gemacht, *weil* der Programmierer überfordert ist, meist zwecks (zumindest vorübergehender) mentaler Entlastung, seltener auch als tatsächlicher Stategiewechsel, um einen anderen Lösungsweg einzuschlagen. Bewältigungsverhalten

Dabei sind reine Irrtumsfehler nicht aufgrund des Verhaltens zu beurteilen, denn sie sind kaum zu trennen von Nicht-Fehlern: in beiden Fällen agiert der Programmierer überzeugt von seinem Tun und somit ähnlich. Desweiteren ist oft das Nicht-Tun ein Fehler, was ebenfalls schwerlich als Verhaltensweise auffällt. Auffälliges

Was jedoch interessant ist, sind auffällige Verhaltens*änderungen*, welche meist ihre Gründe in Symptom- oder Bewältigungsverhaltensweisen haben. Ein Fokus auf Änderungen impliziert eine automatische Kalibrierung auf einen Programmierer und seinen normalen Prozess: Verhaltensweisen können sich zwischen Programmierern deutlich unterscheiden; wenn sich ein Programmierer aber auffällig anders verhält als üblich, dann muss dies einen Grund haben, z.B. geänderte Beanspruchung. Verhaltensänderungen

Dass solcherlei Verhaltensweisen beim Programmieren existieren und auch stereotypisch genug sind, dass man von wiedererkennbaren Programmierfehlerverhaltens*mustern* reden kann, ist die zweite Hypothese dieser Arbeit. Die Arbeit wird sich vor allem um Symptomverhaltensweisen kümmern, die Indikator für problematische Kodierphasen sind. Diese müssen scharf getrennt werden von Bewältigungsverhaltensweisen, die - da schon ein Weg zur Besserung - eher ein Zeichen sind, dass *keine* Fehler auftauchen werden. Hypothese 2

1.4 Mikroprozess

Zur Beschreibung der Verhaltensmuster soll der Begriff des „Mikroprozesses des Programmierens“ (kurz μPP) dienen. Er beschreibt die einzelnen Tätigkeiten (z.B. Codeänderungen, Testen, Nachschlagen, Fragen, usw.) des Programmierers während seiner Arbeit. Da „Prozesse“ in der Softwaretechnik meistens vorschreibend gemeint sind, muss betont werden, dass hier der *tatsächliche* μPP gemeint ist. μPP

Der μPP kann zu einem nicht unerheblichen Teil von außen beobachtet werden; er kann sogar als Ereignisfolge in einigen Teilen automatisch aufgezeichnet werden in Form einzelner Arbeitsschritte, z.B. bei einer Entwicklungsumgebung. Diese Aufzeichnung soll unbedingt nicht unterbrechend, aufdringlich oder sonstwie bemerkbar sein, um den Programmierer nicht zusätzlich zu belasten und die Akzeptanz nicht zu gefährden. μPP-Aufzeichnung

Da der μPP sehr grundlegende Aktionen des Programmierers beschreibt, ist eine Zu- μPP-Analyse

sammenfassung der μPP-Schritte notwendig. So könnte eine Aktionsfolge wie z.B. (Programmiersprache Java)

1. Einfügen einer Anweisung mit `System.out.println` an Stelle x
2. Laufenlassen des Programms
3. Durchlaufen der Stelle x im Bytecode
4. Beenden des Programms
5. {Beliebige andere Aktionen, außer Schritt 6}
6. Löschen der `println`-Anweisung in x

als ein „Debugging mit Testausgabe“ bezeichnet werden. Eine solche Zusammenfassung von μPP-Ereignissen nennt man eine *Episode*. Dabei kann einer Episode durchaus auch andere Episoden zugrunde liegen. Verhaltensweisen im obigen Sinne sind aus μPP-Sicht also nichts anderes als Episoden, welche aus Basisereignissen gebildet werden, andersherum: In den Basisereignissen sind solche Verhaltensmuster erkennbar.

Es ist die dritte Hypothese, dass eine systematische Untersuchung von Verhaltensmustern im Rahmen einer Programmierfehlerforschung auf Basis eines automatisch aufgezeichneten und geeignet verdichteten μPP möglich ist. Hypothese 3

Dies hat aber Grenzen. Zur Verdeutlichung ist es hilfreich, die *μPP-Analyse*, d.h. das Interpretieren und Bewerten des Mikroprozesses, anhand der folgenden Ebenen zu beschreiben: Analyseebenen

Metriken über den μPP stellen einfach und kontinuierlich berechen- und beobachtbare Maße dar: Häufigkeiten, Größen oder Zeitdauern, z.B. wie schnell der Programmierer tippt, wie häufig er die Datei wechselt oder wie viele Tests er macht. Basierend auf mehr oder weniger starken Korrelationen nutzt man solche einfachen Metriken in der Softwaretechnik gerne als Indikator für interessierende, aber schwerer zu messende Eigenschaften, in unserem Fall z.B. die Beanspruchung des Programmierers oder gar direkt die Fehlerhaftigkeit seines Tuns. Dazu muss der Zusammenhang „lediglich“ empirisch belegt werden. Der aufgezeichnete μPP bietet reichlich Möglichkeiten, Metriken zu berechnen. Metrik

Sequenzen beschreiben im Gegensatz zu den skalaren Metriken Abläufe in der Zeit, hier also Abfolgen von direkt oder nicht direkt aufeinander folgenden Tätigkeiten, d.h. Ereignissen. Tauchen solche Sequenzen häufiger in der gleichen Weise auf, werden sie Muster genannt. Sequenz Muster

Zweierlei Arten von Tätigkeitessequenzen sind interessant:

Praktiken bezeichnen Vorgehensweisen zur Erlangung eines Ziels. Es sind Muster-artige Abfolgen von Tätigkeiten, die ein Programmierer einstudiert hat und häufig anwendet, weil er sie für erfolgreich hält. Praktiken können auch unbewusst vorhanden sein. Häufig hat jeder Programmierer seine eigenen im Laufe der Zeit erworbenen Praktik

Praktiken entwickelt; es gibt aber auch einige allgemein anzutreffende Praktiken. Beispiel: Wie kurz sind die Iterationen des Programmierers, d.h. programmiert er lange und testet am Ende oder testet er immer wieder selbst nach kleinen Änderungen? Interessant sind vor allem auch Praktik*änderungen* und -anpassungnen und deren Anlässe, in unserem Fall z.B. aufgrund erhöhter Belastung durch Zeitdruck.

Strategie

Strategien beschreiben im Vergleich zu Praktiken das *warum* und nicht das *was*. Sie sind Problembezogen: Sie beschreiben die Abfolge der Teilziele des Programmierers, seinen Plan von einem Ausgangspunkt zum gewünschten in Ferne liegenden Zielpunkt zu gelangen. Beispiel: Erst das Programmgerüst mit der Ein- und Ausgabe erstellen und danach die nötige komplizierte Berechnung dazwischen. Die Strategie gibt wertvolle Hinweise zu Programmierfehlern. So kann vermutet werden, dass gerade an Grenzen zwischen zwei erreichten Teilzielen (in Form von Programmmodulen) sogenannte Schnittstellenfehler entstehen. Strategien sind meistens bewusst geplant. Sie sind längerfristiger als Praktiken. Praktiken sind das Mittel, um den Zweck der Strategie zu erfüllen. Strategien werden nicht selten im Laufe der Arbeit geändert. Da Programmieraufgaben meistens neue Aufgaben sind, tauchen Strategien zudem eher selten als Muster auf.

Semantische Lücke

Hypothese 3 behauptet, dass Praktiken im μPP erkennbar sind. Diese Arbeit wird sich vor allem mit *Praktiken* beschäftigen, was sie am deutlichsten von ähnlichen Arbeiten absetzt. In dieser Arbeit wird nämlich angenommen, dass man grundsätzlich die Strategie *nicht* allein auf Basis des μPP erkennen kann. Der μPP kann zwar teilweise wiederspiegeln, was der Programmierer tat, aber kaum, was er tun wollte, denn das kognitive Konzept eines *Teilziels*, das der Programmierer mit seinem Tun versucht zu erreichen (also der Kontext, in dem sein Handeln statt findet) lässt sich nicht *umkehrbar* im μPP ausdrücken. Am obigen Beispiel: Im Schritt 1 kann auch eine Anforderung bzgl. der Ausgabe auf der Konsole prototypisch erfüllt worden sein, die in Schritt 6 halt wieder entfernt wird, weil sie danach auf die eigentlich geforderte Weise erfüllt wurde. Oder andersherum kann das Fehlen des Schritts 6 trotzdem eine „Debugging mit Testausgabe“-Episode zulassen, denn in der Tat wird nicht selten vergessen, eine solche Ausgabe am Ende zu entfernen. Ich nenne diese wichtige Erkenntnis im folgenden die *semantische Lücke*. Sie taucht auf ganz ähnliche Weise in der modellgetriebenen Softwareentwicklung als auch bei vielen Problemen der Künstlichen Intelligenz auf.

Aufzeichnungslücke

Ein anderes Problem ist die Lücke in den Aufzeichungen. Wo blickt der Programmierer gerade hin? Diskutiert er mit seinem Kollegen über das Programmierproblem oder über den gestrigen Kinofilm? Editiert er mit dem fremden Programm relevante Quelldateien? Es kann letztlich nicht alles aufgezeichnet werden, was man gerne hätte, vor allem unter dem Anspruch, den Programmierer bei der Arbeit damit nicht zu stören.

Ein Lösungsansatz für diese beiden Lücken – die Hypothese 3 stark einschränken – besteht darin, dass der Programmierer selbst an der μPP-Analyse beteiligt wird.

1.5 Reflexion

Reflexion

Lernen kann man aus Fehlern nur, wenn man eine Rückmeldung bekommt. PSP^{SM} ist ein

wesentlicher Fortschritt im Bereich der konstruktiven Qualitätssicherung, weil es *individualisiert* ist und sich damit prinzipiell nicht nur typische (d.h. regelmäßig wiederkehrende) Irrtümer, sondern auch die auf Belastung basierenden Fehler untersuchen lassen. Der Grundgedanke ist dabei, dass der Programmierer aus seinen Fehlern lernt, indem er die Ursachen und Umstände aufspürt und diese in Zukunft versucht zu vermeiden oder zumindest frühzeitig zu erkennen. Es ist somit ein Prozess, der auf *Reflexion* beruht. Reflexion bietet nicht nur das menschliche Urteil über Richtig und Falsch, sondern auch eine Spezialisierung auf individuelle Bedürfnisse und Eigenarten.

PSP^{SM}-Probleme

PSP^{SM} hat jedoch zwei Probleme: Es ist ein manuelles Verfahren und der Aufzeichner ist gleich dem Programmierer, d.h. es findet nicht nur eine weitere Belastung durch das Aufzeichnen statt (das sich leider nicht unmittelbar bezahlt macht), sondern die Aufzeichnung ist im Falle sowieso schon hoher Belastung nicht mehr oder nur mit Lücken und Mängeln machbar. Sowieso ist der Gedankenwechsel von einer Problemlösungsebene (Programmieren) in eine Prozessebene (Reflexion) nicht leicht und bedarf viel Übung.

μPP-Präsentation

Die Reflexion des Programmierers findet daher idealerweise *nachträglich* statt. Dafür bedarf es aber einer Unterstützung zur Wiedervorlage des vergangenen μPP, welche – wenn nicht ein Kollege aushilft – automatisiert sein muss. Wegen der semantischen Lücke lässt sich allerdings eine PSP^{SM}-ähnliche Reflexion nicht vollständig automatisieren. Der relevante μPP muss dem Programmierer also präsentiert werden (z.B. visuell), und es wird auch nötig sein, dass er sich „seinen Teil“ hinzudenkt.

Hypothese 4

Hypothese vier dieser Arbeit ist, dass der μPP dem Programmierer soweit hilfreich präsentierbar ist, dass er mit zusätzlichen Interpretationen reflektierend aus seinen Fehlerverhaltensmustern lernen kann. Zumindest ist dies für einzelne Verhaltensmuster und Situationen jeweils auf ihre Weise gut möglich. Es wird also explizit nicht die allgemeine Lösung angestrebt.

1.6 Werkzeuge

Werkzeugunterstützung

Abgesehen davon, dass – Hypothese 4 einschränkend – eine interaktive Fehlerverhaltensreflexion nur für bestimmte Muster möglich erscheint, kann eine Werkzeugunterstützung grundsätzlich mehrere, sich ergänzende Formen annehmen:

Mustererkennung

Erkennend hilft das Werkzeug bei der Erforschung des eigenen Verhaltens zur Entdeckung neuer, vielleicht persönlicher Verhaltensmuster oder einfachen Metriken. (Dieser Modus ist insbesondere auch für die empirische Forschung interessant.)

Bewusstmachung

Bewusstmachend wirkt das Werkzeug, wenn es vordefinierte Verhaltensmuster für den Programmierer sichtbar macht. Er kann damit feststellen, wann und wie oft er nach dem Muster handelte. Umgekehrt betrachtet kann ein Prozessbeobachtungssystem (also eins das kontrolliert, ob ein vorgeschriebener Prozess eingehalten wird) bewusstmachend für ein *Nicht*-Verhalten wirken.

Ursachenverdeutlichung

Verdeutlichend ist ein Werkzeug, wenn es bei einem Verhaltensmuster dessen Umstände verdeutlicht und damit beim Finden einer vermuteten Fehlerursache hilft. Dies enthält insbesondere die Bildung von Korrelationen zwischen Metriken, Verhaltensweisen und Ursachen und das Führen von Statistiken darüber.

Beratend agiert das Werkzeug, wenn es anhand erkannter Verhaltensmuster und Ursachen Prozessverbesserungen und Therapien vorschlägt. Beratung

Helfend unterstützt das Werkzeug den Programmierer schließlich bei der Durchführung bestimmter Verhaltensweisen. Dies ist offensichtlich nur bei Therapien ratsam, kann aber auch bei manchen Symptomen sinnvoll sein. Tätigkeitshilfe

Im Rahmen dieser Arbeit soll vor allem die Bewusstmachung im Mittelpunkt stehen.

In allen Fällen ist eine Kalibrierung des Werkzeugs (genauer: der den Mustern zugrunde liegenden μPP-Episoden) an den einzelnen Benutzer sinnvoll, vermutlich ratsam, vielleicht sogar notwendig. Bei der Mustererkennung ist dies ja sogar der eigentliche Inhalt. Kalibrierung

Dass das Werkzeug die Unterstützung des Programmierers selbst benötigt, um hilfreich zu sein, wurde schon in den Hypothesen 3 und 4 verdeutlicht. Sollte dies nicht nötig sein, vielleicht insbesondere nach einer umfangreichen Kalibrierung, so wäre auch eine Echtzeitunterstützung denkbar, d.h. das Werkzeug wird nicht mehr nachträglich eingesetzt, sondern gibt schon im laufenden Programmierprozess hilfreiche Hinweise. Dies wäre insbesondere für eine Bewusstmachung von großem Wert. Echtzeit

1.7 Fazit

Zur individuellen Fehleranalyse wäre es also schön,

- ein Werkzeug zu erstellen und zu evaluieren,
- das eine Nachbetrachtung des persönlichen μPPs
- für verschiedene Verhaltensmuster (speziell derjenigen in Folge erhöhter Beanspruchung)
- zwecks reflektierender Bewusstwerdung möglicher Fehlerursachen

ermöglicht. Der Programmierer soll aus seinen Fehlern einfacher und schneller lernen.

Tätigkeiten in Bereichen, die man vorher nicht genauso schon einmal unternommen hat, sind natürlicherweise fehlerbehaftet. Fehler gänzlich vermeiden zu wollen bedeutet, nicht kreativ und innovativ, sondern ängstlich zu sein. Je mehr man die Grenzen des Wissens und Könnens auslotet (z.B. in Situationen, in denen man lernt) desto eher berührt man diese Grenzen und macht folglich Fehler. Das ist nicht schlimm. Dumm ist nur, aus diesen Fehlern nicht zu lernen. In der Softwareentwicklung passiert dies vermutlich häufig.

Viele Tätigkeiten in der Softwareentwicklung werden alleine ausgeführt und viele Fehler finden einsam statt. Das Lernen aus Fehlern ist in dieser Situation schwer: Man muss sich beobachten können, um Fehlersituationen wiederzuerkennen. Dies ist bei einer so schwierigen Tätigkeit wie dem Programmieren aber nicht einfach, da man voll und ganz mit dem Problem beschäftigt ist. Jede Hilfe, die einem diese verdrängten Situationen bewusst machen kann, ohne selbst belastend zu wirken, ist eine wertvolle Hilfe.

Damit ist die „Story“ dieser Forschung abgesteckt.

2 Titel und Forschungsfrage

Hypothesen

Die Forschungsfragen ergeben sich aus der Betrachtung der vier in der Motivation genannten (und dort teilweise eingeschränkten) Hypothesen:

1. Relevante Programmierfehlerarten äußern sich im Verhalten.
2. Es gibt einen Musterkatalog typischer Verhaltensweisen im Fehlerfall.
3. Relevante Verhaltensmuster sind im Mikroprozess in Form von Episoden semiautomatisch erkennbar.
4. Diese Episoden sind dem Programmierer präsentierbar, so dass er über sein Verhalten reflektieren kann.

Annahmen

Die Hypothesen fußen auf einige in der Motivation implizit gemachte Annahmen, zum Beispiel:

- Programmierfehler sind teuer und lohnen ein Betrachtung und eine Reduzierung.
- Programmierfehler werden (unter anderem) aufgrund hoher Beanspruchung gemacht.
- Eine Reflexion über Programmierfehler hilft, diese in Zukunft zu vermeiden.

Diese Annahmen sollen nicht geprüft werden.

Fragen

Inhalt der Dissertation ist die Prüfung der Hypothesen bzw. das Erstellen und Evaluieren eines oder mehrerer spezieller Werkzeuge zwecks Unterstützung der Hypothesen.

Risiken

Die Hypothesen können falsch sein. Es gibt Warnungen aus zwei Richtungen und ein drittes Neuigkeitsproblem:

- In [109] wird gewarnt, dass ohne die Kenntnis des „deep structure knowledge“ jegliche Defekt- und Fehleranalyse oberflächlich (am „surface“) bleibt. Die Einschränkung der Hypothese 3 geht auf das Problem der sematischen Lücke ein, indem es postuliert, dass der Programmierer bei der Analyse mithelfen muss.
- Dass dem möglich wird zumindest für eine *Echtzeithilfe* durch den Benutzer in [51] widersprochen: PSP^{SM} lässt sich weder automatisieren, noch kann er durch den Benutzer unterstützbar ermittelt werden. Hypothese 4 formuliert daher, dass eine *nachträgliche* Präsentation angestrebt wird, also sozusagen zu Mußezeiten. Es bleibt abzuwarten, ob dies akzeptiert und genutzt wird.
- Eine Präsentation des Mikroprozesses wurde meines Wissens noch nie realisiert. Ähnliches aus anderen Gebieten ist mir leider unbekannt. Die Realisierung des Ansatzes wird daher anhand spezieller Situationen anstatt des Strebens nach einem allgemeinen Ansatz unternommen.

Szenarien

Der Nachweis über die Möglichkeit solcher Werkzeuge wird also an ausgewählten Verhaltensmustern und Fehlerszenarien verlaufen. Die Anwendungsfälle sind:

Metriken zur persönlichen μPP-Beurteilung Es wurden viele Mesoprozessmetriken entwickelt, die genauso, aber eben mit feinerer Granularität im Mikroprozess anwendbar sind: Änderunghäufigkeiten und -größen, Folge von Änderungen, Überlappungen von Änderungen, Frequenz von Arbeitsfokuswechseln, etc. Neben einer allgemeinen Evaluation derer Tauglichkeit kann ein Programmierer persönlich anhand von Stellenmetriken den Charakter von Stellen erkennen, bei ihm kritische Stellen leichter identifizieren und zudem Vergleiche mit seinem eigenen Verhalten in anderen Projekten und Zeiten anstellen. Eine spezielle Anwendung wäre das Erkennen von „Festgefahrensein“ anhand simpler Maße: Häufig kommt man an einer Stelle nicht weiter, probiert aber noch ein oder zwei Stunden herum, ohne voran zu kommen, gibt schließlich auf und kommt am nächsten Tag sofort auf den rettenden Einfall. Die zwei Stunden hätte man sich sparen können. Sind diese zwei Stunden anhand Mikroprozessmaßen erkennbar? μPP-Metriken

Testzyklus- und Ad-Hoc-Testverhalten Recht riskant scheinen lang andauernde Phasen zu sein, in denen der Programmierer wenig testet, weil das Programm nicht compilier- oder offensichtlich nicht lauffähig ist. Durch persönliche μPP-Analyse kann er herausfinden, wann er z.B. bei kurzen Testzyklen erfolgreicher arbeitet und sein Verhalten dementsprechend anpassen. Die Selbsterkenntnis kann dazu führen, dass er kurze Testzyklen als geeignetes Bewältigungsverhalten akzeptiert. Auch andere Testverhaltensweisen sind interessant, z.B. Versuch-und-Irrtums-Phasen, in denen zwischen den Tests gar nicht mehr groß überlegt wird. Testzyklus

Wiederaufsetzungsverhalten nach Unterbrechungen Unterbrechungen sind eine meist unterschätzte, zudem potenziell vermeidbare Fehlerursache und daher ein gutes Thema für diese Arbeit. Es bedarf aber einer persönlichen Untersuchung, wie man auf welche Unterbrechungen reagiert und ob dies überhaupt ein wichtiges Thema für den einzelnen Programmierer ist. Bestehende Schwierigkeiten und Taktiken, an eine offen gelassene Arbeit nach einer Unterbrechung wieder anzuknüpfen, müssten per μPP-Präsentation analysiert und bewertet werden. Konstruktive Fernziele sind eine Unterbrechbarkeitseinstufung und konkrete Arbeitswiederaufnahmehilfen. Wiederaufsetzen

Dazu kann man die Dimensionen „Fehlerart“ (siehe 1.2), „Verhaltensmusterart“ (siehe 1.3), „Analyseebene“ (siehe 1.4) und „Werkzeugunterstützungsart“ (siehe 1.6) kombinieren und den Anwendungsfällen zuordnen.

- Die μPP-Metriken zielen auf *Beanspruchungsfehler* ab, deren *Metrik*-artigen *Symptome bewusstmachend* präsentiert werden.
- Die Testzyklusvariationen deuten auf allgemeine *Beanspruchung*swechsel hin, die entweder selbst das *Fehlverhalten* oder ein *Symptomverhalten* sind, was das Werkzeug zumindest *bewusst* machen kann, wo es bei dem Ad-Hoc-Test aber auch *helfen* kann.
- Die Analyse des Wiederaufsetzens zielt auf *Belastungsfehler* ab, die sich in *Symptom- oder Bewältigungsverhalten* – also kurzzeitigen *Praktiken* – zeigen, die man mit einem Werkzeug *verdeutlichen* kann.

Planung

Die Praxisrelevanz der Anwendungsfälle entspricht der obigen Reihenfolge, die Einfachheit scheint eher andersherum. Die Anwendungsfälle werden sich mit dem laufenden Erkenntnisgewinn vermutlich ändern, vor allem weiter spezialisieren müssen. Da die Schwierigkeiten und Zeitabschätzungen zu diesem Zeitpunkt noch nicht erfasst sind, kann nicht ausgeschlossen werden, dass sogar eines oder zwei der Szenarien nicht realisiert werden können. Zu Details der Planung siehe ansonsten Kapitel 4.

Titel

Der Titel der Arbeit wäre (vorausgesetzt die Hypothesen werden nicht widerlegt): „Bewusstmachung einiger individueller Programmierverhaltensmuster in Beanspruchungssituationen mit Hilfe des Mikroprozesses“. Eine Kurzform, in der man die Hypothese offen lässt und die Mittel in den Vordergrund rückt, wäre „Mikroprozessanalysen von Fehlerverhaltensmustern beim Programmieren“.

Rückfallposition

Im Falle grundsätzlicher und größerer Probleme ändert sich die Arbeit dahingehend, dass ausführlich begründet wird, warum die entdeckten (und möglichst nachgewiesenen) Probleme die Lösung unmöglich machen. Es bleibt auf jeden Fall eine Einführung in den Mikroprozess: Die Untersuchung des μPP kann selbstverständlich nicht nur der Fehleruntersuchung dienen, sondern ist auch in vielen anderen Fällen einsetzbar: von einer Nutzung der Aufzeichnungen zum Bau von Programmierhilfen bis hin zum empirisch-wissenschaftlichen Einsatz. Auch ist der Mikroprozess nicht begrenzt auf das Programmieren, sondern kann auf andere Gebiete der Softwareentwicklung, vielleicht sogar ganz andere Entwicklungstätigkeiten ausgedehnt werden. Es kann wird in jedem Fall ein Anfang zu einer Mikroprozesstheorie und -praxis in der Softwareentwicklung gelegt werden.

3 Stand der Forschung

Das Thema „dieser Arbeit“ (so wird sie im folgenden Text referenziert) ist

1. konstruktive Qualitätssicherung
2. mittels μPP-Analyse
3. zur Vermeidung von Programmierfehlern
4. in einem reflexiven Prozess
5. anhand einiger Anwendungsfälle

Entsprechend sind die in diesem Kapitel erwähnten fremden Arbeiten geordnet.

Meines Wissens gibt es keine fremde Arbeit dieser Art, aber es gibt selbstverständlich einige Arbeiten, die Voraussetzungen dazu liefern, ähnliche Ideen beinhalten oder ein einen Teil der obigen Aspekte berühren. Die Beschreibungen sind knapp gehalten, aber in ihrer Struktur vollständig.

3.1 Konstruktive Qualitätssicherung

Zu der konstruktiven Qualiätssicherung sollen nicht nur ausgefeilte Vorgehensweisen zählen, sondern auch die Mittel dazu wie Vermessungen und empirische Untersuchungen. Geteilt ist dieser Abschnitt nach der zeitlichen Granularität der untersuchten Prozessschritte.

3.1.1 Makro

Das klassische Prozessverbesserungs- oder -wiederholbarkeitsprogramme, das vor allem (wenn auch nicht nur) zur Fehlervermeidung erfunden wurden, ist CMM [45][39], andere sind SPICE oder ISO9000. Diese sehen die Defektursachen aber meist in unterlassenden Prüfungshandlungen; zu einer begründeten Fehlerursachenanalyse kommt es nicht explizit. Spezielle defektbasierte Qualitätsverbesserungsprogramme (eine Übersicht bietet [29]) wie Orthogonal Defect Classification (ODC) [9][4][7] oder Root Cause Analysis (RCA) [8][64] funkionieren mittels Defektanalyse in verschiedenen Dokumenten zwecks späterer Vermeidung und streben damit eine fokussiertere Verbesserung an, der Blick auf die Fehlerauswirkungen (nämlich die Defekte) statt den Fehlern selbst bleibt jedoch.

Eine Reihe von Softwareprozessmaßen wurden entwickelt um die von den Makro-Ansätzen geforderte Prozessanalyse zu vereinfachen. Einen Überblick und eine Kritik bietet [23].

3.1.2 Meso

Eine vergleichsweise junge Technik nutzt den gemessenen, tatsächlichen Prozess zur Analyse, nämlich die in vielen Fällen vorhandenen Software-/Dokumentrevisionen, Defektdatenbanken oder den Projektmailverkehr. Solche Arbeiten werden als Softwarearchäolo-

gie oder „Mining software repositories“ bezeichnet. Hauptthemen sind Softwareevolution [38][22][84], Defektursachen [123][72][79] und Programmierunterstützung [1]. Einige Techniken behandeln eine Defektvorhersage nach oder während der Arbeit, z.B. [33], [105], [65] oder [126]. Häufig werden dort Produktmaße (z.B. Dateigrößen) mit Prozessmaßen (z.B. Größe einer Änderung) kombiniert. Eine irgendwie geartete Theorie der Defektentstehung (die den Menschen ins Zentrum stellen müsste) gibt es auch hier nicht, allenfalls Vermutungen oder Expertenerfahrungen.

3.1.3 Mikro

Programmierlehre In abgewandelter Form werden Mesoprozesstechniken in Lehrsituationen eingesetzt: Softwareversionen sind alle *abgespeicherten* (nicht nur in die Versionsverwaltung überführte) Quelltextdateien. Diese kann man auf verschiedene Weise im Lehrkontext nutzen [68][70]), z.B. für Stilanalysen, Produktmaße oder Testdurchläufe. Durch diese deutlich detaillierter aufgezeichnete Kodierdynamik wird der Übergang vom Meso- und Mikroprozess begonnen. Untersuchungen der Praktiken finden hier aber nicht statt, es bleibt bei einer Reduktion auf skalare Metriken

Software Process Telemetry Dieser Ansatz [52] fügt automatisch ermittelte tatsächliche und aktuelle Prozess- und Produktdaten aller Art zusammen zur manuellen Prozessbeobachtung und Entdeckung von Anomalien. Maße und andere Entwicklungen werden visualisiert und dem Entwicklungsleiter zur Beurteilung überlassen. Dies basiert auf der Software Hackystat, die den μPP aufzeichnet und im Abschnitt 3.2 erwähnt wird.

Software Development Stream Analysis Basierend auf μPP Daten aus Hackystat, automatisiert Zorro [61] eine Prozessbeobachtung, speziell die Praktik des „test driven development“. Es stellt somit einen dieser Arbeit sehr nahe stehenden Ansatz dar: Die Bewusstmachung der Nichteinhaltung einer vorgeschriebenen, in diesem Fall therapeutischen Praktik zur frühzeitigen Entdeckung (aber streng genommen nicht Vermeidung) von Defekten aller Art.

Tätigkeitsstudien von Programmierern In Tätigkeitsstudien (work studies) [104] wird untersucht, wie Programmierer ihre Zeit während des Programmierens verbringen [81] oder was sie während eines Reviews wirklich tun [92]. Diese Untersuchungen sind nicht mehr automatisierbar oder gar auf Basis von eh abfallenden Daten durchführbar. Statt dessen wird meist *in vivo*, also während des Arbeitsalltags beobachtet und manuell aufgezeichnet, evtl. unterstützt durch lautes Denken. Die Analysen haben zwar nur Fallstudiencharakter, betreffen aber häufige, relevante, vor allen den Aspekt der Arbeitspraktiken und -umgebungen betreffende Fehlerursachen und sind daher für diese Arbeit besonders interessant.

Psychologie des Programmierens Psychologische Untersuchungen des Programmierens [100][41][18] haben zunächst das Ziel, ein kognitives Modell zur Erklärung des Verhaltens und Vorgehens von Programmierern zu erstellen mit dem gelegentlich geäußerten

Zweck aus diesem Modell Programmierhilfen (z.B. bessere Programmiersprachen oder -umgebungen) ableiten zu können. Dies ist natürlicherweise eine Untersuchung auf der μPP-Ebene, wobei in der Regel ein μPP-Modell entworfen wird; näheres dazu im Abschnitt 3.2. Es existieren auch direkte Untersuchungen zu Fehlerprozessen (siehe Abschnitt 3.3.2) wobei die Domäne meistens unrealistisch trivial ist.

Programmiertechniken Damit eng zusammenhängend sind die Programmiertipps, also Techniken und Methoden, die – obschon selten so entstanden – als Folgerung der Kenntnisse über die Stärken und Schwächen des Programmierers als Mensch gelten könnten, u.a. Verbesserungen der Programmiersprachen (z.B. strukturiertes Programmieren [19], vertragsbasiertes Programmieren [69], automatische Speicherbereinigung, strenge Typisierung), der Programmierumgebungen und der Programmierpraktiken (z.B. Geheimnisprinzip [80], Dokumentationshilfen, testgetriebene Entwicklung [3]).

Persönlicher Softwareprozess Der PSP^{SM} gehört ebenfalls zu den Mikroprozessverbesserungen, wird aber ausführlicher erst auf Seite 20 erläutert, da er einige relevante Aspekte zusammenfasst.

3.2 Mikroprozess

3.2.1 Modelle

Ein μPP-Modell ist eine Sprache zur Beschreibung von Mikroprozessen. Es bietet ein Vokabular und Grammatik, um konkrete Programmieraktivitäten und deren Abfolge beschreiben zu können. Sobald man über Mikroprozesse spricht, benötigt man eine solche Sprache.

In der Forschung zur Psychologie des Programmierens wurden z.B. häufig Beobachtungsprotokolle gemacht [99] die eine bestimmte Struktur haben. Meistens sind dies einzelne aufeinanderfolgende Aktivitäten aus einem Korb vorgesehener Aktivitätsarten mit spezifischen Parametern. Ausführlich wurden solche Kodierschemata z.B. für Programmverstehprozesse [120][119], für die Tätigkeiten bei Codeänderungen [15][34] oder für das Testen [94] niedergeschrieben. Ich selbst habe eine solche entwickelt [49].

Basierend auf den kognitiven Modellen entstand u.a. eine Sicht des Programmierens als Problemlösen, welches selbst wieder eine Form des Planens ist [106][90][118][5] mit speziellen aus dem Programmieren stammenden Operatoren [44]. Die Theorien laufen also stets über Programmier*strategien* [14][15]. Dies berührt das Problem der sematischen Lücke in jeglicher Fehleranalyse. Viele Arbeiten in diesem Bereich behandeln daher leider nur unrealistisch einfache Programmierprobleme [12], was sich aus Aufwandsgründen schwer vermeiden lässt.

Auch Tätigkeitsstudien von Programmierern basieren auf einem Modell, welches die Aufzeichnungen leitet. So wurde z.B. in [61] speziell das testgetriebene Entwickeln formal in einem Modell beschrieben und auf naheliegende Weise auf Programmiereraktivitäten abgebildet.

Auch manche archäologischen Arbeiten haben ein Modell entwickelt (z.B. [20] und [26]) welche aber nicht mehr dynamisch sind, d.h. keine Tätigkeiten, sondern nur Artefakte repräsentieren können, in sofern für diese Arbeit nicht ausreichend sind. Prozessmodelle für Makroprozesse sind hingegen nicht detailliert genug für die Tätigkeitsschritte einzelner Programmierer.

3.2.2 Aufzeichnung

Die Mikroprozessaufzeichnung ist die semi-automatische Speicherung von μPP-Daten, also einzelner Ereignisse im Laufe des Prozesses. In den seltensten Fällen entspricht dies schon der μPP-Sprache. Um Mikroprozesse konform zu einem μPP-Modell aufzeichnen zu können, bedarf es also einer Abbildung der primitiven Aufzeichnungsdaten (Basisereignisse) auf die im Modell beschriebenen Aktivitäten [124]. In allen bekannten Arbeiten wird dies entweder nicht erwähnt oder im Einzelfall manuell bewerkstelligt.

Folgende Werkzeuge zur automatischen Aufzeichnung von Mikroprozessen sind mir bekannt:

- Hackystat [50][54] ist eine vergleichsweise reife Software, die an Entwicklungswerkzeugen wie IDEs, Editoren, Konfigurationsverwaltungen, Testwerkzeugen, statischem Analysatoren usw. nützliche Ereignisse – aktuelle Datei, Testläufe, Produktmaße usw. – abgreift und zentral zwecks Gesamtanalyse speichert. Ähnliche Werkzeuge sind PROM [102][103], SUMS [76] und in größerem Rahmen EPM [78]. Wichtige Eigenschaft all dieser Werkzeuge ist die Nichtbeinträchtigung des Programmierers während der Aufzeichnung. Sie dienen somit als Vorbild für Aufzeichnungswerkzeuge im Rahmen dieser Arbeit. Wichtig ist jedoch, dass viele der aufgezeichneten Daten nicht so „mikro“ sind, wie für diese Arbeit nötig.

- GRUMPS [112] ist ein Beispiel wie man mit primitiven Ereignissen wie Mausklick und Tastendruck Aufzeichnungen machen und nutzen kann, z.B. für Unterbrechungsanalyse [88]. Ein ähnlicher Ansatz wird in [113] beschrieben. Diese Aufzeichnungsdaten sind wiederum *zu* „mikro“. In den genannten Arbeiten wird dies auch als Problem geschildert.

- Ginger [114] und der Nachfolger Ginger2 [115] sind Umgebungen zur μPP-Aufzeichnung in einer *Laborumgebung* (in vitro), was so ungewöhnliche Daten liefern kann wie Hautleitfähigkeit, Blickrichtung, Armbewegungen usw. So wertvoll diese Daten auch sein mögen, sie sind nicht ohne erhebliche Beeinträchtigung des Programmierers erhebbar, somit zweitrangig für diese Arbeit.

- Marmoset [107] konzentriert sich auf die Codeänderungen im Mikroprozess, indem es alle Dateiabspeicherungen (also nicht nur die expliziten Submits) zwecks nachträglicher Analyse in einem CVS speichert. Ähnliches wird in [97] bewerkstelligt. Dieser Ansatz ist interessant, weil vergleichsweise leicht implementierbar. Zur Analyse stehen dann allerdings nur Code*änderungen* zur Verfügung, nicht Ereignisse wie z.B. Testläufe. Zudem sind für diese Arbeit auch Änderungen interessant, *bevor* sie abgespeichert werden. Marmoset ist sozusagen „nicht mikro genug“.

3.2.3 Analyse

Die μPP-Analyse ist die der Aufzeichnung folgende Verdichtung und Nutzung der μPP-Aufzeichnungen meist zum Zweck der Prozessbeobachtung oder -bewertung. Dabei geschieht die Analyse meist nachträglich, kann aber auch „just-in-time“ ablaufen. Desweiteren kann die Analyse manuell/visuell oder automatisch geschehen. Einen Überblick über Analysemöglichkeiten solcher diskreter, zeitgeordneter Daten geben [27], über Visualisierungstechniken [101].

- Bei Hackystat findet die Analyse visuell statt. Die μPP-Aufzeichnungen werden ohne weitere Verdichtung (Ausnahme sind Durchschnittswertbildungen) in Graphen präsentiert zur Analyse und zum Vergleich mit vergangenen Projekten. Die Metriken sind vergleichsweise einfach.

- Zorro ergänzt Hackystat um erste μPP-Analysewerkzeuge, nämlich der Definition und automatischen Erkennung von Episoden auf Basis von Regeln über μPP-Ereignisse. Ähnliches wird auch in dieser Arbeit benötigt. Angewendet wurden die Analysen bei Zorro für den vorschreibenden Prozess der testgetriebenen Entwicklung. Ähnliches versucht [121] nachträglich anhand von Coderevisionen im Mesoprozess zu erkennen.

- Eine „Process discovery“ nutzt Analysemethoden, um Makroprozesse beschreiben zu können. Es werden diverse Mittel dazu eingesetzt; eine vergleichende Übersicht bietet [11]: Es werden Grammatiken, Neuronale Netze und Markovmodelle gelernt anhand von tatsächlichen, aufgezeichneten Prozessen. Diese Arbeit wird hingegen eher auf eine Visualisierung setzen und eine auf a-priori gegebenenen Episodenschablonen basierende Prozesserkennung dem Programmierer selbst überlassen.

- Die Software Evolution Matrix [63] bietet eine Visualisierung von Softwareänderungen, an denen man die Weiterentwicklung eines Softwarepakets ablesen kann. Es werden typische optische Muster definiert. Dies kann als Vorlage für eine Visualisierung diesen, siehe auch Ansätze in [49].

- Bei Tätigkeitsstudien werden in der Regel lediglich einfache Metriken berechnet über Häufigkeiten von Tätigkeiten, evtl. auch Häufigkeiten von aufeinanderfolgenden Tätigkeiten. Speziell wurde in [55][56] der Einsatz von Copy-Paste beim Programmieren und in [57] die Evolution von Codekopien untersucht, zwei relevante Fälle auch für diese Arbeit.

- Eine sehr relevante Arbeit, die leider nur auf Japanisch vorliegt, ist [125], wo mit Hilfe von Ginger analysiert wurde, was Programmierer kurz vor dem Einbau eines Defektes getan haben. In ihrer Motivation und Methodik ist dies die zu dieser ähnlichste Arbeit.

- Eine Reduktion des Mikroprozesses auf einfache Metriken (z.B. Änderungsgrößen, -verteiltheit und -häufigkeiten) findet sich in [71], [24] oder [37] .

Interessant ist, dass kaum ein Projekt Codeveränderungsanalysen, also den Wandel des Codes über die Zeit, durchführt.

3.3 Fehlerursachen und Verhaltensmuster

Alle Untersuchungen in diesem Unterkapitel konzentrieren sich auf die *Erklärung* von Fehlern auf Basis eines *kognitiven* Modells. Eine Einsicht in die psychologischen Ursachen eines Fehlers dient einer Klassifizierung. Zu einer Fehlervermeidung bedarf es der zusätzlichen Beachtung der persönlichen Umstände des Falschtuns, um bei Wiederauftreten besonders aufmerksam zu arbeiten.

3.3.1 Menschliche Fehler

Forschung über Programmierfehler ist insbesondere Forschung über menschliche Fehler im allgemeinen und kognitive Schwächen im spezielleren. [86] ist das Standardwerk über menschliches Versagen, einen Überblick bietet auch [93], einige andere Quellen vertiefen dies, z.B. spezialisiert auf Planungsfehler [98] oder Fehler in komplexen Umgebungen [21]. Die diskutierten Fehlerarten sind aber schwer zu übertragen auf Programmierfehler. Wenn man einen Programmierfehler (genauer: das Einbauen eines Defektes) beobachtet, kann man aber durchaus eine oder mehrere dieser Fehlerursachen versuchen anzubringen. In der Regel bleibt es aber bei Vermutungen.

Speziell kann das Phänomen der begrenzten Aufmerksamkeit [43] oder der Beanspruchung und Belastung [66] richtungsweisend sein, obwohl dort meist nicht das *Verhalten* als Indikator heranziehen, sondern psychophysiologische Merkmale. Auch hier gilt: Die Erklärungsebene der Psychologen ist kaum geeignet für einen reflexiven Verbesserungsprozess.

Mir ist als einzige Arbeit [111] bekannt, wo konkret von allgemeinen menschlichen Schwächen (hier: Bestätigungsneigung) auf Programmierfehler im weiteren Sinne (hier: Auswahl von Testfällen) geschlossen wird. Einen breiteren, dafür aber nicht nachvollziehbar auf psychologische Forschung basierten Ansatz verfolgt [32].

3.3.2 Fehler in der Softwareentwicklung

Es gibt einige spezielle Arbeiten zu Programmierfehlerursachen im speziellen oder Fehler in der Softwareentwicklung. Das Spezialgebiet ist hier aus psychologischer Sicht die schon oben erwähnte „Psychologie des Programmierens“

In aller Regel wurden Anfänger-Programmierfehler untersucht, z.B. in [108][117][35], manchmal für erfahrene Programmierer [36]. [58][59] unternehmen eine detaillierte, kognitiv basierte Root-Cause-Analysis für eine einfache Lehrprogrammiersprache. Ähnlich geht [74] vor, bleibt aber auf klassischem softwaretechnischen Terrain. [10] versucht, die Theorien von [86] über menschliches Versagen auf die Softwareentwicklung anzuwenden.

Irrtumsfehler [32] führt Irrtumsfehler explizit auf typische kognitive Heuristiken zurück: Strukturerwartung, Kausalitätserwartung und Assoziationsstreben. Er kommt dabei zu dem Schluss, dass ein PSP^{SM}-ähnlicher reflexiver Lernprozess eine Lösung wäre, hat dies

aber nicht weiter vertieft. [110] betrachten typische menschliche Heuristiken aus softwaretechnischer Sicht, kommen aber zu keinem prozessverbessernden Ergebnis.

Überforderungsfehler Die Forschung über die Unterschiede zwischen Programmieranfängern und Experten gehen insbesondere auf die Unterschiede in den Strukturen (also dem passiven Faktenwissen) und den Heuristiken (also dem Handlungswissen) ein. Auftretende Schwächen der Anfänger in diesen Bereichen verursachen vermutlich stets eine Beanspruchung und damit eine Begünstigung von menschlichen Schwächen wegen verschobener Aufmerksamkeit. Arbeiten in diesem Bereich sind mir leider nicht bekannt, siehe aber die eingangs genannten Programmierfehler bei Anfängern.

Belastungsfehler Es gibt einige spezielle Untersuchungen zu Belastungsursachen wie Stress [2][30][85], Arbeitsumgebung [17] und Unterbrechungen [48][116]. Diese Arbeiten sind empirischer Art: Sie beschreiben die Wirkungen der Belastungen und diskutieren anschließend über Prozessverbesserungsmaßnahmen. Diese Arbeiten, obschon wegen einer fehlenden Mikroprozessbetrachtung nicht direkt verwendbar für meine Arbeit, bieten eine gute Motivation für die Wichtigkeit der Belastungsfehler auch bei der Softwareentwicklung.

3.3.3 Verhaltensmuster

Leider gibt es nur wenige Untersuchungen zu speziellen Verhaltensmustern bei der Softwareentwicklung. [104] beschreibt eine „just in time comprehension“, das Verstehen und gleich wieder Vergessen eines Programmablaufes, [81] das Kommunikationsverhalten, [92] das Verhalten bei Reviews und Reviewsitzungen, [55] das Nutzen von Copy-Paste beim Implementieren und [91] die Art und Weise des Navigierens durch den Code. [34] zeigen ähnliches beim Kodieren selbst: Man schreibt ein Stück Code (gnisrap) und muss es fortan immer wieder neu verstehen (parsing). Gerade dies geht in ihrer Analyse schon sehr nah an das hier auch notwendige, ist jedoch leider nicht automatisch μPP-aufzeichenbar.

3.4 Persönliche, reflexive Softwareprozesse

Der PSP^{SM}[46][47][45] rät Programmierern, u.a. regelmäßig Defekteinbauten und -ausbauten in Art, Vorgeschichte und Dauer zu protokollieren. Das Ziel ist, sowohl typische Defekteinbaumuster erkennbar und damit vermeidbar zu machen, als auch die Konzentration auf die teuren Defekte und Umstände zu lenken.

Es wurde empirisch unterstützt [83][122][25][40], dass PSP^{SM} qualitätsverbessernd wirkt, wenn es nur konsequent angewendet wird. Vermutlich ist der wesentliche Aspekt das Aufschreiben selbst, also die umfangreiche Reflexion über das eigene Tun.

Die praktischen Probleme liegen jedoch in der ungewollten Unterbrechung bei Aufzeichnungen [50], der Dysfunktionalität der simplen Metriken [73], der geringen Datenqualität bei Selbstaufzeichnungen [53] und dem nicht überzeugenden (wenn auch nachweislich vorhandenen) Nutzen im Vergleich zum Aufwand [82]. Versuche, das Protokollieren zu (teil-)automatisieren wurden für gescheitert erklärt [51].

Aus den Problemen ergaben sich schließlich neue Herangehensweisen, die entweder den Aufwand minimierten [82] oder durch Automatisierung zumindest einen Teil des Gedankens operabel zu machen [51]. Kernanalyseebene als Grundlage zu Prozessverbesserungen beim PSP blieben aber einfache Metriken. Praktiken oder Strategien rückten erst in jüngerer Zeit in den Fokus der PSP-nahen Forschung [61].

3.5 Drei Anwendungsfälle

μPP-Metriken Für diesen Bereich kann man sich – das ist ja die Grundidee – reichlich bei den Mesoprozessideen „bedienen“. Erwähnt wurden diese Arbeiten ansatzweise schon in Abschnitt 3.1. Je weiter dies Metriken allerdings in Richtung von Praktiken oder gar Strategien gehen, desto eher sind diese auf Mesoprozessebene eh nicht erkennbar und daher vermutlich auch selten beschrieben worden. Desweiteren sind Forschungen im Bereich der „Psychologie des Programmierens“ (siehe 3.1.3) hier interessant, insbesondere der Kodierdynamik, als auch der bei [21] erwähnten Verhaltensänderungen unter Beanspruchung, deren Uminterpretation für das Programmieren allerdings schwer fallen dürfte.

Testzyklusvariationen Es gibt verwandte Konzepte wie die kleinen Iterationen in agilen Prozessen oder der testgetriebenen Entwicklung. Aber die Herangehensweise an dieses Thema ist stets vorschreibend, nicht beschreibend oder analysierend. Interessant sind vermutlich Untersuchungen zur Akzeptanz der testgetriebenen Entwicklung insbesondere dann, wenn die Abweichungen davon beschrieben werden. Immerhin existiert mit Zorro [61] ein Ansatz, der im Mikroprozess ein Test-first-Verhalten zu erkennen versucht. Eine Begründung für den Grund für kurze Iterationen und häufiges Testen kann man bei der Literatur über agile Softwareprozesse finden. Das Verhalten bezüglich Hypothesenbestätigung findet man unter allerdings anderen Arbeitsaufgaben bei [21].

Wiederaufsetzverhalten Die Forschung zu Arbeitsunterbrechungen ist sehr umfangreich, speziell zur Bildschirmarbeit. Sie teilt sich in die Bereiche Häufigkeit und Art von Unterbrechungen [67][17] Auswirkungen von Unterbrechungen je nach Unterbrechungsart [13][6][31], Untersuchungen zur Unterbrechbarkeit eines Arbeiters [16][87][42], Verhalten nach einer Unterbrechung [77] und technische Hilfe zum Wiederaufsetzen nach einer Unterbrechung [28]. Eine teilweise Übertragungen dieser Forschung auf die Softwareentwicklung, z.B. zu den Kosten von Unterbrechungen, gibt es bei [48][116]. Untersucht wurden meist Unterbrechungen während geistiger Problemlösungsarbeit, also etwas dem Programmieren durchaus verwandtes. Unterbrechungen sind ein sehr relevantes Thema. Für die Untersuchung von Wiederaufsetzverhalten ist insbesondere die genaue zeitliche Analyse in [6] bedeutend, die sich allerdings nicht auf das Wie des Weiterarbeitens bezieht. Dies thematisieren zwar [67] und [77], aber lediglich auf der Mesoebene, *ob* eine unterbrochene Tätigkeit nach der Unterbrechung wieder aufgenommen wird. Eine Analyse des Mikroprozesses einer Tätigkeitswiederaufnahme, z.B. den des Weiterschreibens und Reorientierens in einem Text, fehlt in allen mir bekannten Quellen.

4 Planung

4.1 Gliederung

Die Gliederung der Dissertation ist voraussichtlich wie folgt (in Klammern die vermutete relative Anzahl von Seiten):

```
1.  Motivation (5%)
2.  Fehlerverhaltensmuster (10%)
3.  Mikroprozess
3.1  Mikroprozessmodell (10%)
3.2  Mikroprozesstechnik (5%)
3.2.1  Aufzeichnung der Basisereignisse
3.2.2  Bildung von Episoden
3.2.3  Erkennung des Modells
3.3  Mikroprozessanalyse (10%)
3.3.1  Maße
3.3.2  Muster
3.3.3  Präsentation
4.  Werkzeuggestützte Reflexion (5%)
5.  Drei Anwendungsfälle (50%)
5.1  Mikroprozessmetriken
5.1.1  Problem
5.1.2  Verwandte Arbeiten
5.1.3  Lösung
5.1.4  Evaluation
5.2  Testzyklusverhalten
5.2.1  Problem
5.2.2  Verwandte Arbeiten
5.2.3  Lösung
5.2.4  Evaluation
5.3  Wiederaufsetzungsverhalten
5.3.1  Problem
5.3.2  Verwandte Arbeiten
5.3.3  Lösung
5.3.4  Evaluation
5.4  Verallgemeinerungen
6.  Fazit und Ausblick (5%)
```

Die Motivation wird ähnlich ausfallen wie in diesem Text. Die einzelnen Abschnitte der Motivation werden ausführlich in den Kapiteln 2 bis 4 behandelt. Dort werden meine Definitionen, Modelle und Ergebnisse zusammen mit verwandten Arbeiten vorgestellt. Die Darlegung der drei Anwendungsfälle erfolgt jeweils ähnlich, wobei am Ende noch Verallgemeinerungsmöglichkeiten in Richtung eines allgemein einsetzbaren Werkzeuges diskutiert werden sollen. Im Fazitkapitel schließlich werden die Evaluationen bewertet und ein Ausblick bzgl. der weiteren Forschung gegeben.

4.2 Vorleistungen

Wenige Teile des Promotionsvorhabens sind schon geleistet, einiges aber schon begonnen worden, so dass ich nicht bei Null anfange. Die erste knappe Hälfte der Dissertation ist in Grundzügen schon klar:

- Die Motivation wurde mit diesem Exposé einigermaßen fertig gestellt.
- Die Literaturliste ist schon sehr umfangreich, und wird nur noch für sporadisch ergänzt werden müssen. Die Darstellung der Literatur bedarf aber noch einiger Arbeit.
- Zu Fehlerursachen und Verhaltensweisen habe ich schon ein umfangreiches Literaturstudium gemacht, dessen erste Früchte auch schon in diesem Exposé zu finden sind. Hinzu kommen Erkenntnisse aus schon gemachten eigenen Beobachtungen, die allerdings noch ein bischen weiter geführt werden müssen. Die ausgewählten Anwendungsfälle basieren unter anderem aus den Erkenntnissen dieser Literatur und meine Beobachtungen.
- Die Überlegungen zum μPP sind schon recht weit: Ein *μPP-Modell* ist erstellt und auch schon veröffentlicht [49]. Auch die *μPP-Aufzeichnung* ist schon weit fortgeschritten, vor allem aufgrund einiger Abschlussarbeiten von Studierenden [96][95] und Realeinsatz der Software von Kollegen und von mir. Eine Weiterentwicklung, allerdings ganz klar nur auf Basis der Anforderungen dieser Forschung, ist nichtsdestotrotz notwendig und wird leider umfangreich sein.
- Derzeit bearbeite ich die Abbildung der recht primitiven aufgezeichneten μPP-Ereignisse auf die im μPP-Modell genannten Episoden. Dies bezieht sich insbesondere auf die Erkennung von Codestellen und deren Metamorphose, d.h. die Fortentwicklung von zusammenhängenden, kleinen Codefragmenten. Eine Lücke besteht vor allem noch in der *μPP-Analyse*, wo ein paar Ideen existieren (z.B. [60][89]), die aber kaum geprüft sind.
- Auch Überlegungen zur Reflexion und zu den Werkzeugtypen habe ich begonnen. Gedanken zur allgemeinen *μPP-Visualisierung* sind von Stephan Salinger und mir entwickelt und von Stephan prototypisch implementiert. Eine Java-Implementierung läuft [75]. Eine andere videoähnliche Präsentation des μPP wurde ebenfalls realisiert [62]
- Es bleibt aber noch der gänzlich unbearbeitete Teil der konkreten Szenarien. Zum Trial-and-Error-Verhalten, welches sich entweder bei einem μPP-Komplexitätsmaß bemerkbar macht oder zumindest ein spezielles Ad-Hoc-Testverhalten ist, existiert eine Vorarbeit [89].

Die Vorleistungen machen weniger als die Hälfte der (zudem noch nicht geschriebenen) Dissertation aus, was allerdings nur wenig korreliert mit der Dauer der Bearbeitung der Themen, wie aus dem nachfolgenden Zeitplan hervorgeht.

4.3 Zeitplan

Der Vorgehensplan betrachtet generelle Arbeiten, die in allen Fällen zu erledigen oder zumindest sinnvoll sind, und Anwendungsfall-spezifische Arbeiten, die sich aus den drei Szenarien ergeben. Es reicht derzeit leider nicht zu einem Zeitplan, da für eine Abschätzung zu wenig Erfahrungswerte vorliegen. Daher beschränkt sich der Zeitplan auf die Angabe relativer Zeitaufwendungen auf Basis von 100% verbleibende Gesamtzeit. Nicht eingerechnet sind Aufwände für das Schreiben der Dissertation und Konferenzbeiträge.

Generell: Aufzeichnung des μPP :

1. Weitere Ereignisse sammeln, die aus Eclipse und seiner Umgebung erhebbar sind
2. Externe Ereignisse für Unterbrechungsindikatoren (fremde Anwendungen oder Videobilder) erheben
3. Unterstützung für möglichst „sanfte“ Nachfragen an den Entwickler aufgrund spezieller Geschehnisse
4. Dazu dann auch Integration der derzeit Serverprozess-basierten Aufzeichnung in Eclipse

Diese Arbeitseinheiten ergeben sich vor allem aus den Anforderungen der Anwendungsfälle und sind daher noch nicht vollständig festgelegt. Die μPP-Aufzeichnung ist deswegen oft umfangreiche Arbeit, weil es Interna von z.B. Eclipse betrifft, die schwer zu verstehen und zu testen sind. Dies dürfte bei geringen Qualitätsanforderungen etwa 10% des Gesamtaufwandes der Forschung benötigen.

Generell: Erkennung des μPP-Modells :

1. Weitere Verbesserung des „Stellenerkenners“, d.h. der Segmentierung des Codes in hierarchischen Stellen und die Berechnung und Typisierung von Stellenänderungen inklusive Implementation verschiedener passender Strategien und deren Vergleich
2. Versuch der möglichst guten Erkennung von „Betterments“ (so werden im μPP-Modell Defektbehebungen genannt), inklusive der Abgrenzung zu Advancements, Displacements und zu Inner Activities ähnlicher Art, z.B. Tag-Alongs
3. Erkennung diverser anderer μPP-Eigenschaften, z.B. Geschwindigkeit, je nach entstehender Hypothese
4. Klassifikation von Stellenkarrieren

Auch dies ist abhängig von den Bedürfnissen der Szenarien. Da hier die Algorithmen weitestgehend fehlen und mit ihnen experimentiert werden muss, dürfte 20% der Gesamtzeit allein für diesen Teil in Anspruch genommen werden.

Generell: Präsentation des μPP :

1. Stellenhierarchie und -geschichte in Eclipse präsentieren
2. Dort auch Stellenmetriken und andere -eigenschaften einbinden
3. Zeitorientierte Darstellung (Tätigkeiten/Praktik pro Stelle) präsentieren

Für eine saubere, nutzbare Implementation einer Präsentation würde zu viel Zeit benötigt werden, die woanders wichtiger wäre, so dass ich hier eher mittels Prototypen versuchen werde, die Machbarkeit zu zeigen. Der Aufwand reduziert sich damit vermutlich auf 10%, wobei schon bedacht wurde, dass noch einige Konzepte fehlen.

Anwendungsfall μPP-Metriken :

1. Implementieren von aus dem Mesoprozess bekannten Änderungsprozessmetriken und diese anwenden und umfangreichere Statistiken für Projekte machen und dokumentieren
2. Aufgrund Erfahrungen und Vermutungen neue Metriken entwickeln und evaluieren, die die Codierdynamik der Stellen nutzen
3. Überlegungen und qualitative Untersuchungen, um Metriken in Bezug zu setzen zu Defekteinbauten, welche wiederum auf Basis von Defektbehebungen berechnet werden

Der Umfang dieser Arbeiten im Gesamtrahmen dürfte etwa bei 20% liegen. Einige der Aufwände sind in die generellen μPP-Arbeiten verschoben. Obschon die Arbeiten an den Metriken einigermaßen gradlinig sind, möchte ich hier intensiv arbeiten, weil es im Vergleich zum Aufwand und Risiko vergleichsweise wahrscheinlich zu brauchbaren Ergebnissen führt. Das ist auch der Grund, warum dieses Szenario an den zeitlichen Anfang der Bemühungen gestellt wird. Zudem ist hier die Konkurrenz am größten.

Anwendungsfall Testzyklusvariationen :

1. Testpraktiken explorativ und qualitativ beschreiben, ausgehend von der vorhandenen Trial-and-Error-Beschreibung
2. Festlegen der eventuell fehlenden Ereignisse
3. Untersuchung der Korrelation zu Defekteinbauten

Dieses Szenario ist einigermaßen gradlinig. Zentral ist hier die Präsentation, deren Aufwand zum Teil schon generell eingerechnet wurde. Insgesamt verbleiben 15%.

Anwendungsfall Wiederaufsetzverhalten :

1. Theoretische Betrachtungen zum Thema
2. Qualitative Untersuchung von Unterbrechungsindikatoren und deren möglichst weitgehende Implementierung als Ereignisse

3. Untersuchung von „Inner-Programmier-Unterbrechungen“ in Form von Abschweifungen
4. Möglichkeiten eruieren, nach Ursachen für Unterbrechungen nachzufragen
5. Umfangreiche Empirie zur Beurteilung von Wiederaufsetzverhalten und Beschreibung dessen
6. Möglichst weitgehender Versuch, diese Verhaltensweisen automatisch zu erkennen

Wegen der großen Unsicherheit und des umfangreichen Empirieanteils schätze ich einen Aufwand von 25% im Rahmen der gesamten verbleibenden Promotionszeit.

Beim Streichen eines Anwendungsfalls verteilen sich die gesparten Aufwände entsprechend. Das Niederschreiben der Dissertation und das Veröffentlichen von Forschungsergebnissen und Durchführung von Vorträgen ist nicht explizit bedacht worden, obschon es ohne Zweifel viel Zeit kosten wird. Diese Aufwände sind also implizit. Das letztendliche Fertigstellen der Dissertation inkl. die Durchführung des Promotionsvorhabens sollte aber mit 3 MM bedacht werden. Diese Schätzung setzt voraus, dass es im wesentlichen Formulierungsaufwände sind, und dass einiges kopiert werden kann.

Bei einem derzeit laut Arbeitsvertrag verbleibenden Zeitraum von etwa 24 Monaten minus 6 Monate für die Lehre (ein Viertel) minus 2 Monate für die Ausarbeitung, also 16 Monaten wird die Zeit gewiss nicht für alles reichen.

4.4 Erfolgs- und Endekriterien

Generelles Endekriterium wäre der Nachweis der Nützlichkeit der entstandenen Werkzeuge durch subjektive Einschätzungen oder gar mittels Nachweis, dass damit defektfreiere Software erstellt wird. Da aber nur Machbarkeitsstudien das Ziel sind und zudem ein solcher Nachweis alleine schon eine Promotion Wert wäre, muss ich es beschränken und pro Anwendungsfall spezialisieren.

Die Arbeiten zur Aufzeichnung, zur Erkennung des μPP-Modells und dessen Präsentation sind stark bestimmt durch ihren Einsatz in den Szenarien, daher werden sie hier nicht explizit aufgeführt.

μPP-Metriken Es müssen mindestens so plausible Ergebnisse wie im Mesoprozess präsentiert worden sein. Es sollen Metriken nachgebildet worden sein, die aus Mesoprozessarbeiten entstanden. Sind die Mikroprozessergebnisse wider Erwarten nicht eindeutiger als die Mesoprozessergebnisse, dann muss dies begründet worden sein. Es muss eine verständliche Präsentation der Metriken zwecks explorativer Arbeit möglich sein.

Testzyklusvariationen Es muss die Präsentation von verschiedenen Testpraktiken durch Programmierer für gut bewertet worden sein. Es muss realistisch erscheinen, dass der Programmierer diese Praktiken in Relation zu seinen Programmierschwierigkeiten stellen kann.

Wiederaufsetzverhalten Es muss eine angemessene Anzahl von Wiederaufsetzverhaltensmuster beschrieben, grundiert und durch Programmierer bestätigt sein. Es muss eine Mustererkennung implementiert und ihre Differenz zu den Idealmustern beschrieben sein.

Literatur

[1] D.L. Atkins. Version sensitive editing: Change history as a programming tool. *Proceedings of the SCM-8 Symposium on System Configuration Management*, pages 146–157, 1998.

[2] R.D. Austin. The effects of time pressure on quality in software development: An agency model. *Information Systems Research*, 12(2):195–207, 2001.

[3] K. Beck. *Test-Driven Development: By Example.* Addison-Wesley, 2002.

[4] I. Bhandari, M. Halliday, E. Tarver, D. Brown, J. Chaar, and R. Chillarege. A case study of software process improvement during development. *IEEE Trans. Softw. Eng.*, 19(12):1157–1170, 1993.

[5] R. Brooks. Towards a theory of the cognitive processes in computer programming. *International Journal of Human-Computer Studies*, 51(2):197–211, 1999.

[6] I. Burmistrov and A. Leonova. Do interrupted users work faster or slower? The micro-analysis of computerized text editing task. *Proceedings of the International Conference on Human-Computer Interaction: Theory and Practice (Part I)*, 1:621–625, 2003.

[7] M. Butcher, H. Munro, and T. Kratschmer. Improving software testing via ODC: Three case studies. *IBM Systems Journal*, 41(1):31–44, 2002.

[8] D.N. Card. Learning from our mistakes with defect causal analysis. *IEEE Software*, 15(1):56–63, 1998.

[9] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.Y. Wong. Orthogonal defect classification – A concept for in-process measurements. *IEEE Trans. Softw. Eng.*, 18(11):943–956, 1992.

[10] Trevor Cockram, Jim Salter, Keith Mitchell, Judith Cooper, and John May Brian Kinch. *Human Error in the Software Generation Process.* In Redmill, F. Anderson, T. (eds.) Technology and assessment of safety-critical systems. Springer, 1994.

[11] Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 73–82, New York, NY, USA, 1995. ACM Press.

[12] Bill Curtis. By the way, did anyone study any real programmers? In *First workshop on Empirical studies of Programmers*, pages 256–262, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[13] Edward B. Cutrell, Mary Czerwinski, and Eric Horvitz. Effects of instant messaging interruptions on computing tasks. In *CHI '00: CHI '00 extended abstracts on*

Human factors in computing systems, pages 99–100, New York, NY, USA, 2000. ACM Press.

[14] S.P. Davies. The nature and development of programming plans. *Int. J. Man-Mach. Stud.*, 32(4):461–481, 1990.

[15] S.P. Davies. Models and theories of programming strategy. *Int. J. Man-Mach. Stud.*, 39(2):237–267, 1993.

[16] U. Dekel and S. Ross. Eclipse as a platform for research on interruption management in software development. *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 12–16, 2004.

[17] T. DeMarco and T. Lister. Programmer performance and the effects of the workplace. In *Proceedings of the 8th International Conference on Software engineering*, pages 268–272, London, England, August 1985. IEEE Computer Society.

[18] Francoise Detienne. *Software Design: Cognitive Aspects*. Springer, 2002.

[19] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[20] Dirk Draheim and Lukasz Pekacki. Process-centric analytical processing of version control data. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Dietrich Dörner. *Die Logik des Mißlingens*. Rowohlt, 15th edition, 2004.

[22] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.

[23] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, 1999.

[24] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, 2000.

[25] Pat Ferguson, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya. Results of applying the Personal Software Process. *Computer*, 30(5):24–31, 1997.

[26] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, Washington, DC, USA, 2003. IEEE Computer Society.

[27] C. Fisher and P. Sanderson. Exploratory sequential data analysis: exploring continuous observational data. *interactions*, 3(2):25–34, 1996.

[28] Jerry L. Franke, Jody J. Daniels, and Daniel C. McFarlane. Recovering context after interruption. In W. Gray and C. Schunn, editors, *Proceedings of 24th Annual Meeting of the Cognitive Science Society (CogSci 2002)*, pages 310–315, 2002.

[29] Michael Fredericks and Victor Basili. Using defect tracking and analysis to improve software quality. Technical Report DACS-SOAR-98-2, Data & Analysis Center for Software, 1998.

[30] Tsuneo Furuyama, Yoshio Arai, and Kazuhiko Iio. Fault generation model and mental stress effect analysis. *J. Systems Software*, 26(1):31–42, 1994.

[31] Tony Gillie and Donald Broadbent. What makes interruptions disruptive? a study of length, similarity, and complexity. *Psychological Research*, 50(4):243–250, April 1989.

[32] T. Grams. *Denkfallen und Programmierfehler*. Springer, 1990.

[33] Todd L. Graves, Alan F. Karr, J.S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[34] Wayne D. Gray and John R. Anderson. Change-episodes in coding: when and how do programmers change their code? In *Empirical studies of Programmers: Second workshop*, pages 185–197, Norwood, NJ, USA, 1987. Ablex Publishing Corp.

[35] Wayne D. Gray, Albert T. Corbett, and Kurt Van Lehn. Planning and implementation errors in algorithm design. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society*, pages 594–600, Montreal, Canada, 1988.

[36] Raymonde Guindon, Herb Krasner, and Bill Curtis. Breakdowns and processes during the early activities of software design by professionals. In *Empirical studies of Programmers: Second workshop*, pages 65–82, Norwood, NJ, USA, 1987. Ablex Publishing Corp.

[37] Ahmed E. Hassan and Richard C. Holt. The chaos of software development. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 84–94, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Ahmed E. Hassan and Richard C. Holt. Studying the evolution of software systems using evolutionary code extractors. In *IWPSE '04: Proceedings of the 7th International Workshop on the Principles of Software Evolution*, pages 76–81, Washington, DC, USA, 2004. IEEE Computer Society.

[39] James Herbsleb, David Zubrow, Dennis Goldenson, Will Hayes, and Mark Paulk. Software quality and the capability maturity model. *Commun. ACM*, 40(6):30–40, 1997.

[40] Iraj Hirmanpour and Soheil Khajenoori. Personal software process technology: An experiential report. In *ISECON 2000: Proceedings of the 21st Annual Information Systems Education Conference*, Philadelphia, 2000.

[41] J.M. Hoc et al. *Psychology of programming.* Academic Press, San Diego, 1990.

[42] Eric Horvitz and Johnson Apacible. Learning and reasoning about interruption. In *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*, pages 20–27, New York, NY, USA, 2003. ACM Press.

[43] Eric Horvitz, Carl Kadie, Tim Paek, and David Hovel. Models of attention in computing and communication: From principles to applications. *Commun. ACM*, 46(3):52–59, 2003.

[44] Karen E. Huff and Victor R. Lesser. A plan-based intelligent assistant that supports the software development. In *SDE 3: Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 97–106, New York, NY, USA, 1988. ACM Press.

[45] Watts S. Humphrey. *Managing the software process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[46] Watts S. Humphrey. Using a defined and measured personal software process. *IEEE Softw.*, 13(3):77–88, 1996.

[47] Watts S. Humphrey. *Introduction to the Personal Software Process.* Addison-Wesley Longman Ltd., Essex, UK, 1997.

[48] T. Jackson, R. Dawson, and D. Wilson. The cost of email interruption. *Journal of Systems and Information Technology*, 5(1):81–92, 2001.

[49] Sebastian Jekutsch. An annotation scheme to support analysis of programming activities. In *Papers presented at Workshop on Ethnographies of Code.* Infolab21, Lancaster University, 2006.

[50] Philip M. Johnson. Project Hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis. Technical Report CSDL-01-13, Department of Information and Computer Sciences, University of Hawaii, November 2001.

[51] Philip M. Johnson. You can't even ask them to push a button: Toward ubiquitous, developer-centric, empirical software engineering. In *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications*, Nashville, TN, December 2001.

[52] Philip M. Johnson. A continuous, evidence-based approach to discovery and assessment of software engineering best practices. Technical Report CSDL-05-05, Department of Information and Computer Sciences, University of Hawaii, June 2005.

[53] Philip M. Johnson and Anne M. Disney. A critical analysis of PSP data quality: Results from acase study. *Empirical Softw. Eng.*, 4(4):317–349, 1999.

[54] Philip M. Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen, and William E. J. Doane. Beyond the personal software process: metrics collection and analysis for the differently disciplined. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 641–646, Washington, DC, USA, 2003. IEEE Computer Society.

[55] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[56] Miryung Kim, Vibha Sazawal, and David Notkin. Supporting uses of editing process patterns. In *Papers presented at Workshop on Behavior Based User Interface Customization*, January 2004.

[57] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference*, pages 187–196, New York, NY, USA, 2005. ACM Press.

[58] A.J. Ko and B.A. Myers. Development and evaluation of a model of programming errors. In *IEEE Symposium on Human Centric Computing Languages and Environments*, pages 7–14, Oct 2003.

[59] Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16:41–84, 2005.

[60] Christian Kopf. Entwicklung einer Sprache zur einfachen Beschreibung und Implementierung von Episodenerkennern. Bachelorarbeit, Freie Universität Berlin, Institut für Informatik, September 2006.

[61] H. Kou. Studying micro-processes in software development stream. Technical Report CSDL-05-03, University of Hawaii, 2005.

[62] Marco Kranz. Animation vergangener Codeänderungen von Java-Methoden. Studienarbeit, Freie Universität Berlin, Institut für Informatik, Dezember 2006.

[63] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *IWPSE '01: Proceedings of the 4th International*

Workshop on Principles of Software Evolution, pages 37–42, New York, NY, USA, 2001. ACM Press.

[64] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. A case study in root cause defect analysis. In *ICSE '00: Proceedings of the 22nd International conference on Software engineering*, pages 428–437, New York, NY, USA, 2000. ACM Press.

[65] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference*, pages 296–305, New York, NY, USA, 2005. ACM Press.

[66] Dietrich Manzey. Psychophysiologie mentaler Beanspruchung. *Enzyklopädie der Psychologie, Serie Biologische Psychologie*, 5, 1998.

[67] Gloria Mark, Victor M. Gonzalez, and Justin Harris. No task left behind?: examining the nature of fragmented work. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 321–330, New York, NY, USA, 2005. ACM Press.

[68] A. Merceron and K. Yacef. Educational data mining: a case study. *Proceedings of the 12th International Conference on Artificial Intelligence*, pages 467–474, 2005.

[69] Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.

[70] Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson. Mining student CVS repositories for performance indicators. In *MSR '05: Proceedings of the 2005 International workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[71] A. Mockus and D.M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April–June 2000.

[72] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, pages 120–130, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[73] Carlton A. Moore. Lessons learned from teaching reflective software engineering using the leap toolkit. In *ICSE '00: Proceedings of the 22nd International conference on Software engineering*, pages 672–675, New York, NY, USA, 2000. ACM Press.

[74] Takeshi Nakajo and Hitoshi Kume. A case history analysis of software error cause-effect relationships. *IEEE Trans. Softw. Eng.*, 17(8):830–838, 1991.

[75] Nicolas Ngandeu. Visualisierung konzeptioneller Beschreibungen von Programmieraktivitäten. Diplomarbeit, Freie Universität Berlin, Institut für Informatik, 2007.

[76] N. Nystrom, J. Urbanic, and C. Savinell. Understanding productivity through non-intrusive instrumentation and statistical learning. In *Proceedings of the Productivity and Performance in High-End Computing (P-PHEC), Workshop at the 11th International Symposium for High Performcance Computer Architecture*, San Francisco, February 2005.

[77] Brid O'Conaill and David Frohlich. Timespace in the workplace: dealing with interruptions. In *CHI '95: Conference companion on Human factors in computing systems*, pages 262–263, New York, NY, USA, 1995. ACM Press.

[78] Masao Ohira, Reishi Yokomori, Makoto Sakai, Ken-ichi Matsumoto, Katsuro Inoue, and Koji Torii. Empirical project monitor: Automatic data collection and analysis toward software process improvement. In *MSR 2004: International Workshop on Mining Software Repositories*, pages 42–46, Edinburgh, Scotland, UK, May 2004.

[79] T.J. Ostrand and E.J. Weyuker. A tool for mining defect-tracking systems to predict fault-prone files. In *MSR 2004: International Workshop on Mining Software Repositories*, pages 85–89, Edinburgh, Scotland, UK, May 2004.

[80] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[81] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. *Understanding and Improving Time Usage in Software Development*. John Wiley & Sons Ltd, 1995.

[82] Lutz Prechelt. Accelerating learning from experience: Avoiding defects faster. *IEEE Softw.*, 18(6):56–61, 2001.

[83] Lutz Prechelt and Barbara Unger. An experiment measuring the effects of personal software process (PSP) training. *IEEE Trans. Softw. Eng.*, 27(5):465–472, 2001.

[84] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005.

[85] K. S. Rajeswari and R. N. Anantharaman. Development of an instrument to measure stress among software professionals: Factor analytic study. In *CPR '03: Proceedings of the SIGMIS Conference on Computer personnel research*, pages 34–43, New York, NY, USA, 2003. ACM Press.

[86] J. Reason. *Human Error*. Cambridge University Press, 1990.

[87] K. Renaud. Expediting rapid recovery from interruptions by providing a visualization of application activity. In *Proceedings of OZCHI*, pages 348–355, 2000.

[88] Karen Renaud and Phil Gray. Making sense of low-level usage data to understand user activities. In *SAICSIT '04: Proceedings of the 2004 Annual research conference of the South African institute of computer scientists and information*

technologists on IT research in developing countries, pages 115–124, Pretoria, SA, 2004.

[89] Hannes Restel. Automatisches Erkennen von Trial-and-Error-Episoden beim Programmieren. Bachelorarbeit, Freie Universität Berlin, Institut für Informatik, September 2006.

[90] Robert S. Rist. Plans in programming: definition, demonstration, and development. In *Papers presented at the First workshop on Empirical studies of programmers*, pages 28–47, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[91] M.P. Robillard and G.C. Murphy. Program navigation analysis to support task-aware software development environments. In *WoDiSEE2004: Workshop on Directions in Software Engineering Environments*, pages 83–88, Edinburgh, Scotland, UK, May 2004. IEEE Computer Society.

[92] Pierre N. Robillard, Patrick d'Astous, Françoise Détienne, and Willemien Visser. Measuring cognitive activities in software engineering. In *ICSE '98: Proceedings of the 20th International conference on Software engineering*, pages 292–299, Washington, DC, USA, 1998. IEEE Computer Society.

[93] W.B. Rouse and S.H. Rouse. Analysis and classification of human error. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:539–549, 1983.

[94] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 281, Washington, DC, USA, 2003. IEEE Computer Society.

[95] Julia Schenk. Entwurf einer Datenbankanbindung zur Speicherung und Anfrage von Mikroprozessereignissen. Studienarbeit, Freie Universität Berlin, Institut für Informatik, Dezember 2006.

[96] Frank Schlesinger and Sebastian Jekutsch. ElectroCodeoGram: An environment for studying programming. In *Papers presented at Workshop on Ethnographies of Code*. Infolab21, Lancaster University, UK, 2006.

[97] K.A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *MSR 2004: Proceedings of the International Workshop on Mining Software Repositories*, pages 106–110, Edinburgh, Scotland, May 2004.

[98] W. Schönwandt. *Denkfallen beim Planen*. Bauwelt Fundamente 74. Vieweg Verlag, Braunschweig/Wiesbaden, 1986.

[99] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572, 1999.

[100] B. A. Sheil. The psychological study of programming. *ACM Comput. Surv.*, 13(1):101–120, 1981.

[101] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 336–343, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[102] Alberto Sillitti, Andrea Janes, Giancarlo Succi, and Tullio Vernazza. Collecting, integrating and analyzing software metrics and personal software process data. In *EUROMICRO '03: Proceedings of the 29th Euromicro Conference*, page 336, Washington, DC, USA, 2003. IEEE Computer Society.

[103] Alberto Sillitti, Andrea Janes, Giancarlo Succi, and Tullio Vernazza. Monitoring the development process with Eclipse. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing*, volume 2, page 133, Washington, DC, USA, 2004. IEEE Computer Society.

[104] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, pages 209–223. IBM Press, 1997.

[105] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4), 2005.

[106] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *Readings in artificial intelligence and software engineering*, pages 507–521, 1986.

[107] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software repository mining with marmoset: an automated programming project snapshot and testing system. *SIGSOFT Softw. Eng. Notes*, 30(4), 2005.

[108] James C. Spohrer and Elliot Soloway. Novice mistakes: are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, 1986.

[109] James C. Spohrer, Elliot Soloway, and Edgar Pope. Where the bugs are. In *CHI '85: Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 47–53, New York, NY, USA, 1985. ACM Press.

[110] Webb Stacy and Jean MacMillan. Cognitive bias in software engineering. *Commun. ACM*, 38(6):57–63, 1995.

[111] B. Teasley, L.M. Leventhal, R.M. Clifford, and D.S. Rohlman. Positive test bias in software testing by professionals: What's right and what's wrong. *Journal of Applied Psychology*, 79(1):142–155, 1994.

[112] P.G. Thomas and C.B. Paine. Tools for observing study behaviour. In *Proceedings of the Psychology of Programming Interest Group*, pages 221–236, Cosenza, Italy, April 2000.

[113] R. Thomas, G.E. Kennedy, S. Draper, R. Mancy, M. Crease, H. Evans, and P. Gray. Generic usage monitoring of programming students. In *Proceedings of the 20th Annual Conference of the Australian Society for Computers in Learning in Tertiary Education (ASCILITE 03)*, pages 7–10, 2003.

[114] K. Torii, K. Matsumoto, and S. Kusumoto. Ginger: a quantitative analysis environment for improving programmer performance. Technical Report 96001, Nara Institute of Science and Technology, Japan, January 1996.

[115] Koji Torii, Ken-ichi Matsumoto, Kumiyo Nakakoji, Yoshihiro Takada, Shingo Takada, and Kazuyuki Shima. Ginger2: An environment for computer-aided empirical software engineering. *IEEE Trans. Softw. Eng.*, 25(4):474–492, 1999.

[116] Rini van Solingen, Egon Berghout, and Frank van Latum. Interrupts: Just a minute never is. *IEEE Softw.*, 15(5):97–103, 1998.

[117] Iris Vessey. Toward a theory of computer program bugs: an empirical test. *Int. J. Man-Mach. Stud.*, 30(1):23–46, 1989.

[118] W. Visser and J.M. Hoc. Expert software design strategies. In *Proceedings of the Psychology of Programming Group*, pages 235–250, London, 1990. DJ Academic Press.

[119] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Trans. Softw. Eng.*, 22(6):424–437, 1996.

[120] Anneliese von Mayrhauser and Stephen Lang. A coding scheme to support systematic analysis of software comprehension. *IEEE Trans. Softw. Eng.*, 25(4):526–540, 1999.

[121] Christian Wege. *Automated Support for Process Assessment in Test-Driven Development.* PhD thesis, Universität Tübingen, 2004.

[122] Anders Wesslén. A replicated empirical study of the impact of the methods in the PSP on individual engineers. *Empirical Softw. Eng.*, 5(2):93–123, June 2000.

[123] Claes Wohlin, Martin Höst, and Magnus C. Ohlsson. Understanding the sources of software defects: A filtering approach. In *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, pages 9–17, Limerick, Ireland, 2000. IEEE Computer Society.

[124] A.L. Wolf and D.S. Rosenblum. A study in software process data capture and analysis. In *Second International Conference on Software Process*, pages 115–124, Berlin, Germany, 1993.

[125] M. Yanagi, A. Monden, Y. Takada, and K. Torii. A tool detecting patterns of programmers' behavior when they likely inject bugs. Technical Report 334, Nara

Institute of Science and Technology, Nov 1994. (in Japanese, discussion due to personell communication with A. Monden).

[126] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.