Haben wir Programmverstehen schon ganz verstanden?

Rainer Koschke Rebecca Tiarks

Arbeitsgruppe Softwaretechnik
Fachbereich Mathematik und Informatik
Universität Bremen
Postfach 33 04 40
28334 Bremen
{koschke,beccs}@tzi.de

Abstract: Langlebige Systeme müssen kontinuierlich von Entwicklern angepasst werden, wenn sie nicht an Wert verlieren sollen. Ohne ausreichendes Verständnis des Änderungswunsches und des zu ändernden Gegenstands kann die Anpassung nicht effizient und effektiv vorgenommen werden. Deshalb ist es wichtig, das System so zu strukturieren, dass Entwickler es leicht verstehen können. Methoden und Werkzeuge müssen bereitgestellt werden, um die Aktivitäten bei der Änderung zu unterstützen. Dazu ist ein umfassendes Verständnis notwendig, wie überhaupt Entwickler Programme verstehen.

In diesem Beitrag geben wir eine Übersicht über den Stand der Wissenschaft zum Thema Programmverstehen und identifizieren weiteren Forschungsbedarf.

1 Einführung

Die Möglichkeit zur kontinuierlichen Wartung ist die Voraussetzung für langlebige Systeme, weil nur stete Anpassungen an geänderte Rahmenbedingungen Software nützlich erhält. Eine Reihe von Studien zeigt, dass die meisten Ressourcen in der Softwareentwicklung gerade in der Wartung und nicht für die initiale Erstellung von neuen Systemen verwendet werden [AN91, Art88, LS81, Mar83]. In der Wartung werden Fehler beseitigt, umfassende Restrukturierungen vorgenommen oder neue Funktionalität eingebaut.

Wartbarkeit ist keine absolute Eigenschaft eines Systems; sie hängt neben inneren Eigenschaften des Systems immer ab von der Art der Änderung und nicht zuletzt auch von den Entwicklern, die sie durchführen sollen. So können manche Entwickler eine Änderung schnell vornehmen, während andere dafür sehr viel mehr Zeit benötigen. Diese Unterschiede ergeben sich aus unterschiedlichen kognitiven Fähigkeiten, Vertrautheit mit dem System und Erfahrung in der Wartung, aber auch durch unterschiedliche Vorgehensweisen.

Wenn wir also bei langlebigen Systemen für nachhaltige Wartbarkeit sorgen wollen, müssen wir nicht nur zukünftige Änderungen beim initialen Entwurf antizipieren, sondern das System auch so strukturieren und dokumentieren, dass die zukünftigen Wartungsentwickler in der Lage sind, das System für den Zweck ihrer Änderung soweit zu verstehen, dass sie es anpassen und testen können. Dies erfordert ein grundlegendes Verständnis, wie Entwickler

Programme überhaupt verstehen. Wenn dieser Verstehensprozess ausreichend verstanden ist, können wir für verständnisfördernde Systemstrukturen sorgen und Wartungsentwicklern effektive und effiziente Methoden und Werkzeuge bereitstellen. Diese sollten sich nach dem jeweiligen Wartungsziel richten, um sowohl die korrektive als auch die adaptive Wartung optimal unterstützen zu können.

Die Forschung hat sich deshalb dem Thema Programmverstehen intensiv gewidmet. Jedoch flachte das Interesse nach einer Blütezeit in den Achtziger Jahren, in der vorwiegend Theorien über kognitive Strategien entwickelt wurden, wieder ab. Hin und wieder erscheinen auch in den letzten Jahren einzelne Publikationen zu diesem Thema. Dennoch können wir nicht belastbar behaupten, die Forschungsagenda, die im Jahre 1997 bei einem internationalen Workshop zu empirischen Studien in der Softwarewartung aufgestellt wurde, abgearbeitet zu haben [KS97]. Die dort speziell in Bezug auf Entwickler geäußerten offen Fragen lauten:

1. What tasks do people spend time on?
2. Time spent on comprehension?
3. Demographics of maintainers – age, technical experience, language experience, application experience?
4. What skills are needed?
5. What makes a great maintainer great?

Diese Fragen sind noch immer weitgehend offen, auch wenn einzelne Studien versucht haben, Teile von ihnen zu beantworten. Eine sehr bekannte und vielzitierte Studie, die die zweite offene Frage adressiert, stammt aus dem Jahr 1979 [FH79]. In dieser Studie kommen die Autoren zum Schluss, dass Wartungsentwickler rund die Hälfte (bei korrektiver Wartung sogar bis zu 60 %) ihrer Zeit mit dem Verständnis des Problems und des Systems verbringen. Schon auf dem bereits angesprochenen Workshop wurde auf fehlende Vergleichsstudien hingewiesen [KS97]. Umso mehr ist es heute unklar, ob diese Zahlen auf heutige Verhältnisse übertragbar sind. Außerdem hat diese Studie nur eine globale Aussage zum Programmverstehen gemacht (50 % des Aufwands in der Wartung), aber nicht weiter untersucht, wie lange die Programmierer dabei mit einzelnen Aktivitäten beschäftigt sind. Schließlich ist Zweifel an der Verlässlichkeit der Aussage zum Aufwand des Programmverstehens angebracht, da die Zahlen lediglich über Umfragen erhoben wurden, ohne tatsächlich selbst den Aufwand bei Wartungsaufgaben zu messen. Dennoch zitieren viele wissenschaftlichen Publikationen in Ermangelung von Alternativen diese Studie.

In diesem Beitrag geben wir eine Übersicht über den Stand der Wissenschaft zum Thema Programmverstehen und identifizieren weiteren Forschungsbedarf. Die folgende Darstellung zum Stand der Forschung im Programmverstehen gliedern wir in drei Themenfelder:

- **Untersuchung von kognitiven Aspekten:** In dieser Kategorie geht es um die Frage, wie ein Entwickler bzw. eine Entwicklerin vorgeht, wenn er oder sie ein Programm verstehen möchte.

- **Entwicklung von Hilfsmitteln:** Dieses Feld befasst sich mit der Frage, wie Werkzeuge beim Programmverstehen helfen können und wie diese aussehen sollten.
- **Bisherige empirische Studien:** Hier fassen wir frühere Studien zusammen, die empirisch neue Technologien und Werkzeuge sowie kognitive Aspekte untersuchten.

2 Untersuchung von kognitiven Aspekten

Die Erforschung der kognitiven Aspekte des Programmverstehens befasst sich mit menschlichen Denkprozessen von Programmierern bei der Analyse von Programmen. Sie ergründet, welche Strategien ein Programmierer beim Verstehen von Programmen verfolgt. Um diese Strategien zu untersuchen, muss man ermitteln, welche Informationen der Programmierer nutzt, um ein Stück Software und das zugrunde liegende Modell zu begreifen, und wie er diese verwendet. Im Verstehensprozess können zwei grundlegend verschiedene Strategien verfolgt werden. Der *Top-Down*-Ansatz geht davon aus, dass Hypothesen generiert werden und diese in Richtung auf niedrigere Abstraktionsebenen geprüft, verfeinert und gegebenenfalls revidiert werden. Beim *Bottom-Up*-Ansatz hingegen wird die Repräsentation eines Programms von unten nach oben aufgebaut, d.h. bei der Ableitung der Verfahrensweise eines Programmteils wird allein vom Quelltext ausgegangen, ohne dass vorher Annahmen über das Programm getroffen werden.

Ein Beispiel für den *Bottom-Up*-Ansatz ist das Modell von Soloway und Ehrlich [SE84]. Ihre Analyse basiert auf der Erkennung von Konzepten im Programm-Code. Diese Konzepte werden zusammengefasst zu Teilzielen und diese wiederum zu Zielen. Verschiedene Experimente bestätigen dieses Modell, und ein von Letowsky [Let88] entwickeltes Werkzeug führt Teile des Analyseprozesses automatisch aus. Auch Shneiderman und Mayer [SM79] beschreiben ein *Bottom-Up*-Modell. Sie unterscheiden zwei Arten von Wissen über ein Programm: syntaktisches und semantisches Wissen. Das syntaktische Wissen ist sprachabhängig und beschreibt die grundlegenden Einheiten und Anweisungen im Quelltext. Das semantische Wissen hingegen ist sprachunabhängig. Es wird schrittweise aufgebaut, bis ein mentales Modell entsteht, das den Anwendungsbereich beschreibt. Das *Bottom-Up*-Modell von Pennington [Pen87] besteht aus zwei Teilen: dem Situationsmodell und dem Programmmodell. Bei unbekanntem Code wird durch die Abstraktion des Kontrollflusses das Programmmodell gebildet. Dies geschieht in einem *Bottom-Up*-Prozess durch Zusammenfassen von kleinen Programmfragmenten (Anweisungen, Kontrollstrukturen) zu größeren Einheiten. Nach dem Programmmodell wird das Situationsmodell gebildet. Das Situationsmodell ergibt sich *bottom-up*, indem durch Rückverweise auf das Programmmodell eine Abstraktion des Datenflusses und eine funktionale Abstraktion entsteht.

Ein Verfechter des *Top-Down*-Modells ist Brooks [Bro83]. Die Grundlage seines Modells des Programmverstehens bilden drei Pfeiler:

- Der Programmierprozess ist eine Konstruktion von Abbildungen von einem Anwendungsbereich über ein oder mehrere Zwischenbereiche auf einen Programmierbereich.

- Der Verstehensprozess umfasst die Rekonstruktion von allen oder Teilen dieser Abbildungen.
- Der Rekonstruktionsprozess wird durch Erwartungen geleitet, d.h. durch das Aufstellen, Bestätigen und Verfeinern von Hypothesen.

Beim Verstehensprozess werden die Abbildungen, die bei der Entwicklung des Programms als Grundlage dienten, rekonstruiert. Die Menge von Abbildungen und ihre Abhängigkeiten untereinander sind baumähnlich aufgebaut. Hypothesen sind Annahmen über diese Abhängigkeiten. Die primäre Hypothese bildet die Wurzel. Sie beschreibt, was das Programm nach dem gegenwärtigen Verständnis des Programmierers tut. Das Verfeinern der primären Hypothese führt zu einer Kaskade von ergänzenden Hypothesen. Dieser Prozess wird solange fortgeführt, bis eine Hypothese gefunden ist, die präzise genug ist, um sie anhand des Codes und der Dokumentation zu verifizieren oder zu falsifizieren. Präzise genug heißt, dass die Hypothese Abläufe oder Datenstrukturen beschreibt, die mit sichtbaren Elementen im Quellcode oder der Dokumentation in Zusammenhang gebracht werden können. Diese sichtbaren Elemente werden mit dem Begriff *Beacons* beschrieben. Ein typischer *Beacon* für das Sortieren in einem Datenfeld wäre z.B. ein geschachteltes Paar von Schleifen, in denen zwei Elemente miteinander verglichen und vertauscht werden [Bro83].

Ein weiteres Modell, das aus einer Kombination der beiden vorherigen Modelle besteht, ist das *wissensbasierte Modell* von Letovsky [Let86]. Es geht davon aus, dass Programmierer sowohl *top-down* als auch *bottom-up* vorgehen. Es besteht aus drei Komponenten:

- Die Wissensbasis beinhaltet Fachwissen, Wissen über den Problembereich, Stilregeln, Pläne und Ziele.
- Das mentale Modell beschreibt das gegenwärtige Verständnis des Programmierers über das Programm.
- Der Prozess des Wissenserwerbs gleicht Quellcode und Dokumentation mit der Wissensbasis ab. Dadurch entwickelt sich das mentale Modell weiter. Der Prozess kann sowohl *top-down* als auch *bottom-up* erfolgen.

Nach Letovsky [Let86] gibt es drei Arten von Hypothesen:

- Warum-Vermutungen fragen nach dem Zweck eines Code-Fragments.
- Wie-Vermutungen fragen nach der Umsetzung eines Zwecks.
- Was-Vermutungen fragen nach, was etwas ist und was es tut (z.B. eine Funktion, die eine Variable setzt).

Das *integrierte Metamodell* von Mayrhauser und Vans[vMV95] ist ebenso eine Synthese der bereits erläuterten Modelle. Es besteht aus drei Komponenten, die Verstehensprozesse darstellen und verschiedene mentale Repräsentationen von Abstraktionsebenen enthalten. Die vierte Komponente bildet die Wissensbasis, die die Informationen für den Verstehensprozess bereit stellt und diese speichert.

In allen Modellen lassen sich gemeinsame Elemente wiederfinden. So verfügt ein Programmierer über Wissen aus zwei Bereichen: das allgemeine und das systemspezifische

Softwarewissen. Das allgemeine Wissen ist unabhängig von einem konkreten System, das verstanden werden soll, und umfasst Kenntnisse über Algorithmen, Programmiersprachen und allgemeine Programmierprinzipen. Das systemspezifische Wissen hingegen repräsentiert das gegenwärtige Verständnis des spezifischen Programms, das der Programmierer betrachtet. Während des Verstehensprozesses erwirbt der Programmierer weiteres systemspezifisches Wissen, jedoch kann ebenso zusätzliches allgemeines Wissen nötig sein [vMV95]. Der Prozess des Programmverstehens gleicht neu erworbenes Wissen mit Bestehendem ab und prüft, ob zwischen beiden eine Beziehung besteht, d.h. ob sich bekannte Strukturen im neuen Wissen befinden. Je mehr Korrelationen erkannt werden, desto größer ist das Verständnis, und es bildet sich ein mentales Modell. Das mentale Modell beim Programmverstehen ist eine interne Repräsentation der betrachteten Software und setzt sich aus statischen und dynamischen Elementen zusammen [vMV95].

Man unterscheidet beim Programmverstehen zwischen zwei Strategien: die *opportunistische* und die *systematische* Vorgehensweise. Bei einem systematischen Ansatz geht ein Programmierer in einer systematischen Reihenfolge vor, um ein Verständnis für das Programm als Ganzes zu erlangen, z.B. durch zeilenweises Verstehen des Codes [vMVL98]. Beim opportunistischen Ansatz geht der Programmierer nach Bedarf vor und konzentriert sich nur auf die Teile des Codes, von denen er denkt, dass sie für seine aktuelle Aufgabe von Bedeutung sein könnten, ohne die weiteren Abhängigkeiten zu anderen Programmteilen weiter zu betrachten. [LPLS86, KR91, LS86].

Die kognitive Psychologie befasst sich außerdem mit der Frage, welche weiteren Faktoren einen Einfluss darauf haben, wie gut ein Programm verstanden werden kann. Mögliche Faktoren sind z.B. die Programmeigenschaften, die individuellen Unterschiede von Programmierern und die Aufgabenabhängigkeit [SWM00]. Zu den Programmeigenschaften gehören unter anderem die Dokumentation, die Programmiersprache und das Programmierparadigma, das einer Sprache zugrunde liegt. Die Programmierer selber unterscheiden sich ebenso durch eine Reihe von individuellen Eigenschaften in Bezug auf ihre Fähigkeiten und Kreativität. Weiterhin können sich die Vorgehensweisen beim Programmverstehen je nach Aufgabe und Wartungsfragestellung unterscheiden. Programmverstehen stellt kein eigenständiges Ziel im Wartungsprozess dar, sondern ist vielmehr ein nötiger Schritt, um Änderungen an einem Programm durchzuführen, Fehler zu beheben oder Programmteile zu erweitern. Somit hängt auch der Verstehensprozess von der jeweiligen Aufgabe ab.

3 Entwicklung von Hilfsmitteln

Computergestützte Hilfsmittel können beim Programmverstehen von großem Nutzen sein, indem sie Programmierern helfen, Informationen herzuleiten, um daraus Wissen auf einem höheren Abstraktionsniveau zu schaffen.

Reverse-Engineering ist ein wichtiger Begriff im Zusammenhang mit der Unterstützung des Programmverstehens. Nach Chikofsky und Cross [CC90] ist Reverse-Engineering der Analyseprozess eines Systems, um die Systemkomponenten und deren Beziehungen zu identifizieren. Das Ziel des Reverse-Engineerings ist es, ein Softwaresystem zu verstehen, um das Korrigieren, Verbessern, Umstrukturieren oder Neuschreiben zu erleichtern

[Rug92]. Reverse-Engineering kann sowohl auf Quellcodeebene als auch auf Architektur- oder Entwurfsebene einsetzen.

Die maschinelle Unterstützung von Programmverstehen ist Gegenstand der aktiven Forschung. Die resultierenden Werkzeuge lassen sich in drei Kategorien einteilen [TS96]: Werkzeuge zur Extraktion, zur Analyse und zur Präsentation. Werkzeuge, die sich mit der Extraktion von Daten befassen, sind beispielsweise Parser. Analysetools erzeugen Aufrufgraphen, Modulhierarchien oder Ähnliches mit Hilfe von statischen und dynamischen Informationen. Bei der statischen Analyse werden die Informationen über ein Programm aus dem zugrunde liegenden Quelltext extrahiert. Im Gegensatz dazu wird bei der dynamischen Analyse das Programm mit Testdaten ausgeführt. Zu den Tools, die sich mit der Präsentation der Informationen befassen, gehören Browser, Editoren und Visualisierungswerkzeuge. Moderne Entwicklungsumgebungen und Reverse-Engineering-Werkzeuge vereinen oft Eigenschaften aus mehreren Kategorien in einem integrierten Werkzeug.

Die existierenden Analysen werden in *grundlegende* und *wissensbasierte* Analysen unterschieden [KP96]. Die grundlegenden Analysen ermitteln Informationen über das Programm und präsentieren diese in geeigneter Form. Der Betrachter nimmt mithilfe dieser Informationen die Abstraktion selber vor. Grundlegende statische Analysen erleichtern den Zugriff auf Informationen aus dem Quelltext. Sie verwenden ausschließlich Daten, die direkt aus dem Quelltext oder anderen Dokumenten abgeleitet werden können. Ein Beispiel hierfür sind Kontrollflussanalysen. Grundlegende dynamische Analysen hingegen ermitteln Informationen während des Programmlaufs.

Die wissensbasierten Analysen verfügen über Vorwissen. Dieses spezielle Wissen über Programmierung, Entwurf oder den Anwendungsbereich ermöglicht es den Analysen, automatisch zu abstrahieren. Nach Soloway und Ehrlich [SE84] verfügen erfahrene Programmierer über eine Basis an Wissen, die es ihnen ermöglicht, schneller als Unerfahrene zu abstrahieren. Anhand dieser Wissensbasis versuchen wissensbasierte Analysen, automatisch zu abstrahieren.

Die Forschung auf dem Gebiet der Softwarevisualisierung hat zu einer Reihe von Werkzeugen zur Unterstützung des Programmverstehens geführt. Bei der Bildung eines mentalen Modell spielt die Präsentation der extrahierten Information eine wichtige Rolle. Ein Modell zur Bewertung von Visualisierungen wurde von Pacione et al. [PRW04] vorgestellt. Mit Hilfe dieses Modells können Visualisierungen anhand ihrer Abstraktion, ihres Blickwinkels und ihren zugrundeliegenden Informationen evaluiert werden. Neuere Ansätze befassen sich außerdem mit der Integration von auditiver Unterstützung in moderne Entwicklungsumgebungen [HTBK09, SG09].

Es ergeben sich verschiedene Anforderungen an die Werkzeuge, die das Programmverstehen unterstützen sollen. Nach Lakhotia [Lak93] sollte ein Werkzeug die partielle Rekonstruktion des System-Designs ermöglichen. Singer et al. [SLVA97] identifizieren drei wichtige Funktionen, die ein Werkzeug bieten sollte. Zum einen sollte es dem Benutzer ermöglichen, nach Artefakten im Programm zu suchen, und zum anderen sollte das Werkzeug die Abhängigkeiten und Attribute der gefundenen Artefakte in geeigneter Form darstellen können. Um bereits gefundene Informationen zu einem späteren Zeitpunkt wie-

derverwenden zu können, sollte das Werkzeug außerdem die Suchhistorie speichern.

Einen weiteren wichtigen Aspekt formulieren Mayrhauser und Vans [vMV93]: ein Werkzeug sollte die verschiedenen Arbeitsweisen von Programmierern unterstützen, statt die Aufgaben im gesamten Wartungsprozess fest vorzugeben.

Leider ist festzuhalten, dass bisherige empirische Studien zur Eignung von Werkzeugen für das Programmverstehen oft nur mit einer geringen Anzahl von Teilnehmern und an kleineren Systemen durchgeführt wurden. Außerdem stehen bei den meisten Untersuchungen die Werkzeuge im Vordergrund und nicht die Verhaltensweisen und kognitiven Prozesse der Programmierer. Der steigende Bedarf an weitreichenderen Werkzeugen, die ein Benutzermodell entwickeln und den Entwickler somit besser proaktiv unterstützen können, erfordert jedoch mehr empirische Grundlagen auf diesem Gebiet.

4 Bisherige empirische Studien

Sobald der Nutzen einer Methode, einer Theorie oder eines Werkzeugs von menschlichen Faktoren abhängt, sind Studien und Experimente mit Menschen erforderlich. In Bezug auf das Programmverstehen sind empirische Studien sowohl für die Erforschung kognitiver Aspekte als auch für die Entwicklung von unterstützenden Anwendungen wichtig. Sie bieten die einzige Möglichkeit, Hypothesen systematisch zu bestätigen oder zu widerlegen, da mathematische Beweisführung für dieses Feld nicht in Frage kommt [Tic98, RR08].

Eine Reihe von Experimenten zum Programmverstehen beschreiben Mayrhauser und Vans [vMV95]. Die meisten dokumentierten Versuche beziehen sich allerdings auf kleine Programme mit einigen hundert Zeilen Code. Es stellt sich die Frage, ob die Ergebnisse auch für größere Programme geltend gemacht werden können. Außerdem fehlen Aussagen über die Anwendung von Fachkenntnissen, da den meisten Untersuchungen lediglich Wissen über die statische Struktur eines Programms zu Grunde liegt. Ein großer Teil der Erkenntnisse stammt zudem von Studien, die bereits über zehn Jahre zurück liegen, aber bisher nicht wieder bestätigt oder widerlegt worden sind; so z.B. auch die Ergebnisse von Fjeldstad und Hamlen [FH79]. Bei den neueren Studien stellt sich die Frage nach der externen Validität und der statistischen Signifikanz, da sie auf einer starken Vereinfachung des Umfelds beruhen. So wurden bei der Untersuchung von Ko et al. [KDV07] beispielsweise nur ein Entwickler in einem einzelnen Unternehmen beobachtet. Murphy et al. [MN97] begründen den Nutzen der Reflektionsmethode anhand einer Fallstudie, an der nur ein Programmierer beteiligt war. Außerdem sind viele der Studien nur auf ein Werkzeug oder eine Programmiersprache beschränkt (z.B. Eclipse: [SDVFM05, KAM05, LOZP06], Java: [DH09b], C++:[MW01].

Die Resultate aus der Studie von Soloway und Ehrlich [SE84] belegen, dass Programmierer Programme besser verstehen, wenn sie mit bestimmten Konzepten erstellt worden sind und sie bestimmten Programmierrichtlinien folgen, jedoch waren die verwendeten Programme kurze, künstlich erstellte Code-Fragmente, so dass sich wieder die Frage nach der Übertragbarkeit auf große Programme stellt. Littman et al. [LPLS86] haben Wartungsprogrammierer beobachtet und dabei festgestellt, dass diese entweder einer *opportunistischen* oder einer *systematischen* Vorgehensweise folgen. Die Programmierer, die systematisch

versucht haben, den Code zu verstehen, konnten ein mentales Modell des Programms aufbauen. Dadurch waren sie erfolgreicher bei der Umsetzung von Änderungen im Code. Ergebnisse zur Nützlichkeit von Werkzeugen stammen von Bennicke und Rust [BR04]. Die Experimente belegen, dass der Einsatz von Werkzeugen dazu führen kann, dass die Wartung effizienter und effektiver wird. Auch die Ergebnisse von Storey et al. [SWM00] zeigen, dass die drei von ihnen verglichenen Werkzeuge die bevorzugten Verstehensstrategien der Programmierer beim Lösen der gestellten Aufgabe unterstützen. Aktuellere Experimente haben sich mit der Frage befasst, inwieweit eine auditive Unterstützung, neben der visuellen, beim Programmverstehen hilfreich sein kann [SG09, HTBK09]. Sie belegen, dass der Einsatz von auditiven Informationen den Verstehensprozess positiv beeinflusst.

Eine Studie von Robillard et al. [RCM04] befasst sich mit den Kennzeichen effektiven Programmverstehens, also mit der Frage, inwieweit Strategien beim Programmverstehen Einfluss haben auf den Erfolg einer Änderungsaufgabe. Die Ergebnisse belegen zwar, dass systematisches Vorgehen beim Programmverstehen bessere Erfolgsquoten bei der Änderungsaufgabe erzielt als das zufällige Vorgehen; allerdings waren bei dem Experiment nur fünf Programmierer beteiligt. Weitere Studien untersuchen Behauptungen über den positiven Einfluss bestimmter neuer Techniken auf das Programmverstehen. Beispiele für diese Kategorie von Studien sind die Arbeiten von Prechelt et al. [PULPT02] über den Einfluss von Dokumentation von Entwurfsmustern, von Arisholm et al. [AHL06] über den Einsatz von UML sowie von Murphy et al. [MWB98] über den Einfluss von aspektorientierten Konzepten auf Aktivitäten in der Wartung. Ein weiterer Mehrwert dieser Studien über die Überprüfung einer konkreten Hypothese hinaus ist der Beitrag zur empirischen Methodik. Sie adaptieren und verfeinern allgemeine empirische Methoden für konkrete Untersuchungen im Bereich Programmverstehen. Arbeiten zur empirischen Methodik finden sich unter anderem bei Di Penta et al. [PSK07, LP06].

Mit den individuellen Unterschieden von Programmierern befasst sich eine Studie von Crosby et al. [CSW02]. Sie untersucht, welchen Einfluss der Wissensstand eines Programmierers auf das Erkennen der von Brooks [Bro83] identifizierten *Beacons* hat. Demnach erkennen erfahrenere Programmierer eher die sogenannten *Beacons* und finden dadurch schneller die für das Verständnis wichtigen Teile im Quelltext. Nach Höfer und Tichy [HT06] fehlen allerdings empirische Untersuchungenen zu den individuellen Eigenschaften von Programmierern, die neben der Softwaretechnik auch andere Disziplinen wie Sozialwissenschaften und Psychologie miteinbeziehen.

Eine Untersuchung von Ko [Ko03] belegt, dass die Erfahrung eines Programmierers Einfluss hat auf die Strategie, die er anwendet, wenn er mit einer unbekannten Programmierumgebung und Programmiersprache konfrontiert ist. Die Ergebnisse zeigen auch, dass für Wartungsarbeiten im Bereich der Fehlerbeseitigung das systemspezifische Wissen eher von Bedeutung ist als die Erfahrung eines Programmierers. Wie Programmierer vorgehen, wenn sie nach einer Unterbrechung eine Aufgabe erneut aufnehmen und somit sich somit an bereits erlangtes Wissen erinnern müssen, beschreibt eine Studie von Parnin und Rugaber [PR09].

Die Frage nach dem Einfluss von verschiedenen Codecharakteristika auf das Programmverstehen hat in den letzten Jahren zu einer Reihe von Untersuchungen geführt. Eine ältere Studie von Arab [Ara92] untersucht, inwieweit die Formatierung von Quelltext und Kom-

mentare das Programmverstehen unterstützen können. Weitere Studien über den Nutzen von Dokumentation stammen unter anderem von [BC05, DH09a, DH09b, SPL+88]. Alle Untersuchungen haben gezeigt, dass verschiedene Codecharateristika das Programmverstehen beeinflussen.

Inwieweit sich geschlechtsspezifische Unterschiede bei der räumlichen Wahrnehmung auf den Verstehensprozess übertragen lassen, untersuchen Fisher et al. [FCZ06] basierend auf der Hypothese von Cox et al. [CFO05], dass sich zwischen dem Programmverstehen und der Raumkognition Gemeinsamkeiten identifizieren lassen. Diese Studie ist eine der wenigen, die explizit die Unterschiede des Programmverstehens hinsichtlich des Geschlechtes betrachtet.

5 Fazit

Das Ziel, den Aufwand der Wartung und damit die Gesamtkosten der Softwareentwicklung zu reduzieren, hat den Anstoß zu einer Reihe von Untersuchungen zum Programmverstehen gegeben. Diesen Studien liegen jedoch hauptsächlich Experimente mit relativ kleinen Programmen zu Grunde. Die meisten Untersuchungen wurden zudem nur mit einer kleinen Anzahl von Teilnehmern durchgeführt, wodurch sich die Frage nach der Verallgemeinerbarkeit der Ergebnisse stellt. Es fehlen Aussagen zu größeren Systemen sowie eine klare Methodologie für das Programmverstehen, die methodisches Vorgehen nach der Wartungsaufgabe differenziert. Außerdem liegen viele der Studien bereits Jahrzehnte zurück, und die heutige Gültigkeit der Ergebnisse angesichts moderner Programmiersprachen und Entwicklungsumgebungen steht in Frage. Die Trends der heutigen Softwareprojekte, wie z.B. das stärkere Einbinden von Open-Source-Komponenten, die zunehmende Verteilung der Entwickler oder die kürzeren Entwicklungszyklen, haben auch zu neuen Erkenntnisse über die Entwicklungsmethoden, die Eigenschaften der entwickelten Programme und das Verhalten der Entwickler geführt. Das hinterfragt zusätzlich die früheren Erkenntnisse über das Programmverstehen und erhöht den Bedarf nach aktuellen, detaillierten und signifikanten Untersuchungen.

Auf Basis unserer Literaturstudie zum Thema Programmverstehen formulieren wir die folgenden offenen Fragen, die zukünftige Forschung adressieren sollte:

- **Vorgehensweisen:** Wie gehen Programmierer – abhängig von der Art ihrer Aufgabe – genau vor, wenn sie Programme ändern sollen? Aus welchen Einzelaktivitäten besteht der Programmverstehensprozess für welche Art von Aufgaben? Welche Informationen werden abhängig von der Aufgabe benötigt? Warum gehen Programmierer so vor? Worin unterscheiden sich die Vorgehensweisen unterschiedlicher Programmierer? Welche Vorgehensweisen führen eher zum Erfolg?
- **Aufwand:** Wie hoch ist der Aufwand des Programmverstehens in der Wartungsphase im Verhältnis zu Änderung und Test? Wie hoch ist der Aufwand der Einzelaktivitäten des Programmverstehens? Wie hoch sind die Kosten, wenn bestimmte Informationen fehlen?

- **Werkzeuge:** Wie werden moderne Entwicklungswerkzeuge beim Programmverstehen benutzt? Wie gut eignen sie sich für ihren Zweck? Wie lassen sie sich verbessern? Für welche Aspekte existiert noch unzureichende Werkzeugunterstützung? Welches Wissen kann kaum von Werkzeugen verwaltet werden?
- **Methoden:** Was sind gut geeignete Vorgehensweisen beim Programmverstehen für wiederkehrende Aufgabenarten? Welche Informationen und Werkzeuge sind geeignet, um diese Vorgehensweisen zu unterstützen?
- **Code-Strukturen:** Welche Auswirkungen haben die *Bad Smells* von Beck und Fowler auf das Programmverständnis?
- **Einfluss von Architektur** Wie wird Architektur in der Praxis dokumentiert? Wie wird sie von Programmierern benutzt? Welche Probleme gibt es für den Austausch des Architekturwissens?
- **Wissenschaftliche Methoden:** Welche Methoden aus den Sozial- und Kognitionswissenschaften und der Psychologie lassen sich für die Forschung im Programmverstehen wie übertragen? Wie können empirische Beobachtungen und Experimente durch Softwaresysteme unterstützt werden?

Literatur

[AHL06] L.C. Arisholm, E.and Briand, S.E. Hove und Y. Labiche. The impact of UML documentation on software maintenance: an experimental evaluation. *IEEE Computer Society TSE*, 32(6):365–381, Juni 2006.

[AN91] A. Abran und H. Nguyenkim. Analysis of maintenance work categories through measurement. In *Software Maintenance, 1991., Proceedings. Conference on*, Seiten 104–113, Oct 1991.

[Ara92] Mouloud Arab. Enhancing program comprehension: formatting and documenting. *SIGPLAN Not.*, 27(2):37–46, 1992.

[Art88] Lowell Jay Arthur. *Software Evolution: A Software Maintenance Challenge*. John Wiley & Sons, 2 1988.

[BC05] Scott Blinman und Andy Cockburn. Program comprehension: investigating the effects of naming style and documentation. In *Proc. of AUIC'05*, Seiten 73–78, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[BR04] M. Bennicke und H. Rust. Programmverstehen und statische Analysetechniken im Kontext des Reverse Engineering und der Qualitätssicherung, April 2004. Virtuelles Software Engineering Kompetenzzentrum.

[Bro83] Ruven E. Brooks. Towards a Theory of the Comprehension of Computer Programs. *IJMMS*, 18(6):543–554, 1983.

[CC90] Elliot J. Chikofsky und James H. Cross. Reverse Engineering and Design Recovery: a Taxonomy. *IEEE Software*, 7(1):13–17, Januar 1990.

[CFO05] Anthony Cox, Maryanne Fisher und Philip O'Brien. Theoretical Considerations on Navigating Codespace with Spatial Cognition. In *Proc. of PPIG'05*, Seiten 92–105, 2005.

[CSW02] Martha E. Crosby, Jean Scholtz und Susan Wiedenbeck. The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *Proc. PPIG'02*, Seiten 18–21, 2002.

[DH09a] Uri Dekel und James Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proc. of ICSE'09*, 2009.

[DH09b] Uri Dekel und James D. Herbsleb. Reading the Documentation of Invoked API Functions in Program Comprehension. In *Proc. of ICPC'09*, Seiten 168–177, Vancouver, Canada, 2009. IEEE Computer Society.

[FCZ06] Maryanne Fisher, Anthony COx und Lin Zhao. Using Sex Differences to Link Spatial Cognition and Program Comprehension. In *Proc. ICSM'06*, Seiten 289–298, Washington, DC, USA, 2006. IEEE Computer Society.

[FH79] R.K. Fjeldstad und W.T. Hamlen. Application Program Maintenance Study: Report to our Respondents. In *Proc. of GUIDE 48*, Philadelphia, PA, 1979.

[HT06] Andreas Höfer und Walter F. Tichy. Status of Empirical Research in Software Engineering. In *Empirical Software Engineering Issues*, Seiten 10–19, 2006.

[HTBK09] Khaled Hussein, Eli Tilevich, Ivica Ico Bukvic und SooBeen Kim. Sonification DEsign Guidelines to Enhance Program Comprehension. In *Proc. ICPC'09*, Seiten 120–129, Vancouver, Canada, 2009. IEEE Computer Society.

[KAM05] Andrew J. Ko, Htet Htet Aung und Brad A. Myers. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *Proc. ICSE'05*, Seiten 126–135, 2005.

[KDV07] Andrew J. Ko, Robert DeLine und Gina Venolia. Information Needs in Collocated Software Development Teams. *Proc. ICSE'07*, Seiten 344–353, 2007.

[Ko03] Andrew Jensen Ko. Individual Differences in Program Comprehension Strategies in an Unfamiliar Programming System. In *IWPC'03*, 2003.

[KP96] R. Koschke und E. Plödereder. *Ansätze des Programmverstehens*, Seiten 159–176. Deutscher Universitätsverlag/Gabler Vieweg Westdeutscher Verlag, 1996. Editor: Franz Lehner.

[KR91] Jürgen Koenemann und Scott P. Robertson. Expert problem solving strategies for program comprehension. In *Proc. of SIGCHI'91*, Seiten 125–130, New York, NY, USA, 1991. ACM.

[KS97] Chris F. Kemerer und Sandra Slaughter. Methodologies for Performing Empirical Studies: Report from WESS'97. *EMSE*, 2(2):109–118, Juni 1997.

[Lak93] Arun Lakhotia. Understanding someone else's code: Analysis of experiences. *JSS*, 23(3):269–275, 1993.

[Let86] Stanley Letovsky. Cognitive Processes in Program Comprehension. In *Workshop ESP*, Seiten 58–79, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[Let88] Stanley Letovsky. Plan analysis of programs. Bericht Research Report 662, Yale University, Dezember 1988.

[LOZP06] Andrea De Lucia, Rocco Oliveto, Francesco Zurolo und Massimiliano Di Penta. Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment. *Proc. of ICPC'06*, 0:317–326, 2006.

[LP06] Giuseppe A. Di Lucca und Massimiliano Di Penta. Experimental Settings in Program Comprehension: Challenges and Open Issues. In *Proc. of ICPC'06*, Seiten 229–234, Washington, DC, USA, 2006. IEEE Computer Society.

[LPLS86] D. C. Littman, Je. Pinto, S. Letovsky und E. Soloway. Mental Models and Software Maintenance. In *Workshop ESP*, Seiten 80–98, Norwood, NJ, USA, 1986. Ablex Publishing Corp. Reprinted in Journal Systems and Software 7, 4 (Dec. 1987), 341–355.

[LS81] Bennet P. Lientz und E. Burton Swanson. Problems in application software maintenance. *Commun. ACM*, 24(11):763–769, 1981.

[LS86] S. Letovsky und E. Soloway. Delocalized Plans and Program Comprehension. *Software, IEEE*, 3(3):41–49, May 1986.

[Mar83] James Martin. *Software Maintenance the Problem and Its Solution*. Prentice Hall, third printing. Auflage, 6 1983.

[MN97] Gail C. Murphy und David Notkin. Reengineering with Reflexion Models: A Case Study. *IEEE Computer*, 30(8):29–36, August 1997. Reprinted in Nikkei Computer, 19, January 1998, p. 161-169.

[MW01] Russel Mosemann und Susan Weidenbeck. Navigation and Comprehension of Programs by Novice Programmers. In *Proc. of IWPC'01*, Seite 79, Washington, DC, USA, 2001. IEEE Computer Society.

[MWB98] G. Murphy, R. Walker und E. Baniassad. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-oriented Programming. Technical Report TR-98-10, University of Britsh Columbia, Computer Science, 1998.

[Pen87] N. Pennington. Comprehension Strategies in Programming. In *Workshop ESP*, Seiten 100–113, Norwood, NJ, USA, 1987. Ablex Publishing Corp.

[PR09] Chris Parnin und Spencer Rugaber. Resumption Strategies for Interrupted Programming Tasks. *Proc. of ICPC'09*, 0:80–89, 2009.

[PRW04] Michael J. Pacione, Marc Roper und Murray Wood. A Novel Software Visualisation Model to Support Software Comprehension. In *Proc. of WCRE'04*, Seiten 70–79, Washington, DC, USA, 2004. IEEE Computer Society.

[PSK07] Massimiliano Di Penta, R. E. Kurt Stirewalt und Eileen Kraemer. Designing your Next Empirical Study on Program Comprehension. In *ICPC*, Seiten 281–285. IEEE Computer Society, 2007.

[PULPT02] L. Prechelt, B. Unger-Lamprecht, M. Philippsen und W.F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Computer Society TSE*, 28(6):595–606, Juni 2002.

[RCM04] M. Robillard, W. Coelho und G. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Computer Society TSE*, 30(12):889–903, 2004.

[RR08] R. L. Rosnow und R. Rosenthal. *Beginning behavioral research: A conceptual primer*. Pearson/Prentice Hall, 6th. Auflage, 2008.

[Rug92] Spencer Rugaber. Program Comprehension for Reverse Engineering. In *AAAI Workshop*, Juli 1992.

[SDVFM05] J. Sillito, K. De Voider, B. Fisher und G. Murphy. Managing software change tasks: an exploratory study. In *ESE'05*, Seiten 10 pp.–, Nov. 2005.

[SE84] Elliot Soloway und Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE TSE*, 10(5):595–609, 1984.

[SG09] Andreas Stefik und Ed Gellenbeck. Using Spoken Text to Aid Debugging: An Emperical Study. *Proc. of ICPC'09*, 0:110–119, 2009.

[SLVA97] Janice Singer, Timothy Lethbridge, Norman Vinson und Nicolas Anquetil. An examination of software engineering work practices. In *Proc. of CASCON'97*, Seite 21. IBM Press, 1997.

[SM79] B. Shneiderman und R. Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *IJCIS*, 8(3):219–238, 1979.

[SPL+88] E. Soloway, J. Pinto, S. Letovsky, D. Littman und R. Lampert. Designing Documentation to Compensate for Delocalized Plans. *Commun. ACM*, 31(11):1259–1267, November 1988.

[SWM00] M.-A. D. Storey, K. Wong und H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[Tic98] Walter F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, Mai 1998.

[TS96] Scott R. Tilley und Dennis B. Smith. Coming Attractions in Program Understanding. Bericht, Software Engineering Institute, Pittsburgh, PA 15213, Dezember 1996.

[vMV93] Anneliese von Mayrhauser und A. Marie Vans. From Program Comprehension to Tool Requirements for an Industrial Environment. In *IWPC*, Seiten 78–86. IEEE Computer Society Press, Juli 1993.

[vMV95] Anneliese von Mayrhauser und A. Marie Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, 28(8):44–55, 1995.

[vMVL98] Anneliese von Mayrhauser, A. Marie Vans und Steve Lang. Program Comprehension and Enhancement of Software. In *Proc. of the IFIP World Computing Congress*. IEEE, August-September 1998.