

Felix Raab, Regensburg

Interaktionsdesign menschzentrierter Refactoring-Tools

Refactoring, die Umstrukturierung von Software unter Beibehaltung des äußeren Verhaltens, ist eine allgemein akzeptierte Entwicklungspraktik. Da manuelle Transformationen von Code fehleranfällig und langsam sind, ist Unterstützung durch Tools notwendig. Der Artikel gibt einen Überblick zum Stand der Forschung und zeigt, wie sich durch den Einsatz menschzentrierter und neuartiger Interaktionsmethoden Verbesserungen dieser Tools erzielen lassen.

Deskriptoren: Software, Entwicklung, Programmierung, Benutzerfreundlichkeit, Mensch-Maschine-Schnittstelle

Interaction design of user-centered refactoring tools

Refactoring describes the practice of restructuring software without changing its external behaviour. One major goal is to keep the functionality of software while improving its code structure and style. Usually, refactoring by hand is very error-prone, so adequate software tools are needed. This article describes the state of the art of refactoring tools. Furthermore, it shows how the interaction design of such tools can be improved using user-centered evaluation and design methods.

Keywords: software, development, programming, usability, HCI, refactoring

Le design d'interaction des outils de refactorisation centrés sur l'homme

La refactorisation, c.à.d. la restructuration des logiciels tout en maintenant le comportement extérieur, est une pratique de développement généralement reconnue. Partant du principe que les transformations manuelles de codes sont lents et forment une source d'erreurs, le soutien par des outils devient nécessaire. L'article donne un aperçu de l'état de la recherche et montre comment obtenir des améliorations de ces outils par l'utilisation de méthodes d'interaction novatrices et centrées sur l'homme.

Mots-clés: logiciel, développement, programmation, utilisabilité

1 Einleitung

Unter *Refactoring* versteht man die Restrukturierung von Software, ohne das beobachtbare Verhalten zu ändern (Fowler 1999). Das Hauptziel von Refactoring ist, lesbaren, einfach verständlichen und wartbaren Code zu produzieren. Das Verfahren wurde ursprünglich bereits 1992 in William Opdykes Dissertation formalisiert und ist heute weit verbreitete, allgemein akzeptierte Entwicklungspraktik. Refactoring ist deswegen wichtig, weil Software ständig angepasst werden muss, Quellcode im Laufe eines Projektes immer komplexer wird und ohne regelmäßige Refactoring-Maßnahmen degeneriert. Je länger Software existiert, desto wahrscheinlicher ist es, dass Teile davon geändert werden müssen und je komplexer das System, desto komplexer die Wartungsarbeiten (Kanat-Alexander 2012).

Es ist allgemein bekannt, dass der Großteil der Kosten von Softwareentwicklungsprojekten in der Wartungsphase entsteht. Insofern ist es wirtschaftlicher, den Aufwand für Wartungsarbeiten zu reduzieren, als den erstmaligen Implementierungsaufwand (Kanat-Alexander 2012). Studien haben gezeigt, dass Refactoring nicht nur die Softwarequalität erhöht, sondern auch die Produktivität von Entwicklungsteams steigern kann und damit letztendlich Kosten spart (Moser et al. 2008). Zusätzlich kann Refactoring die Ad-hoc-Wiederverwendbarkeit von Code erhöhen (Moser et al. 2006). Umgekehrt gibt es Belege, dass die Anhäufung von schlecht geschriebenem Code, sog. *technical debt*, negativen Einfluss auf die Softwarequalität hat, die sich unter anderem in höheren Defekt-Raten äußert (Zazworka et al. 2011).

Nach Johnson (2010) ist Softwareentwicklung in den meistens Fällen nichts anderes als kontinuierliche Programmtransformation. Dig und Johnson (2005) haben beispielsweise herausgefunden, dass 80 Prozent der API-Evolution durch Refactoring entsteht. Agile Entwicklungsmethoden wie *Extreme Programming (XP)* haben Refactoring mit ihren vielen kurzen Code-Refactor-Zyklen längst als festen Prozessbestandteil integriert. Nicht zuletzt deswegen, weil diese Methode zu besserem Programmverständnis beitragen kann und somit andere Prinzipien wie die kollektive Wissenskonstruktion unterstützt.

Zusammenfassend lässt sich also feststellen, dass Refactoring aus den oben genannten Gründen als regelmäßige und essenzielle Praktik in Entwicklungsprozesse integriert werden sollte. Diese regelmäßige Integration muss allerdings von geeigneten Tools unterstützt werden, weil manuelles Umstrukturieren langsam und fehleranfällig ist (Ge et al. 2012). Inzwischen unterstützen viele gängige Entwicklungsumgebungen wie z. B. Eclipse, Xcode oder Visual Studio automatische und semi-automatische Refactorings. Sie weisen aber teilweise gravierende Mängel hinsichtlich der Benutzerfreundlichkeit und Interaktion auf. Bevor diese Schwächen bestehender Tools und mögliche Optimierungen detaillierter aufgezeigt werden, wird im folgenden Abschnitt der Refactoring-Prozess näher erläutert. Daraus ergeben sich Anforderungen an das Design menschenzentrierter Tools für Entwickler, die im darauf folgenden Teil diskutiert werden.

2 Funktionsweise von Refactoring-Prozessen

Im Wesentlichen unterteilt sich der Refactoring-Prozess in drei Hauptphasen: Die Identifizierung sog. *Code Smells*, die Anwendung geeigneter Refactorings und die Kontrolle, dass das beobachtbare Verhalten durch die vorausgehenden Maßnahmen nicht verändert wurde. *Code Smells* (Fowler 1999) sind Programmstrukturen, die eine Verletzung allgemein anerkannter objekt-orientierter Design-Prinzipien darstellen und somit Verständnis und Wartung des Programms erschweren. In Fowlers Standwerk *Refactoring* sind jedem Code Smell bestimmte Refactorings zugeordnet, die diese Smells beheben können. Entscheidend dabei ist, dass das Programmverhalten beibehalten wird, was entweder durch formale Überprüfungen (statische Analyse) oder automatisierte Tests (z. B. Unit Tests) sichergestellt werden kann.

Das beizubehaltende Verhalten ist abhängig vom jeweiligen Anwendungsgebiet. Ebenso ist statische Analyse des Codes abhängig von verwendeter Programmiersprache und Compiler. Häufig werden hierfür der *Abstract Syntax Tree (AST)* eines Programms analysiert und damit sog. *Preconditions*, *Invariants* oder *Postconditions* (oder alles zusammen) überprüft. Ergebnis ist ein transformierter AST, der Code mit demselben äußeren Verhalten abbildet. Verfahren und Werkzeuge, die solche Transformationen möglichst korrekt und effizient durchführen können, sind Teil eines aktiven Forschungsfeldes (vgl. Overbey und Johnson 2011).

Besondere Herausforderungen an die Toolunterstützung stellen dabei dynamische Programmiersprachen wie z. B. Python, Ruby oder JavaScript, die in moderner Softwareentwicklung eine immer größere Rolle spielen (Schäfer 2012). Während bei statisch typisierten, objekt-orientierten Programmiersprachen Syntax und Semantik analysiert werden können, fehlen bei dynamischen Sprachen diese Informationen über die statische Struktur. Objekte und Eigenschaften können sich verändern, Fehler können damit oft erst zur Laufzeit erkannt werden. Manche Refactoringtools führen einfaches Suchen und Ersetzen über den AST aus, vernachlässigen dabei allerdings wichtige Zusammenhänge und verändern damit das Programmverhalten (Schäfer 2012).

Tools lassen sich grob in drei Kategorien einteilen: Manuelle, semi-automatische und vollautomatische Tools. Viele der Tools in Entwicklungsumgebungen arbeiten semi-automatisch, d. h. das Werkzeug führt die eigentliche Transformation aus, erfordert aber mit Wizards, Konfigurationsdialogen und anderen manuellen Einstellungen diverse Eingaben vom User. Vollautomatische Ansätze erkennen durch die Anwendung bestimmter Metriken Code Smells, visualisieren diese evtl. oder geben Refactoring-Vorschläge und führen anschließend die Transformation durch.

3 Wie refaktorisieren Entwickler?

Um herauszufinden, wie Entwickler refaktorisieren, und um frühere Studien zu diesem Thema zu validieren, haben Negara et al. (2012) detaillierte Nutzungsdaten durch spezielle Monitoring-Anwendungen gesammelt und ausgewertet: Automatisierte Refactorings über IDE-Tools machten nur die Hälfte aller durchgeführten Refactorings aus, die andere Hälfte wurde manuell ohne Tool-Unterstützung ausgeführt. (In einer älteren Studie analysierten Murphy-Hill et al. (2009a), dass bis zu 90 Prozent händisch refaktoriert wird.) In zusätzlichen Interviews hat sich bestätigt, dass Entwickler häufig nicht wissen, dass automatische Funktionen für bestimmte Refactorings vorhanden sind. Interessanterweise wurden einige der Mechanismen aber ebenfalls nicht verwendet, *obwohl* Entwickler Kenntnis davon hatten.

Die populärsten, automatisch ausgeführten Refactorings waren: *Extract Local Variable*, *Rename Local Variable*, *Rename Class*, *Inline Local Variable* und *Rename Method*. Die populärsten, manuell ausgeführten Refactorings: *Rename Local Variable*, *Rename Field*, *Extract Method*, *Rename Method* und *Convert Local Variable To Field*. Die

Autoren teilen die Refactorings in drei verschiedene Bereiche ein: *API Level* (z. B. *Rename Class*), *Partially Local* (z. B. *Extract Method*) und *Completely Local* (z. B. *Extract Local Variable*).

Mehr als ein Drittel und bis zu 55 Prozent der Refactorings waren gruppiert und wurden als zusammenhängender Block ausgeführt. Beweise für häufig ausgeführte sog. *Composite Refactorings* liefern auch Vakilian et al. (2012a). *Composite Refactorings* setzen sich aus *Primitive Refactorings* (Verschieben von Code, Umbenennen von Methoden etc.) zusammen und bilden größere, komplexe Transformationen. Ein Beispiel für eine derartige Kombination ist die Verkettung der Refactorings *Extract Local Variable*, *Extract Method* und *Inline Local Variable*. Auffällig war, dass bestimmte einfache Refactorings häufiger genutzt wurden, auch wenn mehr Einzelschritte nötig waren als bei zusammengesetzten Refactorings, die zusätzliche Konfiguration eines IDE-Tools erforderten (Vakilian et al. 2012a). *Composite Refactorings* haben den Vorteil, dass die Wahrscheinlichkeit möglicher Kollisionen und Fehler bei der Transformation geringer ist als bei einer Ausführung mehrerer einzelner *Primitive Refactorings*. Durch solche Verkettungen kann auch leichter hin zu *Design Patterns* refaktoriert werden.

In einem Artikel zur Frage, warum Entwickler Refactoringtools nur selten verwenden, schlugen Murphy-Hill und Black (2007a) die grundsätzliche Unterscheidung zwischen *Root Canal* und *Floss Refactoring* vor. Mit dieser Dental-Metapher soll unterschieden werden zwischen häufigen, proaktiven Refactorings (*Floss*) und selteneren, komplexeren Refactorings (*Root Canal*), die erst durchgeführt werden, wenn das Programm bereits in schlechtem Zustand ist. Entwickler nutzen häufiger *Floss Refactoring* (Murphy-Hill et al. 2009a), das sich gut mit agilen Entwicklungspraktiken vereinbaren lässt. Viele der Tools fokussieren allerdings eher auf die problematischeren *Root Canal Refactorings*.

4 Usability-Probleme aktueller Tools

Aus den oben genannten Refactoring-Prozessen ergeben sich unterschiedliche Anforderungen an geeignete Werkzeuge, die von aktuellen Entwicklungsumgebungen allerdings nur teilweise umgesetzt werden und somit Schwächen in punkto Benutzerfreundlichkeit und Interaktion aufweisen. Eines der ersten Refactoringtools, der *Smalltalk Refactoring Browser* (Roberts 1997), war anfangs angeblich so unpopulär, dass sogar die Entwickler

der Anwendung selbst den Umgang damit vermieden (Murphy-Hill und Black 2008). Einige der gängigen Tools orientieren sich allerdings nach wie vor an diesem Design und haben sich kaum verändert (Vakilian et al. 2012b).

Studien haben bereits mehrfach gezeigt, dass Refactoringtools selten benutzt werden, was einerseits bedeuten kann, dass Entwicklern die korrekten Tools unbekannt sind, die falschen Refactorings unterstützt werden oder die Bedienung zu schlecht ist (Murphy-Hill und Black 2007a, Vakilian et al. 2011). Schlechte Bedienung äußert sich beispielsweise darin, dass der Auslöse-Mechanismus oft zu aufwändig ist. Befehle sind in überfrachteten linearen Menüs mit unpräzisen Benennungen untergebracht oder müssen mit überladenen Tastaturbefehlen, deren Mapping zur eigentlichen Funktion oft inkonsistent ist, ausgelöst werden.

Weiteres Hindernis stellen modale Wizards und Konfigurationsdialoge dar, die umständlich konfiguriert werden müssen und den Workflow unterbrechen. Nach Vakilian et al. (2012b) ist der Kosten-Nutzen-Faktor für Entwickler oft nicht gegeben. Einfachere Methoden wie *Quick Assist* (Inline-Menüs direkt im Code-Editor) sind daher beliebter, allerdings nicht bei allen Entwicklern bekannt. Interessant ist in diesem Zusammenhang auch die Tatsache, dass Preview-Funktionen von Refactoring-Dialogen kaum genutzt werden, Standard-Einstellungen in Konfigurationsdialogen meist unverändert belassen und Warnungen ignoriert werden, weil Entwickler den Tools nicht trauen und größerer Refactorings lieber manuell durchführen (Vakilian et al. 2012b).

Zusätzlich werden Fehlermeldungen oft unzureichend genau kommuniziert. Als Beispiel nennen Murphy-Hill und Black (2008) das Refactoring *Extract Method*: Damit die Transformation korrekt arbeiten kann, müssen bestimmte *Preconditions* erfüllt sein (z. B. darf keine mehrfache Zuweisung lokaler Variablen im umschließenden Block auftreten). Bevor das Tool aufgerufen wird, müssen Teile des Quellcodes selektiert werden. Unklar ist häufig allerdings, was genau selektiert werden muss, damit das Refactoring ausgeführt werden kann. Fehler, die aufgrund falscher Selektionen gemeldet werden, müssen von Usern erst richtig interpretiert werden und können dazu führen, dass das Tool in Zukunft möglicherweise gar nicht mehr benutzt wird. Hinzu kommt, dass das oben erwähnte wichtige *Floss Refactoring* und *Composite Refactorings* nur schlecht von Entwicklungsumgebungen unterstützt werden.

5 Verbesserte Refactoringtools

Im Folgenden werden vorhandene Lösungsansätze für einige dieser Probleme vorgestellt. Um Entwickler bei manuellen Refactorings in Entwicklungsumgebungen auf automatische Refactorings hinzuweisen, entwickelten Ge et al. (2012) *BeneFactor*. Das Tool analysiert den Programmier-Workflow und erinnert Entwickler, dass eingebaute Refactorings vorhanden sind, die den aktuellen Vorgang automatisch abschließen können.

In „*A Model of Refactoring Tool Use*“ schlägt Murphy-Hill (2009b) Phasen und Übergänge für Refactoringtools vor: *Identify, Select, Initiate, Configure, Execute, Interpret* und *Clean Up*. Einzelne Phasen können dabei erweitert, anders angeordnet, aufgeteilt, zusammengefasst, parallelisiert oder auch weggelassen werden. Beispielsweise könnte die Error-Phase weggelassen werden, weil man davon ausgehen kann, dass Code temporär zu einem höheren Zweck umstrukturiert wird und daher für kurze Zeit nicht mehr ausführbar ist. Zusätzlich wissen Entwickler möglicherweise besser, wie Kompilierfehler zu beheben sind, als Fehlermeldungen, die ein Refactoring-tool produziert (Murphy-Hill 2009b). Durch Anpassungen dieser Phasen lassen sich also Werkzeuge entwickeln, die einige der angesprochenen Usability-Probleme verhindern.

So entwickelten Murphy-Hill und Black (2008) zur Verbesserung der Selektionsphase im Code-Editor Tools wie *Selection Assist* (farbige Hervorhebungen für Selektionen) und *Box View* (einfache Darstellung verschachtelter Statements). Zur Verbesserung von Feedback und Errors wurden sog. *Refactoring Annotations* in Eclipse integriert, die mit farbigen Überlagerungen und Pfeilen detaillierter anzeigen, wann Refactorings durchgeführt werden können. *Marking Menus* (Murphy-Hill und Black 2007b) können die Trigger-Phase durch schnelle Aktivierung mit Gesten optimieren: Ein radiales Menüsystem erlaubt schnelleren Zugriff auf einzelne Menüeinträge, die zusätzlich durch einfache Mausbewegungen in die Richtung eines Eintrags effizienter aufgerufen werden können, ohne dabei das Menü anzuzeigen. Allerdings ist diese Variante in Bezug auf Mapping einzelner Refactorings und Skalierbarkeit limitiert. *Refactoring Cues* (Murphy-Hill und Black 2007b) verbessern das Selektionsproblem, indem Phasen umgekehrt werden: Zuerst wählt der Entwickler das Refactoring aus, legt Konfigurationsoptionen fest und führt die Transformation dann basierend auf farbigen Hervorhebungen im Editor aus.

Neuere Ansätze (Lee et al. 2012) nutzen Drag-and-Drop zwischen einzelnen AST-Knoten oder innerhalb der Eclipse-Ansichten *Package Explorer* und *Outline View*.

Vorteile hierbei sind unter anderem, dass keine Menüeinträge gesucht, keine Tastaturkürzel gelernt und keine Konfigurationsoptionen festgelegt werden müssen. Zusätzlich unterstützt Drag-and-Drop *Floss Refactoring* besser, allerdings entstehen dadurch häufige Wechsel zur Mauseingabe. Inzwischen gibt es auch in Entwicklungsumgebungen wie Eclipse oder IntelliJ rudimentäre Unterstützung von Refactorings, die direkt im Editor (inline) oder mit Drag-and-Drop ausgelöst werden können.

6 Potenziale natürlicher Refactoringtools

Trotz bekannter Usability-Probleme und einiger Weiterentwicklungen, gibt es vielfältige Potenziale, Refactoringtools zu verbessern. Vor allem neuere Interaktionsmöglichkeiten, die für eine natürlichere Benutzung sorgen können, wurden bisher kaum ausgeschöpft. Tablets eignen sich mit ihren hoch auflösenden, scharfen Displays und Touch-Bedienung möglicherweise gut für das Lesen, Verstehen und Warten von Quellcode. Es ist allgemein bekannt, dass Code mehr gelesen als geschrieben wird und Entwickler einen Großteil ihrer Arbeitszeit mit Navigation von Code verbringen (Bragdon et al. 2010). Geräte mit touch-basierten Oberflächen ohne Hardware-Buttons und Tastatur stellen allerdings besondere Herausforderungen an das Interaktionsdesign.

Eine Möglichkeit der Interaktion mit solchen Geräten ist der Einsatz von Gestensteuerung, die wichtige kognitive Vorteile mit sich bringt (Appert und Zhai 2009). Im Gegensatz zu herkömmlichen linearen Menüs, Tastaturbefehlen, Wizards und Konfigurationsdialogen, können gestenbasierte Refactoringtools intuitivere und effizientere Interaktion ermöglichen: Beispielsweise können mit Gesten bzw. einem Pen zusammenhängende Codeteile selektiert werden, in derselben Interaktion Parameter enkodieren und die Transformation damit zusammen als *einen* flüssigen Interaktionsschritt ausführen. Bei der Verwendung eines Pens können mit Druck, Neigung und konfigurierbaren Buttons weitere Parameter enkodiert werden. Häufig genutzte *Composite Refactorings* werden mit user-definierbaren Gesten als zusammenhängende Blöcke ausgeführt. Kontinuierliches *Floss Refactoring* wird gefördert, da lokale Mikro-Transformationen schnell zwischen Verständnisprozessen mit dem Pen ausgeführt werden. Ein derartiges *RefactorPad* (in Anlehnung an *CodePad* von Parnin et al. (2010)) kann somit als zusätzliches Gerät mit spezialisierten Werkzeugen Restrukturierungsprozesse ergänzen. Wie geeignete Interaktionstechniken

dafür konkret aussehen, wird aktuell an der Universität Regensburg untersucht.

Zukünftige Entwicklungswerkzeuge können auch von multimodaler Interaktion profitieren, die erwiesenermaßen kognitive Anstrengung reduziert (Oviatt et al. 2004), was gerade bei Wartungsarbeiten entscheidend ist. Wenn entsprechende Technik auf sog. *Commodity Hardware* verfügbar ist, wäre beispielsweise folgendes Zukunftsszenario denkbar: *Brain-Computer-Interfaces* messen die Anstrengung beim Entwicklungsvorgang und annotieren problematische Stellen im Code automatisch im Hintergrund. Bei der späteren Durchsicht werden mit *Recommender-Systemen* und *Eye-Tracking* automatische Refactoring-Hinweise im Blickfeld des Users angezeigt. Mit *Gesteninteraktion* und der Verarbeitung von *Bewegungssensoren* auf dem Gerät navigiert der Entwickler das Programm und strukturiert Quellcode mit einem *Pen* um. In allen Phasen wird unterstützend *haptisches Feedback* für Hinweise und Fehler verwendet.

7 Fazit

Refactoring ist eine essenzielle Entwicklungspraktik, um Softwarequalität zu erhöhen und damit Kosten zu senken, weil der Aufwand in der Wartungsphase eines Projektes um ein Vielfaches höher ist als der initiale Implementierungsaufwand. Für diese Wartungsarbeiten ist Toolunterstützung wichtig, weil manuelle Umstrukturierungen fehlerhaft und langsam sind. Aktuelle Refactoringtools unterstützen diese Transformationen von Quellcode allerdings teilweise unzureichend: einerseits, weil Refactoring-Prozesse technisch komplex sind (z. B. Refactoring von dynamischen Sprachen), andererseits, weil die Benutzerfreundlichkeit aktueller Werkzeuge stark verbesserungswürdig ist. Schlechte Usability führt wiederum dazu, dass diese Tools nachweisbar wenig von Entwicklern benutzt werden. Die Funktionsweise von Refactoring-Prozessen aus Entwicklersicht und Potenziale für Verbesserungen zukünftiger Tools wurden in diesem Artikel aufgezeigt.

Literatur

Appert, C., Zhai, S. (2009). Using strokes as command shortcuts: cognitive benefits and toolkit support. 27th international conference on Human factors in computing systems (CHI '09), 2289–2298.

- Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeptura, F., LaViola Jr., J. J. (2010). Code bubbles: a working set-based interface for code understanding and maintenance. 28th international conference on Human factors in computing systems (CHI '10), 2503–2512.
- Dig, D., Johnson, R. (2005). The Role of Refactorings in API Evolution. 21st IEEE International Conference on Software Maintenance (ICSM '05), 389–398.
- Fowler, M. (1999). Refactoring: improving the design of existing code. Reading, MA: Addison-Wesley.
- Ge, X., DuBose, Q. L., Murphy-Hill, E. (2012). Reconciling manual and automatic refactoring. ICSE 2012, 211–221.
- Johnson, R. E. (2010). Software development is program transformation. FSE/SDP workshop on Future of software engineering research (FoSER '10), 177–180.
- Kanat-Alexander, M. (2012). Code Simplicity: The Science of Software Development. Sebastopol, CA: O'Reilly Media.
- Lee, Y. Y., Chen, N., Johnson, R. E. (2012). Drag-and-Drop Refactoring: Intuitive Program Transformation. Technical report. <http://hdl.handle.net/2142/30011>.
- Moser, R., Sillitti, A., Abrahamsson, P., Succi, G. (2006). Does refactoring improve reusability? Proceedings of the 9th international conference on Reuse of Off-the-Shelf Components (ICSR'06), 287–297.
- Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., Succi, G. (2008). A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. In: Balancing Agility and Formalism in Software Engineering. Lecture Notes In: Computer Science, 252–266. Berlin: Springer.
- Murphy-Hill, E., Black A. P. (2007). Why Don't People Use Refactoring Tools? In: 1st Workshop on Refactoring Tools, 60–61.
- Murphy-Hill, E.; Black A. P. (2007). High velocity refactorings in Eclipse. In: OOPSLA workshop on eclipse technology eXchange (eclipse '07), 1–5.
- Murphy-Hill, E.; Black, A. P. (2008). Breaking the barriers to successful refactoring: observations and tools for extract method. In: 30th international conference on Software engineering (ICSE '08), 421–430.
- Murphy-Hill, E.; Parnin, C.: Black, A. P. (2009). How we refactor, and how we know it. In: 31st International Conference on Software Engineering (ICSE '09), 287–297.
- Murphy-Hill, E. (2009). A Model of Refactoring Tool Use. In: 3rd Workshop on Refactoring Tools.
- Negara, S.; Chen, N.; Vakilian, M.; Dig, D.; Johnson, R. E. (2012). Using Continuous Change Analysis to Understand the Practice of Refactoring. Under Submission to OOPSLA 2012.
- Opydyke, W. F. (1992). Refactoring Object-Oriented Frameworks. Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign.
- Overbey, J. L.; Johnson, R. E. (2011). Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 303–312.
- Oviatt, S.; Coulston, R.; Lunsford, R. (2004). When do we interact multimodally? Cognitive load and multimodal communication patterns. In: 6th international conference on Multimodal interfaces (ICMI '04), 129–136.
- Parnin, C.; Görg, C.; Rugaber, S. (2010). CodePad: interactive spaces for maintaining concentration in programming environments. In: 5th international symposium on Software visualization (SOFTVIS '10), 15–24.

- Roberts, D.; Brant, J.; Johnson, R. (1997). A refactoring tool for Smalltalk. In: Theory and Practice of Object Systems 3, 253–263.
- Schäfer, M. (2012). Refactoring tools for dynamic languages. In: Fifth Workshop on Refactoring Tools (WRT '12), 59–62.
- Vakilian, M.; Chen, N.; Negara, S.; Rajkumar, B. A.; Moghaddam, R. Z.; Johnson, R. E. (2011). The Need for Richer Refactoring Usage Data. In: PLATEAU 2011.
- Vakilian, M.; Chen, N.; Negara, S.; Moghaddam, R. Z.; Johnson, R. E. (2012). Composite refactoring. Under submission to OOPSLA, 2012.
- Vakilian, M.; Chen, N.; Negara, S.; Rajkumar, B. A.; Bailey, B. P.; Johnson, R. E. (2012). Use, Disuse, and Misuse of Automated Refactorings. To Appear in ICSE 2012.
- Zazworka, N.; Shaw, M. A.; Shull, F.; Seaman, C. (2011). Investigating the impact of design debt on software quality. In: Proceedings of the 2nd Workshop on Managing Technical Debt (MTD'11), 17–23.



Felix Raab, M.Sc.
Lehrstuhl für Medieninformatik
Universität Regensburg
Universitätsstraße 31
93053 Regensburg
felix.raab@ur.de

Felix Raab ist wissenschaftlicher Mitarbeiter und Doktorand am Lehrstuhl für Medieninformatik der Universität Regensburg. In seiner Forschungsarbeit befasst er sich mit dem Schnittfeld von Human-Computer-Interaction und Software Engineering und der Frage, wie Softwareentwicklungswerkzeuge durch neue Interaktionsmethoden verbessert werden können.