Das Re-Engineering variantenreicher Systeme verstehen: Eine empirische Arbeit über Kosten, Wissen, Nachvollziehbarkeit und Methoden[1]

Jacob Krüger[2]

Abstract: Durch variierende Anforderungen existieren die meisten Softwaresysteme in verschiedenen Varianten. Entwickler beginnen üblicherweise durch Klonen und Anpassen diese wiederzuverwenden, bis Wartungsprobleme zum Re-Engineering einer Softwareplattform führen. Obwohl dies das verbreitetste Szenario ist, wurde es in der Forschung nur unzureichend untersucht. Im Rahmen der Dissertation wurden vier Kernfaktoren empirisch analysiert, was zu folgenden, stark zusammengefassten, Ergebnissen führt: Entwickler sollten versuchen die Wiederverwendung in Richtung einer Softwareplattform zu systematisieren. Wissen über Features und deren Sourcecode ist essentiell, weshalb Featurecode proaktiv dokumentiert werden sollte, da ansonsten Wissen aufwändig wiedergewonnen werden muss. Ein aktualisiertes Prozessmodell mit zugehörigen Richtlinien hilft Organisationen bei der Durchführung von Re-Engineering Projekten und zeigt neue Forschungsrichtungen auf. Die Ergebnisse stellen eine Synthese und Erweiterung des existierenden Wissensstandes zum (Re-)Engineering variantenreicher Systeme sowie weiterer grundsätzlicher Problemstellungen dar, durch die viele etablierte Annahmen mit verlässlichen und aktuellen Daten bestätigt aber auch einige widerlegt werden.

1 Einleitung

Moderne Softwaresysteme in allen Domänen existieren in verschiedenen Varianten um unterschiedliche Anforderungen von Kunden, bestimmter Hardware oder anderen Beschränkungen zu erfüllen. Solche variantenreichen Systeme sind in verschiedenen Ausprägungen sowohl in der Industrie als auch im Open-Source Bereich weit verbreitet, wie beispielsweise bei Betriebssystemen (z.B. Linux Kernel und Distributionen, Windows), eingebetteten Systemen (z.B. 3D Drucker, Roboter, Kameras, Steuerungseinheiten), Programmiersprachen (z.B. GCC, Python), Android Apps, Datenbanken (z.B. Oracle Berkeley DB) oder Autos (z.B. Volve, Opel, Rolls Royce) [Be20; Fo16; KB20b; Ku18; LSR07; SSW15; YGM06]. Da sich die entstehenden Varianten sehr ähneln, ist es eine wichtige strategische Entscheidung für jede Organisation, wie sie die Wiederverwendung bereits existierender Softwareartefakte (z.B. Code, Modelle, Anforderungen) organisiert.

[1] Englischer Titel der Dissertation [Kr21a]: „Understanding the Re-Engineering of Variant-Rich Systems: An Empirical Work on Economics, Knowledge, Traceability, and Practices“

[2] Eindhoven University of Technology, Department of Mathematics and Computer Science, Groene Loper 3, 5612 AE Eindhoven, The Netherlands, j.kruger@tue.nl

Wiederverwendung ist ein Kernprinzip des Software Engineering um Entwicklungs- und Wartungskosten zu reduzieren, eine neue Variante schneller zu liefern und die Softwarequalität zu erhöhen [KB20b; LSR07; St84]. Es existiert eine Vielzahl an Techniken zur Wiederverwendung von Softwareartefakten, die in zwei Strategien unterteilt werden können:

Klonen bezeichnet die Strategie eine existierende Variante (oder größere Teile von dieser) zu Kopieren und an geänderte Anforderungen anzupassen [SSW15]. Die Vorteile dieser Strategie liegen darin, dass sie ohne weitere Vorbereitung umsetzbar und dadurch ohne größere Investitionen nutzbar ist, sowie von vielen Versionskontrollsystemen unterstützt wird [KB20b; SSW15]. Aus diesem Grund beginnen viele Organisationen mit dieser Strategie, bis die Wartung der unabhängigen Klone zu aufwendig wird, beispielsweise weil Änderungen propagiert werden müssen oder die Entwickler den Überblick verlieren [Be20; Ku18; YGM06].

Softwareplattformen hingegen erfordern, dass die Organisation ein systematisch wiederverwendbares Portfolio an Softwareartefakten entwickelt, das es erlaubt Varianten zu konfigurieren und automatisch zu generieren. Diese Strategie baut typischerweise auf Produktlinienkonzepten auf (z.B. Variabilitätsmechanismen, Feature-Modelle) und organisiert existierende Artefakte entlang (konfigurierbarer) Funktionalitäten der Varianten—welche als Features bezeichnet werden [LSR07]. Im Gegensatz zum Klonen erfordert eine Softwareplattform hohe Investitionen in technische (z.B. Plattformarchitektur) und nicht-technische (z.B. Prozesse) Aspekte [KB20b; Li21; LSR07]. Allerdings versprechen Softwareplattformen verglichen mit dem Klonen erhebliche Einsparungen bei der Entwicklung und Wartung von Varianten, erhöhte Softwarequalität und eine schnellere Auslieferung.

Da die meisten Organisationen mit Klonen beginnen, ist das Re-Engineering von geklonten Varianten in eine Softwareplattform eines der häufigsten praktischen Re-Engineering Szenarien mit hohen Risiken und Kosten [Be13; Be20; Kr16; Kr19a; Ku18; YGM06]. Beispielsweise berichten Fogdal et al. [Fo16], dass Danfoss für das Re-Engineering von geklonten Varianten in eine Softwareplattform 10 Monate geplant hatte (inklusive Sourcecode, Tools, Feature-Modell und weiteren Artefakten). Stattdessen benötigte das Unternehmen alleine 36 Monate um 80 % des Sourcecodes zu überführen.

Im Rahmen der Dissertation [Kr21a] wurden vier Faktoren des Re-Engineerings variantenreicher Systeme erforscht und als Forschungsziele (FZ) definiert, die für das Verständnis und damit die Planbarkeit solcher Projekte essentiell sind. Aktuelle Literaturanalysen, Umfragen unter Experten und Erfahrungsberichte zeigen, dass diese Faktoren unzureichend verstanden sind und keine systematisch erhobenen Daten existieren. Dementsprechend basieren viele Entscheidungen bezüglich des (Re-)Engineerings variantenreicher Systeme lediglich auf einzelnen Erfahrungen, Spekulationen und Bauchgefühlen, trotz Jahrzehnten an Forschung. Abb. 1 zeigt einen groben Überblick über die folgenden vier Faktoren, ihre Beziehungen zueinander, was im Rahmen der Dissertation analysiert wurde sowie

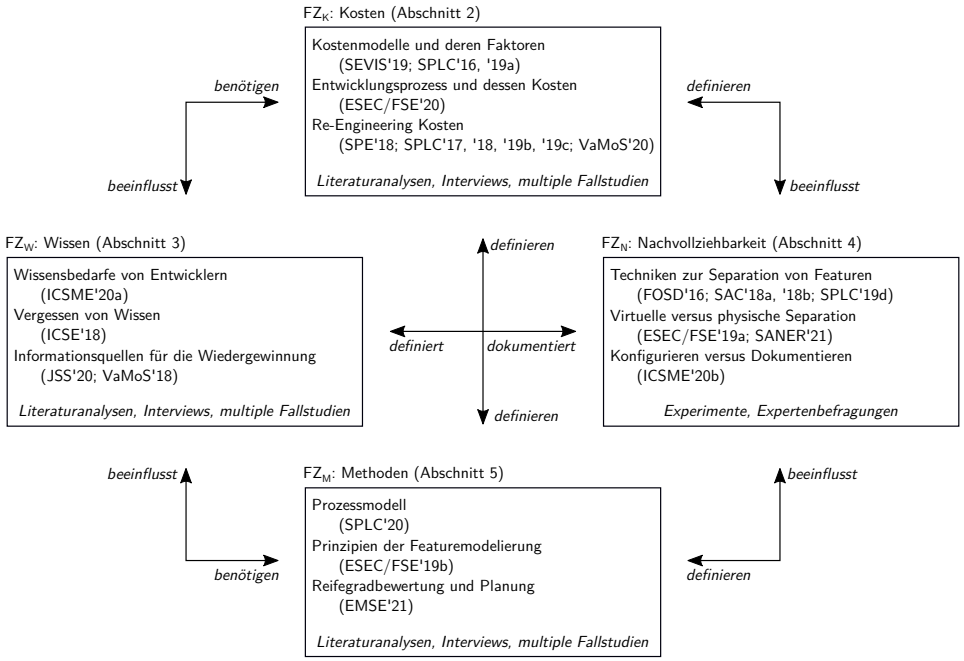

Abb. 1: Grobüberblick über die Forschungsziele, ihre Beziehungen (Pfeile) und zugehörigen Beiträge mit den wichtigsten Publikationen (in Klammern) sowie Forschungsmethoden (kursiv).

die wichtigsten Publikationen (in Klammern) und Forschungsmethoden (kursiv)—eine detailliertere Beschreibung folgt in den zugehörigen Abschnitten:

FZ_K – Kosten (Abschnitt 2): Die Kosten des (Re-)Engineerings variantenreicher Systeme zu verstehen hilft, strategische Entscheidungen bezüglich der zu nutzenden Strategie zu treffen, das Projektmanagement zu unterstützen (FZ_W) und besonders anspruchsvolle Aktivitäten zu identifizieren sowie zu unterstützen (FZ_N, FZ_M).

FZ_W – Wissen (Abschnitt 3): Für das erfolgreiche (Re-)Engineering variantenreicher Systeme benötigen die involvierten Entwickler Wissen, was den größten Kostenfaktor darstellt (FZ_K). Zu verstehen, welches Wissen dabei (nicht mehr) verfügbar ist und von wo es zurückgewonnen werden kann hilft, Entwickler beim Verstehen eines Systems zu unterstützen (FZ_N) und in Entwicklungsprozesse einzubinden (FZ_M).

FZ_N – Nachvollziehbarkeit (Abschnitt 4): Das teuerste (FZ_K) Wissen (FZ_W) im Bezug auf das (Re-)Engineering variantenreicher Systeme sind die genauen Stellen im Sourcecode die ein bestimmtes Feature implementieren („Feature Locations“). Eine geeignete Technik (FZ_M) für die expliziete Nachvollziehbarkeit von Featuren proaktiv einzuführen ist essentiell um später Probleme und Kosten zu vermeiden.

FZ_M – Methoden (Abschnitt 5): Existierende Richtlinien für die Entwicklung variantenreicher Systeme sind seit Jahrzehnten nicht systematisch aktualisiert worden. Ein Prozessmodell und Analysemethoden die moderne Software Engineering Techniken (FZ_N) berücksichtigen helfen bei der Planung und beim Monitoring von (Re-)Engineering Projekten (FZ_K) sowie der Organisation von Wissen (FZ_W).

Um systematisch Daten bezüglich dieser Faktoren zu erheben, wurde auf Methoden zur Kreation und Verifikation von Wissen im Rahmen des Evidence-Based Software Engineerings aufgebaut [SSS08], beispielsweise Literaturanalysen, Experimente und multiple Fallstudien. Durch das bessere Verständnis und die verlässlicheren Daten, die im Rahmen der Dissertation bezüglich der vier Faktoren entstanden sind, können Organisationen verlässlicher Entscheidungen treffen und Projekte managen, sowie Forscher praxisrelevante Ansätze entwickeln und einige etablierte Missverständnisse auflösen. Dies ist durch Kollaborationen (z.B. mit Axis AB, pure-systems GmbH) während der Forschungsarbeiten sichergestellt und durch spätere Diskussionen der Ergebnisse (z.B. mit Danfoss SE, Robert Bosch GmbH) bestätigt worden. Zudem gehen einige der Einsichten, beispielsweise zu Wissen, Nachvollziehbarkeit und dem verbindenden Thema Programmverständnis, über den Bereich des (Re-)Engineerings variantenreicher Systeme hinaus und sind für das Software Engineering im Allgemeinen von großer Bedeutung.

2 FZ_K: Kosten variantenreicher Systeme

Die Kosten und Nutzen beider Wiederverwendungsstrategien gegeneinander abzuwägen ist essentiell damit eine Organisation eine verlässliche Entscheidung treffen und Projekte planen kann. Zudem hängt von dieser Entscheidung ab, was für Wissen (z.B. einzelne Variante versus gemeinsame Softwareplattform), Nachvollziehbarkeit (z.B. Kunden zu geklonten Varianten oder zu Features der Softwareplattform) und Methoden (z.B. kontinuierliche Integration bei einer Softwareplattform) benötigt werden. Existierende Kostenmodelle und Analysemethoden basieren oft auf einzelnen Erfahrungen anstatt von systematisch erhobenen Daten, weshalb die meisten Entscheidungen auf Bauchgefühlen beruhen [KB20a; KB20b; Kr16; Li21]. Verschiedene Literaturanalysen [As17; LC13] und Expertenbefragungen [Be20; GMA12; Ra19] verdeutlichen, dass es sich dabei um ein langanhaltendes und fundamentales Problem handelt—was insbesondere in der Komplexität von Kostenanalysen im Software Engineering begründet liegt [Bo84; JB09].

Im Rahmen der Dissertation wurden existierende Kostenmodelle für Softwareplattformen und deren Faktoren analysiert [KBL19; Kr16]. Basierend auf dieser Analyse wurden Forschungsarbeiten gesammelt und Interviews mit Entwicklern bei Axis AB durchgeführt um quantitative als auch qualitative Daten zur Wiederverwendung in über 100 Organisationen zu sammeln [KB20b]. Der Vergleich zwischen beiden Wiederverwendungsstrategien hat viele etablierte Annahmen zu den Kosten bestätigt (z.B. Features einer Softwareplattform sind teurer als die von Klonen), aber auch einige widerlegt (z.B. Änderungen propagieren ist

teurer in Klonen). Zuletzt wurden multiple Fallstudien durchgeführt um die Probleme und Kosten des Re-Engineerings von Klonen zu einer Softwareplattform zu verstehen [De19; KB20a; Kr17; Kr18c; Ku18]. Aus allen drei Teilbereichen ging hervor, dass eine Plattform deutliche Kostenersparnisse gegenüber dem Klonen bietet, aber auch dass dies primär vom Wissen der Entwickler und der Softwarequalität abhängt. Auf einer sehr hohen Abstraktionsebene ist die Kerneinsicht zu diesem Faktor:

FZ_K: Kosten

Die Wiederverwendung von Softwareartefakten in Richtung einer Softwareplattform zu systematisieren ist die ökonomisch sinnvollste Strategie.

3 FZ_W: Das Wissen der Entwickler

Obwohl eine Vielzahl von Faktoren (z.B. Systemgröße, Methoden) die Kosten des (Re-) Engineerings variantenreicher Systeme beeinflussen, hat sich in den Analysen das (verbliebene) Wissen der Entwickler als besonders wichtig herausgestellt. Vor allem zu identifizieren, welcher Sourcecode welches Feature implementiert und das zugehörige Programmverständnis wiederzuerlangen sind Herausforderungen in diesem Bereich. Beides benötigt hohen manuellen Aufwand, da eine Automatisierung bestenfalls unzuverlässig möglich ist und meist keine Methoden zur Nachvollziehbarkeit verwendet oder aktualisiert wurden [BMW93; KBL19; Ra18b; Wa13]. Andere Studien mit Entwicklern zeigen, dass diese weiteres Wissen bezüglich der Features benötigen und dies wiederum nicht umfangreich genug untersucht worden ist [Be20; GMA12; Ra18a; Ta10].

Im Rahmen der Dissertation wurde eine Literaturanalyse durchgeführt um darauf aufbauend Entwickler zu interviewen um zu verstehen, welches Wissen ihnen wichtig ist und welches sie sich merken [KH20]. Hieraus ging hervor, dass Entwickler sich vorrangig abstraktes Wissen (z.B. welche Features existieren) merken, aber nicht Details (z.B. die zugehörigen Stellen im Sourcecode). Mit einer weiteren Befragung von Entwicklern wurde untersucht, wie sich deren verbliebenes Programmverständnis zu einen Stück Sourcecode über die Zeit entwickelt [Kr18d]. Da Entwickler ihr Wissen in verschiedenen Bereichen aber insbesondere bezüglich ihres Programmverständnisses wiedererlangen müssen, wurde in einer multiplen Fallstudie untersucht, welche Informationsquellen sie dafür nutzen können [Kr18a; Kr19c]. Aus diesen drei Teilbereichen lassen sich verschiedene Empfehlungen für Organisationen ableiten, beispielsweise, dass mit dem System erfahrene Entwickler neuen Entwicklern die abstrakte Struktur vermitteln können. Auf einer sehr hohen Abstraktionsebene ist die Kerneinsicht zu diesem Faktor:

FZ_W: Wissen

Wissen bezüglich der Features eines variantenreichen Systems und deren zugehöriger Sourcecode sollten dokumentiert werden um teure Wiedergewinnung zu vermeiden.

4 FZ_N: Nachvollziehbarkeit von Wissen

Nachvollziehbarkeit zielt darauf ab, Wissen explizit zu dokumentieren und über verschiedene Artefakte verfolgen zu können. Da die vorigen Ergebnisse gezeigt haben, dass insbesondere das Wissen welcher Sourcecode ein bestimmtes Feature implementiert wichtig ist und die Wiedergewinnung Kosten verursacht, lag der weitere Fokus der Dissertation auf der Nachvollziehbarkeit von Featuren. In variantenreichen Systemen werden meist nur optionale Features zu einem gewissen Grad durch den Variabilitätsmechanismus im Sourcecode dokumentiert, der aber zusätzliche Komplexität hinzufügt und oftmals nicht eindeutig ist (z.B. können Präprozessor Direktiven beliebig viele Features komplex miteinander verbinden). Methoden zur Nachvollziehbarkeit von Featuren einzuführen ist herausfordernd und es fehlt an empirischen Daten, die die Wirksamkeit verschiedener Methoden untermauern sowie von Variabilitätsmechanismen abgrenzen [As17; Be20; LC13; Va17].

Im Rahmen der Dissertation wurden die Dimensionen der Nachvollziehbarkeit von Featuren basierend auf Codeanalysen und Expertenbefragungen erforscht [Kr18b; LKL19]. Darauf aufbauend entstanden zwei Experimente mit verschiedenen Fokussen: Erstens, ob Features durch Annotationen oder durch die physische Trennung (z.B. als Module) das Programmverständnis besser unterstützen [Kr19b; Kr21b]. Die Ergebnisse deuten auf Annotationen im Sourcecode als geeignetere Methode hin, da sie nicht zu Indirektionen zwischen interagierenden Teilen des Systems führen. Zweitens, welchen Einfluss die Konfigurierbarkeit von Featureannotationen auf das Programmverständnis hat [Fe20]. Neben einer überraschenden Diskrepanz zwischen der Wahrnehmung und der objektiven Leistung der Teilnehmer des Experiments, deuten die Ergebnisse darauf hin, dass die Nachvollziehbarkeit von Featuren besser über Annotationen sichergestellt wird die nicht konfigurierbar sind. Dementsprechend lässt sich aus diesen drei Teilen eine klare Empfehlung zur Implementierung der Nachvollziehbarkeit von Featuren ableiten. Auf einer sehr hohen Abstraktionsebene ist die Kerneinsicht zu diesem Faktor:

FZ_N: Nachvollziehbarkeit

Features sollten im Sourcecode mit nicht konfigurierbaren Annotationen dokumentiert werden um durch die Nachvollziehbarkeit das Programmverständnis zu vereinfachen.

5 FZ_M: Methoden des (Re-)Engineerings

Die Bearbeitung der vorigen Forschungsziele war eng gekoppelt an bestimmte Prozesse, Aktivitäten und Methoden für das (Re-)Engineering variantenreicher Systeme (z.B. entstehen Kosten durch Aktivitäten). Dabei wurde klar, dass die existierenden Prozessmodelle und viele Methoden für solche Systeme seit Jahrzehnten nicht aktualisiert worden sind—was durch weitere Literaturanalysen [Ra18a] und industrielle Studien [Be20] ebenfalls untermauert wird. Ein aktualisiertes Prozessmodell und neue Methoden die die vorigen Ergebnisse berücksichtigen bieten Organisationen konkrete Unterstützung bei ihren Projekten und vermitteln Forschern den aktuellen Stand der Praxis.

Im Rahmen der Dissertation wurden die vorigen Ergebnisse synthetisiert und kontextualisiert indem sie, erweitert um eine zusätzliche Literaturanalyse, für die Entwicklung des neuen Prozessmodells promote-pl genutzt wurden [KMB20]. Dies inkorporiert die Erfahrungen mit Mischformen der variantenreichen Systeme die in anderen Studien auftraten und beinhaltet besonders Wissens- und Nachvollziehbarkeitsaspekte. Für die initiale Phase eines (Re-)Engineeringprozesses in dem der Umfang und die Kosten des Systems definiert werden, entstanden zudem Prinzipien für die Feature Modellierung aus einer Literaturanalyse und Experteninterviews um damit verbundene Fallstricke zu vermeiden [Ne19]. Zudem ist es wichtig, dass ein variantenreiches System vor und während seiner Lebenszeit nach verschiedenen Dimensionen (z.B. Kosten, Methoden) analysiert wird. Für diesen Zweck existieren Reifegradmodelle, von denen eines im Rahmen einer multiplen Fallstudie in einem Unternehmen angewandt wurde [Li21]. Daraus entstanden Empfehlungen zur Anwendung, den Vorteilen und den Problemen dieses Modells. Diese drei Teile bilden ein Rahmenwerk für Organisationen, ihre (Re-)Engineeringprojekte unter Berücksichtigung aktueller Standards effizient zu planen und durchzuführen. Auf einer sehr hohen Abstraktionsebene ist die Kerneinsicht zu diesem Faktor:

FZ_M: Methoden

Das Prozessmodell und die Methoden bilden ein Rahmenwerk um das (Re-)Engineering eines variantenreichen Systems durchzuführen und den Nutzen zu maximieren.

6 Zusammenfassung

Die Ergebnisse der Dissertation tragen dazu bei, das Re-Engineering variantenreicher Systeme besser zu verstehen. Dabei sind unter Anderem verlässliche ökonomische Daten (FZ_K), ein besseres Verständnis für Wissensbedarfe (FZ_W), Empfehlungen für die Implementierung von Nachvollziehbarkeit (FZ_N) und Methoden für das Projektmanagement (FZ_M) entstanden. Diese Beiträge erweitern den wissenschaftlichen Wissensstand beträchtlich durch belastbare empirische Einsichten und werden bereits in der Praxis genutzt. Allerdings kann diese Zusammenfassung lediglich einen sehr abstrakten Überblick über die einzelnen Forschungsziele der Dissertation geben, die bedeutend tiefer gehen.

Literatur

[As17] Assunção, W. K. G.; Lopez-Herrejon, R. E.; Linsbauer, L.; Vergilio, S. R.; Egyed, A.: Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. Empirical Software Engineering 22/6, 2017.

[Be13] Berger, T.; She, S.; Lotufo, R.; Wąsowski, A.; Czarnecki, K.: A Study of Variability Models and Languages in the Systems Software Domain. IEEE Transactions on Software Engineering 39/12, 2013.

[Be20] Berger, T.; Steghöfer, J.-P.; Ziadi, T.; Robin, J.; Martinez, J.: The State of Adoption and the Challenges of Systematic Variability Management in Industry. Empirical Software Engineering 25/, 2020.

[BMW93] Biggerstaff, T. J.; Mitbander, B. G.; Webster, D. E.: The Concept Assignment Problem in Program Understanding. In: WCRE. IEEE, 1993.

[Bo84] Boehm, B. W.: Software Engineering Economics. IEEE Transactions on Software Engineering SE-10/1, 1984.

[De19] Debbiche, J.; Lignell, O.; Krüger, J.; Berger, T.: Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In: SPLC. ACM, 2019.

[Fe20] Fenske, W.; Krüger, J.; Kanyshkova, M.; Schulze, S.: #ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference. In: ICSME. IEEE, 2020.

[Fo16] Fogdal, T. S.; Scherrebeck, H.; Kuusela, J.; Becker, M.; Zhang, B.: Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In: SPLC. ACM, 2016.

[GMA12] Ghanam, Y.; Maurer, F.; Abrahamsson, P.: Making the Leap to a Software Platform Strategy: Issues and Challenges. Information and Software Technology 54/9, 2012.

[JB09] Jørgensen, M.; Boehm, B. W.: Software Development Effort Estimation: Formal Models or Expert Judgment? IEEE Software 26/2, 2009.

[KB20a] Krüger, J.; Berger, T.: Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In: VaMoS. ACM, 2020.

[KB20b] Krüger, J.; Berger, T.: An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In: ESEC/FSE. ACM, 2020.

[KBL19] Krüger, J.; Berger, T.; Leich, T.: Features and How to Find Them - A Survey of Manual Feature Location. In: Software Engineering for Variability Intensive Systems. CRC Press, 2019.

[KH20] Krüger, J.; Hebig, R.: What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In: ICSME. IEEE, 2020.

[KMB20] Krüger, J.; Mahmood, W.; Berger, T.: Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines. In: SPLC. ACM, 2020.

[Kr16] Krüger, J.; Fenske, W.; Meinicke, J.; Leich, T.; Saake, G.: Extracting Software Product Lines: A Cost Estimation Perspective. In: SPLC. ACM, 2016.

[Kr17] Krüger, J.; Nell, L.; Fenske, W.; Saake, G.; Leich, T.: Finding Lost Features in Cloned Systems. In: SPLC. ACM, 2017.

[Kr18a] Krüger, J.; Gu, W.; Shen, H.; Mukelabai, M.; Hebig, R.; Berger, T.: Towards a Better Understanding of Software Features and Their Characteristics. In: VaMoS. ACM, 2018.

[Kr18b] Krüger, J.; Ludwig, K.; Zimmermann, B.; Leich, T.: Physical Separation of Features: A Survey with CPP Developers. In: SAC. ACM, 2018.

[Kr18c] Krüger, J.; Pinnecke, M.; Kenner, A.; Kruczek, C.; Benduhn, F.; Leich, T.; Saake, G.: Composing Annotations Without Regret? Practical Experiences Using FeatureC. Software: Practice and Experience 48/3, 2018.

[Kr18d] Krüger, J.; Wiemann, J.; Fenske, W.; Saake, G.; Leich, T.: Do You Remember This Source Code? In: ICSE. ACM, 2018.

[Kr19a] Krüger, J.: Are You Talking about Software Product Lines? An Analysis of Developer Communities. In: VaMoS. ACM, 2019.

[Kr19b] Krüger, J.; Çalıklı, G.; Berger, T.; Leich, T.; Saake, G.: Effects of Explicit Feature Traceability on Program Comprehension. In: ESEC/FSE. ACM, 2019.

[Kr19c] Krüger, J.; Mukelabai, M.; Gu, W.; Shen, H.; Hebig, R.; Berger, T.: Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. Journal of Systems and Software 152/, 2019.

[Kr21a] Krüger, J.: Understanding the Re-Engineering of Variant-Rich Systems: An Empirical Work on Economics, Knowledge, Traceability, and Practices, Diss., Otto-von-Guericke University Magdeburg, 2021.

[Kr21b] Krüger, J.; Çalıklı Gül, G.; Berger, T.; Leich, T.: How Explicit Feature Traces Did Not Impact Developers' Memory. In: SANER. IEEE, 2021.

[Ku18] Kuiter, E.; Krüger, J.; Krieter, S.; Leich, T.; Saake, G.: Getting Rid of Clone-And-Own: Moving to a Software Product Line for Temperature Monitoring. In: SPLC. ACM, 2018.

[LC13] Laguna, M. A.; Crespo, Y.: A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. Science of Computer Programming 78/8, 2013.

[Li21] Lindohf, R.; Krüger, J.; Herzog, E.; Berger, T.: Software Product-Line Evaluation in the Large. Empirical Software Engineering 26/30, 2021.

[LKL19] Ludwig, K.; Krüger, J.; Leich, T.: Covert and Phantom Features in Annotations: Do They Impact Variability Analysis? In: SPLC. ACM, 2019.

[LSR07] van der Linden, F. J.; Schmid, K.; Rommes, E.: Software Product Lines in Action. Springer, 2007.

[Ne19] Nešić, D.; Krüger, J.; Stănciulescu, S.; Berger, T.: Principles of Feature Modeling. In: ESEC/FSE. ACM, 2019.

[Ra18a] Rabiser, R.; Schmid, K.; Becker, M.; Botterweck, G.; Galster, M.; Groher, I.; Weyns, D.: A Study and Comparison of Industrial vs. Academic Software Product Line Research Published at SPLC. In: SPLC. ACM, 2018.

[Ra18b] Razzaq, A.; Wasala, A.; Exton, C.; Buckley, J.: The State of Empirical Evaluation in Static Feature Location. ACM Transactions on Software Engineering and Methodology 28/1, 2018.

[Ra19] Rabiser, R.; Schmid, K.; Becker, M.; Botterweck, G.; Galster, M.; Groher, I.; Weyns, D.: Industrial and Academic Software Product Line Research at SPLC: Perceptions of the Community. In: SPLC. ACM, 2019.

[SSS08] Shull, F.; Singer, J.; Sjøberg, D. I. K., Hrsg.: Guide to Advanced Empirical Software Engineering. Springer, 2008.

[SSW15] Stănciulescu, S.; Schulze, S.; Wąsowski, A.: Forked and Integrated Variants in an Open-Source Firmware Project. In: ICSME. IEEE, 2015.

[St84] Standish, T. A.: An Essay on Software Reuse. IEEE Transactions on Software Engineering SE-10/5, 1984.

[Ta10] Tang, A.; Couwenberg, W.; Scheppink, E.; de Burgh, N. A.; Deelstra, S.; van Vliet, H.: SPL Migration Tensions: An Industry Experience. In: KOPLE. ACM, 2010.

[Va17] Vale, T.; de Almeida, E. S.; Alves, V. R.; Kulesza, U.; Niu, N.; de Lima, R.: Software Product Lines Traceability: A Systematic Mapping Study. Information and Software Technology 84/, 2017.

[Wa13] Wang, J.; Peng, X.; Xing, Z.; Zhao, W.: How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. Journal of Software: Evolution and Process 25/11, 2013.

[YGM06] Yoshimura, K.; Ganesan, D.; Muthig, D.: Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In: SEAS. ACM, 2006.

Jacob Krüger ist Assistant Professor für Software Engineering an der Technischen Universität Eindhoven in den Niederlanden. Er hat zuvor an der Ruhr-Universität Bochum als Akademischer Rat gearbeitet nachdem er an der Otto-von-Guericke Universität Magdeburg seinen Masterabschluss in Wirtschaftsinformatik (2016) sowie seinen Doktor in der Arbeitsgruppe Datenbanken und Software Engineering (2021) erlangte. Dabei arbeitete er als wissenschaftlicher Mitarbeiter an dem DFG-Projekt EXPLANT, zuerst an der Fachhochschule Harz und danach an der Otto-von-Guericke Universität. Während seiner Promotion hat er einen einjährigen Forschungsaufenthalt an der Technischen Hochschule Chalmers | Universität von Göteborg in Schweden sowie einen dreimonatigen Forschungsaufenthalt an der Universität von Toronto in Kanada wahrgenommen. Seine Forschung wurde durch Preise, Stipendien und eigene Projekte ausgezeichnet beziehungsweise unterstützt und in führenden Konferenzen (z.B. ICSE, ESEC/FSE, ICSME) sowie Journalen (z.B. EMSE, JSS) publiziert. Er fokussiert sich auf menschliche Faktoren während der Softwareevolution, insbesondere auf Kosten, Programmverständnis und variantenreiche Systeme.