



24. Workshop Software-Reengineering und -Evolution (WSRE)

der GI-Fachgruppe Software-Reengineering (SRE)

13. Workshop Design for Future (DFF)

des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2)

WSRE 2022

02.-04. Mai 2022, Physikzentrum Bad Honnef



24. Workshop Software-Reengineering und -Evolution (WSRE) der GI-Fachgruppe Software-Reengineering (SRE)

13. Workshop „Design for Future“ des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2)

02.-04. Mai 2022

Besondere Zeiten, besondere Bestimmungen, besonderer WSRE. Seit 1999 fanden die Workshops „Software-Reengineering & -Evolution“ (WSRE) immer Anfang Mai und immer im Physikzentrum in Bad Honnef statt. Nicht so 2020 und 2021. Dieses Mal empfinden wir es deshalb als besonders, wieder an den üblichen Ort zur üblichen Zeit zurückkehren zu dürfen.

Die Workshopreihe WSRE (am Anfang noch WSR) wurde 1999 von Jürgen Ebert und Franz Lehner ins Leben gerufen, um neben den internationalen erfolgreichen Tagungen auch ein deutschsprachiges Diskussionsforum zum Thema Reengineering zu schaffen.

Ziel der Treffen ist es, einander kennen zu lernen und auf diesem Wege eine direkte Basis für Wissensaustausch und Kooperationen zu schaffen, so dass das Themengebiet eine Stärkung, Konsolidierung und Weiterentwicklung erfährt. Durch aktive und gewachsene Beteiligung vieler Personen aus Forschung und Praxis hat sich der WSRE als zentrale Reengineering-Konferenz im deutschsprachigen Raum etabliert. Dabei wird er weiterhin als Low-Cost-Workshop ohne eigenes Budget durchgeführt. Bitte tragen auch Sie dazu bei, den WSRE weiterhin erfolgreich zu machen, indem Sie interessierte Kolleginnen und Kollegen sowie Bekannte darauf hinweisen. Auf Basis der erfolgreichen WSR-Treffen der ersten Jahre wurde 2004 die GI-Fachgruppe Software-Reengineering (SRE) gegründet, siehe <https://fg-sre.gi.de/>), welche zusätzlich zum WSRE auch bei anderen Aktivitäten rund um das Thema Reengineering mitwirkt.

Seit 2010 ist der Arbeitskreis Langlebige Softwaresysteme (L2S2) mit seinen „Design for Future“-Workshops (DFF) aufgrund der inhaltlichen Nähe ebenfalls bei der Fachgruppe Software-Reengineering aufgehängt. Zunächst in der Regel alle zwei Jahre, inzwischen jährlich findet seitdem ein gemeinsamer Workshop von WSRE und DFF statt. Diese Kombination soll den Austausch zwischen den beiden Gruppen fördern. Der inhaltliche Schwerpunkt beim DFF liegt auf Ansätzen, um langlebige Systeme wartbar, evolvierbar und modernisierbar zu entwickeln.

Die WSRE-Workshops sind die zentrale Tagungsreihe der Fachgruppe Software-Reengineering (SRE). Sie bieten eine Vielzahl aktueller Themen aus den Bereichen Software-Reengineering, Software-Wartung und Software-Evolution, die gleichermaßen aus praktischer und wissenschaftlicher Perspektive betrachtet werden.

In diesem Jahr gab es wieder Beiträge zu einem breiten Spektrum von Software-Reengineering- und Software-Evolutionsthemen.

Das diesjährige Workshop-Programm enthielt insbesondere die folgenden Programmpunkte:

- **Vorträge:** Einblick in sowie Rückblick und Ausblick auf interessante Arbeiten und Ergebnisse rund ums Software-Reengineering.
- **13. Workshop "Design For Future"** des Arbeitskreises Langlebige Softwaresysteme (L2S2) als besonderer Track des WSRE.
- **Best Student Paper Award:** Ein Novum: Wir prämierten den besten studentischen Beitrag (Papier und Vortrag), bewertet durch eine Jury aus Wissenschaft und Praxis. Danke an Teilnehmer, Juroren und Sponsoren (Caruso GmbH, Ismaning; itemis AG, Lünen; S&N Invent GmbH, Paderborn)!
- **Fachgruppensitzung** der GI-Fachgruppe mit Wahl des Leitungsgremiums: In 2022 fanden turnusgemäß die Neuwahlen der Leitung der Fachgruppe Software-Reengineering (SRE) statt.
- **SRE Body of Knowledge (SREBOK):** Diskussion der Erstellung einer "Wissensbasis" zum Software-Reengineering unter Einbezug aller Teilnehmerinnen und Teilnehmer.
- **Networking:** Vernetzung zwischen den Teilnehmenden.
- **Social Event:** Besuch der Ausstellung „Das Gehirn. In Kunst & Wissenschaft“ in der Bundeskunsthalle, Bonn.

Im Rahmen der Fachgruppensitzung am 2. Mai haben die Mitglieder der Fachgruppe eine neue Fachgruppenleitung gewählt. **Jens Knodel**, Caruso GmbH (bisheriger Sprecher), **Torsten Görg**, itemis AG und **Matthias Gutheil**, itemis AG, sind satzungsgemäß nach der zweiten Amtszeit ausgeschieden. Ihnen gebührt der besondere Dank der Fachgruppe für Ihre engagierte Arbeit! **Marco Konersmann**, Universität Koblenz-Landau (bisher stellvertretender Sprecher) und **Stefan Sauer** (Universität Paderborn) kandidierten erneut. Sie wurden in ihren Ämtern bestätigt. Das neue Leitungsgremium begrüßt darüber hinaus die neuen Mitglieder **Jochen Quante**, Robert Bosch GmbH, **Daniela Schilling**, Delta Software Technology GmbH und **Sandro Schulze**, TU Braunschweig und dankt den ausgeschiedenen Mitgliedern vielmals für ihren Einsatz. Als neuer Sprecher der Fachgruppe wurde Jochen Quante durch die Leitung gewählt.



Die bisherige Leitung der Fachgruppe SRE (v.l.): Stefan Sauer, Jens Knodel (Sprecher), Marco Konersmann (stellv. Sprecher), Torsten Görg; Matthias Gutheil fehlt auf dem Bild.



Das neu gewählte Leitungsgremium der Fachgruppe SRE (v.l.): Daniela Schilling, Stefan Sauer, Sandro Schulze, Marco Konersmann, Jochen Quante (Sprecher).

Die Organisatoren danken allen Beitragenden für ihr Engagement – insbesondere den Autorinnen und Autoren, den Vortragenden, und allen Teilnehmerinnen und Teilnehmern für die lebhaften, kontroversen und interessanten Diskussionen. Unsere Grüße gelten in diesem Jahr insbesondere auch den Mitarbeiterinnen und Mitarbeitern des Physikzentrums Bad Honnef, verbunden mit der Hoffnung, den WSRE nun wieder alljährlich wieder in üblicher Form stattfinden lassen zu können.

Für die FG SRE:

Jochen Quante, Robert Bosch GmbH (Sprecher)
 Marco Konersmann, Universität Koblenz-Landau
 (stellv. Sprecher)
 Stefan Sauer, Universität Paderborn
 Daniela Schilling, Delta Software Technology
 Sandro Schulze, TU Braunschweig

Für den AK L2S2:

Robert Heinrich, KIT Karlsruhe (Sprecher)
 Marco Konersmann, Universität Koblenz-Landau
 Stefan Sauer, Universität Paderborn

Best Student Paper Award des 24. Workshop Software-Reengineering und -Evolution (WSRE) der GI-Fachgruppe Software-Reengineering (SRE)

03. Mai 2022

Zum 24. Workshop Software-Reengineering und -Evolution (WSRE) wurde durch die Fachgruppe Reengineering erstmals in der Geschichte der Workshopreihe der Preis für den besten studentischen Beitrag ausgelobt. Der **Best Student Paper Award** ist mit einem **Preisgeld in Höhe von 250 €** verbunden, das von Sponsoren bereitgestellt wird.

Teilnehmen konnten Studierende mit ihrem eigenen Projekt, z.B. ihrer **Master- oder Bachelorarbeit** oder einem andersartigen **studentischen Projekt**, das sie im Rahmen oder im Kontext ihres Studiums (bspw. auch innerhalb einer Werkstudententätigkeit) durchgeführt haben. Hierbei konnte es sich um Einzel- oder Gruppenarbeiten handeln. Beiträge mussten sich mit einem Thema aus dem Themenspektrum des WSRE beschäftigen, so wie es im allgemeinen Call for Papers des WSRE dargestellt war. Mögliche Themen waren u.a. Software-Analyse und -Transformation, Software-Qualität und Metriken, (intelligente) Wissensgewinnung aus Software-Repositorien, Software-Visualisierung, Analyse- und Reengineering-Werkzeuge, Continuous-Development-Ansätze, Wartung und Refactoring, Software-Migration und -Modernisierung, Modelle, Methoden, Prozesse für Reengineering und Software-Evolution, empirische Forschung und der Faktor Mensch im Reengineering sowie Wirtschaftlichkeit von Reengineering-Maßnahmen.

Bewerbungen/Einreichungen wurden in Form von Kurzbeschreibungen in deutscher oder englischer Sprache erwartet. Die Beiträge sollten das Thema, die Motivation und Zielsetzung, die Durchführung und die Ergebnisse umreißen. Die Beiträge sollten maximal zwei Seiten lang sein und im Format der Softwaretechnik-Trends eingereicht werden, denn die für die Finalrunde akzeptierten Beiträge sollten anschließend (und werden nun) auch im Workshopband (Proceedings) des WSRE in der Softwaretechnik-Trends veröffentlicht.

Eine Experten-Jury traf aus den Einreichungen eine Vorauswahl, die für den Best Student Paper Award nominiert und zur Präsentation beim Finale während des Workshops eingeladen wurden. Mit der Einladung zur Finalrunde war ein Reisekostenzuschuss in Höhe von 200 € pro Beitrag verbunden, der von den Sponsoren beigesteuert wurde. Die Präsentation erfolgte im Hauptprogramm des WSRE 2022. Auf Basis der Begutachtung der schriftlichen Beiträge und der Präsentation bestimmte dann die Jury den Beitrag,

der die Auszeichnung "Best Student Paper Award" erhielt. Alle Finalisten wurden mit ihren Beiträgen in den Workshopband aufgenommen.

Die nominierten Beiträge im Finale waren:

- **Falko Galperin.** Visualisierung von Code-Smells in Code-Cities.
Universität Bremen
- **Philipp Gnoyke.** On the Evolution of Architecture Smells and Technical Debt.
Otto-von-Guericke Universität Magdeburg
- **Felix Grabler.** Automatisierte Migration von Legacy-Dateien in relationale Datenbanken.
TU Chemnitz
- **Bjarne Sauer.** Analyse von Entwurfsentscheidungen in natürlichsprachlicher Softwarearchitekturdokumentation.
Karlsruher Institut für Technologie

Die Beiträge aller zum Finale Nominierten zeichneten sich nach Ansicht der Jury durch eine hohe Qualität aus und wurden mit einer Urkunde als Finalisten gewürdigt.

Wir gratulieren Philipp Gnoyke als Sieger des Best Student Paper Awards des WSRE 2022.

Die Fachgruppe dankt der Jury für ihr Engagement:

- Jens Borchers, Borchers Bfl, Hamburg
- Dr. Marco Konersmann, Universität Koblenz-Landau
- Prof. Dr. Rainer Koschke, Universität Bremen
- Dr. Jochen Quante, Robert Bosch GmbH, Stuttgart

Ebenso danken wir den ausgeschiedenen Mitgliedern der Fachgruppenleitung für ihre Unterstützung des Awards:

- Jens Knodel, Caruso GmbH
- Torsten Görg, itemis AG
- Matthias Gutheil, itemis AG

Für die FG SRE:

Jochen Quante, Robert Bosch GmbH (Sprecher)
Marco Konersmann, Universität Koblenz-Landau (stellv. Sprecher)
Stefan Sauer, Universität Paderborn
Daniela Schilling, Delta Software Technology
Sandro Schulze, TU Braunschweig

Preisverleihung



Foto (Stefan Sauer):
Marco Konersmann (Jury),
Philipp Gnoyke
(Preisträger)

Finalisten des Best Student Paper Awards



Foto (Stefan Sauer): die Finalisten (v.l.n.r.)
Falko Galperin, Felix Graßler,
Philipp Gnoyke, Bjarne Sauer

Finalisten, Jury & FG-Leitung



Foto (Stefan Sauer): die
Finalisten (vorn), Jens
Borchers (Jury), Stefan
Sauer (Leitung FG SRE),
Jochen Quante (Jury),
Torsten Görg (Leitung FG
SRE), Jens Knodel
(Sprecher FG SRE), Marco
Konersmann (Jury & stellv.
Sprecher FG SRE)

Wir danken den **Sponsoren des WSRE 2022 Best Student Paper Awards** für Ihre freundliche Unterstützung:



CARUSO GmbH, Ismaning
<https://www.caruso-dataplace.com/>



itemis AG, Lünen
<https://www.itemis.com/>



S&N Invent GmbH, Paderborn
<https://www.sn-invent.de/>

On the Evolution of Architecture Smells and Technical Debt

Philipp Gnoyke
Otto-von-Guericke University Magdeburg
Magdeburg, Germany
philipp.gnoyke@t-online.de

Abstract

In this paper, I summarize my master’s thesis about the evolution of software-architecture smells and (architectural) technical debt [2]. Parts of the thesis have been published at ICSME 2021 [3].

1 Background

If a software project shall be maintained and further developed for a long time with evolving requirements, negative symptoms in the code naturally arise and are accelerated by neglecting quality assurance. With the codebase becoming increasingly complex, the effort for changing or extending functionalities, reusing parts of the code, testing for detecting potential defects, and resolving existing defects tends to increase.

The metaphor of *technical debt* (TD) is used to describe the aforementioned patterns. TD is often taken on to enable faster software deployment. In return, interest rates occur, e.g., as reduced maintainability.

TD can usually be attributed to not complying with principles of designing software. This ranges from issues in single code lines up to module interaction. Such issues can be identified by searching for symptoms, which are called *smells*. This includes *code smells* (CS) on the micro level and *architecture smells* (AS) on the macro level. Knowledge about the location, number, severity, and evolution of smells enables developers to identify efficient spots for refactoring.

2 Motivation

In previous research, similar to CSs, a catalogue of different AS types and their properties, negative consequences from the presence of ASs, tools for detecting them, and methods for the automatic calculation of TD from ASs have been identified and devised.

However, the evolution of ASs has received minor research effort so far. A single study by Sas et al. investigated the evolutionary properties of single ASs, including their lifetime, evolution in severity, and circumstances of removal [4]. For researchers and practitioners, a relevant remaining question is whether the number and severity of ASs, as well as the manner of resolving existing ASs, influence, e.g., exacerbate the introduction and properties of new ASs. Moreover, it can be asked how these aspects of the evolution of

ASs relate to the evolution of *architectural TD* (ATD), i.e., TD on an architectural level. In figurative terms: Do ASs show compound interest? Respectively: Does the early resolving of ASs not only prevent “interest payments” in the form of degraded maintainability, but also prevent new ATD and ASs from being introduced? To answer these questions, suitable and comprehensible metrics for the evolution of ASs have to be defined and could serve as a basis for enhancing AS tracker tools. Such metrics and the gained knowledge about ASs could in return facilitate the continuous supervision and well-targeted removal of ASs and thus contribute to assuring the quality of software systems.

3 Research questions

To address the broad questions above, I answer the following research questions in my thesis:

RQ1: How can the long-term evolution of ASs and ATD be adequately quantified and represented?

“Long-term evolution” refers to the evolution over several versions and years. “Adequate” result quality implies a balance between accuracy, computation efficiency, and intuitive understandability. With the presence of suited evolutionary metrics, RQ2 and RQ3 can be addressed. They aim to answer whether compound interest applies to ASs and ATD.

RQ2: How do existing ASs and ATD influence the introduction of new ASs and ATD?

RQ3: How does the rate of reducing existing ASs and ATD influence the introduction of new ASs and ATD?

Lastly, it can be asked whether the severity and other properties of ASs influence how long they persist in a system and thus contribute to potential compound interest effects. Thus, RQ4 is:

RQ4: Which factors influence the length of the time span in which ASs persist in a system?

4 Methods

I first reviewed existing tools and algorithms for detecting ASs and calculating TD for usage in this study. To answer RQ1, I propose a refined approach for tracking ASs across versions depending on their type.

I focus on three types of ASs that have been commonly observed by previous studies: A *cyclic dependency* (CD) is a set of components (classes or packages) that depend on each other, so that their de-

pendency graph forms a cycle. A *hub-like dependency* (HD) represents a component with a high number of incoming and outgoing dependencies. An *unstable dependency* (UD) describes a stable component that depends on at least one less stable component, where stability refers to the amount of work to change a component. A high stability thus results in fewer changes.

For CDs, I introduce the distinction between *subcycles* and *supercycles*. In graph theory, a supercycle represents a *strongly connected component* in a version’s dependency graph, which contains one to n subcycles, i.e., *simple cycles*. As supercycles can merge or split between versions, I track their evolution in a novel way as *cyclic dependency evolution graphs*. Furthermore, I propose to distinguish *intra-* and *inter-version smells*, with the former being a smell instance in a particular version and the latter being a set of related intra-version smells over multiple versions.

As the research questions require data on how many ASs are introduced and removed per version and how long they persist, I derive definitions on lifetime properties of ASs. Lastly, I define properties that provide insights on the evolution of projects, ASs, and TD. For example, CD evolution graphs can be described by their *width*, i.e., number of concurrent intra-version smells, as well as their *order*, i.e., total number of intra-version smells.

To retrieve all defined properties and metrics according to the conceptual model for efficiently analyzing the evolution of software projects, I modified and built upon the existing open-source tool for smell detection *Arcan*, which resulted in the framework *ASTdEA*. In two analyzed systems, *Arcan* achieved a precision of 100%, as well as a recall of 60 and 66% [1]. I document the framework’s implementation and changes to *Arcan* in my thesis. Moreover, I explain the usage of *ASTdEA* and how the implementation was verified to ensure correct results.

I conducted an empirical study to answer RQ 2 to RQ 4. To this end, I selected a dataset from the *Qualitas Corpus* that comprises 14 software projects with different numbers of releases. On the results of the analysis, a series of queries was performed to gather information. This includes general queries about the evolution of the projects in the dataset and how values of all defined properties of ASs are distributed. Moreover, I performed a regression and correlation analysis to check whether the presence of many ASs and high TD levels amplify the growth of AS numbers and AS severity akin to compound interest. In a similar way, I checked whether a high degree of removing ASs induces the introduction of fewer new ASs according to RQ 3. Finally, RQ 4 was approached by analyzing whether the value of each defined AS property influences the length of AS lifespans.

5 Results

Among others, I obtained the following insights:

RQ 1: Tracking CDs as CD evolution graphs is relevant as nearly a quarter of class-level and nearly a third of package-level intra-version CDs occur in inter-version CDs that involve splitting or merging.

RQ 2: I found that TD and the number of ASs tend to increase along with the code size of systems. When looking at their relative levels, they remained mostly stable and decreased in some systems, independent of the magnitude. I could thus not confirm exponential growth patterns. However, I found many ASs that persisted for the entire observed period after their introduction. Secondly, I noted that some AS types like CDs had a greater impact on TD and system degradation, while not always corresponding to their share on AS numbers. Rather, aspects like the complexity seem to be more pivotal on their impact.

RQ 3: Contradicting the assumption that an active effort to remove ASs and TD would result in the introduction of fewer new issues, I observed that AS and TD introduction mostly correlated with the rate of removing them. This might be attributed to high development activity resulting in concurrent AS introductions and removals and requires further research.

RQ 4: Lastly, I found no universal patterns between the lifespan and certain properties of all AS types. However, complex CDs among classes, as well as UD that overlapped with other smells or had a large difference in stability, tended to persist longer.

The replication package¹ contains the thesis, the source code of *ASTdEA* and modified *Arcan*, the raw output dataset, the queries that were used to aggregate information, as well as additional diagrams.

References

- [1] Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, Damian Tamburri, Marco Zanoni, and Elisabetta Di Nitto. *Arcan: A tool for architectural smells detection*. In *Proceedings of the International Conference on Software Architecture Workshops (ICSAW)*, pages 282–285. IEEE, 2017.
- [2] Philipp Gnoyke. Does architectural technical debt exhibit compound interest? – On the evolution of architecture smells and technical debt. Master’s thesis, University Magdeburg, 2021.
- [3] Philipp Gnoyke, Sandro Schulze, and Jacob Krüger. An evolutionary analysis of software-architecture smells. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 413–424. IEEE, 2021.
- [4] Darius Sas, Paris Avgeriou, and Francesca Arcelli Fontana. Investigating instability architectural smells evolution: An exploratory case study. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 557–567. IEEE, 2019.

¹ <https://figshare.com/s/fa17e81cf4f27c84d059>

Ein kontrolliertes Experiment zur Visualisierung von Code-Smells in Code-Cities

Falko Galperin*
Universität Bremen

1 Einführung

Wir vergleichen in diesem Paper mit Hilfe eines kontrollierten Experiments einen Ansatz zur Visualisierung von Code-Smells in Code-Cities mit tabellarischen Darstellungen, wie sie bei kommerziellen Tools häufig zum Einsatz kommen, im Hinblick auf Genauigkeit, Effizienz und Usability.

Die Code-Cities werden mit SEE (Software Engineering Experience) visualisiert. SEE ist eine an der Universität Bremen entwickelte, in Unity implementierte interaktive Visualisierung von Software, welche die Code-City-Metapher am Datenmodell eines Abhängigkeitsgraphen umsetzt. Code-Komponenten und -Module werden dabei im Graphen als *Knoten* (in der Metapher „Gebäude“), Abhängigkeiten zwischen ihnen als *Kanten* (in der Metapher „Straßen“) repräsentiert. Metriken können als visuelle Attribute dieser Knoten und Kanten repräsentiert werden – die Höhe eines Gebäudes kann z. B. der Anzahl an Codezeilen entsprechen. Ein Vorteil dieser Darstellungsart im Vergleich zu „traditionellen“ IDEs liegt z. B. darin, dass Eigenschaften des Quellcodes oft schneller erkannt werden können.

2 Kontrolliertes Experiment

Wir vergleichen die Visualisierung von Code-Smells mittels Code-Cities umgesetzt in SEE mit rein tabellarischen Darstellungen, in diesem Paper exemplarisch betrachtet an dem State-of-the-Art-Dashboard der Firma AXIVION. Sowohl der als Code-City dargestellte Abhängigkeitsgraph als auch die Metriken zu den Code-Smells wurden mit einer statischen Analyse der AXIVION-Suite extrahiert. Somit enthalten die verglichenen Darstellungen dieselbe Information.

2.1 Abhängige Variablen der Studie

In dieser Vergleichsstudie untersuchen wir fünf abhängige Variablen, die jeweils zwischen den beiden Systemen verglichen werden: Die **Korrektheit** (Anteil korrekter Antworten), **Geschwindigkeit** (benötigte Zeit) und **Usability**, wobei die Usability einerseits durch den Post-Study-Fragebogen *System Usability Scale* (SUS) [Bro96] und andererseits durch den Post-Task-Fragebogen des *After-Scenario Questionnaire* (ASQ) [Lew91] gemessen wird. Beim ASQ erfassen wir dabei den empfundenen **Aufwand** (Mühsehnlichkeit) separat von der empfundenen **Komplexität** (kognitive Herausforderung).

*galperin@uni-bremen.de

¹Als Code-Smell-Typen unterscheiden wir zyklische Abhängigkeiten, toten Code, Klone, sowie Architektur-, Metrik- und Stil-Verletzungen.

²Eine Ebene besteht aus allen Knoten, die den gleichen Abstand zum Wurzelknoten haben.

2.2 Vergleichene Darstellungen

In den Code-Cities werden Code-Smells als Icons über betroffenen Knoten dargestellt, aufgeteilt nach den Typen¹ vorhandener Code-Smells. Die Rotfärbung eines Icons entspricht dabei der relativen Häufigkeit dieses Code-Smell-Typs innerhalb einer Ebene² des hierarchischen Abhängigkeitsgraphen. So können Stellen mit ungewöhnlich vielen Code-Smells schnell erkannt werden. Durch das Hovern über einem Knoten werden die genauen Zahlen eingeblendet, die die Anzahl an Vorkommnissen dieses Typs im jeweiligen Knoten repräsentiert – dies kann z. B. in [Abbildung 1](#) gesehen werden.

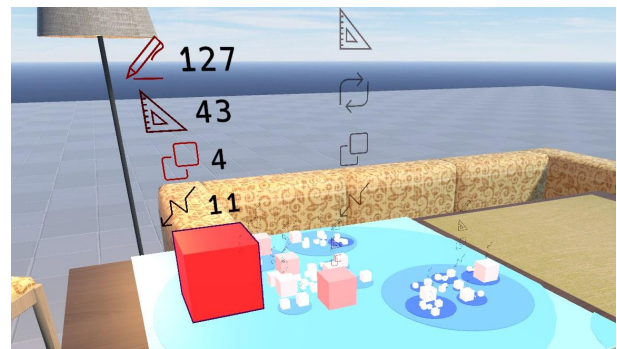


Abbildung 1: Code-City mit Code-Smell-Icons.

Im AXIVION-Dashboard hingegen werden die Code-Smells tabellarisch aufgelistet – dort wird der Ordnerbaum des zu untersuchenden Projekts mit den zugehörigen aggregierten und nach Typ getrennten Code-Smell-Zahlen angezeigt. Diese auch *Tree View* genannte Ansicht lässt sich einerseits durch die Angabe eines (Teil)pfs filtern und andererseits nach der Code-Smell-Anzahl insgesamt oder eines bestimmten Typs sortieren. Durch das Anklicken einer Datei öffnet sich der Code-Viewer des Dashboards für diese Datei, worin der Quellcode und die Zeilenanzahl (LOC) der Datei gefunden werden kann.

2.3 Versuchsaufbau

Damit die den Probanden gestellten Aufgaben repräsentativ sind, wurden AXIVION-Mitarbeiter nach typischen Nutzungsszenarien des Dashboards befragt. Auf Basis dieser Erkenntnis wurden drei Aufgaben entwickelt, die die Erkennung von Gottklassen als Thema haben. Als zugrunde liegendes, zu untersuchendes Projekt wurde hier der Kern des Quellcodes von SEE (≈50.000 LOC) verwendet, welcher aus

	1 _a	2 _a	3 _a	1 _b	2 _b	3 _b
Korrektheit	D ($p \approx 0,03$)	D ($p \approx 0,01$)	–	S ($p \approx 0,02$) ⁴	–	–
Geschwindigkeit	S ($p \approx 10^{-5}$)	S ($p \approx 0,008$)	–	S ($p \approx 0,007$)	S ($p \approx 10^{-4}$)	–
ASQ (Aufwand)	S ($p \approx 0,03$)	–	–	S ($p \approx 0,003$)	S ($p \approx 0,01$)	–

Tabelle 1: Signifikante Unterschiede der Variablen zugunsten **SEE** bzw. **Dashboard** in den einzelnen Aufgaben.

ca. 9.000 Knoten besteht. Dabei wurden alle Aufgaben von allen Probanden einmal mit SEE und einmal mit dem Dashboard bearbeitet, wobei die Reihenfolge der beiden Systeme randomisiert wurde, um einem Einfluss der Bearbeitungsreihenfolge entgegen zu wirken. Eine Aufgabe i bezeichnen wir als i_a , wenn wir über den ersten Durchlauf reden und als i_b , wenn es um den zweiten Durchlauf geht. Wir trennen dabei die Architektur- von den Stil- bzw. Metrik-Verletzungen, da die letzteren beiden in ihrer Anzahl deutlich stärker miteinander korrelieren ($r \approx 0,66$) als Architektur- und Stil- ($r \approx 0,24$) oder Architektur- und Metrik-Verletzungen ($r \approx 0,13$). Die anderen drei Code-Smell-Typen kamen zu selten vor und wurden ausgelassen.

Die gestellten Aufgaben waren wie folgt (X, Y, Z sind Platzhalter für Ordner der Implementierung von SEE, sie wurden für die Durchläufe variiert belegt, um Lerneffekte zu vermeiden; die jeweiligen Ordner für eine Aufgabe, die von jedem Probanden einmal für die Code-City und einmal für die tabellarische Darstellung bearbeitet wurde, sind jedoch in ihren relevanten Eigenschaften vergleichbar gewählt):

1. Finde im Ordner X die fünf größten Dateien. Wie viele Architektur-Verletzungen haben diese jeweils?
2. Finde im Ordner Y die fünf größten Dateien. Wie viele Stil-Verletzungen (2_a) bzw. Metrik-Verletzungen (2_b) haben diese jeweils?
3. Finde für die fünf gegebenen größten Dateien des Ordners Z jeweils den häufigsten Code-Smell-Typen.

2.4 Ergebnisse

Die Studie wurde asynchron online durchgeführt. Es nahmen insgesamt $n = 20$ Probanden teil, die wir durch Convenience Sampling erhalten haben. Die Korrektheit und Geschwindigkeit wurden automatisiert gemessen, für die anderen untersuchten Variablen wurden die erwähnten Fragebögen abgefragt. Wir haben uns hier im Vorfeld auf ein Signifikanzniveau von $\alpha = 0,05$ festgelegt und verwenden zum Feststellen signifikanter Unterschiede den *Mann-Whitney-U-Test*, der keine bestimmte Verteilung voraussetzt. Dabei vergleichen wir jeweils den gleichen Durchlauf der gleichen Aufgabe zwischen den zwei Systemen. Die Ergebnisse dieses Vergleichs für Korrektheit, Geschwindigkeit und empfundenem Aufwand sind in [Tabelle 1](#) gelistet – hier lässt sich zusammenfassen, dass die benötigte Zeit und der empfundene Aufwand

signifikant geringer in SEE war, beim Anteil korrekter Antworten hat das Dashboard jedoch signifikant besser abgeschnitten. Die empfundene Komplexität der Aufgaben erwies keine signifikanten Unterschiede.

Der Mann-Whitney-U-Test zeigt eine signifikant höhere Usability des Dashboards ($p \approx 0,03$) gemessen durch den SUS – in SEE lag der Median-SUS-Score bei 64, im Dashboard bei 79 von maximal 100 Punkten.

2.5 Threats to Validity

Selektionseffekte wurden durch die randomisierte Zuweisung adressiert. Teilnehmer wurden außerdem u.A. nach ihrer Erfahrung mit Dashboard und SEE gefragt. Der Mann-Whitney-U-Test zwischen den verglichenen Gruppen ergab bei diesen Aspekten keinen signifikanten Unterschied.

Alle Teilnehmer waren männlich, außerdem waren zwölf Teilnehmer Studenten, was problematisch hinsichtlich der Repräsentanz sein kann.

Im Dashboard musste zusätzlich der Code-Viewer geöffnet werden, um die LOC zu ermitteln. In SEE hingegen gab es keine genauen Größenangaben; diese konnten nur im Vergleich abgeschätzt werden.

3 Fazit

Wir erhalten hier als Ergebnis, dass das Dashboard als tabellarische Darstellung von Code-Smells zwar grob einen höheren Anteil korrekter Antworten erzielt und bei der Usability besser abschneidet, dass aber eine Analyse mit SEE als Code-City-Visualisierung dafür sowohl weniger Zeit kostet, als auch als weniger mühselig eingeschätzt wird. Dies liefert empirische Evidenz für die häufig aufgestellte Hypothese, dass sich Code-Cities eher für eine schnelle Übersicht und Tabellen hingegen besser für Detailanalysen eignen. Details zur Visualisierung und Studie können in meiner Bachelorarbeit³ nachgelesen werden.

Literatur

- [Bro96] John Brooke. „SUS: A ‘Quick and Dirty’ Usability Scale“. In: *Usability Evaluation In Industry*. CRC Press, 1996, S. 189–194.
- [Lew91] James R. Lewis. „Psychometric evaluation of an after-scenario questionnaire for computer usability studies: the ASQ“. In: *ACM SIGCHI Bulletin* 23.1 (1991), S. 78–81.

³Abrufbar als PDF unter <https://v.gd/CodeSmellsCities>.

⁴Hier wurde das Ergebnis aufgrund zu vieler „Ties“ mit dem Fisher-Pitman-Randomisierungstest bestätigt.

Automatisierte Migration von Legacy-Dateien in relationale Datenbanken

Felix Graßler

pro et con Innovative Informatikanwendungen GmbH, Reichenhainer Straße 29a, 09126 Chemnitz

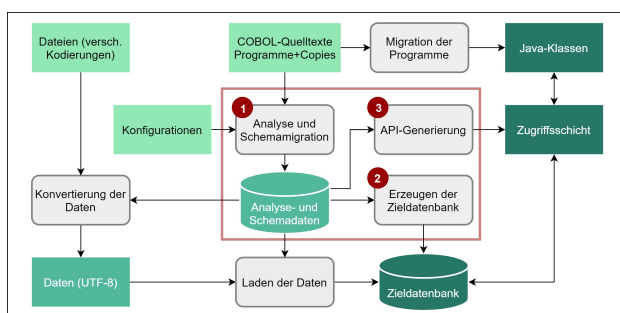
felix.grassler@proetcon.de

Abstract

Dieses Paper basiert auf der Masterarbeit des Autors [5]: Große Unternehmen setzen noch vielfach auf historisch gewachsene Legacy-Systeme mit dateibasierter Datenhaltung. Oft besteht der Wunsch, diese durch moderne Systeme abzulösen. Das Ziel dieser Arbeit war die Entwicklung einer allgemeinen Technologie für die werkzeuggestützte Migration einer dateiorientierten Datenhaltung zu einer relationalen Datenbank (DB). Dies wird im Kontext einer zeitgleichen COBOL-zu-Java-Programm migration des umgebenden Softwaresystems betrachtet. Um auch die Migration großer Datenbestände zu ermöglichen, wird eine hohe Automatisierung angestrebt. Der Fokus dieser Arbeit liegt auf zwei Aspekten: Der Migration des Datenschemas (dateiorientiert vs. relationale DB) sowie einer Migration der Datenzugriffe des umgebenden Softwaresystems. Bei letzterem sind die Unterschiede in den Programmiersprachen (COBOL vs. Java) sowie die abweichenden Arbeitsweisen (prozedural vs. SQL) zu beachten.

1 Überblick

Im Bereich der Datenmigration existieren bereits einige theoretische Ansätze sowie erfolgreich absolvierte Projekte [1][2][3]. Das Ziel dieser Arbeit war eine Vertiefung der bestehenden Ansätze und Technologien. Der Fokus lag besonders auf einer offenen, konfigurierbaren Vorgehensweise sowie einem hohen Grad an Automatisierung. Als Alternative zu einer Eigenentwicklung wurden auch existierende ETL-Technologien untersucht. Damit können Datenbestände aus einem Quellsystem extrahiert, transformiert und in ein Zielsystem geladen werden [6][4]. Es kann jedoch festgehalten werden, dass diese Werkzeuge für eine Datenmigration im Sinne dieser Arbeit nicht geeignet sind. Daher wurde stattdessen der nachfolgende Ansatz gewählt.



Wie dargestellt, umfasst eine Migration der Datenhaltung mehrere Aspekte. Es ist das Datenschema des Legacy-Systems zu ermitteln. In COBOL sind die Informationen über die verwendeten Dateien und die Strukturierung der Datensätze nicht Bestandteil der Datenhaltung (also der Dateien). Stattdessen sind sie im umgebenden Programmsystem enthalten. Sie werden in einer Analysephase extrahiert und sind die Grundlage für die Schemamigration (1). Bei dieser entsteht eine Menge von Tabellen- und Spalten-

beschreibungen, mit denen die Datenbestände des Legacy-Systems in einer relationalen DB abgebildet werden können. Darauf basierend erfolgt die Erzeugung der Ziel-DB (2). Im Zielsystem unterscheidet sich die Art der Zugriffe auf die Daten in mehreren Aspekten. Daher ist die Generierung einer Zugriffsschicht notwendig (3).

2 Analyse des Programmsystems

In COBOL existieren verschiedene Organisations- und Zugriffsarten von bzw. auf Dateien. In einer Datei können unterschiedliche Arten von Datensätzen gespeichert werden, welche sich in Länge, Strukturierung und Datentypen unterscheiden. Diese und weitere Informationen stellen die Grundlage der weiteren Schritte dar. Sie sind Bestandteil der COBOL-Programme und müssen in einer Analysephase extrahiert werden. Dafür stehen bei pro et con ausgereifte Werkzeuge zur Verfügung [2].

Die Programmiersprache COBOL bietet einige Features, welche kein Äquivalent in Java besitzen, aber Einfluss auf die Datenhaltung haben. Diese sind insbesondere Redefinitionen und Filler. Mit Redefinitionen können Teilbereiche eines Datensatzes unterschiedlich interpretiert werden. Filler kommen i. d. R. zum Einsatz, wenn mehrere Strukturbeschreibungen jeweils nur einen Teil des Datenbereiches beschreiben. Alle genannten Informationen sind in einem Analyseprozess aus den Programmen zu gewinnen. Dadurch entsteht das Datenschema des Legacy-Systems.

3 Migration des Datenschemas

Bei der Transformation des Datenschemas sind verschiedene Aspekte zu beachten:

(1) Die Abbildung von Dateien zu Tabellen. Aus einer Datei können eine oder mehrere Tabellen hervorgehen, bspw. in Abhängigkeit der verwendeten Strukturbeschreibungen sowie der Anzahl unterschiedlicher Datensatzarten, welche in derselben Datei gespeichert werden. Zudem sind Spalten für Verwaltungsinformationen vorzusehen. Damit wird Verhalten explizit im Zielsystem nachgebildet, welches im Legacy-System implizit umgesetzt wurde. Das betrifft z. B. die garantierte Reihenfolge der Datensätze bei sequentiellen Dateien sowie die Schlüsselverwaltung bei indexsequentiellen und relativen Dateien. Das entwickelte Werkzeug unterstützt die Definition allgemeiner und dateispezifischer Regeln für die Abbildung.

(2) Große Datenfelder können in Untertabellen ausgelagert werden. Das erhöht die Übersicht und Wartbarkeit des Zielschemas. Eine Auslagerung ist z. T. technisch notwendig. Auch hier können Regeln für spezifische Datenfelder definiert werden, sowie allgemeine Regeln in Abhängigkeit der Anzahl der Datenelemente, der Größe in Zeichen und der Verschachtelungstiefe der entstehende Tabelle.

(3) Redefinitionen beschreiben denselben Speicherbereich unterschiedlich. Es ist jeweils zu entscheiden, welche Beschreibung(en) in das Zielschema übernommen wird/werden. Das kann für spezifische Redefinitionen erfolgen, so-

wie durch allgemeine Regeln. So kann bspw. automatisch stets die detaillierteste Redefinition gewählt werden.

(4) Die Datentypen des Zielsystems müssen mit den Datenbeständen des Legacy-Systems zusammenpassen. Dafür sind Informationen aus der Analysephase sowie ein Kenntnis des exakten Ziel-DBS notwendig. Daher kann letzteres für die Migration konfiguriert werden.

(5) Bezeichner können nicht immer 1:1 übernommen werden. Das liegt z. B. an unterschiedlichen Namenskonventionen in Legacy- und Zielsystem. Zudem stellen unterschiedliche DBS eigene Restriktionen an mögliche Bezeichner, bspw. bei der Länge und der Auswahl der erlaubten (Start-)Zeichen. Für die Namenskonvertierung können allgemeine und spezifische Regeln definiert werden.

Die genannten Konfigurationen werden in separaten Dateien vorgenommen. Dies ermöglicht eine Wiederholung der Schemamigration mit verschiedenen Konfigurationen. Zudem können diese in einer Versionsverwaltung abgelegt werden, um durchgeführte Schemamigrationen reproduzieren zu können. Beide Punkte sind im Kontext von komplexen Migrationsprojekten relevant. Das Ergebnis der Schemamigration ist eine Menge von Tabellen- und Spaltenbeschreibungen (das Datenschema des Zielsystems).

4 Datenzugriffe im Zielsystem

4.1 Erzeugen der Datenbank

Aus dem gewonnenen Zielschema entsteht im nächsten Schritt die DB des Zielsystems. Dafür werden DDL-Skripte mit *create table*-Statements generiert. Diese müssen den SQL-Dialekt des konkreten Ziel-DBS berücksichtigen. Sie übersetzen damit das Schema in konkrete Anweisungen für das spezifische DBS des Zielsystems. Um eine hohe Performanz bei Zugriffen auf ehemals index-sequentiell oder relativ organisierte Datenbestände zu gewährleisten, werden zusätzlich Indizes generiert.

4.2 Zugriffsschicht

Das umgebende Programmsystem wird von COBOL nach Java migriert und die dateibasierte Datenhaltung in eine relationale DB überführt. Dadurch unterscheiden sich bei Legacy- und Zielsystem sowohl die Organisation der Datenbestände, als auch die Datenzugriffe: (1) Im Legacy-System greifen die Programme direkt auf die Dateien zu. Im Zielsystem wird ein DBMS als Zwischenschicht genutzt. (2) Die Persistierung eines Datensatzes erfolgt im Legacy-System als zusammenhängender Speicherbereich, während im Zielsystem die Datenelemente separat gespeichert werden. (3) Zur Laufzeit werden im Zielsystem Datenklassen genutzt, wohingegen im Legacy-System ein Speicherbereich interpretiert wird. (4) Das Legacy-System und die migrierten Programme nutzen (logische) Dateioperationen wie READ, WRITE und START, um auf die Daten zuzugreifen. Im Zielsystem werden DB-Operationen genutzt.

Aus diesen Gründen ist eine Zwischenschicht für den Zugriff der migrierten Programme auf die Datenbestände erforderlich. Sie übersetzt logische Dateioperationen in DB-Operationen und übernimmt eine objektrelationale Abbildung. Damit werden DB-Spalten und Attribute der Java-Datenklassen einander zugeordnet und somit Datensätze

und Java-Objekte ineinander übersetzt. Dabei ist eine hohe Performanz erforderlich. Aus diesem Grund wird für die Zugriffsschicht ein hybrider Ansatz gewählt: Eine Menge abstrakter Klassen setzt allgemeine Funktionalität um und Interfaces stellen den Programmen logische Dateioperationen zur Verfügung. Zusätzlich wird für jede Tabelle eine separate Hilfsklasse generiert. Diese übernehmen die objektrelationale Abbildung. Besonderheiten wie die Verwendung von Redefines und das Auslagern von Datenfeldern sind darüber abgebildet. Es wurden auch alternative Vorgehensweisen untersucht, wie z. B. die Nutzung von Reflection und Annotation Processing. Ersteres ist jedoch aufgrund der Performanz nicht geeignet und letzteres stellte große Herausforderungen dar, wenn im Legacy-System keine einfache 1:1-Zuordnung von einer Strukturbeschreibung zu einer Datei existiert.

5 Test der Datenmigration

Der Test des Gesamtprozesses erfolgt im direkten Vergleich COBOL vs. Java. Zunächst werden COBOL-Programme mit Dateiarbeiten ausgeführt. Dabei entstehen Referenz-Datenbestände. Anschließend werden die Programme nach Java übersetzt und anhand des hier beschriebenen Prozesses werden eine DB und die Zugriffsschicht generiert. Eine Ausführung der migrierten Java-Programme ergibt nun ebenfalls Datenbestände (in der DB). Ein Vergleich mit den Referenz-Datenbeständen stellt sicher, dass Schreiboperationen korrekt funktionieren. Für den Vergleich von Leseoperationen lesen die COBOL- und Java-Programme jeweils Datensätze aus den Dateien bzw. der DB ein und geben die Ergebnisse aus. Die so entstehenden Ausgaben können automatisiert verglichen werden.

6 Zusammenfassung

In der Arbeit wurden existierende Ansätze und Technologien für die Datenmigration erweitert. Es entstand ein automatisierter, konfigurierbarer Prozess, mit welchem Datenbestände aus Legacy-Systemen modernisiert werden können. Die Automatisierung ermöglicht eine praktikable Migration umfangreicher Systeme. Durch die offene Gestaltung können diverse Besonderheiten und Individualitäten abgebildet werden. Für eine vollständige Datenmigration ist, zusätzlich zu den hier beschriebenen Prozessen, eine Überführung der reinen Datenbestände notwendig. Diese war nicht Gegenstand der Arbeit.

Literaturverzeichnis

- [1] Becker, C.; Kaiser, U.: Toolbasierte Software-Migration nach Plan. WSRE 2016
- [2] Erdmenger, U.; Prof. Dr. Kaiser, U; Loos, A.; Uhlig, D.: Methoden und Werkzeuge für die Software Migration. WSRE 2008
- [3] De Marco, A.; Iancu, V.; Asinofsky, I.: COBOL to Java and Newspapers Still Get Delivered. 2018 IEEE International Conference on Software Maintenance and Evolution.
- [4] Biplob, Md. B.; Sheraji, G. A.; Khan, S. I.: Comparison of Different Extraction Transformation and Loading Tools for Data Warehousing. 2018. 2nd ICISSET.
- [5] Grabler, F.: Untersuchung und Implementierung von Verfahren für die automatisierte Migration von Dateien in relationale Datenbanken. Masterarbeit, 2021. An TU Chemnitz Fakultät für Informatik.
- [6] Vassiliadis, P.: A Survey of Extract-Transform-Load Technology. 2009. International Journal of Data Warehousing and Mining.

Analyse von Entwurfsentscheidungen in natürlichsprachiger Softwarearchitekturdokumentation

Bjarne Sauer

KASTEL - Institut für Informationssicherheit und Verlässlichkeit

Karlsruher Institut für Technologie

Karlsruhe, Deutschland

bjarne.sauer@t-online.de

Zusammenfassung—Entwurfsentscheidungen bilden das Fundament zur Entwicklung qualitativ hochwertiger Softwaresysteme. Ihre Extraktion aus und Klassifikation in natürlichsprachiger Softwarearchitekturdokumentation ermöglichen die Informationsgewinnung für Implementierungs- und Wartungsprozesse sowie die Ausführung nachgelagerter Analysen, etwa Konsistenzprüfungen.

Das in dieser Arbeit entwickelte hierarchische Klassifikationsschema (Taxonomie) für Entwurfsentscheidungen erweitert bestehende Ansätze in der hierarchischen Tiefe und um feingranularere Trennlinien. Es entsteht aus einem iterativen Prozess, in dem ein auf Literatur basierendes initiales Schema entlang von 17 Softwarearchitekturdokumentationen aus Open-Source-Projekten weiterentwickelt wird. Dabei wird die Taxonomie wiederholt und final hinsichtlich Struktur, Vollständigkeit und Anwendbarkeit evaluiert.

Das entstandene Textkorpus wird anschließend zur automatischen Identifikation und Klassifikation von Entwurfsentscheidungen in Softwarearchitekturdokumentationen verwendet. Die Anwendung verschiedener Ansätze des maschinellen Lernens ermöglicht die Identifikation von Sätzen mit Entwurfsentscheidungen mit einem F1-Wert über 0,9 und einer Klassifikation anhand der Ebenen des Schemas mit F1-Werten von 0,6 bis 0,7.

I. MOTIVATION

Da der Erfolg eines Softwareprojekts maßgeblich von der Qualität der Dokumentation abhängig ist, ist es essentiell, die getroffenen Entwurfsentscheidungen ausführlich und nachvollziehbar zu dokumentieren. Entwurfsentscheidungen betreffen dabei insbesondere die Architektur der Software, aber auch den Einsatz bestimmter Technologien und Werkzeuge. Während die Softwarearchitektur textuell oder graphisch weitgehend dokumentiert wird, wird die explizite Dokumentation der zugrundeliegenden Entwurfsentscheidungen oft vernachlässigt. Dies führt dazu, dass Entwurfsentscheidungen in späteren Entwicklungs- und Wartungsprozessen übersehen werden, nur noch veraltet dokumentiert sind oder die Entscheidungsbegründung nicht mehr zu rekonstruieren ist. Dieser Wissensverlust führt zu wiederkehrenden Problemen beim Entwurf komplexer Systeme, hohen Kosten für Veränderungen und einer Erosion der Softwarearchitektur.

Eine Möglichkeit, Zeit- und Budgeteinsparungen bei gleichzeitig hoher Dokumentationsqualität zu erreichen, ist der Ein-

satz automatisierter Verfahren während des Entwicklungsprozesses. Durch die automatisierte Extraktion von Entwurfsentscheidungen lassen sich die für den Softwareentwickler in Implementation und Wartung relevanten Informationen aus der Dokumentation übersichtlich darstellen. Darüber hinaus ermöglicht eine feingliedrige Klassifikation die Entwicklung von Anwendungen für Rückverfolgbarkeits- und Konsistenzanalysen hinsichtlich verschiedener Dokumentations- und Softwareartefakte, wie etwa beschrieben von Keim et al. [1] und Wohlrab et al. [2]. Bestehende Schemata und Modelle sind für diese Zwecke zu grob gegliedert.

II. ZIELSETZUNG

Um Entwurfsentscheidungen zu erfassen und adäquat in die Entwicklung des Systems zu übersetzen, müssen Softwareentwickler den Anwendungsbereich der Entscheidungen und die intendierten Auswirkungen verstehen. Untersucht wird daher, wie natürlichsprachige Softwaredokumentation strukturiert ist und wie Entwurfsentscheidungen darin beschrieben werden. Hierfür ist es sinnvoll, die Entwurfsentscheidungen durch eine Klassifikation voneinander abzugrenzen, und ihnen damit eine Repräsentation erster Ordnung zu geben.

Die Anwendbarkeit der Taxonomie ist maßgeblich davon abhängig, die Klassifikation in einem zweiten Schritt auch automatisiert durchzuführen. Durch die manuelle Analyse der Fallstudien entsteht ein mit Labels versehener, englischsprachiger Textkorpus. Dieser wird genutzt, um eine Proof-of-Concept-Implementation der automatisierten Analyse anhand der Taxonomie mittels verschiedener Ansätze des überwachten, maschinellen Lernens (ML) umzusetzen.

III. DURCHFÜHRUNG

Den Ausgangspunkt bildet ein initiales Klassifikationsschema, entwickelt anhand von verwandter Literatur und theoretischen Vorüberlegungen. Es basiert insbesondere auf Kruchten's Ontologie für architektonische Entwurfsentscheidungen [3], aus der insbesondere die Grobstruktur mit den Oberklassen *Existenz*-, *Eigenschafts*- und *Ausführungsentscheidung* übernommen wird. Im Anschluss wird das Schema iterativ auf

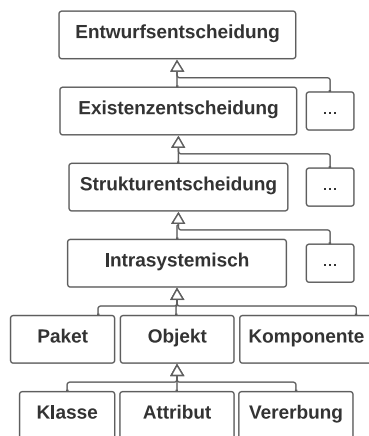


Abbildung 1. Ausschnitt aus dem Klassifikationsschema mit feingranularen Unterklassen für strukturelle Existenzentscheidungen

die Fallstudien angewandt. Die darin auffindbare Dokumentation der Softwarearchitektur wird dazu genutzt, verschiedene Entwurfsentscheidungen manuell zu identifizieren und zu klassifizieren. Dabei wird die Passform der aktuellen Version des Schemas anhand der Fallstudien geprüft und bei auftretenden Unzulänglichkeiten das Schema durch Erweiterungen und Verminderungen angepasst. Der iterative Weiterentwicklungsprozess wird beendet, sobald sich nachhaltig, d.h. in zwei kompletten Iterationen mit jeweils drei enthaltenen Fallstudien, keine Änderungen am Schema ergeben und das Schema somit final konvergiert. Diesen Vorgaben folgend wird insgesamt die Dokumentation aus 17 Open-Source-Projekten betrachtet, die allesamt eine hinreichend gute Dokumentationsqualität hinsichtlich Länge und Grammatik aufweisen, sich jedoch in der Domäne voneinander unterscheiden.

Nach einer Evaluation des finalen Schemas anhand der Prinzipien nach Bedford [4] und einer abschließenden Aktualisierung aller Label wird das resultierende Textkorpus für die automatisierte Analyse mithilfe von natürlicher Sprachverarbeitung mittels ML verwendet. Hierfür wird mit verschiedenen Konfigurationen des überwachten Lernens experimentiert, hinsichtlich Textvorverarbeitung, Vektorrepräsentationen und Klassifikationsalgorithmen. Die Klassifikation wird anschließend anhand einer dreifach-wiederholten fünffachen Kreuzvalidierung anhand der Korrektorklassifizierungsrate und des F1-Wertes evaluiert. Dies dient dazu, die Möglichkeit einer automatisierten Identifikation und Klassifikation von Entwurfsentscheidungen anhand des entwickelten Schemas zu beurteilen und zudem um verschiedene ML-Methoden für diesen Anwendungsfall miteinander zu vergleichen.

IV. ERGEBNISSE

Aus dem beschriebenen Entwicklungsprozess resultiert ein baumartiges Klassifikationsschema, welches insgesamt 43 Knoten aufweist, wovon 26 Blattknoten sind. Gegenüber bestehenden Ansätzen ist es in der hierarchischen Tiefe deutlich stärker ausdifferenziert und zieht feingranuläre Trennlinien

zwischen verschiedenen Klassen von Entwurfsentscheidungen und den damit verbundenen Auswirkungen auf die Architektur der Software. Damit ist es besonders für die Klassifikation im Rahmen von Rückverfolgbarkeits- und Konsistenzanalysen geeignet. Ein Ausschnitt aus dem Klassifikationsschema ist in 1 dargestellt und zeigt beispielhaft die Aufgliederung in spezielle Unterklassen struktureller Existenzentscheidungen.

Insgesamt lassen sich 1427 Entwurfsentscheidungen in den 17 Fallstudien identifizieren und vollständig anhand des Schemas klassifizieren, wobei etwa in 2/3 der Fälle nur eine Entwurfsentscheidung pro Satz auftritt und anderenfalls noch mindestens eine weitere, gegebenenfalls implizite Entscheidung getroffen wird. Das resultierende Textkorpus bildet eine solide Grundlage, um ML zur automatisierten Identifikation und Klassifikation anzuwenden. Dabei zeigt sich, dass für das gegebene Korpus die Textvorverarbeitung mittels Reduktion auf Kleinschreibung sowie Stammformreduktion und Lemmatisierung einen positiven Effekt hat. Die Überführung in eine Vektorrepräsentation kann durch das Zählen von Wortvorkommen in Trigrammen zielführend umgesetzt werden. Mit einem Random-Forest-Klassifikator sowie alternativ dem Sprachmodell BERT wird in der 5-fachen Kreuzvalidierung ein F1-Wert von über 90% für die Abgrenzung von Sätzen mit und ohne Entwurfsentscheidung erzielt. Für die Klassifikation innerhalb der Ebenen des Schemas erzielen Logistische Regression und Multinomiale Naive Bayes mit F1-Werten zwischen 60% und 70% die vielversprechendsten Ergebnisse.

V. FAZIT & AUSBLICK

In dieser Arbeit wurde ein feingranulares Klassifikationsschema für Entwurfsentscheidungen entwickelt, welches die Rückgewinnung dieser aus Dokumentationsartefakten ermöglicht. Die Anwendbarkeit des Schemas für eine automatisierte Analyse wurde unter Verwendung verschiedener ML-Ansätze gezeigt.

Aufbauend auf dem entwickelten Klassifikationsschema und orientiert an diesen Ergebnissen können nun weitere Anwendungen geschaffen werden, etwa um Inkonsistenzen zwischen natürlichsprachiger Softwarearchitekturdokumentation und formalen Modellen eines Softwaresystems aufzuspüren. Möglich erscheint beispielsweise ein hierarchisch gegliederter, mehrstufiger Klassifikator, der sich entlang des Schemas auf jeder Ebene für die wahrscheinlichste Option entscheidet, bis ein Blattknoten erreicht ist.

LITERATUR

- [1] J. Keim and A. Koziol, "Towards Consistency Checking Between Software Architecture and Informal Documentation," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2019, pp. 250–253.
- [2] R. Wohlrab, U. Eliasson, P. Pelliccione, and R. Heldal, "Improving the Consistency and Usefulness of Architecture Descriptions: Guidelines for Architects," in *2019 IEEE International Conference on Software Architecture (ICSA)*, Mar. 2019, pp. 151–160.
- [3] P. Kruchten, "An ontology of architectural design decisions in software-intensive systems," *2nd Groningen Workshop on Software Variability*, pp. 54–61, 2004.
- [4] D. Bedford, "Evaluating classification schema and classification decisions," *Bulletin of the American Society for Information Science and Technology*, vol. 39, no. 2, pp. 13–21, 2013.

Eine Waschmaschine für Software - Automatisiert technische Schulden bereinigen

Daniela Schilling
dschilling@delta-software.com
Delta Software Technology GmbH

Abstract

Seit mehr als 30 Jahren entwickelt und pflegt RDW Anwendungen zur Kraftfahrzeugsverwaltung. Die Anwendungen funktionieren zuverlässig, doch mit der Zeit haben sich technische Schulden angesammelt, die die Wartung und Weiterentwicklung erschweren. Das Tagesgeschäft sowie Größe und Kritikalität der Anwendung lassen eine manuelle Bereinigung der Software nicht zu. Die Lösung für das Problem: eine Waschmaschine, die die Software automatisiert von den technischen Schulden bereinigt.

1 Die Anwendungen laufen, aber...

RDW ICT ist der IT-Dienstleister der nationalen Kraftfahrzeugverwaltung der Niederlande. Zu den Hauptaufgaben gehören die Inspektion und Registrierung von Fahrzeugen, die Marktzulassung von Fahrzeugen, die Verwaltung von Führerscheinen, Auskunftsdienste für die Polizei - insgesamt mehr als 300 Millionen Transaktionen pro Jahr.

Die für die Aufgaben notwendigen Anwendungen werden seit den 1980er Jahren von RDW entwickelt. Die Anwendungen wurden in den Jahrzehnten gepflegt und kontinuierlich weiterentwickelt, an neue Anforderungen angepasst, neue Technologien eingeführt, Plattformen ausgetauscht und vieles mehr. Bis heute laufen sie stabil und verrichten zuverlässig ihren Dienst, aber: die in den Jahrzehnten durchgeführten Änderungen und Anpassungen, sowie Generationen von Entwicklern haben ihre Spuren hinterlassen. Es haben sich technische Schulden angesammelt, die

- eine effiziente Wartung behindern,
- ein Zusammenspiel mit Komponenten in "modernen" Sprachen und mit neuen Technologien erschweren,
- keine agile Entwicklung zulassen und
- eine Integration in neuere Entwicklungsprozesse erschweren.

Kurz: die die Zukunftsfähigkeit der Anwendung und damit die Geschäftsprozesse gefährden.

Größe, Komplexität und Kritikalität der Anwendung lassen ein Neuschreiben nach aktuellen Paradigmen und mit modernen Sprachen jedoch nicht zu; Aufwand und Risiko wären zu groß.

2 Automatisierte Bereinigung

Statt alles neu zu machen, entwickelten RDW und Delta Software Technology ein Konzept wie die bewährten Anwendungen in überschaubaren Schritten bereinigt werden können und das so, dass die reguläre Wartung und Weiterentwicklung und noch viel weniger der laufende Betrieb beeinträchtigt werden. Wie bei vielen (Modernisierungs-)projekten waren zu Beginn noch nicht alle Anforderungen bekannt. Statt langwierige Analysen vorab durchzuführen, wurde beschlossen das Projekt zu beginnen und neue Erkenntnisse sukzessive einfließen zu lassen.

2.1 Eine Waschmaschine für Software

In einem iterativen Prozess wurde eine Factory aufgesetzt, die die Sourcen automatisiert von den technischen Schulden bereinigt, aus diesem Grund auch als Washing Machine bezeichnet. Die Washing Machine arbeitet regelbasiert und nach einem strengen Clean Room-Konzept, d.h. mit strikten Prozessen, in die nicht von außen eingegriffen werden kann. Dadurch lässt sie sich jederzeit um neue Regeln zur Analyse und Transformation erweitern. Außerdem sind die von der Washing Machine durchgeführten Änderungen immer reproduzierbar.



Figure 1: Eine Waschmaschine für Software

2.2 Schritt für Schritt zu sauberer Software

In einem ersten Schritt wurde eine Version der Washing Machine erstellt, die die Sourcen von RDW verarbeiten kann und ein Set von Bereinigungsregeln umsetzt. Mittels Meta Level Test wurde anhand einer Teilapplikation die Korrektheit der Regeln geprüft. Noch während die Tests für das erste Regel-Set liefen, wurden die nächsten Bereinigungsmaßnahmen definiert und in einem weiteren Regel-Set umgesetzt und erneut nach dem Konzept des Meta-Level-Tests überprüft.

Die einzelnen Regel-Sets bilden jeweils ein Waschprogramm der Washing Machine. Sobald ein neues Waschprogramm zur Verfügung stand, konnten die Anwendungsentwickler bei RDW entscheiden, ob sie ihre Teilanwendung bereits mit einem oder mehreren existierenden Waschprogrammen bereinigen lassen wollen oder auf weitere in Arbeit befindliche Waschprogramme warten wollen. Durch die Reproduzierbarkeit der Bereinigungen konnten Teilanwendungen, die schon frühzeitig erste Waschgänge durchlaufen haben, zu einem späteren Zeitpunkt weitere relevante Waschgänge durchlaufen.

2.3 Der Meta-Level-Test

Die Kritikalität der Anwendung erforderte eine sehr gründliche Überprüfung der Bereinigung. Die Größe der Anwendung und die Anzahl der durchgeführten Änderungen verhinderte jedoch, dass jede durchgeführte Änderung einzeln getestet werden konnte. Aus diesem Grund entschied sich RDW, Tests nach dem Konzept des Meta Level Testings durchzuführen. Beim Meta Level Test wird ausgenutzt, dass die Washing Machine regelbasiert arbeitet und die durchgeführten Bereinigungen immer wieder reproduziert werden können. Die Annahme ist, wenn die Washing Machine eine Regel einmal korrekt anwendet, dann wird sie das immer wieder tun. Aus diesem Grund ist es nicht notwendig jede gemachte Änderung zu testen, sondern alle Regeln.

Die Washing Machine lieferte RDW zusätzlich zu den bereinigten Programmen auch eine Übersicht darüber, welche Regeln existieren und in welchen Programmen diese angewendet wurden. Auf diese Weise war es für RDW leicht, geeignete Test-Sets zu ermitteln.

2.4 Paketweise Produktivsetzung

Statt die gesamten Anwendungen in einem Big-Bang zu bereinigen, wurden Teilpakete gebildet. Diese wurden passend zu den regulären Wartungszyklen durch die Washing-Machine bearbeitet.

Während bei Delta die einzelnen Waschprogramme in der Factory umgesetzt wurden, lief bei RDW die Wartung und Entwicklung ungestört weiter. Nach erfolgreichem Meta-Level-Test für das jeweilige Waschprogramm, lieferte RDW den aktuellen Source-Stand der gewünschten Teilpakete an Delta. Diese Sourcen wurden dann durch die Washing-Machine von ihren technischen Schulden bereinigt und an RDW zurückgeliefert. Dazu war lediglich ein sehr kurzer Freeze der Sourcen von wenigen Stunden oder einem Wochenende erforderlich. Die bereinigten Sourcen konnten bei RDW integriert und produktiv gesetzt werden. Weitere Tests waren dank des Meta-Level-Tests nicht notwendig. Aus dem gleichen Grund entstand für RDW auch kein Mehraufwand, wenn Sourcen mehrfach durch die Washing Machine bereinigt wurden.

Durch die Washing Machine wurden 16.800 Artefakte (COBOL-Programme und -Copybooks, sowie Delta ADS-Macros) der Anwendung verarbeitet. Bis heute wurden in 4.236 Artefakten Bereinigungen durchgeführt, dies führte zu insgesamt 53.740 Änderungen.

Die zu Beginn des Projekts angedachten Bereinigungsmaßnahmen sind inzwischen alle umgesetzt. Allerdings haben sich inzwischen weitere Möglichkeiten herauskristallisiert. Zur Zeit wird z.B. geprüft, wie das Prinzip der Washing Machine auch für Refaktorisierungen der Anwendungen oder eine Bereinigung von Datenmodellen inkl. der dazu notwendigen Änderungen in den Sourcen eingesetzt werden kann.

3 Fit für die Zukunft

Es liegt in der Natur der Sache, dass sich in Anwendungen, die über viele Jahre entwickelt und gepflegt werden, technische Schulden ansammeln. RDW entschied sich, diese technischen Schulden automatisiert entfernen zu lassen. Da zunächst nicht alle Anforderungen bekannt waren, wurde in einem iterativen Prozess eine passgenaue Washing Machine aufgesetzt. Die Anwendung wurde paketweise bereinigt, wobei bei jedem Waschgang entschieden werden konnte, welche Bereinigungsmaßnahmen vorgenommen werden sollten. Die Bereinigung erfolgte so, dass die geänderten Sourcen in die regulären Wartungszyklen eingeschleust werden konnten und durch die Änderungen kein Mehraufwand entstanden ist. Damit ist ein Schritt in Richtung Zukunftssicherheit der Anwendung gemacht, weitere Maßnahmen sind geplant und sollen dafür sorgen, dass RDW auch weiterhin zuverlässig 24/7 Kraftfahrzeuge verwalten kann.

Clang Preprocessor Tricks for Setting up Source Code Analysis Tools

Jochen Quante

Robert Bosch GmbH, Corporate Research
Renningen, Germany

jochen.quante@de.bosch.com

Abstract

clang¹ is in widespread use for development of C/C++ source code analysis tools. Many professional tools like Astrée² use clang as a C++ frontend, specially because the continuously evolving C++ standard causes a lot of effort on the tool side, and clang provides an adequate infrastructure. At the same time, many hardware-specific compilers use gcc³ as a basis. Although gcc and clang are compatible to a certain degree, analyzing gcc-based code with clang always runs into problems. In this paper, we propose a lightweight approach to address recurring problems in clang-based software analysis tool usage.

1 Motivation

When analyzing C or C++ code, it has to have been preprocessed before in order to be able to parse it. Unfortunately, preprocessing as such is a quite intransparent process: You only see the original code and the final result. It is often not clear how certain directives influence which code is active and which is not, and it is usually a lot of effort to trace these dependencies. For example, a `#if` at a given point may depend on how a certain macro is defined at this point, and the macro in turn may be defined differently in different places and depend on other `#defines` – or it may even be redefined differently. All this information is usually distributed over hundreds of header files. This makes it hard to track, especially in larger libraries like, e. g., the GNU C/C++ standard libraries. For the latter, there are hundreds of configuration switches for customizing the libraries to different compilers, hardware platforms, C/C++ standards, and so on. Therefore, it would be quite helpful to be able to trace what the preprocessor is actually doing.

We specifically investigate the use case of analyzing code with a tool that is based on a different frontend (e. g., clang instead of gcc). Given the libraries of a certain compiler, you get a configured set of C/C++ standard library files with it. This configuration was chosen to suit the respective compiler, as the compiler has certain capabilities and a number of built-in macros. The compiler and the library files work perfectly together. However, when we want to analyze

code that is usually processed by the compiler, we also have to process the original library header files, as certain functionalities that are specific to the compiler may be used – especially in hardware-dependent low-level code. On the other hand, these functionalities may use features that are not available in the analysis frontend. Examples include things like 128 bit floating point types or certain long long functions. This means we need to adapt the libraries somehow to make them work with the analysis frontend. And this usually means changing the configuration of the library by modifying `#defines`. But how can we find out which `#defines` lead to a working configuration?

2 Related Work

The problem of understanding code containing preprocessor directives has been investigated for a long time. Livadas et al. [3] already proposed a graph representation for preprocessor resolution in 1994. Kullbach and Riediger [2] proposed folding to help understanding macro expansions. Building an AST for preprocessor analysis was also proposed by Vidács [4], and tools like Bauhaus⁴ provide internal representations for macros and their resolution. Other approaches support the developer by highlighting code guarded by `#if` in the code [1] to support understanding. All these approaches require quite some infrastructure and an interactive environment. In contrast to that, we propose a very lightweight approach which is specifically targeted at helping users to find the right preprocessor settings for configuring a library.

3 Preprocessor Logging

clang provides the possibility to add callbacks to the preprocessor (class `PPCallbacks`⁵). The callbacks are notified whenever a preprocessor directive is processed. This includes definition or undefinition of macros, evaluating `#if` or `#ifdef` conditions, or checking whether a macro is defined or not. It also includes definition and use of a compiler's builtin macros. These callbacks make it very easy to create a preprocessor log that exactly shows what the preprocessor is doing.

¹<https://clang.llvm.org/>

²<https://www.absint.com/astree/index.htm>

³<https://gcc.gnu.org/>

⁴<https://www.axivion.com/en/products/axivion-suite/>

⁵https://clang.llvm.org/doxygen/classclang_1_1PPCallbacks.html

```

28 #ifndef _STDBOOL_H
29 #define _STDBOOL_H
31 #ifndef __cplusplus
33 #define bool _Bool
34 #define true 1
35 #define false 0
37 #else /* __cplusplus */
40 #define _Bool bool
42 #if __cplusplus < 201103L
44 #define bool bool
45 #define false false
46 #define true true
47 #endif
49 #endif /* __cplusplus */
52 #define __bool_true_false_are_defined 1
54 #endif /* stdbool.h */

```

```

File Change: include\stdbool.h:1:1
[28] #ifndef _STDBOOL_H => 1
[29] #define _STDBOOL_H
[31] #ifndef __cplusplus => 0
[31-37] skipped
[37] #else
[40] #define _Bool bool
[42] macro __cplusplus expands to 201402L
[42] #if => False
[42-47] skipped
[47] #endif (started in line 42)
[49] #endif (started in line 31)
[52] #define __bool_true_false_are_defined 1
[54] #endif (started in line 28)

```

Figure 1: Code example (left) and corresponding preprocessor debug log file (right).

Figure 1 shows an example for a header file code snippet and the corresponding preprocessor log for this code. The heading number is the respective line number in the code. Non-preprocessor code is omitted from the log file. We can see the evaluation results of macros, along with its consequences for inclusion or omission of code. In the example, it is easy to see that the compiler is running in C++ mode (`__cplusplus` defined), so the `bool` type is already defined. It is C++14, so macros for `true` and `false` need not be defined. When a macro has a certain value at a given point in the log, it is now very easy to trace back to the last `#define` for that macro by searching backwards (as all included files are handled in the same log file). This, in turn, enables the user of the analysis tool to find out the right `#defines` required to make the library compile using a different frontend.

4 Adapting a 3rd Party Standard Lib

The availability of certain compiler features (as described in Section 1) is usually hard-coded into some config file (e.g., `c++config.h` in `g++`). To override these settings, you would need to change these files, which you usually don’t want to do, as the original compiler should use the same header files.

A solution to this is to use the `PPCallbacks` mechanism to suppress definition of certain macros. This can be done by listening to `#define` events and immediately inserting an `#undef` with the same macro name⁶. An implementation can also allow overwriting `#defines` when specifying the new value in the compiler’s command line.

5 Practical Experiences

We have successfully used the two described techniques to adapt compiler-specific GNU-based libraries

for analysis with Astrée. The latter has a clang-based frontend and partially replaces the standard libraries with annotated ones to improve analysis precision. This requires even more specific changes to make Astrée’s standard libraries work along with the original libraries. Using this preprocessor logging approach, we can now quickly find working configurations for Astrée and other analysis tools, which was a time-consuming manual process before. Without the preprocessor log, we often ended up changing `#defines` that were local to the problematic code, but which later turned out not to be the real root-cause. Only the preprocessor log allows to quickly follow dependency chains and identify the key `#define`. Using the macro definition suppression mechanism, we can now easily overwrite configurations that lead to code that is not supported by clang – which was not possible before without duplicating the library files (which in turn imposes a maintenance problem). In summary, the two techniques make the process of finding working configurations much faster and easier.

References

- [1] J. Feigen span, C. Kästner, M. Frisch, R. Dachsel, and S. Apel. Visual support for understanding product lines. In *Proc. 18th Int’l Conf. on Program Comprehension*, pages 34–35, 2010.
- [2] B. Kullbach and V. Riediger. Folding: an approach to enable program understanding of preprocessed languages. In *Proc. of 8th Working Conference on Reverse Engineering*, pages 3–12, 2001.
- [3] P. E. Livadas and D. T. Small. Understanding code containing preprocessor constructs. In *Proc. of 3rd Workshop on Program Comprehension*, pages 89–97, 1994.
- [4] L. Vidács and Á. Beszédes. Opening up the C/C++ preprocessor black box. In *Proc. of 8th Symp. on Programming Languages and Software Tools (SPLST)*, pages 45–57, 2003.

⁶`Preprocessor.appendMacroDirective(new UndefMacroDirective())`

Towards Detecting Algorithm Implementations in Code Bases

Denis Neumueller
denis.neumueller@uni-ulm.de
Ulm University

Matthias Tichy
matthias.tichy@uni-ulm.de
Ulm University

1 Abstract

Developing an understanding of a software system is an integral part of a software-reengineering effort. Even though many approaches for supporting the process of software understanding exist, to the best of our knowledge, none focuses on leveraging information from the algorithms implemented in a system. We believe that detecting well known algorithms in the code base can be helpful to gain knowledge about, which concerns are present in the code base, how they are solved and which components are involved. Our envisioned solution consists of a Domain Specific Language (DSL) designed to describe key features of an algorithm, a search algorithm to find these features and a set of “ready to use” descriptions for common algorithms.

2 Introduction

One of the first steps for both software-reengineering and -evolution is to develop an understanding of the software system. Unfortunately, this is often a tedious and time-consuming process. The documentation is frequently outdated or non-existent, colleagues and experts are pressed for time or have even left the company [1] [2].

Many different tools and approaches have been developed to support the process of software understanding and maintenance. Some examples include the generation of diagrams from source code, the detection of design patterns, or the clustering of components with the aim of understanding their interactions and improving the software architecture [1] [2] [3]. To the best of our knowledge, no approach exists which tries to detect which algorithms are implemented in a code base.

We are convinced that detecting well known algorithms during the early stages of software-reengineering is helpful in gaining knowledge about which concerns are present in the code base, how they are solved, and which components are involved. As these are core questions during the process of code understanding and therefore also software maintenance, such an approach would be a valuable addition to the state of the art.

One example for an algorithm whose recognition could be helpful for the understanding and system analysis is the “Raft” algorithm, which is a fault-tolerant consensus algorithm for distributed systems [4]. Detecting an implementation of Raft reveals that the system in question is distributed, that fault-tolerance

is an essential requirement, and how the consensus is implemented. Additionally, the user can analyse the reported code locations to identify which data is replicated in the system and, therefore, especially important. The reported code locations also provide information about which components implement the different elements of the algorithm. Moreover, the approach could report which of the optional features of Raft, such as log compaction, pre-voting, or membership changes, are implemented. A user acquainted with the Raft algorithm will also know about the provided guarantees like log matching or leader completeness, which are important to understand but not obvious from simply reading the code. Lastly, making the user aware of the implemented algorithm enables her or him to consult official documentation about Raft, which is probably better suited for gathering an overview of the system than trying to painstakingly reverse-engineer legacy code.

3 Related Work

Textual code search tools like OpenGrok [5] or Sourcegraph [6] offer full-text search in code bases with limited code understanding, e.g. differentiating between the definition and usage of functions. However, such tools focus on finding exact matches and do not offer a flexible way of defining key aspects of an algorithm that can abstract from implementation specific details. Furthermore, searching for different parts of an algorithm implementation by using multiple different query snippets and aggregating results related to one algorithm is not supported by such tools.

Code clones are source code fragments which appear multiple times in a code base, typically due to copy and paste. Approaches for the detection of code clones search for similar code and often allow for some variance between the clones. Many approaches use an intermediate form such as Abstract Syntax Trees (ASTs) or Program Dependency Graphs (PDGs) for code representation. The search is then carried out by extracting and comparing features or graph matching [7]. Unfortunately, since the goal of these tools is to support refactoring, the focus is often on clones which are as similar as possible. Additionally, code clone detection tools offer no possibility for the user to specify what to search for.

Design patterns are generalised and well-proven solutions to recurring problems in software design.

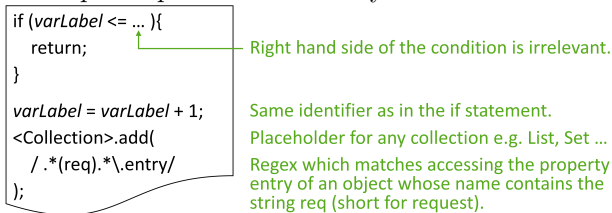
Approaches for the (semi-)automatic identification of patterns in a code base often share our goal of assisting the user with the understanding and re-engineering of software systems. Unfortunately, the formalisation of these patterns and the search-algorithm typically rely heavily on object-oriented concepts, which by themselves are not well suited for the description of algorithms.

4 Proposed Solution

The problem of detecting well known algorithms in a code base can be divided into three different sub-problems, for which we envision the following solutions:

1. Defining the key features of an algorithm in an abstract way: We want to recognise different algorithm implementations created by various developers. It is therefore necessary to define a way of capturing the key characteristics of an algorithm in an abstract form. This representation must be specific enough to eliminate as many false positives as possible but at the same time general enough to describe any algorithm and enable the abstraction of some implementation specific details. A user might for example, be interested in all the implementations of the "Quicksort" algorithm regardless of how the pivot element is computed exactly. Although abstracting from implementation-specific details allows for false positives, we nevertheless believe that it is reasonable to tolerate some false positives in order to detect algorithms whose implementation details differ.

We currently envision a DSL which makes it possible to describe an algorithm with multiple independent short snippets. These snippets should allow it to mix key characteristics of the algorithm like code structure definition, keywords, and other features. While the snippets themselves are independent from one another it should be possible to hierarchically compose multiple snippets to describe (parts of) an algorithm. This idea is similar to the composition of pattern descriptions in Niere [2]. Incorporating advanced search features offered by textual code search tools like wildcards, regular expressions, and structural search, which specifically matches code structures, could be a good starting point for the description language. Which other features are a good fit for the description and the specification of algorithms is part of our future research. Figure 1 shows an example for such a snippet and explains points of variability.



```

if (varLabel <= ... ){
    return;
}

varLabel = varLabel + 1;
<Collection>.add(
    /.*(req).*\entry/
);

```

Right hand side of the condition is irrelevant.

Same identifier as in the if statement.

Placeholder for any collection e.g. List, Set ...

Regex which matches accessing the property entry of an object whose name contains the string req (short for request).

Figure 1: Example for a description snippet

2. Defining and implementing a search algorithm to find implementations of the described

algorithm in an unknown code base: Given a code base of interest and a set of algorithm descriptions, the search algorithm should look for the specified key characteristics of a description in the code. Potential matches for all the snippets of a description should be reported with their corresponding location in the code. Each detected snippet can be seen as a single piece of evidence for the existence of the algorithm it describes and should, therefore, increase an overall belief for the existence of the specified algorithm. This is necessary since the abstraction of implementation details enables us to detect different algorithm implementations at the expense of false positives. It is therefore desirable to indicate the confidence in the retrieved results to the user. Possible ways to handle the uncertainty of the results include fuzzy logic [8], which is also used by Niere [2], or bayesian statistics [9].

The design of the search algorithm depends on the specified description language. One possible solution could be graph matching between the AST of the program and the pattern described with the DSL, which is similar to the techniques used in code clone detection approaches [7]. Challenges include the selection of the best search approach, optimisation of the runtime, and finding a good trade-off between precision and recall.

3. Publishing a set of common algorithm specifications in the description language: In order to be immediately beneficial to a user, the descriptions for a set of well known algorithms should be provided. This should be seen as a starting point and demonstration of the capabilities of the approach.

5 Future Work

To understand which code analysis and search techniques are best suited for our use case, we are currently conducting a systematic literature study in the code search domain. Furthermore, we are working on an early AST-based prototype which supports an initial set of features from the description language and uses graph matching for the search.

References

- [1] Laser *et al.*, "ARCADE: An Extensible Workbench for Architecture Recovery, Change, and Decay Evaluation," in *ESEC/FSE 2020*, pp. 1546–1550.
- [2] J. Niere, "Inkrementelle Entwurfsmustererkennung," Ph.D. dissertation, University of Paderborn, Germany, 2004.
- [3] Schröder *et al.*, "Search-Based Software Re-Modularization: A Case Study at Adyen," in *ICSE-SEIP 2021*, pp. 81–90.
- [4] Ongaro *et al.*, "In Search of an Understandable Consensus Algorithm," in *USENIX ATC 2014*, ISBN: 978-1-931971-10-2.
- [5] *OpenGrok*, Oracle. [Online]. Available: <https://oracle.github.io/opengrok/> (visited on 03/20/2022).
- [6] *Sourcegraph*. [Online]. Available: <https://about.sourcegraph.com/> (visited on 03/20/2022).
- [7] Ain *et al.*, "A Systematic Review on Code Clone Detection," *IEEE Access*, vol. 7, pp. 86 121–86 144, 2019.
- [8] "Fuzzy sets as a basis for a theory of possibility," *Fuzzy Sets and Systems*, vol. 1, no. 1, 1978, ISSN: 0165-0114.
- [9] Gelman *et al.*, *Bayesian Data Analysis*. 2013.

Möglichkeiten Metriken-gestützter Reviews zur Quellcodebewertung

Andreas Schmietendorf, Sandro Hartenstein

Hochschule für Wirtschaft und Recht Berlin

andreas.schmietendorf@hwr-berlin.de, sandro.hartenstein@hwr-berlin.de

1. Motivation

Entsprechend ([Fowler 2018] S. 4) sollten Softwareänderungen nur mit gut strukturiertem bzw. wartbarem Quellcode einhergehen:

„When you have to add a feature to a program but the code is not structured in a convenient way, first refactor the program to make it easy to add the feature, then add the feature.“

Bei agilen Methoden zur Softwareentwicklung wird der kontinuierliche Einsatz inhärenter Review-Techniken unterstellt. Typische Ansätze finden sich mit den Techniken des „Pair Programmings“ oder auch dem „Refactoring“ - kontinuierliche Optimierung des Quellcodes bei gleichbleibender Funktionalität.

Dennoch existieren aus Sicht des Autors nach wie vor Bedürfnisse, klassische Review-Techniken unabhängig vom originären Entwicklungsteam zum Einsatz zu bringen. Der vorliegende Beitrag geht auf ausgewählte Aspekte einer sowohl im praktischen als auch akademischen Umfeld erprobten Vorgehensweise ein.

2. Einordnung

Bei Review-Techniken wird auch von statischen bzw. manuell auszuführenden Testverfahren gesprochen. Obwohl mit Reviews zumeist hohe Kosten verbunden sind, besitzen diese Alleinstellungsmerkmale. Insbesondere semantische Eigenschaften des Quellcodes lassen sich nur so erfassen. Ebenso bieten diese ein gutes Mittel zur struktur- und architekturorientierten Qualitätsbewertung, zur Codeoptimierung und zur Wissensverteilung.

Entsprechend dem „IEEE Standard for Software Reviews and Audits“ (vgl. [IEEE 2008]) werden folgende Reviewtypen unterschieden.

- Management Reviews (Managementführung)
- Technical Reviews (Projektleitungsführung)
- Inspections (externe Führung)
- Walk-through (Entwickler geführt)
- Audits

Der vorliegende Beitrag konzentriert sich auf technische Reviews. Diese werden u.a. benötigt, um die Möglichkeiten zur Wartung und Pflege existierender Software einschätzen zu können.

Bei den aktuell stark genutzten Scrum-Projekten unterscheidet [Herby 2022] zwischen den folgenden einsetzbaren Review-Techniken:

- Instant Code Review – im Zusammenhang mit der Technik des „Pair Programmings“,
- Synchronous Code Review – unmittelbares Prüfen durch erfahrenen Entwickler,
- Asynchronous Code Review – werkzeuggestütztes Vorgehen,
- Code Review once in a while – Team-Meeting orientierter Ansatz.

Gerade im Zusammenhang mit existierenden Altsystemen, aber auch bei extern entwickelten Systemen spielen vom originären Entwicklerteam losgelöste (asynchrone) Reviews eine wichtige Rolle. Folgende Review-Einsatzszenarien lassen sich identifizieren:

- Übernahme von Altanwendungen in die eigene Verantwortung (Wartung/Pflege).
- Identifikation durch Änderungen betroffener Komponenten (z.B. beim DBMS-Tausch).
- Qualitätsbewertung (-sicherung) extern gefertigter Software(-komponenten).

Grundsätzlich bedarf es für die aufgezeigten Einsatzszenarien klarer Zielstellungen und akzeptierter Bewertungsgrundlagen.

3. Konzeptionelles Vorgehen

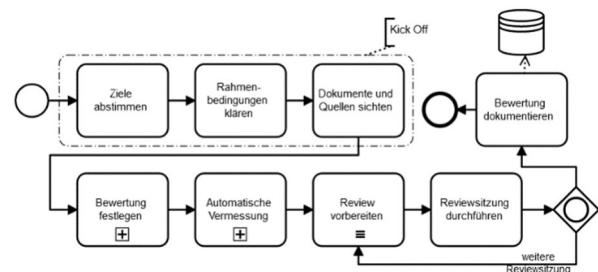


Abb. 1: angepasster Review-Prozess

Der Prozess zur Durchführung eines asynchronen Reviews entsprechend Abb. 1 beinhaltet die toolgestützte Vermessung der Software als Unterstützung der Vorbereitung und Durchführung des Reviews. Die Ergebnisse dieser Vermessung beziehen sich auf Bewertung eingehaltener Programmierkonventionen (ggf. Styleguides), Metriken im Diskurs eingesetzter Versionsmanagementlösungen (z.B. Commit-Häufigkeiten) und korrespondierender Quellcodemetriken wie z.B.:

- Umfangsmetriken zu Klasseigenschaften (Methoden, Attribute, ...),
- Zyklomatische Komplexitäten als Antipattern zur Identifikation unzureichender OO-Entwürfe,
- Kopplungsmaße bezüglich eingesetzter Subsysteme, Komponenten und Services,
- Grad der Kommentierung und Konsistenz im Gesamtsystem,
- Aufwandsschätzung mit Hilfe der Ermittelten Umfangsmetriken (u.a. Backfire).

Häufig ist die zyklische Erhebung dieser Bewertungen durch die Integration fertiger oder eigenentwickelter Messwerkzeuge (beides hat Vor- und Nachteile) in genutzte Entwicklungsumgebungen bzw. Versionsmanagementsysteme zu empfehlen. Bei reifen Softwareproduktionsumgebungen lassen sich diese Information darüber hinaus über Dash-Board-Darstellungen verdichten und zur Projektsteuerung verwenden.

4. Beispiel einer Analyse

Die Qualitätsbewertung von durch einen externen Dienstleister entwickelten Komponenten ist ein häufig benötigter Anwendungsfall für ein Review. Auf dieser Grundlage soll u.a. die ggf. eigenständige Weiterentwicklung sichergestellt werden. Das Beispiel bezieht sich auf eine Java-basierte Wrapper-Applikation, die auf einem bestehenden Altsystem (Legacy Application) aufsetzt. Aufgabe der Erweiterung (vgl. Abb. 2) war es, den Verkauf eines telekommunikationstechnischen Produkts auf einem existierenden Order-Management-System abzubilden und eine serviceorientierte Schnittstelle (WebAPIs) für potentielle Front-End-Applikationen (u.a. Shop-System) zur Verfügung zu stellen.

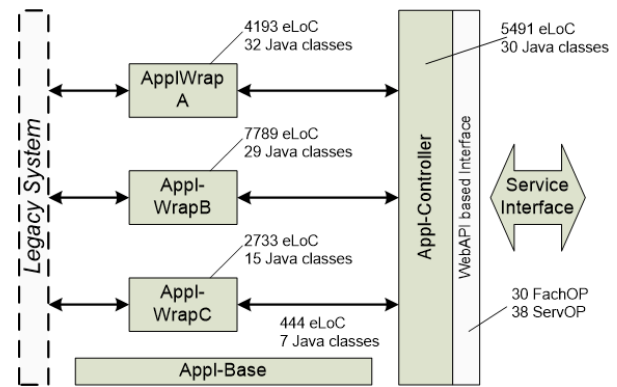


Abb. 2: Komponenten der Wrapper Applikation

	Anzahl Klassen	Public Methods	Protected Methods	Private Methods	Public Attributes	Protected Attributes	Private Attributes	eLoC	Commend LoC
Appl-Base									
mspi	7	21	25	1	9	8	4	444	295
Appl-WrapA									
bsi	9	75	23	45	52	0	18	2256	808
bsi/distrib	2	24	0	1	0	1	6	75	81
bsi/info	3	25	0	0	0	0	1	432	81
bsi/manage	5	85	0	12	0	1	17	526	20
bsi/receiver	4	23	1	19	5	0	3	534	103
bsi/types	9	144	0	4	0	0	65	370	0
Appl-Controller									
controller	7	95	17	1	81	0	50	4078	900
framework	2	6	2	1	0	0	2	22	42
framework/handler	6	18	1	9	0	2	14	240	189
framework/imbean	1	28	0	1	0	0	22	122	113
framework/mgmt	5	8	0	7	0	0	15	179	87
ep	6	42	3	18	56	1	14	727	156
ep/manage	3	13	0	4	0	1	2	123	10
Appl-WrapB									
buad	2	5	4	0	5	0	0	48	50
buad/manage	6	88	3	4	5	1	38	2169	652
buad/parts	7	56	1	2	5	26	11	955	557
mnp	3	5	19	0	99	3	0	490	160
mnp/info	3	49	0	0	0	0	0	1388	209
mnp/manage	3	109	2	3	0	1	25	534	4
mnp/receiver	1	6	0	0	0	0	0	836	78
mnp/sender	4	32	0	0	0	0	0	1369	186
Appl-WrapC									
sim	5	64	2	5	45	0	0	1861	333
sim/mp	5	28	2	8	0	0	10	546	141
sim/manage	5	55	0	4	2	5	7	326	138
Total	113	1104	105	149	359	50	324	20650	5393

Abb. 3: Umfangsmetriken Gesamtsystem

Abb. 3 und 4 zeigen die Aufnahme klassischer Umfangs- und Komplexitätsmetriken.

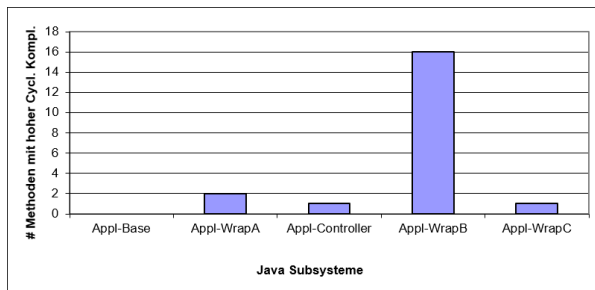


Abb. 4: Methoden mit hoher Komplexität

Hier nicht dargestellt sind Analysen zum Gebrauch von „Cut & Paste“, Abhängigkeiten bzw. Kopplungsmaße zwischen Komponenten des Systems oder auch abgeleitete Umfangsmaße (Function Points) zur Plausibilisierung der verbrauchten Ressourcen. Bei den eingesetzten Messwerkzeugen handelte es sich um am Markt verfügbare Lösungen (u.a. Resource Standard Metrics¹) bzw. Eclipse PlugIns (u.a. CodeMR²). Für spezielle Messungen macht ggf. auch die eigenständige Entwicklung entsprechender (Parser-) Ansätze Sinn. Mit Hilfe der Vermessung konnte eine schnelle (modellbasierte) Erfassung der Implementierungsdetails unterstützt werden. Auf der Grundlage von überschrittenen Schwellwerten bzw. durch die Messwerkzeuge ausgegebenen

Warnungen war es möglich, Probleme zu identifizieren und für das Review zu priorisieren. Konkret bezogen sich diese auf semantische und logische Fehler, unzureichend generische Lösungsansätze, schwer lesbaren Quellcode, ungenügende Kommentierungen oder auch Verstöße gegen Programmierkonventionen.

5. Zusammenfassung

Werkzeuggestützte Code Reviews bieten eine gute Möglichkeit zur strukturorientierten Bewertung der Wartbarkeit von Software. Keinesfalls dürfen die originären Vermessungen unkommentiert übernommen und verabsolutiert werden. Diese dienen ausschließlich der Priorisierung zu diskutierender Problembereiche innerhalb der Reviewsitzung. Für die Review-Bewertung können auch Metriken von Versionsmanagement-Lösungen wie GitHub (REST API für Metriken³ bzw. die GitHub GraphQL API⁴) herangezogen werden.

6. Quellenverzeichnis

- [Fowler 2018] Fowler, M.: Refactoring - Improving the Design of Existing Code, Addison Wesley; 2nd edition, 2018
- [Herby 2022] 4 Types Of Code Reviews Any Developer Should Know, <https://www.scrum-tips.com/article/types-of-code-reviews/>, Upload: 19.04.2022
- [IEEE 2008] IEEE 1028-2008 - IEEE Standard for Software Reviews and Audits

¹ <https://msquaredtechnologies.com/>

² <https://www.codemr.co.uk/>

³ <https://docs.github.com/en/rest/reference/metrics>

⁴ <https://docs.github.com/en/graphql>

Mit Feature-Modellen das Komplexitätsmanagement vereinfachen

Vasil L. Tenev Martin Becker
Fraunhofer IESE, Kaiserslautern
{vasil.tenev, martin.becker}@iese.fraunhofer.de

Zusammenfassung

In dieser Arbeit stellen wir ein aktuelles Projekt mit einem Unternehmen vor, in dem es darum geht, in Produktkonfigurationen kodiertes Konfigurationswissen zu extrahieren und dies maschinen-verarbeitbar zu machen. Auf dieser Grundlage sollen Entscheidungen über die Optimierung der Modularität einzelner Produkte getroffen werden. Bei den Produkten geht es darum, bei mehr als 60.000 Konfigurationsparametern einen Überblick zu schaffen. Dabei wird ein Feature-Modell aus den Daten heraus entwickelt, dass alles miteinander vernetzt – ein Managementsystem für die gesamte Abwicklung.

1 Einführung

Mehr als 100 Jahre nachdem Henry Ford die Verkleinerung der gesamten Produktfamilie auf dem *Model T* ankündigte [1] ist der Druck zur Massen Anpassung und zur Erreichung von kürzeren Produktfreigabezyklen zunehmend hoch. Die Massenproduktion von Waren, die auf die individuellen Bedürfnisse der Kunden zugeschnitten sind, ist schon im Konfigurator eines jeden deutschen Automobilherstellers Realität. In Zukunft werden die Möglichkeiten durch die 3D-Drucktechnologien für die Massenproduktion noch deutlich erweitert [2].

Die ständig wachsende Komplexität insbesondere softwareintensiver Systeme ist vor allen in dem Konfigurationsswissen zu orten [3]. Dieses Wissen liegt oft in den Köpfen einiger weniger Experten. Dadurch wird der Transfer von Know-how immer aufwendiger und kostspieliger. Sowohl die Ausbildung von neuen Fachexperten, als auch einzelne Schritte im Konfigurationsprozess müssen in solchen Anwendungsfällen computerunterstützt werden.

Ziel ist es, die Werkzeuge sowohl für die Speicherung als auch für die Verarbeitung von Daten zu modernisieren und umzugestalten. Im Ergebnis soll Konfigurationswissen extrahiert und für den Menschen nachvollziehbar gemacht werden. Der Umgang damit soll nicht nur automatisiert, sondern auch bei der Entscheidungsfindung computergestützt sein. Der Aufbau eines Feature-Modells ist einer der ersten Schritte einer solchen Systemmodernisierung. In diesem Erfahrungsbericht stellen wir einen automatisierten Ansatz anhand eines realen Firmenbeispiels vor.

2 Firmenbeispiel

Dieser Beitrag basiert auf den Erfahrungen in dem Marktsegment des Kraftstoff-Einzelhandels. Es handelt sich hier um einen Hersteller von Tankstellen-Management-Systemen (TMS). Wie bei vielen anderen Hersteller software-intensiver Systeme ist das Wissen über Produktmerkmale und deren Abhängigkeiten verstreut und versteckt. Einerseits befinden sich relevante Daten in Realisierungsartefakten, aber andererseits auch in den persönlichen Notizen und in den Köpfen der Experten.

Das implizite Konfigurationswissen stellt diesen Hersteller vor mehreren Herausforderungen. Diese lassen sich in zwei Gruppen einordnen:

Management

- Stetig wachsendes Produktportfolio und Verlängerungen des Time-to-Market.
- Konfigurationswissen und Know-how in den Köpfen einiger weniger Experten.
- Ausbildung und Wissenstransfer angehender Experten dauert 3 – 5 Jahre.

Konfigurationsraum

- Exponentiell viele mögliche Varianten.
- Kundenindividuelle und länderspezifische TMS.
- Customizing von Kundenreleases dauert bis zu 3 Monate.

In den letzten drei Jahren wurden die TMS für 49 verschiedene Kunden in Westeuropa bereitgestellt. Zuerst werden kundenspezifische Releases erstellt. In dem betrachteten Zeitraum waren das in Summe 97 kundenspezifische Releases, die danach schrittweise an über 10.000 unterschiedliche Objekte ausgerollt wurden.

Umfang und Komplexität des Konfigurationswissens lassen sich an der umfassenden Datenmenge erahnen. An jedem Objekt befinden sich 137 Datenbank-Tabellen, die kundenspezifische Einstellungen und Daten enthalten. Für jeden kundenspezifischen Release werden 48.774 Parameter eingestellt, die in 3.363 Konfigurationsdateien verteilt sind. Der daraus resultierende Konfigurationsraum impliziert über 10^{1000} mögliche Varianten, die für jeden neuen Release in Frage kommen können.

Im folgenden Kapitel gehen wir diese Herausforderungen an, mit dem leitenden Ziel, das versteckte Konfigurationswissen mittels automatisierten Datenanalysen explizit, verständlich und nutzbar zu machen.

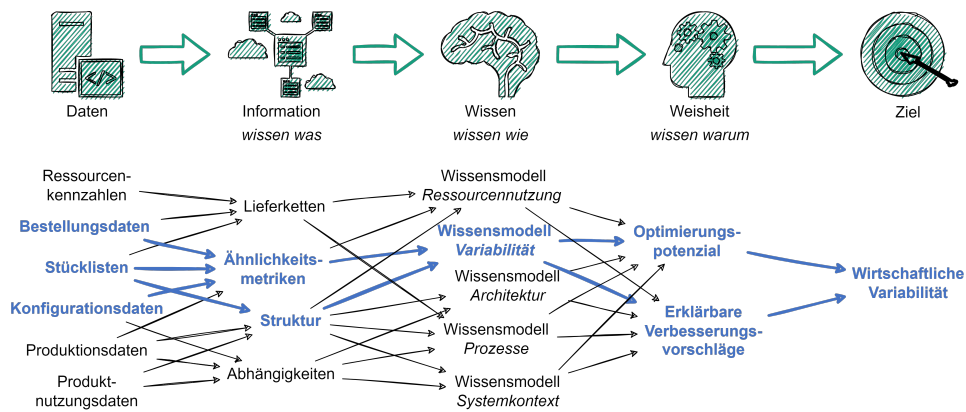


Abbildung 1: Analyse-Ansatz (blau hervorgehoben) im Gesamtkontext von VARIOUS (basierend auf der Wissenspyramide [4, 5])

3 Analyse-Ansatz

Der Ansatz nutzt das Konzept des Feature-Modells [6], um das Management von komplexem Konfigurationswissen einfacher zu machen. Das Vorgehen ist in vier nacheinander folgenden Aktivitäten aufgeteilt und ist an der Wissenspyramide (s. Abbildung 1) angelehnt:

Extrahieren aus Daten. Aus den zur Verfügung stehenden Artefakten (s. Abschnitt 2) werden als erstes relevante Merkmale und Konfigurationsparameter identifiziert und extrahiert. Dabei werden zusätzlich Referenzen zwischen den Datenpunkten erhoben.

Analysieren der Information. Danach werden aus den jeweiligen Releases kundenspezifische Feature-Modelle aufgebaut und computergestützt vorstrukturiert. Parallel dazu wird die Ähnlichkeit zwischen den atomaren Elementen (d. h. Features) analysiert.

Strukturieren des Wissens. Basierend auf den Informationen über Ähnlichkeit und Struktur werden die kundenspezifischen Feature-Modelle zu einem holistischen Feature-Modell zusammengeführt. Somit kann das Konfigurationswissen hierarchisch aufgebaut werden. Als Teil eines übergeordneten Variabilitätsmodells wird dieses Wissen für weitere Zwecke verwendet: Darstellung und Kommunikation von Produktmerkmalen und Abhängigkeiten; Unterstützung angehender Experten; Identifizierung des Einflusses von Features; und Aufhebung von nicht-maschinenlesbarem Expertenwissen.

Optimieren durch Weisheit. Das gesammelte Wissen ermöglicht die Identifikation von Optimierungspotential in diesem Schritt und liefert die Basis für erklärable und faktengestützte Verbesserungsvorschläge. Die Beherrschung der Komplexität bedeutet somit die Fähigkeit, die Wirtschaftlichkeit von Produktvarianten fundiert zu bewerten und Konfigurationsprozesse zu automatisieren.

Dieser Ansatz ist Teil von Fraunhofer VARIOUS –

ein Tool-Framework für die datenzentrierte Integration bestehender und neuer Analysewerkzeuge. Ziel dieses Frameworks ist es, einen integrierten modellbasierten Ansatz (s. Abbildung 1) aufzubauen, um typische analytische Fragen im Kontext großer und sich schnell entwickelnder Produktlinien zu adressieren. Die daraus resultierenden Erkenntnisse und Werkzeuge helfen bei der Abstimmung zwischen den Geschäftszielen, Marketing- und Entwicklungsstrategien.

4 Fazit

In diesem Beitrag wurde ein Ansatz zur niedrighen datengetriebenen Entscheidungsunterstützung bei der Identifikation von unwirtschaftlicher Variabilität im Produktportfolio vorgestellt. Der Ansatz nutzt das Konzept der Feature-Modelle und füllt die Lücke zwischen Daten und Wissen mit dem integrierten Wissensmodell von Fraunhofer VARIOUS. Mit diesem Ansatz kann das Komplexitätsmanagement bei Herstellern variantenreicher Produktfamilien vereinfacht und automatisiert werden.

Literatur

- [1] H. Ford and S. Crowther, *My Life and Work*. Auckland, New Zealand: Floating Press, The, 1980.
- [2] Volkswagen Newsroom, “3D printing in action,” 02.12.2021.
- [3] D. Faust and C. Verhoef, “Software product line migration and deployment,” *Softw. Pract. Exper.*, vol. 33, no. 10, pp. 933–955, 2003.
- [4] J. Rowley, “The wisdom hierarchy: representations of the DIKW hierarchy,” *Journal of Information Science*, vol. 33, no. 2, pp. 163–180, 2007.
- [5] M. Zeleny, “Management support systems: Towards integrated knowledge management,” vol. 7, pp. 59–70, 1987.
- [6] D. Batory, “Feature Models, Grammars, and Propositional Formulas,” in *Software product lines* (H. Obbink and K. Pohl, eds.), vol. 3714 of *Lecture notes in computer science*, 0302-9743, pp. 7–20, Berlin and London: Springer, 2005.

Managing Software Complexity in Automotive Software Development

Jochen Quante

Robert Bosch GmbH, Corporate Research
Renningen, Germany

jochen.quante@de.bosch.com

Thomas Grundler

Robert Bosch GmbH, Powertrain Solutions
Schwieberdingen, Germany

thomas.grundler@de.bosch.com

Abstract

In the last decades, software has become more and more important in the automotive domain. With features like autonomous driving and increasing connectivity, the software's sheer volume has increased by an order of magnitude. This ever-growing complexity has to be accompanied by processes that limit its negative effect on maintainability. Also, the prospect of "end of combustion" demands reduction of development effort for combustion engine control software. In this paper, we report on our approach to control software complexity in the powertrain domain. We describe the basic idea for measuring and managing maintainability, the challenges on adopting such an approach in practice, like having to measure on different kinds of artifacts, and the factors that have lead to success.

1 Maintainability Index

In order to control maintainability, we need to measure it. Early approaches introduced threshold values for individual metrics, which highlight certain aspects, but fail to give an overall picture about maintainability [1]. The idea to combine several software product metrics to build a maintainability index (MI) prediction model was proposed by Oman and Hagemeister in 1994 [3]. The core idea is to find a weighted sum of metrics that best approximates expert opinion on maintainability using principal components analysis and regression analysis. We adapted this approach for automotive software development [6]. Meanwhile, it is an established way in our organization to identify problematic software modules and track their improvement through reengineering. We elaborate on the integration of the MI into our processes in Section 3.

Our experience shows that the main contributors to poor maintainability of C code are size (e.g., LOC), control complexity (e.g., cyclomatic complexity or number of paths), and interface size (e.g., number of incoming or outgoing data flows).

2 Model-based Development

Model-based development is in widespread use for developing automotive control applications. Simulink[®]¹ or ASCET² models often are the main artifacts that developers work on. From these models, code (usually C code) is generated for the target device – a step in which structures from the model are mostly lost. This induces a problem for any approach that is based on code metrics: Common measurement tools work on C code and not on the original models. However, if maintainability is to be assessed, the measurement should be performed on the model and not on the generated code. Usually, the model is much easier to understand and has more structure than the generated code. Metrics on generated code thus give a wrong impression about the complexity that the developer actually faces.

This made us investigate model metrics [2, 5] and use those to create an additional MI for models. However, even within Simulink[®] models, there are different kinds of models supported (e.g., block diagrams and state machines), and there may even be MATLAB[®] code that is generated by other model-based tools. Therefore, each modeling tool or meta-model that is used has to be considered in this approach and incorporated into the overall MI formula, along with the metrics for classical C code.

For models, application of the MI approach reveals that the main contributors to (bad) maintainability are about the same as for code – they just have to be interpreted for the model. For example, in case of a block diagram, the size is measured in terms of number of blocks.

3 Processes

When a new engine control project starts, the software is usually not built from scratch, but is based on existing code. The project team then first measures the present software's maintainability and starts a discussion with experts on the findings. This results in a set of modules that have bad maintainability according to the experts' opinion and for which frequent changes

¹<https://www.mathworks.com/products/simulink.html>

²<https://www.etas.com/en/products/ascet-developer.php>

are expected in the future. For these modules, additional actions are enforced whenever the next code change is due. This can result in refactoring, but it can also mean that additional verification measures are performed, depending on what is most adequate. Modules that are proven in use and not expected to change are kept as they are. The decision for each module has to be well-founded and documented.

4 Success Factors

Any metrics-driven approach faces the problem that it will change behavior, but not necessarily in the intended direction. Since it is not possible to measure every relevant aspect, the measured aspects often end up over-optimized while the unmeasured aspects become worse. Restructuring should never have the goal to just improve the MI metric value, but to really improve the software's quality. The metric should then reflect the improved maintainability. To account for this, the metric is just an indicator for us: The ultimate decision is with the experts. However, a bad MI value enforces a *discussion* about the maintainability of a given module, which also has to be documented – and not an immediate restructuring activity. This way, the MI is perceived as being helpful to improve software quality.

Although the MI formula should be regarded as a black box, its ingredients can give some hint about what the main contributors to bad maintainability are (as discussed above). However, this may change when the MI is recalibrated at some point in the future. The MI always reflects the current maintainability issues. This should be motivation to improve code structures and not the (current) MI metric.

The ease of use is also crucial for success. In our case, the relevant metrics including the MI are automatically calculated along with other static code checks and provided to the developers. The concentration on just a few metrics also adds to the acceptance of this approach. Furthermore, we have established a support team for developers. This team supports in maintainability assessments and in concrete restructuring activities. It turns out that in the embedded domain, people are often experts in their technical domain, but not necessarily software structuring experts. Therefore, this support is highly appreciated and helps to further improve maintainability.

5 Conclusion

With the right mindset and concentration on a few relevant aspects, metrics are a useful and important tool for software development. As our case shows, the MI can be a means to foster an open and objective discussion about maintainability. This paves the road towards better maintainable and less complex modules. Metrics can also create higher transparency about the internal quality of software – even for management. For example, based on our measurements, we could

show that software ageing [4] (i.e., continuously increasing complexity) is a common effect also in our software. It is hard to accept for managers that software ages although it doesn't really change. However, it has to be adapted to a changing environment from time to time, which causes the ageing. We can also confirm that restructured modules are a very good basis for future development that pays off quickly. This is based on experiences from a large-scale reengineering project on several hundred modules.

Overall, the introduction of a maintainability index and its inclusion into processes enables us to continuously improve the quality of our software and prevent it from degrading unnoticed.

References

- [1] J. Heidrich, A. Morgenstern, J. Quante, and T. Grundler. New framework for measurement-based evaluation of quality in automotive software development. *ATZelectronics worldwide*, 17:8–13, Jan 2022.
- [2] M. Olszewska, Y. Dajsuren, H. Altinger, A. Serebrenik, M. Waldén, and M. G. J. van den Brand. Tailoring complexity metrics for Simulink models. In *Proc. of 10th European Conf. on Software Architecture – Workshops*, pages 5:1–5:7, 2016.
- [3] P. Oman and J. Hagemester. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251 – 266, 1994.
- [4] D. L. Parnas. Software aging. In *Proc. of 16th Int'l Conf. on Software Engineering*, pages 279–287, 1994.
- [5] J. Quante. Approximating the number of execution paths in Simulink models. *Softwaretechnik-Trends*, 40(2), 2020.
- [6] J. Quante, T. Grundler, and A. Thums. Maintainability index revisited: Adaption and evaluation for Bosch automotive software. In *Proc. 3rd SQMB Workshop*, pages 67–70, 2010. <https://mediatum.ub.tum.de/doc/1094249/>.

Towards a Software Reengineering Body of Knowledge (SREBOK)

Marco Konersmann
University of Koblenz-Landau
Koblenz, Germany
konersmann@uni-koblenz.de

Jens Borchers
Borchers BfI
Hamburg, Germany
jens@borchers-bfi.de

Leif Bonorden
Universität Hamburg
Hamburg, Germany
leif.bonorden@uni-hamburg.de

Andres Koch
Object Engineering GmbH
Uitikon-Waldeg, Switzerland
akoch@objeng.ch

Sandro Schulze
Otto von Guericke University Magdeburg
Magdeburg, Germany
sandro.schulze@iti.cs.uni-magdeburg.de

Abstract

The special interest group software reengineering (FG-SRE) of the German Informatics Society (GI e.V.) meets to discuss and move forward the state-of-practice and research of software reengineering in the German-speaking countries since 1999. In 2021 we started an initiative to collect and share the knowledge about the state-of-the-art in software reengineering to help practitioners and researchers to get an understanding of the field. In this paper and the associated talk, we present the progress and expected results of this initiative.

Introduction Software reengineering (SRE) is concerned with “the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form” [2]. The field of SRE is very broad, incorporating technical and organizational aspects, which stretch through all phases of software development. This broadness makes it difficult to get an overview of the field, e.g., for practitioners to learn about SRE and apply it to software systems and for researchers to identify valuable points for research.

To tackle this problem, on the 22nd Workshop on Software Reengineering and Evolution 2021, the special interest group software reengineering (FG-SRE¹) of the German Informatics Society (GI e.V.) started an initiative to collect their knowledge and experience and share it with the public. The initiative’s goal is to describe the state-of-the-art of SRE as an entry point for interested software engineers to learn about SRE and as a reference to existing further literature. In this contribution and the associated talk, we present the current state & planned activities for achieving this goal and how the audience, members of our special interest group and further experts in the field can contribute to the project.

¹<https://fg-sre.gi.de>

Current State of the Initiative We decided to create a citable open access publication that collects and describes the relevant aspects of SRE. The “Software Reengineering Body of Knowledge” (SREBOK) is intended to describe the aspects deeply enough to serve as a self-contained overview and reference document while referencing further reading for details comparable to the Software Engineering Body of Knowledge [1]. The publication will present definitions of SRE alongside reasons for and goals of SRE. It will present challenges, risks and chances of SRE. Major chapters will describe foundational practices, processes and technical and organizational aspects. It will also set the context of the current state-of-the-art by commenting on the history and the envisioned future of SRE. Besides SRE, it will also describe means to *design for future*, i.e. how to design software products and development processes in a way that they are beneficial with respect to maintenance, evolution and SRE in the future.

Currently, the initiative has defined the intended format, content and structure of the publication. In a first review phase, the current results were reviewed by further experts in the field to ensure high quality and completeness.

Planned Activities In the Workshop on Software Reengineering and Evolution we will present the current state of the document and invite the workshop participants to show their interest in contributing to the document, e.g., by co-authoring a chapter or section. As the next steps, we plan to:

1. identify potential co-authors for chapters and sections of the SREBOK in the workshop,
2. sketch the chapters’ contents in detail, together with all (potentially new) co-authors, to create an overview of the whole document,
3. contact potential publishers in parallel,
4. iteratively write the document as a group, and

5. publish when ready.

Final Remarks We are currently actively searching for co-authors for the SREBOK. We call every interested party to join our effort to share our knowledge. If you are interested in joining, please address Marco Konersmann via konersmann@uni-koblenz.de.

References

- [1] Pierre Bourque. *SWEBOK : Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, Los Alamitos, CA, 2014.
- [2] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

Hypothesen zu API-basierten Entwickler-Profilen

Hakan Aksu and Ralf Lämmel

The Software Languages Team, Universität Koblenz-Landau
{hakanaksu, laemmel}@uni-koblenz.de

Abstract Wir analysieren Java-basierte Softwareprojekte aus Github. Das Ziel der Analyse ist es, API-basierte Entwickler-Profile zu erstellen, um beispielsweise Bug assignment zu betreiben oder Projektmanagement-Entscheidungen zu treffen. In diesem Artikel werden mögliche Hypothesen vorgestellt, die anhand der Analyse evaluiert werden können.

1 Einleitung

In dieser Arbeit analysieren wir verschiedene Java-basierte Softwareprojekte aus Github. Die Projekte erfüllen dabei bestimmte Kriterien, wie eine Mindestbeliebtheit und Mindestgröße. Die verwendeten Bibliotheken werden vorzugsweise über Projektmanagement-Tools wie Maven oder Gradle ermittelt. Aus den ausgewählten Projekten und mithilfe der Bibliotheken werden für jeden Entwickler über die gesamte Commit-Historie mithilfe des Java-Parsers die verwendeten API-Elemente ermittelt. Eine Nutzung der API-Elemente kann z.B. ein Methodenaufruf oder eine Klasseninstanziierung aus einer bestimmten Bibliothek sein. Als API selbst bezeichnen wir alle API-Elemente aus einer bestimmten Paketstruktur. Dabei verwenden wir die ersten beiden Prefixe der Java-Paketstruktur aus der Bibliothek, um die API zu definieren. Beispielsweise werden alle API-Elemente aus dem Paket (und Unterpaketen) 'java.awt' der AWT-API zugeordnet. Einen verwandten Ansatz zur API-Analyse findet man z.B. in [1] oder [2]. Wir nutzen die API-Analyse als Grundlage für ein Entwickler-Profil. Das Entwickler-Profil kann als ein zusätzliches Kriterium bei der Zuweisung von Fehlern dienen und würde die Arbeiten in [3] erweitern.

Bereits bei der Datenanalyse behandeln wir diverse, meist Github-bedingte Besonderheiten. Beispielsweise wird Identitätsmanagement betrieben, da ein Entwickler die Möglichkeit hat, verschiedene Namen und E-Mails bei den Commits zu verwenden [4].

Als Resultat der Datenextraktion erhalten wir die Anzahl der verwendeten API-Elemente jedes Entwicklers über die gesamte Commit-Historie. In dieser erweiterten Kurzfassung gehen wir auf die Möglichkeiten ein, um aufbauend auf dieser Datenanalyse passende Hypothesen zu bilden und diese zu evaluieren.

2 Hypothesen

Eine strukturierte und systematische Hypothesenbildung und -auswertung ist in der zugrunde liegenden Dissertation noch in Bearbeitung. Manche der Hypothesen zeigen die Existenz eines API-Profils und andere Hypothesen weisen auf spezifische Problematiken hin. In diesem Abschnitt stellen wir nur einige ausgewählte Hypothesen vor.

Trivial-Hypothese: Je größer die Projektbeteiligung, desto größer ist die allgemeine API-Nutzung. Die Projektbeteiligung wird anhand der Lines of Code (loc) bestimmt. Die allgemeine API-Nutzung beschreibt die Nutzung von API-Elementen aus allen möglichen APIs. Wir erwarten eine Korrelation zwischen dem geschriebenen Code und den generell verwendeten API-Elementen. Um die Relation in typischen Repositories zu zeigen, analysieren wir eine Stichprobe von Repositories aus Github und berechnen jeweils die Korrelation zwischen der Projektbeteiligung und der API-Nutzung jedes Entwicklers.

API-Diversität-Hypothese: Die Entwickler unterscheiden sich in der relativen API-Nutzung. Die relative API-Nutzung eines Entwicklers ist das Verhältnis der jeweiligen API-Nutzung zu seiner gesamten API-Nutzung. Wir nehmen dafür API-weise die relative API-Nutzung aller Entwickler und testen diese Verteilung auf verschiedene Formen der Beta/Dirichlet-Verteilung. Die Verteilung gibt uns Auskunft darüber, wie divers/uniform die API-Nutzung der Entwickler ist.

Ähnlichkeit-Hypothese: Entwickler können zueinander ähnlich bezüglich ihrer API-Nutzung sein. In Softwareprojekten werden häufig Ähnlichkeitsfunktionen zwischen Entwicklern verwendet. Wir definieren eine neue Ähnlichkeitsfunktion basierend auf API-Nutzung. Wir vergleichen eine existierende Ähnlichkeitsfunktion beispielsweise die Ähnlichkeit an bearbeiteten Dateien und Ordnern mit unserer Ähnlichkeitsfunktion. Wir suchen dabei nach (Un-)Regelmäßigkeiten über mehrere Projekte.

Prediction-Hypothese: Die nahe Zukunft der API-Nutzung kann vorausgesagt werden. Unsere Annahme ist dabei, dass ein Entwickler unter bestimmten Voraussetzungen über die Zeit zu sich selbst ähnlich ist und wir dadurch die nahe Zukunft voraussagen können. Dies evaluieren wir mithilfe von Ähnlichkeitsmaßen oder Voraussagemodellen.

3 Diskussion von Evaluationsmöglichkeiten

In diesem Kapitel nehmen wir uns die Ähnlichkeit-Hypothese und die Prediction-Hypothese vor und diskutieren Evaluationsmöglichkeiten und zu beachtende Kriterien.

Als Datenbasis haben wir für jeden Entwickler die API-Nutzung aus jedem seiner Commits, welche wir als Vektor interpretieren können. Ein einzelner Commit und die darin enthaltenen API-Nutzungen sind für einen Entwickler meist nicht repräsentativ. Daher werden wir die Daten mehrerer Commits aggregieren. Beispielsweise können wir die Daten nach einer festen Commit-Anzahl oder einem festgelegten Zeitintervall zusammenführen. Die entstehenden Vektoren lassen sich dann auf Ähnlichkeit überprüfen.

Die Abbildung 1 enthält die Kosinusähnlichkeit der Top-8-Entwickler aus dem Projekt Elasticsearch basierend auf der gesamten API-Nutzung jedes einzelnen Entwicklers. Die blaue Säule zeigt jeweils die Ähnlichkeit zu sich selbst (perfekte Ähnlichkeit = 1). Die grauen Säulen zeigen jeweils die Ähnlichkeitswerte zu den anderen Entwicklern. Beispielsweise hat der dritte Entwickler ('Jason') den größten Ähnlichkeitswert mit dem fünften Entwickler ('Martijn') und der erste Entwickler ('Adrien') den kleinsten Ähnlichkeitswert zum zweiten Entwickler ('Armin'). Verwendet wurden hierbei die aggregierten Vektoren der gesamten Commit-Historie. Diese Berechnung ist jedoch erst einmal eine naive Anfangsbetrachtung. Insbesondere wurden hier noch keine möglichen Einfluss- und Störfaktoren betrachtet. Kann man beispielsweise Entwickler, die in unterschiedlichen Zeitintervallen gearbeitet haben, miteinander vergleichen? Inwieweit haben sich die APIs innerhalb des Projektes über die Zeit verändert? Mussten alle Entwickler diversifiziert arbeiten oder hatten sie auch die Möglichkeit sich zu spezialisieren? Das sind nur einige Beispielfragen, die beim Vergleich von Entwicklern geklärt werden müssen. Wir werden entsprechend verschiedene Aggregations- und Filtermöglichkeiten anwenden, die Ursachen für Einflüsse und Störungen suchen und diverse Ähnlichkeitsmaße miteinander vergleichen. Zusätzlich werden wir untersuchen, ob die API-Nutzungsähnlichkeit von anderen Ähnlichkeiten begleitet wird. Eine These könnte lauten, dass Entwickler mit einer ähnlichen API-Nutzung auch in ähnlichen Lokalisationen gearbeitet haben.

Wir schauen uns auch die Ähnlichkeit des Entwicklers über die Zeit zu sich selbst an, was uns zur Prediction-Hypothese leitet. Für die Evaluation erstellen wir zwei Vektoren. Die beiden Vektoren enthalten aggregierte API-Nutzungen von aufeinanderfolgenden Zeitslots. Die Erwartung ist hierbei, dass die Ähnlichkeit von aufeinanderfolgenden Vektoren eines Entwicklers größer als zu den anderen Entwicklern ist. Auch hier sind geeignete Zeitslots, sowie Einfluss- und Störgrößen zu ermitteln. Eine Alternative Evaluation wäre die Anwendung eines Vorhersagemodells.

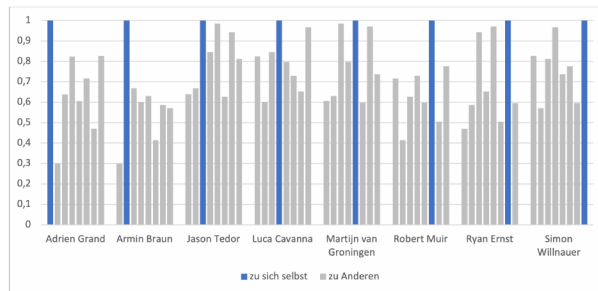


Abbildung 1: Kosinusähnlichkeit der Top-8-Entwickler aus dem Projekt Elasticsearch basierend auf der API-Nutzung

Dafür trainieren wir einen Klassifizierer mit der API-Nutzung der ersten t Zeitslots aller Entwickler an und überprüfen, mit welcher Genauigkeit die API-Nutzung des jeweiligen Zeitslots $t+1$ dem richtigen Entwickler zugeordnet werden konnte.

4 Zusammenfassung

Diese Hypothesen stellen einen Teil unserer größeren Arbeit der Erstellung von API-basierten Entwickler-Profilen dar. Die Hypothesen hängen meist sehr eng zusammen und die Erkenntnisse verleiten uns dazu, immer mehr Hypothesen zu erstellen. Die drei Hauptbereiche lassen sich dabei in die Analyse von API-Nutzung und -Abdeckung von einzelnen Entwicklern, die Vergleichbarkeit bzw. Ähnlichkeit von Entwickler-Profilen und die Verwendung des Profils für zukünftige Entscheidungen aufteilen. Wir betrachten diese Datenanalyse generell aus wissenschaftlicher Sicht, jedoch ist auch die Praxisnähe gegeben, so dass die Erkenntnisse aus unserer Analyse auch beispielsweise bei Projektmanagement-Entscheidungen hilfreich sein könnten.

Literatur

- [1] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *Proc. SAC 2011*. ACM, 2011, pp. 1317–1324.
- [2] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *Proc. MSR 2008*. ACM, 2008, pp. 121–124.
- [3] A. S. Badashian and E. Stroulia, "Investigating the information value of different sources of evidence of developers' expertise for bug assignment in open-source projects," *IET Softw.*, vol. 14, no. 7, pp. 748–758, 2020. [Online]. Available: <https://doi.org/10.1049/iet-sen.2019.0384>
- [4] M. Goeminne and T. Mens, "A comparison of identity merge algorithms for software repositories," *Sci. Comput. Program.*, vol. 78, no. 8, pp. 971–986, 2013.

Improving API Design Skills with the API Design Fest: Insights from a Preliminary Experiment with Students

Achim Röhl

Universität Hamburg
post@achimroell.de

Leif Bonorden

Universität Hamburg
leif.bonorden@uni-hamburg.de

Abstract

APIs should be stable and demand a careful evolution, which requires a good initial design. Such API design skills usually come from experience, but the *API Design Fest* intends to compress such experience into a dense course. While the original training event was intended for practitioners, we are interested in the applicability in software engineering education. Thus, we conducted a slightly adopted *API Design Fest* with students and report on initial insights. While we find the overall event and its API design activities suitable, we recommend extended preparation on breaking changes for future *API Design Fests* with students.

1 API Evolution

As software systems evolve, APIs should remain relatively stable—serving as a layer of abstraction: Even if implementation details change, the interface ensures continued compatibility. However, if interfaces are set in stone, they may hinder evolution. Thus, interfaces may also change, and some of their modifications (*breaking changes*) may impair compatibility [2].

If APIs are well-designed, the need for breaking changes is minimized. Yet, API design skills are often learned from experience as most API designers don't have specific training [3].

2 API Design Fest

The *API Design Fest* by Jaroslav Tulach [1] is a training event that confronts API designers with typical situations of API evolution. The idea is to experience the challenges of API evolution that would typically occur over a long time in a dense course. Furthermore, as the participants attempt to break each other's solutions, a peer-learning effect is expected.

The original *API Design Fest* uses the Java programming language and comprises three phases:

1. **Initial Design:** The participants design an API and a corresponding implementation for boolean circuits supporting AND, OR and NOT.
2. **Evolution:** The participants extend their boolean circuits to circuits that handle all double

values from 0 (equivalent to `false`) to 1 (equivalent to `true`) with adjusted definitions for AND, OR and NOT. Furthermore, clients should be able to define new circuit elements, e.g., GTE (\geq), by providing a suitable mathematical expression. At the same time, any client code working with the old API version must also work with the new API version.

3. **Competition:** The solutions are shared between the participants. Subsequently, the participants try to break each other's APIs by writing tests that succeed with the version from phase 1 but fail with the version from phase 2. Participants win points by breaking others' APIs and earn bonus points if their own API remains unbroken.

The event has been carried out several times, and Tulach has shared the main ideas of participants' solutions and their respective challenges [1].

3 Experiment with Students

We adopted the *API Design Fest* for an experimental setting with students: First, we extended the task descriptions to provide guidance for novel programmers. Second, we automated the provision of tasks and submission of solutions in GitLab. Finally, we added a fourth phase to compare the students' API design before and after the *API Design Fest*:

4. **Check:** The participants design an API for aggregators—independent from the circuit API. An aggregator takes an initial value and an aggregate function. The aggregate function updates the current value if a new value is added. A simple example is the sum aggregation: A new value is added to the current value. Additionally, the participants were given a hint that the API might be extended for more complex aggregate functions in the future, e.g., arithmetic mean aggregation.

We invited students from several informatics-related study programs to participate in the *API Design Fest*, leading to 12 participants from various bachelor's and master's programs with diverse programming experiences. Due to the COVID-19 pandemic,

we conducted this *API Design Fest* remotely. Furthermore, we scheduled 2 days for each phase to allow for asynchronous participation, although each phase comprises at most 2 hours of work.

In addition to unit tests and the competition results, we further analyzed the participants’ solutions from all phases to identify common patterns (e.g., object factories), evaluate general code quality (e.g., names), and check API design principles (e.g., immutability of method parameters).

4 Insights

The participants chose various design approaches in the first phase, including a single static class, comprehensive class hierarchies, object factories, and extensive string parsing. All of these approaches were also observed by Tulach in his *API Design Fests*. On the other hand, some solutions described by Tulach were not given in our experiment. In the second phase, participants who applied information hiding in the first phase could easily extend their solutions. However, others encountered problems as their solutions from the first phase were too rigid or too specific.

In the third phase, the participants in our *API Design Fest* deviated more from Tulach’s observations: Attempts to break each other’s APIs were relatively simple and did not use more comprehensive techniques, e.g., Java reflection. Moreover, some participants missed even simple breaking changes, e.g., method renaming.

Analyzing the solutions further, we found general high quality regarding basic quality criteria, e.g., consistently and well-named methods and acceptable method length. The solutions were more diverse for other criteria: API documentation and code readability stretched from ‘very well’ to ‘unacceptable’, only some solutions used appropriate types or handled exceptions and corner cases.

In the fourth phase, we observed more comprehensive solutions, particularly solutions with more complex structures using appropriate (generic) types and factories. However, many solutions still did not handle corner cases or hide internal information.

Overall, we clustered the solutions into three categories: (1) collection of methods, (2) basic structure, (3) advanced design. The number of solutions per category is given in Table 1—indicating a general increase in quality from phase 1 to phase 4.

Category	(1)	(2)	(3)	N/A
# of solutions, phase 1	4	4	2	2
# of solutions, phase 4	1	4	6	1

Table 1: Observed solutions for each category from first and fourth phase (higher is better)

Finally, we asked the participants to reflect on the experiment: All students rate their solution in phase

4 equal or better than their respective solution in phase 1. However, asked about their learning effect, their statements range from ‘learned almost nothing’ to ‘learned a lot’.

The first author’s master’s thesis [4] presents the experiment’s results in more detail and discusses them further.

5 Conclusion

We conducted a preliminary experiment with students on the learning effects of the *API Design Fest* and found the training event generally suitable for educational settings. While we saw similar designs to Tulach’s original training event, we observed a lack of comprehensive breaking attempts. We will use our observations to improve future *API Design Fests*:

Experiments with students: For future experiments with students, we will use on-site settings and provide extended preparation about breaking changes. Furthermore, we will investigate the respective influence of the second and third phases on the overall learning effect.

Experiments with practitioners: We intend to conduct experiments with experienced API designers to compare solutions and learning opportunities.

Transfer to other APIs: While Java APIs—and programming language APIs in general—remain an important topic, remote APIs—in particular, web APIs via REST—gained importance. Future work on the *API Design Fest* may include an adoption for such APIs.

References

- [1] J. Tulach. *Practical API Design – Confessions of a Java Framework Architect*. Apress, 2012.
- [2] L. Xavier et al. “Historical and impact analysis of API breaking changes: A large-scale study”. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017, pp. 138–147.
- [3] L. Murphy et al. “API Designers in the Field: Design Practices and Challenges for Creating Usable APIs”. In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2018, pp. 249–258.
- [4] A. Röhl. “Wirksamkeit eines Programmierwettbewerb zur Verbesserung von API-Design-Fähigkeiten”. Master’s Thesis. Hamburg, Germany: Universität Hamburg, 2022.

Understanding the Purpose of Source-Code Scopes Based on API Classifications

Philipp Seifer, Katharina Gorjatshev and Ralf Lämmel
The Software Languages Team, Universität Koblenz-Landau
{pseifer,gorjatshev,laemmel}@uni-koblenz.de

Abstract API classifications can give semantic hints regarding the purpose of source-code scopes. In this extended abstract, we present a number of research challenges related to comprehending the purpose of source code utilizing API classifications. We demonstrate preliminary work on applying Maven metadata (e.g., MCR categories) to different scopes in source code and detecting implications between combined usage of APIs; we visualize results as colored treemaps.

1 Introduction

Classification of Application Programming Interfaces (APIs), e.g., by clustering [1, 2], can provide strong semantic hints regarding the purpose or domain of any specific API. The Maven¹ Central Repository provides MCR categories (e.g., `Testing Frameworks`) and MCR tags (e.g., `testing` or `matching`) serving as user-curated classifications of the purpose of APIs.

APIs are used in different *scopes* – such as methods, classes or packages – of software projects. Applying an API classification scheme, such as MCR categories or tags, to the different scopes of a software project may serve as a predictor for the purpose of the respective scopes. Such an *abstraction* may aid the comprehension of the purpose of these source-code scopes. This gives rise to a number of research challenges.

C1 (META) – How can we abstract from usage of specific APIs in source-code scopes? We may abstract from the concrete involved APIs to their API categories (e.g., utilizing MCR *metadata* such as categories or tags) for source-code scopes, such as classes or methods. Is such abstraction effective?

C2 (IMPL) – What are the implications of combined API usage on understanding the purpose of scopes? Any scope may involve several APIs and therefore different API categories as well. For example, one API may be *implied* by another. Can we exploit this relationships to improve insights about the purpose of scopes, e.g., by simplifying the involved API categories based on implications? What different kinds of implications exist?

C3 (EXPL) – Can API categories be combined in a meaningful manner? Instead of relying solely on implication relationships between combined APIs, all involved categories may impact the purpose of a scope. In particular, APIs may be combined meaningfully (or unnecessarily) without a clear implication relationship. To this end, we must provide advanced *explanations* for combined usage: Can scopes be considered to be dominated by APIs of a specific category? Can we combine categories under weighted influences of meaningfully combined API categories?

2 Methodology

We will now present some preliminary insights gained by abstracting from concrete APIs using MCR categories (META), as well as utilizing the implication relationship between combined API usage (IMPL).

To this end, we first collected a data set of relevant Java repositories from GitHub that rely on Maven, filtering by standard popularity metrics such as number of stars, commits and contributors. We determined the dependencies declared for each project based on the included `pom.xml` files. We next processed each project to determine which scopes (packages, classes and methods) use one or more of the declared dependencies. Across all projects we could then identify candidate APIs for the implication relationship from instances of combined API usage by looking at frequent co-occurrence of APIs. We verified manually that usage of one such API is indeed implied by the other. One example for these verified implications of combined API usage is `org.mockito:mockito-core` which implies `org.junit:junit`.

3 Illustration

The treemaps in Figure 1 visualize classes and methods for one package of an example repository². Figure 1a shows the combined usage of APIs. Of particular interest are the usage of the testing framework `junit` (blue), the mocking API `mockito` (red) as well as the methods and classes tagged with `None` (green), that do not use any API. The figure shows significant

¹<https://maven.apache.org/>

²<https://github.com/garbagemule/MobArena> (pkg. signs)

combined usage of `junit` and `mockito` in most methods that involve either, and some combined usage with other APIs, such as the testing framework `hamcrest` (`lime`).

In Figure 1b we instead use the MCR categories of the respective involved APIs, which shows a combined usage of the *Test Frameworks* (both `junit` and `hamcrest`) and *Mocking* categories for all test-related classes. We also see some instances of what may perhaps be utility methods (green), as well as classes not related to testing without any detected API usage (also green).

Finally, in Figure 1c we use the implication relationship between `mockito` and `junit` to further simplify the visualization. In essence, we can now see most test-related methods and classes tagged with the *Test Frameworks* MCR category, clearly indicating the purpose of the respective classes.

4 Conclusion

We outlined interesting challenges in comprehending the purpose of source code utilizing API categories. We also demonstrated some initial work on applying MCR categories to different scopes with combined API usage, utilizing the implication relationship between APIs. Based on illustrative visualizations it seems plausible that both MCR categories as well as utilization of the implication relationship between combined API usage can aid the comprehensibility of the purpose of source code.

This is a first step in addressing challenges **META** and **IMPL**. With regards to challenge **META**, we demonstrate how to abstract to API categories, but empirical validation (e.g., user study or MSR techniques) is still left as future work. Our methodology identifies a strong implication relationship of combined API usage and relies on a straight-forward application of this implication relationship to API categories (**IMPL**). In future work, other relationships, such as conditional implication, must be comprehensibly identified using MSR techniques. In addition, future work needs to determine and detect additional combined API relationships and consider the question on how different API categories should be combined in the general case, beyond implication (**EXPL**).

References

- [1] Johannes Härtel, Hakan Aksu, and Ralf Lämmel. “Classification of APIs by hierarchical clustering”. In: *Proc. of Conference on Program Comprehension, ICPC*. ACM, 2018, pp. 233–243.
- [2] Bofei Xia et al. “Category-Aware API Clustering and Distributed Recommendation for Automatic Mashup Creation”. In: *IEEE Trans. Serv. Comput.* 8.5 (2015), pp. 674–687.



(a) API usage in terms of all concrete APIs used in the method, class and package scopes of the example package.



(b) API usage in terms of MCR categories, showing a more abstract view of the example package where both `junit` and `hamcrest` are reduced to the *Test Frameworks* MCR category.



(c) API usage in terms of MCR categories, while utilizing the implication relationship between `mockito` and `junit`, eliminating the *Mocking* MCR category.

Figure 1: Visualization of API usage showing the increasing level of abstraction from concrete APIs, MCR categories to implications between MCR categories for a studied GitHub project.

Migration des SüdLeasing COBOL-Kernbanken-Systems nach JAVA mit einem iterativ-inkrementellen Ansatz

Michael Maleika, Sebastian Seek
SüdLeasing GmbH, Pariser Platz 7, 70173 Stuttgart

Abstract

Anfang 2019 hat die *SüdLeasing GmbH*, eine der führenden herstellerunabhängigen Leasing-Gesellschaften in Deutschland mit 21 Standorten, das Projekt „Technisches Reengineering Bestandssystem LEASCO“ begonnen. Ziel war es, die Programmiersprache COBOL zu ersetzen und ein zukunftsfähiges System zu erhalten – Unter Einbehaltung sämtlicher Qualitätsvorgaben in einem agilen Projekt. Der folgende Beitrag geht insbesondere auf die Organisation des Gesamtverfahrens ein und spart die technischen Details der eigentlichen Migration von COBOL nach Java aus. Diese Details werden im separaten WSRE2022-Beitrag der *pro et con Innovative Informatikanwendungen GmbH* detailliert erörtert.

Ausgangssituation

Das System *LEASCO* war ursprünglich eine Standardsoftware, die seit über zwei Jahrzehnte in Eigenregie der *SüdLeasing GmbH* weiterentwickelt wird. Zu Beginn des Projekts bestand *LEASCO* aus über 5.000.000 Zeilen COBOL, die sich auf ca. 1.400 COBOL-Programme verteilen, und knapp 500.000 Zeilen PL/SQL-Code. Zudem bietet *LEASCO* ca. 1.900 Masken und enthält über 1.500 Datenbank-Tabellen. Wegen der großen Menge an Code und einer knappen Projekt-Zeitleiste wurde zu Beginn des Projekts beschlossen, dass ein manuelles Refactoring nicht in absehbarer Zeit zum Erfolg führen wird. Fachlich sind in *LEASCO* sowohl das Haupt- und Nebenbuch der *SüdLeasing GmbH* verortet. Ebenso, findet die gesamte Vertragsverwaltung im System statt. Aus diesem Grund war es für die Planung des Projekts unabdingbar, Weiterentwicklungen betreiben zu können. Und dies parallel zur Modernisierung unter Einbehaltung sämtlicher Qualitätsvorgaben der Aufsicht, Revision und Qualitätsprozesse.

Vorgehen des Modernisierungsprojekts

Zu Beginn des Projekts sind das gesamte System und die Systemlandschaft analysiert worden. Ergebnis für das Vorgehen im Projekt waren folgende Schritte:

1. Aufräumen von nicht verwendeten Code
2. Erneuerung des Frontends – Abschaltung der alten Oberfläche
3. (Automatische) Migration von COBOL nach JAVA

In Schritt 1 wurden sämtliche COBOL-Programmaufrufe ausgewertet. In Abhängigkeit zum letzten Aufruf (noch nie, länger als 1 Jahr, länger als 2

Jahre) wurde eine priorisierte Liste erstellt. Die Programme, die noch nie verwendet wurden, konnten relativ zeitnah wegarchiviert werden. Die restlichen Programme sind in Abstimmung mit den Fachbereichen analysiert und ebenfalls entfernt worden. Auf diese Weise konnte knapp die Hälfte des COBOL-Codes eingespart werden. Dies war möglich, da *LEASCO* als Standardsystem verschiedene Module anbietet, die die *SüdLeasing GmbH* jedoch noch nie verwendet und benötigt hat. Der übrige Source-Code stellt die Ausgangslage für die Gesamtmigration dar. Die neue Web-Oberfläche ist in Schritt 2 auf Basis von AngularJS, Kotlin und Spring Boot als Client-Server-Applikation entwickelt worden. Sie ersetzte die bestehende Windows-Applikation, die in den 90er Jahren als Ersatz für die ursprüngliche ASCII-Terminal-Bedienung entwickelt worden war. In Schritt 3 wurde nach einer Herstellerwahl von Anbietern, die eine automatische Migration von Cobol nach Java unterstützen, die *pro et con Innovative Informatikanwendungen GmbH* als Partner für das Migrationsprojekt ausgewählt.

Proof of Concept

Nach dem Auswahlprozess eines geeigneten Partners zur Unterstützung unseres Vorhabens, galt es zunächst den Ansatz des Dienstleisters zur Übersetzung des Cobol Codes nach Java zu validieren. Bevor ein komplexes und kostenintensives Projekt gestartet werden kann, muss sichergestellt sein, dass das anschließende Ergebnis auch den eigenen Erwartungen entspricht. Dies bedeutete, dass ein erster Test zunächst zeigen musste, dass nach der Migration die entsprechenden Programme in Java 1:1 genauso funktionieren und ebenso performant sind wie in Cobol. Hierzu haben wir uns eines der wichtigsten und zugleich ein relativ großes und komplexes Massenverarbeitungs-Programm herausgesucht. Vorteil hiervon war, dass wir sehen konnten, ob die enthaltenen Rechenoperationen korrekt ausgeführt wurden. Zugleich konnten wir prüfen, ob der verhältnismäßig lange laufende Prozess eine vergleichbare Laufzeit aufweist. Die Ergebnisse des POCs haben die Vorgaben ausreichend erfüllt. Daher ist nahtlos in die Planung und Durchführung des Gesamtprojekts übergegangen worden.

Die Migrationspakete

Um ein agiles und handhabbares Projektvorgehen zu erreichen, ist der gesamte COBOL-Code in Pakete aufgeteilt worden. Die erste Idee war es hier, die Pakete fachlich zu schneiden. Der erhoffte Vorteil war,

abgesteckte Bereiche zu erstellen, die dann wiederum unabhängig voneinander getestet werden können. Im Projekt hat sich schnell herausgestellt, dass es zu viele Überschneidungen zwischen den fachlichen Prozessen gibt, weswegen Programme nicht eindeutig einem fachlichen Prozess zugeordnet werden konnten. Die Vorgehensweise wurde somit von einer prozessbasierten in eine programm-basierte abgeändert. Die über 1.000 Cobol-Programme und Funktionen sind in eine Liste überführt worden, die dann von oben nach unten durchgearbeitet werden konnte. Ein Paket enthielt am Ende ca. 120-150 Programme und wurde innerhalb eines vierwöchigen Sprints umgesetzt. Für den gesamten Code sind 14 Migrationspakete erstellt worden. Diese sind dann innerhalb von knapp 12 Monaten umgesetzt worden. Ein großer Vorteil bei diesem Vorgehen, im Vergleich zu klassischen Reengineering Projekten, lag darin, dass wir regelmäßig Feedback zum Vorgehen und über die Qualität des generierten Java-Codes bekamen. Hierdurch konnte Fehler schnell erkannt und bei künftigen Migrationspaketen vermieden werden.

Testautomatisierung

Hunderte von Masken, Programmen und Funktionen mussten regelmäßig getestet werden. Die Entscheidung war, jedes einzelne migrierte Paket direkt im Anschluss nach Auslieferung zu testen. Auf diese Weise konnten Auffälligkeiten/Fehler direkt beseitigt und in zukünftigen Paketen vermieden werden. Aufgrund der großen Menge an benötigten Testfällen und der Notwendigkeit diese mehrfach ausführen zu können, kamen hierfür nur automatisierte Tests in Frage.

Die Umsetzung der Testautomatisierung erfolgt mittels Cucumber. Die Testfälle wurden zunächst in Zusammenarbeit mit den Fachbereichen in Gherkin formuliert und anschließend in Kotlin ausdetailliert bzw. programmiert. Die Ausführung erfolgte browserbasiert über Selenium. Zu Beginn wurde sich auf einfache Testfälle wie Maskenaufrufe und Ausführungen simpler Prozesse fokussiert. Diese wurden entsprechend der in den Paketen umgesetzten Programme parallel zur Migration erstellt, damit sie direkt nach Fertigstellung der Pakete für Tests verwendet werden konnten. Nach Abschluss sämtlicher Migrationsarbeiten belief sich das Repertoire an automatisierten Testfällen auf über 1.000 Stück. Neben den simplen Aufrufen wurden darüber hinaus im Laufe des Projekts auch immer komplexere Fachbereichstests abgebildet.

Abnahme des Java Codes

Zur Sicherstellung eines funktionierenden Systems sind ergänzend zu den automatischen Tests regelmäßige Fachbereichstests eingeplant worden. Die Abnahme der migrierten Software erfolgte über einen

zweistufigen Abnahmetest. Der erste erfolgte direkt nach Beendigung der Migration des gesamten Codes und unter der Bedingung, dass sämtliche automatisierte Tests erfolgreich waren. Diese erste Feuertaufe beinhaltete ca. 250 Testfälle mit über 1.000 Testschritten. Im Anschluss an die erste Test-Phase folgte eine ca. zwei-monatige Defect-Fixing-Phase, in der Fehler beseitigt und identifizierte Performance-Probleme behoben werden konnten. Als letzte Hürde vor der finalen Abnahme gab es einen zweiten Abnahmetest. Dieser musste ohne produktionsverhindernde Defects abgeschlossen werden, damit die Fachbereiche grünes Licht für eine erste Produktivnahme mit Pilotusern geben konnten. Die nächsten Wochen und Monate werden mehr und mehr User auf die neue Produktivumgebung freigeschaltet, bis in der Endausbaustufe das alte COBOL-Backend abgeschaltet werden kann.

Zusammenfassung

Für ein Migrationsprojekt dieser Größe ist es unabdingbar, Wege zu finden, häufig und schnell neue Erkenntnisse über die migrierte Software zu erhalten. Aus diesem Grund wurde ein agiles Vorgehen gemeinsam mit der *pro et con* beschlossen. Das bedeutete konkret:

- Das Aufteilen des Source-Codes in mehrere kleinere Pakete
- Der Aufbau von Testfällen, die automatisiert bei jedem Deployment durchlaufen können
- Das schnelle Ausliefern, Deployen und Testen neuer Software-Pakete

Durch das gemeinsame Projektvorgehen zwischen der *SüdLeasing GmbH* und der *pro et con* konnte das gesamte Team kontinuierlich arbeiten und sinnvoll ausgelastet werden. Dies hat eine sehr hohe Qualität sichergestellt und das Vertrauen des Fachbereichs in das neue System gestärkt. Ebenfalls konnte der Leistungsumfang der *pro et con* direkt erweitert werden, wenn sich neue Anforderungen oder alternative Lösungsansätze im Detail herauskristallisiert haben. Dies wäre bei einem konventionellen Wasserfall-Ansatz schwer möglich gewesen. Die intensive Zusammenarbeit zusammen mit dem Dienstleister hat sich sehr bezahlt gemacht und wird auch in zukünftigen Projekten der SL im vergleichbaren Maß angestrebt.

Zum Zeitpunkt der Einreichung dieser Veröffentlichung ist das Projekt nahezu abgeschlossen. Auf dem produktiven System arbeiten bereits zwei Drittel aller Nutzer und - mit Ausnahme von einigen komplexen Massenverarbeitungsläufen - werden sämtliche Prozesse über Java verarbeitet. Der nächste Schritt ist das Abschalten von COBOL LEASCO und das Starten erster Weiterentwicklungen am neuen Java-System.

Jedes COBOL-Java-Migrationsprojekt birgt neue Überraschungen

Uwe Erdmenger, Uwe Kaiser

pro et con Innovative Informatikanwendungen GmbH, Reichenhainer Straße 29a, 09126 Chemnitz

{uwe.erdmenger,uwe.kaiser}@proetcon.de

Abstract

Der Wunsch nach Modernisierung von Legacy-Software ist aktuell ungebrochen [1]. Die toolgestützte Software-Migration hat sich dabei als eine Technologie für diese Modernisierung etabliert. Werkzeuge für die Software-Migration weisen heute einen beachtlichen Automatisierungsgrad auf. Der von pro et con entwickelte COBOL-Java-Converter CoJaC konvertiert z.B. mehr als 90 % aller COBOL-Programme semantisch äquivalent nach Java. Der vorliegende Beitrag beschreibt ausgewählte Aspekte eines Migrationsprojektes bei der SüdLeasing GmbH. Die SüdLeasing GmbH ist eine der größten Leasing-Gesellschaften Deutschlands und agiert im Verbund der LBBW-Gruppe. Das zu migrierende Legacy-System beinhaltete 1.500 COBOL-Programme mit ca. 2,2 Millionen Codezeilen. Diese wurden mit CoJaC automatisiert nach Java konvertiert. Im Zielsystem ersetzt ein Spring-Boot-Server die proprietäre Middleware des originalen Systems. Die Benutzeroberfläche bestand aus 1.300 ASCII-orientierten Bildschirmmasken, welche über Messages mit den COBOL-Programmen kommunizierten. Die Maskenmigration war nicht Projektbestandteil. SüdLeasing entwickelte dafür mit Angular Weboberflächen identischer Funktionalität, welche an den Spring-Boot-Server angebunden wurden. Das Projekt wurde im geplanten Zeitraum abgeschlossen. Auch in diesem Migrationsprojekt existierten neue Herausforderungen, welche die aktuellen Migrationstechnologien und -werkzeuge noch nicht unterstützten. Dieser Beitrag beschreibt einige der teils unkonventionellen Lösungen.

1 COBOL ist nicht gleich COBOL

Die COBOL-Programme des Legacy-Systems lagen nicht in einem COBOL-Standarddialekt vor. Vielmehr wurde eine COBOL-Erweiterung verwendet, welche zusätzliche Sprachkonstrukte für die strukturierte Programmierung bereitstellt. Das folgende Codefragment zeigt exemplarisch die Definition einer Prozedur (`#entry`) sowie eine `#while`- und eine `#if`-Anweisung und den Prozeduraufruf mit `#pass`:

```
VERARBEITUNG #entry
    initialize v01daten.
    #while not ende-loop #do
        #pass LESE-URTV.
        #if fcode not = zero #then
            #pass LESE-UD0.
        #bend
    #bend
#end
```

Ein Precompiler *precob* übersetzt diesen Code in den Micro-Focus-Dialekt. Dabei werden die strukturierten Anweisungen in Folgen von `if`-Anweisungen und `GO-TO`-Sprüngen konvertiert. Diese COBOL-Programme sind für

eine toolgestützte Migration nach Java ungeeignet. Es entsteht durch die große Anzahl von `GO-TO`-Anweisungen schwer wartbarer Java-Code im Zielsystem. Eine Erweiterung des CoJaC um den originalen Dialekt wurde geprüft, aber verworfen. Der zu realisierende Aufwand hätte in keinem Verhältnis zur Nutzung in einem einzelnen Projekt gestanden. Alternativ wurde folgende Lösung gewählt: Es wurde ein neuer Precompiler *precob2* entwickelt. Dieser erzeugt im Gegensatz zu *precob* strukturierten Standard-COBOL-Code, der von CoJaC ohne weitere Anpassungen nach Java konvertiert werden kann. Das oben beschriebene Codefragment liefert nach der Bearbeitung durch *precob2* das nachfolgende Ergebnis:

```
VERARBEITUNG SECTION.
    initialize v01daten
    PERFORM WITH TEST BEFORE
        UNTIL NOT (not ende-loop)
    PERFORM LESE-URTV
    IF fcode not = zero
        PERFORM LESE-UD0
    END-IF
END-PERFORM.
```

Dieses Vorgehen ist auch auf andere Migrationsprojekte übertragbar. Insbesondere in den Anfangszeiten von COBOL, als die Sprache selbst noch keine strukturierte Programmierung unterstützte, wurden eine Reihe von „Vorübersetzern“ entwickelt, um diese Funktionalität bereitzustellen. Da diese heute nur noch vereinzelt im Einsatz sind, lohnt es nicht, neue Konvertierungswerkzeuge zu entwickeln oder existierende anzupassen, welche diese Programme direkt nach Java konvertieren. Hier ist ein Precompiler die bessere und kostengünstigere Wahl. Gleiches gilt auch für die Behandlung eingebetteter Sprachen wie bspw. die Transaktionssteuerung mit CICS von IBM oder `ENTER-TAL`-Anweisungen von HPE. Auch dafür existieren spezielle Precompiler, welche z.B. bei CICS die Transaktionsbefehle in `CALL`-Befehle von COBOL übersetzen. Bei der Konvertierung nach Java werden diese `CALL`-Befehle zu Aufrufen einer Bibliothek für die Transaktionssteuerung im Zielsystem.

2 Durchblick ordnet Durcheinander

Eine Aufteilung des Gesamtprojektes in Pakete hat sich bei vergangenen Projekten als erfolgreiche Migrationsstrategie erwiesen [2]. Diese Aufteilung wird auf Basis der toolbasierten Analyse von *vertikalen* Geschäftsprozessen (zusammenhängende Komponenten von Masken, Programmen und Daten) und dem Fachwissen des Kunden vorgenommen. Sie reduziert die Komplexität, sowohl bei der eigentlichen Migration als auch beim Test einzelner Pakete parallel zur Migration. Voraussetzung ist das Identifizieren von separat konvertier- und testbaren Paketen.

Beim zu migrierenden System handelte es sich um ein

Online-System für das Leasinggeschäft, welches unterschiedliche Geschäftsbereiche strukturiert abbildet (Sollstellung, Benutzerverwaltung, Angebotskalkulation, ...). Die Nutzeroberfläche bestand aus einer Menge von Masken, welche über mehrstufige Menüs erreichbar sind. Diese Menüstruktur wurde für das Identifizieren von Paketen genutzt. SüdLeasing erstellte eine Programmliste mit Menüpunkten und dazugehörigen Einstiegsprogrammen pro Geschäftsbereich. Diese bildete die Basismenge der im jeweiligen Paket zu migrierenden Programme. Ausgehend davon erfolgte eine toolgestützte Callgraph-Analyse. Dynamische CALL-Aufrufe, bei denen der Name des zu rufenden Programms über eine Variable übergeben wird, wurden durch zusätzliche Datenflussanalysen ermittelt. Die durch CALL referenzierten Programme erweiterten die Basismenge. Das wurde in einem ersten Ansatz solange fortgesetzt, bis eine transitive Hülle ermittelt war und keine weiteren Programme mehr gefunden wurden. Die identifizierten Pakete waren im Ergebnis sehr komplex und so für eine Migration ungeeignet. Die Ursache bestand darin, dass zwischen den Paketen viele Querverweise und gemeinsam genutzte Komponenten existierten.

Um Komplexität zu reduzieren, wurde ein pragmatischer Ansatz gewählt: Dieser bestand darin, nur bis zu einer CALL-Tiefe von drei aufzulösen. Programme, die nur über mehr als drei Stufen erreichbar waren, wurden für das jeweilige Paket ignoriert. Existierten Aufrufe für diese Programme, dann wurden diese in der Migration als Dummy-Programme konvertiert, welche lediglich eine `NotImplemented-Exception` warfen. Erst in nachfolgenden Paketen ersetzten die eigentlichen, migrierten Programme diese Dummy-Programme. Das hatte zur Folge, dass in den betroffenen Paketen nicht die vollständige Funktionalität realisiert und demzufolge auch keine vollständige Testabdeckung pro Paket zu erreichen war. Die Praxis bestätigte jedoch, dass dieser pragmatische Ansatz sowohl für die Paketgröße als auch für den paketweisen Test einen vertretbaren Kompromiss darstellte, mit welchem ein praktikables Ergebnis erzielt werden konnte.

3 Eine Benutzeroberfläche – zwei Systeme

Bei der Benutzeroberfläche des Legacy-Systems handelte es sich um eine SüdLeasing-Eigenentwicklung auf Windows-Basis mit ca. 1.300 ASCII-Masken. COBOL-Server und Windows-Client tauschten ihre Informationen in Form von COBOL-Messages aus. Bereits vor dem eigentlichen Migrationsprojekt entwickelte SüdLeasing mit dem Framework Angular die gesamte Benutzeroberfläche als Webapplikation neu. Der Informationsaustausch zu den unveränderten COBOL-Servern erfolgte nach wie vor über COBOL-Messages. Diese wurden durch Zusatzinformationen in Form von Kommentaren angereichert. Der COBOL-Precompiler wertete diese zusätzlichen Informationen aus, so dass diese dem Client zur Laufzeit zur Verfügung standen. So konnte die Darstellung der Informationen in der Weboberfläche feingranular gesteuert werden. Das folgende Beispiel zeigt den Ausschnitt einer typischen COBOL-Message mit einem zusätzlichen Kommentar als Erweiterung:

```
10 bld1-b30e0117 pic 9(02). 02 B
```

Der Kommentar 02 B steuert die Breite der Anzeige und die Ausrichtung der angezeigten Zahl.

Die Webapplikation sollte im Zielsystem auch zukünftig genutzt werden. Es bestand somit die Aufgabe, eine Schnittstelle zwischen der Webapplikation und den im Ergebnis der Migration entstandenen Java-Webservices zu entwickeln:

Bei der Migration entstehen aus den COBOL-Servern Java-Webservices. Der Informationsaustausch mit der Weboberfläche erfolgt über Java-Klassen, welche durch Konvertierung der COBOL-Messages entstanden. Die zusätzlich notwendigen Informationen werden durch Annotationen an den Java-Attributen realisiert. CoJaC wurde dazu um nutzerspezifische Übersetzungsregeln erweitert. Das nachfolgende Beispiel dokumentiert eine solche Annotation:

```
@LeascoDataField(  
    name = "bld1-b30e0117", length = 2,  
    feldtyp = GanzzahligRechtsbueendig)  
public CobolNumber bld1B30e0117 =  
    createNumber(2);
```

Die Annotation beinhaltet z.B. zusätzlich den originalen Message-Namen, der für die Maskensteuerung benötigt wird. Zur Laufzeit stehen diese Informationen der Webapplikation zur Verfügung. Somit wird eine Kompatibilität der Webapplikation sowohl zum originalen COBOL-System als auch zum generierten Java-System erreicht.

Diese Kompatibilität wurde im Testprozess ausgenutzt. SüdLeasing entwickelte eine komplexe Sammlung von Gherkin-Testszenarien unter Nutzung des Frameworks Cucumber. Mit jeder neuen Version wurden diese Testszenarien automatisch unter der Steuerung von Jenkins sowohl für das COBOL-System als auch für das migrierte Java-System abgearbeitet und die Ergebnisse wurden ausgewertet. Die Möglichkeit des automatisierten Vergleichs der Ergebnisse des COBOL- und Java-Systems reduzierte den Testaufwand wesentlich.

4 Das nächste Projekt wird wieder anders

Kein Projekt gleicht dem anderen. Trotz hoch entwickelter Werkzeuge und Technologien bietet jedes neue Herausforderungen. Das vorgestellte Projekt hat bestätigt, dass fertige Werkzeuge und Technologien nicht existieren. Der Aufwand für deren Anpassung bei einem neuen Migrationsprojekt muss bei der Planung berücksichtigt werden. Im Projekt ist die Kompetenz des Projektpartners hervorzuheben, sowohl beim Aufbau der Infrastruktur, beim Test unter Nutzung moderner Technologien als auch beim „Leben“ dieser Prozesse im Projektverlauf. Dies und die sehr gute Zusammenarbeit der Projektpartner waren ein Garant für den Projekterfolg.

Literaturverzeichnis

- [1] Moussavi-Amin, W.: Kosten senken und Innovationen schaffen. Computerwoche 2019, 51 - 52
- [2] Becker, C.; Kaiser, U.: Toolbasierte Software-Migration nach Plan. Softwaretechnik-Trends, Band 36, Heft 2, Mai 2016

A Bayesian Update to Software Quality Modeling

Johannes Härtel Ralf Lämmel
Software Languages Team, University of Koblenz

Abstract

Software reengineering profits from quantitative definitions of software quality. Such definitions are often given in terms of software quality models. We show a Bayesian reformulation of an established software quality model, in particular, of a software defect model. We evaluate correspondence of the results, and show an acceptable computation overhead of the Bayesian model. We argue on why the Bayesian version may be an improvement, discussing its definition and the representation of results.

1 Motivation

The evaluation of software reengineering profits from quantitative definitions of quality, like software performance, code comprehensions or defect proneness.

Unfortunately, these qualities may be hard to measure during reengineering. Hence, approaches use source code metrics that are easy to measure (e.g., lines of code), and conclude/predict on software quality. We summarize this as *software quality modeling*.

One typical approach is to model a software quality as a function of source code metrics. A measurement of a software quality is required during model fitting/learning (which may be expensive to get). However, the model then can be transferred.

Road-map: We focus on modeling the quality of defect proneness (a practice often referred to as defect modeling [5]). Methodological insights can be transferred to other qualities. We present a reformulation of an established logistic regression model for defects, converting it into a Bayesian version. We demonstrate how to define both models and evaluate correspondence. Finally, we argue why we believe that the Bayesian version is an improvement. A formal evaluation of this claim remains necessary.

2 Established Practice

A very basic software quality modeling scenario, in the scope of defect proneness, relates a software metrics X to a defect classification Y using a logistic regression.

We let X be the *lines of code*. The SZZ algorithm [4] approximates defect Y based on a revision history (see [2]). This is not possible during reengineering; thus a reason we need a model. We borrow data on *spring-boot* from [1] and fit a logistic model (in R). We will refer to this as the *established model*.

	Est.	SE	z value	Pr(> z)
Intercept	-5.05	0.096	-51.98	< 2e - 16***
log(1+X)	0.62	0.019	31.56	< 2e - 16***

Table 1: Excerpt of the model summary.

```
1 m <- glm(Y ~ log(X + 1), family = binomial(  
  link = "logit"))
```

Listing 1: An established model relates X and Y (R).

We provide an excerpt of the model summary in Table 1. The column *Est.* reports on a low average defect rate, reflected by *intercept* -5.0. Transformed lines of code $\log(X+1)$ have a defect increasing effect, reflected by *slope* 0.62. Strongly simplified, reengineering may now use this summary and attempt to decrease lines of code to also decrease defects.

The rest of the table reports on the uncertainty that arises within such data. We do not dive into details buried in this model and the summary – the upcoming Bayesian model will be more comprehensive on that.

3 A Bayesian Update

The following is the Bayesian ‘equivalent’ to the established model in Listing 1 (in STAN language¹).

```
1 data{  
2   int N;  
3   vector[N] X;  
4   int Y[N];  
5 }  
6 parameters{  
7   real intercept;  
8   real slope;  
9 }  
10 model{  
11   vector[N] prob;  
12   prob = inv_logit(intercept + slope * log  
13     (1 + X));  
14   Y ~ binomial(1, prob);  
15 }
```

Listing 2: A Bayesian model relates X and Y (STAN).

This Bayesian model defines how variables relate to each other. Some variables, like X and Y , are observed. We can feed them to the model. Some variables are unobserved, and need to be inferred, such as *intercept* and *slope*. Listing 2 gives a clean definition:

- The *data block* (line 1-5) lists the observed variables. We know the number of observations N

¹<https://mc-stan.org/>

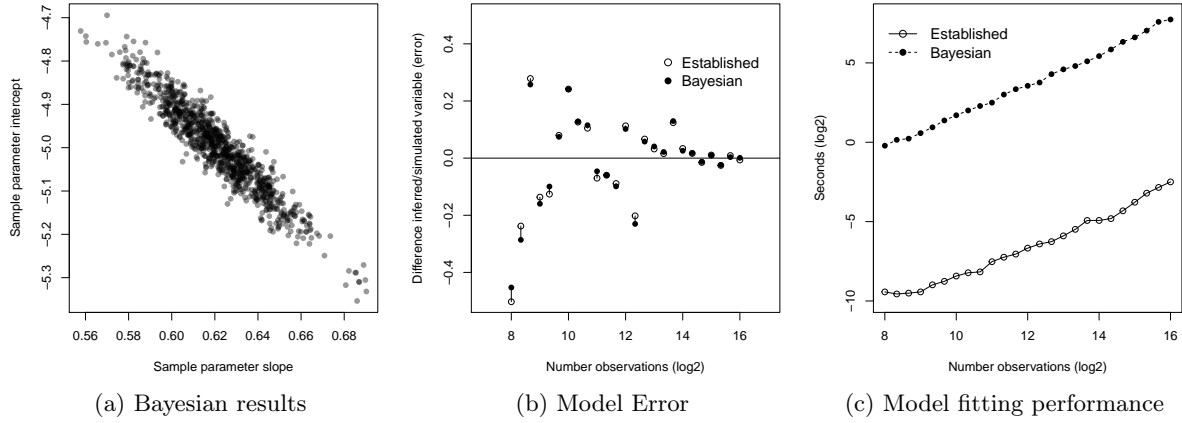


Figure 1: Results and performance.

(type int), the metric X (real valued arrays are type vector), and the defects X (array of int).

- The *parameters block* (line 6-9) lists unobserved variables to be inferred (*intercept* and *slope*).
- The *model block* (line 10-14) defines how the variables (data and parameters) relate to each other in a functional manner. Line 11 and 12 contain a vectorized definition of a new vector *prob*, given in terms of a function (`=`) of the parameters and data X . This definition corresponds to the internals of the established model. To produce a valid probability between 0.0 and 1.0, we apply the inverted logistic function (see `link="logit"` of the established model). Line 13 sets the distribution of defects Y , defining it as a vectorized stochastic function (`~`). A binomial distribution is used, with one trial and the probability being the vector *prob* (that we have just produced).

The Bayesian results, computed for the example, need to be presented in an entirely different way, because STAN reports on samples of parameters. Figure 1a shows a standard scatter plot of samples regarding parameter intercept and slope. The center of the depicted distribution reflects the parameters that match the data best. Comparable to the established model, the center shows a slope parameter of 0.62 and an intercept of -5.0 . Moreover, the plot depicts uncertainty intuitively, given by the density of points. A slope outside the range $[0.56, 0.68]$ is not compatible.

4 Evaluation

Figure 1b shows the relation between inferred parameters for both models. For the Bayesian model, we report on the parameter at the center of the output distribution. The underlying data is simulated [3]. The plot reports on the inferred parameters (by both models) relative to the parameter used to simulate the data. Precision for both models is comparable and improves with increasing number of observations.

Figure 1c shows the model fitting performance in relation to the number of observations (on log-log

scale). The established model is much faster, as it is exactly tailored to this particular task; however, both models scale linear with the number of observations.

5 Discussion

Our evaluation did focus on showing the correspondence, and not the superiority of Bayesian models, regarding results and performance.

However, our opinion is that Bayesian models are an improvement because: **The models are better to share and discuss.** We try to support this by showing both definitions next to each other, hinting at the absence of many details in the established definition. Furthermore, Bayesian models start to get attention in teaching, where sharing and discussing is inevitable. **The Bayesian model definition is more flexible.** One can fluently evolve our Bayesian model to advanced models, without changing the underlying infrastructure, while for the established model, one needs to switch software packages. **The interpretation of results is more intuitive.** We try to support this by comparing the established model’s summary, and the Bayesian results. The latter just needs interpretation of a standard scatter plot. However, a formal evaluation of these claims is still necessary.

References

- [1] Filipe Falcão, Caio Barbosa, Balduino Fonseca, Alessandro Garcia, Márcio Ribeiro, and Rohit Gheyi. On Relating Technical, Social Factors, and the Introduction of Bugs. In *SANER*, pages 378–388. IEEE, 2020.
- [2] Johannes Härtel and Ralf Lämmel. Incremental Map-Reduce on Repository History. In *SANER*, pages 320–331. IEEE, 2020.
- [3] Johannes Härtel and Ralf Lämmel. Operationalizing Threats to MSR Studies by Simulation-Based Testing. In *MSR*. ACM, 2022. To appear.
- [4] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR*. ACM, 2005.
- [5] Chakkrit Tantithamthavorn and Ahmed E. Hassan. An experience report on defect modelling in practice: pitfalls and challenges. In *ICSE (SEIP)*, pages 286–295. ACM, 2018.

Towards Maintainable Resilient Production Systems

Jonas Hansert*, Sandro Koch†, Tim Wunderlich‡, Steffen Ihlenfeldt‡, Thomas Schlegel*

Abstract

Predictable and unpredictable errors both pose threats to production systems and need to be addressed to avoid standstills, damages or even injuries. For this reason, we derive requirements for the structure of resilient production systems and propose an architecture to address these requirements. A decoupled interaction schema allows to reconfigure the compilation of the production system’s electrical and mechanical parts. The collaboration between its services enables the production system to automatically propose mitigation strategies in case of an unforeseen error. This is achieved by monitoring the process execution and extending the process models as well as the workflow engine to allow for dynamically addressing the executing components.

1 Introduction

Despite all sophisticated engineering, Cyber-Physical Production Systems (CPPS) are not immune to errors. Incidents occurring during production can never be prevented completely, not even by state-of-the-art technology and intelligent software. Tolerances of workpieces, sensors, and actuators have shrunk over time, yet they can still cause issues in a CPPS. Single component wear and tear, for example, raises the likelihood of error over time. Such errors are predictable due to their deterministic character and monitoring certain parameters of production systems with the objective of predictive maintenance is common practice [2]. However, in addition to the mechanical, electrical, and software components of a CPPS, humans can be a source of error, particularly erratic human behaviour. Such errors can cause the entire CPPS to halt, which leads to high costs. In the worst case, human life is in danger, so the goal is to minimise potential life-threatening errors and create a resilient environment that can deal with unexpected errors.

The system presented in this paper is part of the research project *RESPOND*. *RESPOND* aims to develop a flexible and dynamic production system of the Industrial Internet of Things (IIoT) that monitors its own state and can react dynamically to errors.

*firstname.lastname@h-ka.de, Karlsruhe University of Applied Sciences, Moltkestr. 30, 76133 Karlsruhe, Germany

†firstname.lastname@kit.edu, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany

‡firstname.lastname@iwu.fraunhofer.de, Fraunhofer IWU, Reichenhainer Str. 88, 09126 Chemnitz, Germany

2 Requirements for Resilient Production Systems

The purpose of a resilient production system is to deal with both predictable and unpredictable errors in order to limit CPPS downtime and prevent damage or injuries. To cope with these errors, we derive the requirement **R1**: *The CPPS must be able to handle unpredictable errors as good as manual error handling by the operator*. Coping with predictable errors is examined extensively in the field of predictive maintenance [2] and therefore outside of the scope of this paper. Furthermore, a CPPS must be flexible during runtime to reduce downtime. Aside from the CPPS, the processes related to the system’s surroundings must be taken into account. Processes that affect CPPS and personnel, in particular, must be adaptive to deal with errors. In addition, changes to the CPPS during runtime or during maintenance necessitate a modular structure to assure the CPPS’s longevity. As a result, the requirements **R2**: *Processes must be re-configurable without restarting the running process or stopping the CPPS* and **R3**: *The nodes of a CPPS must be exchangeable during runtime* are derived.

3 Interaction Schema for Components in Resilient Production Processes

In this section, we explain the architecture of a resilient production system that aims to fulfill the requirements derived in section 2, how its components interact with each other, and how the production processes are modelled to dynamically address their agents.

3.1 Architecture and Interaction

The architecture of the proposed system is based on our previous work [4]. It consists of six services, which primarily communicate with each other via a message bus: 1. Workflow Management System (WfMS) 2. Complex event processing (CEP) engine 3. Context service 4. Self-healing service 5. Process repository 6. User interface (UI). Furthermore, the process repository and the context service provide interfaces for synchronous requests. Sensors and actuators are represented as so-called *RESPOND objects* and can be aggregated to *RESPOND nodes*, which in turn are connected to the message bus. The purpose of these *RESPOND nodes* is to provide a common interface in order to allow for flexible exchangeability of the respective components. Therefore, the six services listed above are also represented as *RESPOND nodes*.

During normal operation, the operator selects a process model via the UI. The UI then deploys a message to the message bus, specifying the ID of the process model to be executed. This message is received by the WfMS, which retrieves the corresponding model from the process repository, instantiates the model and executes this instance. Further details about the process models and the activation of the involved agents are presented in section 3.2. Deviations from the expected process execution are detected by the CEP engine, e.g., if an agent takes too long to respond to a request or if anomalies occur in the data measured by the *RESPOND* objects on the field level. The CEP engine then sends an error message to the message bus containing details of the context of the deviations, e.g., the process step which was executed during the deviation or the underlying error type. With the aid of information provided by the context service, the *self-healing service* utilises this error message to deduct a recovery strategy, e.g., rescheduling the current process or notifying an operator [3]. The resulting recovery process is then pushed to the process repository and the UI is notified about the proposed mitigation strategy via the message bus. Finally, the operator is asked whether they want to stop the WfMS from executing the faulty process and to start the recovery process.

```

1  "timestamp": 1650461820,
2  "messageId": "eW91dHVVi",
3  "nodeId": "ZWRRdzR3",
4  "event": {
5    "type": "message.EventTaskRequest",
6    "taskRequestId": "OVdnWGNR",
7    "task": { "type": "FILL", "variables":
8      [{ "name": "Water", "value": "300" }] } }

```

Listing 1: *TaskRequest* message sent by the WfMS.

3.2 Process Design

Business Process Model and Notation 2 (BPMN2)¹ was developed to model business processes [1], but is also suitable for modelling production processes. Due to its widespread use, a large number of editors and WfMS already exist, hence we use BPMN2 as the basis for our approach. To comply with the proposed architecture, we have extended the WfMS so that messages can be sent and received via the message bus. In the skill-based approach, instead of assigning a task to a machine, the WfMS queries machines with the capability to perform the task and selects the most appropriate one [5]. To simplify the creation of production processes, these processes can be defined at different levels of detail. At the lowest level, two activities are provided: *TaskRequest* represents a request for a task to all nodes (listing 1) and starts a listener waiting for responses from the nodes. Upon positive response, *TaskAssign* instructs one of the available nodes to execute this task. The WfMS then waits for the message that the task has been completed and

continues to the next task. Selecting which node is supposed to take over the task can be modelled in BPMN2. All defined processes can be reused by other processes as sub-processes. The WfMS retrieves all required sub-processes from the process repository.

4 Conclusion and Future Work

In this paper, we presented requirements for resilient production systems and introduced a component interaction model of an architecture addressing these requirements. The proposed architecture consists of independent components that communicate with each other via message bus. Annotations in the BPMN2 model define the messages sent by the WfMS extension to execute tasks in our skill-based approach. Furthermore, components can be added, changed, and removed during operation due to the shared *RESPOND* node interface and the decoupled communication via the message bus. Unpredictable errors can be detected by the CEP engine, which monitors the process execution. The self-healing service then suggests a troubleshooting strategy based on the data from the CEP engine and the context service. For the future, a BPMN2 editor which incorporates our extensions to the WfMS would ease the modelling of resilient production processes. In addition, we aim to automatically generate the sub-processes which represent the *TaskRequest* and *TaskAssign* messages in order to further simplify modelling these resilient processes at a higher level.

Acknowledgement

This work was partially funded by the German Federal Ministry of Education and Research under grant 01IS18067A/B/D (RESPOND).

References

- [1] M. Dumas et al. *Fundamentals of business process management*. Vol. 1. Springer, 2013.
- [2] S. Selcuk. “Predictive maintenance, its implementation and latest trends”. In: *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 231.9 (2017), pp. 1670–1679.
- [3] S. Ihlenfeldt et al. “Increasing Resilience of Production Systems by Integrated Design”. In: *Applied Sciences* 11.18 (2021), p. 8457.
- [4] S. Koch et al. “Tackling Problems on Maintenance and Evolution in Industry 4.0 Scenarios Using a Distributed Architecture.” In: *Software Engineering (Satellite Events)*. 2021.
- [5] M. Vathoopan, K. Dorofeev, and A. Zoitl. “31 Skill-Based Engineering of Automation Systems: Use Case and Evaluation”. In: *AutomationML: The Industrial Cookbook*. Ed. by R. Drath. De Gruyter Oldenbourg, 2021, pp. 555–578.

¹<https://www.omg.org/spec/BPMN/2.0/>

Messung und Bewertung der Ergebnisse eines UI-Reengineerings im Bereich der Betreuung von demenzerkrankten Personen

Sergio Staab

sergio.staab@hs-rm.de

RheinMain University of Applied Sciences
Wiesbaden, Hessen, Deutschland

Ludger Martin

ludger.martin@hs-rm.de

RheinMain University of Applied Sciences
Wiesbaden, Hessen, Deutschland

ZUSAMMENFASSUNG

Gegenstand der hier vorgestellten Arbeit ist die automatisierte Analyse der Benutzungsoberflächen einer Informations-, Abstimmungs-, Kommunikations- und Dokumentationsplattform für Pflegekräfte von demenzerkrankten Patienten. Die Analyse erfolgt über ein eigens entwickeltes Analysetool, welches auf vier unterschiedlichen Analyseverfahren aufbaut. Mittels Performance-, Nutzerinteraktions-, Aufwandsbasierter und Reaktionsanalyse werden Anforderungen und Probleme analysiert, die sich bei der Interaktion der Pflegekräfte mit der digitalen Betreuungsdokumentation ergeben. Darauf folgt ein Software-Reengineering der Betreuungsdokumentation zur Qualitäts- und Akzeptanzsteigerung für das Pflegepersonal. Des Weiteren werden die vorherigen Erkenntnisse mit der überarbeiteten Benutzungsoberfläche gegenübergestellt und das Konzept der Dialoggestaltung gebrauchstauglicher Systeme im Kontext der Pflege um den Grundsatz der Ambientfreiheit von Objekten erweitert.

1 EINLEITUNG

Die Motivation dieser Arbeit beruht auf einer überproportional wachsenden Zahl an Pflegeempfängern. Prognosen des Statistischen Bundesamtes [2] zeigen, dass in den nächsten 15 Jahren allein in der ambulanten Pflege über 66.000 Fachkräfte fehlen werden. Der Pflegekräftemangel wird durch nicht besetzte Stellen bis 2030 einen Wertschöpfungsverlust und gesamtheitliche Einbußen von 35 Milliarden Euro nach sich ziehen. Wir analysieren die Betreuungsdokumentation mittels der Pflegesoftware INFODOQ, die zurzeit von zwei Demenz Wohngemeinschaften aktiv genutzt wird. Weiterhin wenden wir Metriken hinsichtlich Akzeptanz und Gebrauchstauglichkeit zur richtigen Softwareentwicklung von Pflegesystemen anhand der neugestalteten Betreuungsdokumentation an. **Die Beiträge dieser Arbeit sind wie folgt zu nennen:** Vorstellung einer im Rahmen der Digitalisierung von Demenz-Wohngemeinschaften entwickelten Informationsplattform; Erörterung von Anforderungen und Problemen, die sich bei der Interaktion der Pflegekräfte mit der digitalen Betreuungsdokumentation ergeben; Überarbeitung der bestehenden Software unter Berücksichtigung der Evaluationen; Aufstellung von Analyseverfahren bezüglich Akzeptanz und Gebrauchstauglichkeit zur Softwareentwicklung von Pflegesystemen; Gegenüberstellung der bestehenden und überarbeiteten Software mittels Analyseverfahren; Aufzeigen von Verbesserungen in der Interaktion zwischen Pflegern und Software durch diese Arbeit.

2 INFODOQ

INFODOQ [6] ist eine webbasierte Informationsplattform für den Einsatz in ambulant betreuten Demenz-Wohngemeinschaften. Die Plattform bietet einen transparenten Weg zur Nutzung von

Informationen sowie zur Abstimmung und Terminplanung von Angehörigen, Pflegern und Hilfskräften.

Als wichtigstes Teilstück der Software ist die Betreuungsdokumentation zu sehen. Sie zielt auf die Dokumentation und Bereitstellung von Betreuungsleistungen ab. Die Aktivitäten reichen dabei von der Mithilfe beim Kochen über Bastelstunden bis hin zum Training von Alltagskompetenzen und Tagesstrukturen. Ein Tag in den Wohngemeinschaften besteht aus drei Schichten mit jeweils unterschiedlichem Pflegepersonal, welches die Aktivitäten in eine Aktivitätenmatrix als eine binäre Entscheidung unter Angabe ihres Namenkürzels eintragen.

3 ANALYSEVERFAHREN

Das Analysetool baut auf vier unterschiedlichen Analyseverfahren auf:

Interaktion der Anwender: Atterer und Schmidt beschreiben in ihrer Arbeit [1] eine Lösung zur detaillierten Protokollierung von Nutzerinteraktionen mit AJAX-basierten Webanwendungen und lassen sich Maus-Interaktionen, Scroll-Anschläge, Klicks, Tastendrücke und Verweildauer protokollieren.

Performance-Analyse: Tullis und Albert [8] beschreiben in ihrem Buch die Berechnung von Leistungskennzahlen auf Grundlage bestimmter Nutzerverhalten, Szenarien oder Aufgaben. Diese Verhaltensweisen bilden den Eckpfeiler der Leistungskennzahlen, welche sich als Performance Metrics Task Success, Time on Task, Errors, Efficiency und Lernfähigkeit errechnen lassen.

Aufwandsbasierte Analyse: Feldman, Tamir, Komogortsev und Mueller [3] stellen in ihrer Arbeit die Hypothese auf, dass Usability eine Funktion von Aufwand und Zeit ist. Die Differenz zwischen dem Interaktionsaufwand eines Benutzers und dem Aufwand des Entwicklers der selbigen Oberfläche ist laut den Autoren die Lernphase des Benutzers bis hin zu dem Zeitpunkt, an dem Benutzer und Entwickler den fast gleichen Aufwand haben. Die Differenz zwischen den Kurven stellt die Verständlichkeit der Interaktion bezüglich der Aufgabe beziehungsweise dem der Aufgabe zugrundeliegenden Interface dar. Je größer der Unterschied, desto schwieriger scheint der optimale Umgang mit der Benutzungsoberfläche.

Psychologische Reaktionsanalyse: Freeman und Ambady [4] stellen in ihrer Arbeit die Echtzeitverarbeitung ihrer Maus-Tracking-Methode zur Bewertung von Aufgaben vor. Motorische Reaktionen können als Endergebnis einer Feed-Forward-Pipeline aus Wahrnehmung, Kognition und Aktion betrachtet werden. Es lassen sich Rückschlüsse über den zeitlichen Verlauf der wahrnehmungsbezogenen, kognitiven Verarbeitung ziehen.

4 USABILITY-ANALYSE

Die zuvor beschriebenen Analyseverfahren wurden gemäß der Literatur in unserem Analysetool angewandt. In einer 50-tägigen Usability-Analyse aller drei täglichen Protokollierungen durch

das Pflegepersonal, die 3.135 Dokumentationsaktivitäten von 22 Pfleger umfasste, wurden Benutzerverhalten, Performance, Kompetenz, Regelmäßigkeiten, Nutzerirritationen sowie der Umgang mit unterschiedlichen Interfaces eines oder mehrerer Pfleger getrackt, verglichen und analysiert.

Die Analyse der Betreuungsdokumentation, in Abbildung 1 zu sehen, zeigt eine hohe Lernkurve, diverse Nutzerirritationen (darunter die falsche Bedienung der Buttons und Images und eine umständliche Benutzung der Tabelle sowie Verrutschen in Spalte und Zeile) und hoher physikalischer Aufwand durch Scrollen und Klicken.

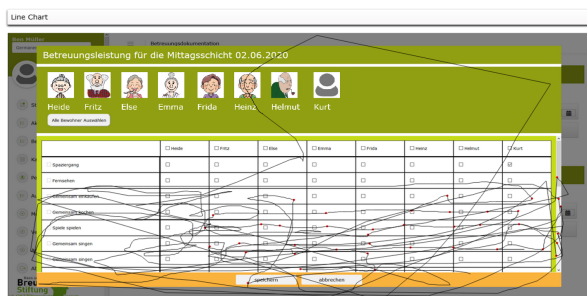


Abbildung 1: Maus-Tracking-Visualisierung altes Dokumentationsdesign

4.1 Reengineering

Angelehnt an empirische Untersuchungen von Tractinsky [7] sowie Kirosu und Kashimura [5] erfolgte daraufhin die Vereinfachung der Betreuungsdokumentation in mehreren Schritten. Das neue Design der Betreuungsdokumentation fällt deutlich sparsamer aus, sowohl im Kopfbereich durch den Verzicht der statischen Auflistung der Bewohnerbilder als auch im Fußbereich durch Entfernen der Fußleiste. Diese Entscheidung basiert auf der nicht Erkenntnis von Funktionen hinter den Bewohnerbilder. Spalten und Zeilen wurden durch den Hover-Effekt maskiert, was die Übersichtlichkeit verbessert und ein Verrutschen verhindern soll. Da einer der größten Fehlerquellen das Vertuschen in Zeile und Spalte ist. Die Auswahl aller Bewohner ist in die Tabelle integriert worden. Das Bild eines Bewohners wechselt mittels Mausinteraktion mit dem jeweiligen Bewohnernamen in der Auflistung oder dem Tabellenkopf. Die Platzersparnisse offerieren dem Anwender auf den ersten Blick deutlich mehr Inhalt der Dokumentationsmatrix, mit dem großen Vorteil das nun der enorme physikalische Aufwand der Betätigung des Scrollbalkens deutlich minimiert wird, vergleiche Abbildung 2.

Die Mauspfad-Analyse zeigt nun einen deutlich zielgerichteteren Weg über die verschiedenen Elemente hinweg. Die Platzersparnisse haben zwei Vorteile: Zum einen entstehen durch weniger Elemente weniger Nutzerirritationen. Zum anderen lässt die freigegebene Fläche mehr Platz für die zu bearbeitende Dokumentationsmatrix, welche weniger Scroll-Bewegungen mit sich bringt. Dies lässt eine Aussage über den Vergleich der beiden Designs im nächsten Kapitel zu.

5 ZUSAMMENFASSUNG

Diese Arbeit stellt eine automatisierte Analyse der Benutzerinteraktionen von Pflegern in interaktiv einer Betreuungsdokumentation vor. Die Auswertung wurde teils ebenfalls automatisiert, so

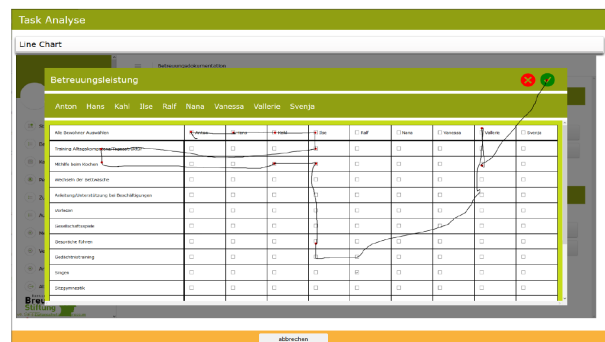


Abbildung 2: Maus-Tracking-Visualisierung neues Dokumentationsdesign

ließen sich diverse Performancemetriken zwischen unterschiedlichen Benutzungsschnittstellen der Betreuungsdokumentation automatisiert vergleichen.

Auf Basis unserer Untersuchung erweitern wir das Konzept der Dialoggestaltung gebrauchstauglicher Systeme im Kontext der Pflege um den Grundsatz der Ambientefreiheit von Objekten, d. h. die Verhältnismäßigkeit (Nutzen - Nutzerirritation) jedes einzelnen Elementes (Img, Button, etc.) auf einer Benutzungsoberfläche. Das bedeutet, dass das Ergebnis der Nutzung im Kontext der Granularität der Betreuungsdokumentationssoftware stehen muss, um den zuvor beschriebenen Nutzerirritationen möglichst vorzubeugen. Als Eigenschaften der Dokumentation sind in diesem Zusammenhang Selbstlokalisierung, Klarheit und Einfachheit zu nennen. Viele Elemente auf einer Benutzungsoberfläche tragen nicht zu dem Hauptfunktionsumfang bei, aber verunsichern in einem unverhältnismäßigen Maß. Es wurde jedes Element im Verhältnis seiner Funktion zu hinterfragt und die Benutzungsoberfläche auf ein Minimum an Elementen reduziert um die erkannten Haupt-Nutzerirritationen zu eliminieren.

LITERATUR

- [1] Richard Atterer and Albrecht Schmidt. 2007. Tracking the interaction of users with AJAX applications for usability testing. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1347–1350.
- [2] Statistisches Bundesamt. 2017. *Pflege im Rahmen der Pflegeversicherung Deutschlandergebnisse*. Statistisches Bundesamt. https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Pflege/Publikationen/Downloads-Pflege/pflege-deutschlandergebnisse-5224001179004.pdf?__blob=publicationFile
- [3] Liam Feldman, Carl J Mueller, Dan Tamir, and Oleg V Komogortsev. 2009. Usability testing with total-effort metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 426–429.
- [4] Jonathan B. Freeman and Nalini Ambady. 2010. MouseTracker: Software for studying real-time mental processing using a computer mouse-tracking method. *Behavior Research Methods* 42, 1 (Feb. 2010), 226–241. <https://doi.org/10.3758/brm.42.1.226>
- [5] Masaaki Kiroso and Kaori Kashimura. 1995. Apparent usability vs. inherent usability: experimental analysis on the determinants of the apparent usability. In *Conference companion on Human factors in computing systems*. 292–293.
- [6] Sergio Staab, Ludger Martin, Maren Ewald, and Stephanie Völs. 2018. INFO-DOQ Onlinebasierte Applikation zur transparenten Betreuungsdokumentation für Wohn-Pflegegemeinschaften. *Bundesweites Journal für Wohn-Pflege-Gemeinschaften* 7, 7 (Nov. 2018), 28. <https://bit.ly/3dHPkBA> BJFWPG.
- [7] Noam Tractinsky. 1997. Aesthetics and apparent usability: empirically assessing cultural and methodological issues. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. ACM, 115–122. <https://doi.org/10.1145/258549.258626>
- [8] Thomas Tullis and William Albert. 2013. *Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics* (2 ed.). Morgan Kaufmann, Amsterdam. <http://www.sciencedirect.com/science/book/9780124157811>

Reengineering a Large-Scale Hybrid Cloud Car Rental System

Markus Kassay, Sebastian Künne, Colin McBride, Sören Frey, Oliver Alth
Mercedes-Benz Tech Innovation GmbH, Wilhelm-Runge-Straße 11, 89081 Ulm, Germany
{markus.kassay, sebastian.s.kuenne, colin.mcbride,
soeren.frey, oliver.alth}@mercedes-benz.com

Abstract

Car rental logistics is backed by complex IT processes that often manifest in large-scale applications covering several markets. We describe our experiences and observations of reengineering such a hybrid-cloud-based car rental system over several years.

1 Introduction

To optimize fleet utilization, the car rental domain has to address complex decision problems, such as pool segmentation and fleet planning [1]. Corresponding IT systems support these decision making processes and also enable day-to-day fleet management operations like order and inventory management. We provide an industry view and report on our multi-year experiences of reengineering such a complex, globally distributed car rental system (*CARS*). We cover several architecture evolution steps, ranging from a partial cloud migration to the introduction of an event-driven architecture. Hybrid cloud architectures involve specific challenges like cloud data integration [2] that also have to be considered during the reengineering of *CARS*.

The paper is structured as follows. Section 2 provides an overview of *CARS*. The architecture evolution steps are detailed in Section 3, before Section 4 draws the conclusions.

2 CARS Overview

From a small single-language booking application, *CARS* has evolved over seven years to a fully configurable rental booking platform. It now supports the day-to-day operations of retailer outlets running multiple vehicle fleets in over 20 markets and languages. The system encompasses a range of fleet functionalities, for example, vehicle pool management and vehicle damage processing. It uses complex algorithms to provide flexible pricing across retailers, tenants, and markets, and it includes online payment features with invoicing and accountancy.

The platform also provides document template creation with the ability to incorporate booking data and digital signatures. KPI and ad-hoc reporting, lead creation, emailing, and the possibility for full system configuration form a complete solution for its users in

retailer outlets and head offices. The system also includes a suite of API endpoints, which enable its data to be exchanged with third party development teams to provide customer-facing booking applications tailored to the specific needs of each market.

3 Architecture Evolution Steps

The four architecture evolution steps (AES1-AES4) of *CARS* are shown in Figure 1 and detailed in the following sections. We focus on seven respective key challenges (CH1-CH7) of AES1-AES3 that necessitate subsequent reengineering measures.

3.1 AES1: Initial On-Premise Monolith

Overview *CARS* end users can rent vehicles via the **Booking Customer Frontend**. Fleets are managed by dealers via a specific UI. Several third party systems, such as **VMD** (vehicle master data) and **CMD** (customer master data), provide and consume data to and from *CARS*. This data is stored in *CARS* in one central relational database. Each system component reads and writes data directly from and to the database. Business logic is contained in multiple classes, some classes duplicate the business logic with only minor adjustments. *CARS* is deployed manually to on-premise environments in three global regions. Infrastructure resource adjustments are done by another company on the basis of issued tickets.

Challenges (CH1) The lack of direct access to computing resources results in inefficiencies when modifying the infrastructure. Necessary changes are time-consuming, costly, and error-prone. (CH2) Furthermore, the initial monolith exhibits insufficient separation of concerns resulting in maintainability problems and slower feature development. (CH3) Moreover, unit tests are difficult to implement because of the tight coupling of system components.

3.2 AES2: 3-Tier Hybrid Cloud

Overview CH1 is addressed by moving the environments of region 2 & 3 to a public cloud. One environment is maintained on premise due to legal restrictions concerning user data in region 1. This results in a hybrid cloud architecture. We also introduce a **CI/CD** system for simplifying and accelerating

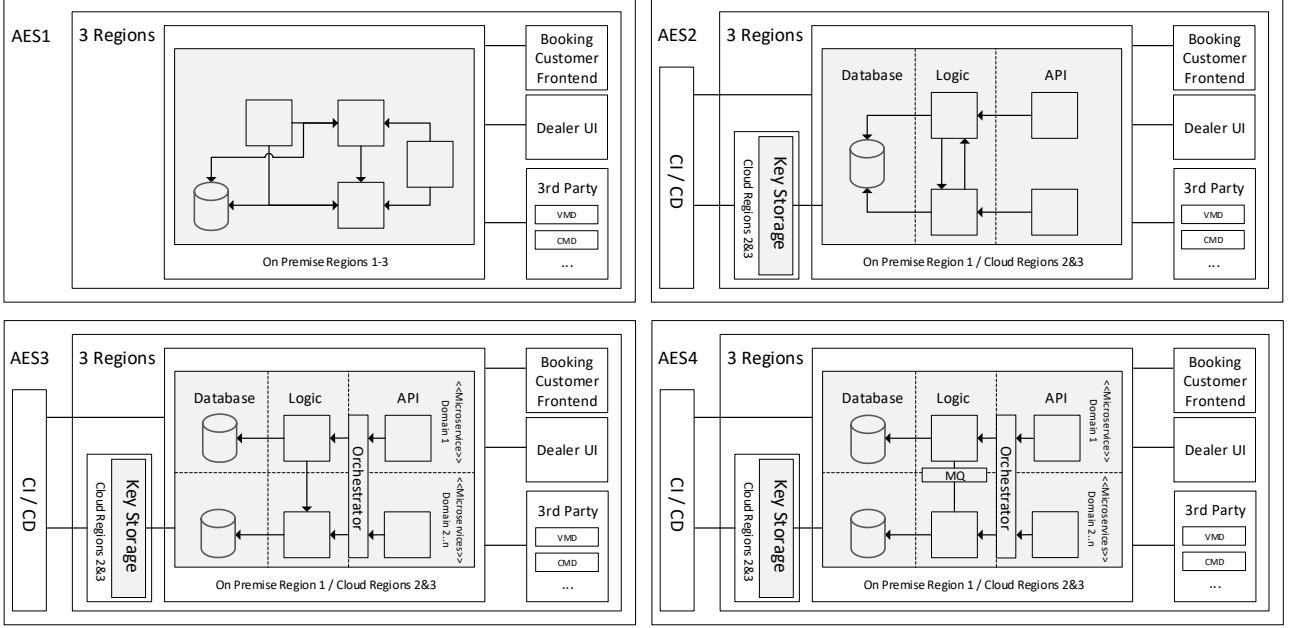


Figure 1: The four architecture evolution steps of the system *CARS* (gray)

our deployments. **Key Storages** are used in both of the cloud regions 2 & 3 to provide system credentials. On premise region 1 has to be integrated using an additional **Key Storage** in cloud region 2. For tackling CH2, we refactor the data access-, business logic-, and endpoint components into the three tiers **Database**, **Logic**, and **API**. Regarding CH3, we split the code into small testable units of work. Dependency injection is used to improve testability and to mitigate tight coupling.

Challenges (CH4) The development team is large while knowledge of the system’s complex parts is limited. (CH5) During refactoring, we discover several cyclic dependencies. (CH6) Development is difficult to scale because different parts of the system cannot be developed and deployed independently.

3.3 AES3: Domain-Driven Architecture

Overview We use domain-driven design to address CH4. Based on expert workshops we define n domains such as *Booking* and *Payment* and assign them to subteams. Each domain owns a set of domain objects (D_{Owned}) and is responsible for changing their state.

We rework the **Database** layer and split the code into separate packages according to the identified domains. Cross-domain use cases are enabled by the introduction of an **Orchestrator**. This new component coordinates calls between the different domains, enforces a call hierarchy, and hence solves CH5. To meet CH6, we extract all components per domain into a corresponding, self-contained (micro-)service. We also enable simplified bootstrapping and configuration of new services. Hence, we can run, develop, test, and scale the services independently.

Challenges (CH7) It is sometimes not possible to assign a domain object to exactly one domain. For example, the *Vehicle* object is used in the domains *Booking* and *Vehicle Damage*.

3.4 AES4: Event-Driven Architecture

Overview Objects relevant for multiple domains have to be synchronized. Hence, we tackle CH7 by implementing a publish/subscribe mechanism using a message queue (MQ). If a microservice experiences a downtime the messages can be processed after returning to operation again. The assumption is that the message processing operation is idempotent for ensuring an exactly-once semantic. In *CARS* each service writes update events for their corresponding D_{Owned} to the message queue and the subscribing services retrieve the update events.

4 Conclusions

Partial cloud migrations can form a viable basis for a sustainable evolution of complex, large-scale systems. Nevertheless, continuous refactoring remains crucial.

References

- [1] Yazao Yang, Wenzhou Jin, and Xiaoni Hao. Car rental logistics problem: A review of literature. In *2008 IEEE International Conference on Service Operations and Logistics, and Informatics*, volume 2, pages 2815–2819, 2008.
- [2] Pankaj Goyal. Enterprise Usability of Cloud Computing Environments: Issues and Challenges. In *19th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 54–59, 2010.

Deriving Goals From Well-Known Industrial Cases of Product Line Engineering Adoption

David Morais Ferreira
TU Kaiserslautern
moraisferreira@cs.uni-kl.de

Vasil L. Tenev Martin Becker
Fraunhofer IESE, Kaiserslautern
{vasil.tenev, martin.becker}@iese.fraunhofer.de

Abstract

Product Line Engineering (PLE) enables strategic reuse within an organization, allowing for a decrease in cost, improved time to market and higher product quality. While a wide variety of approaches have been developed to support reengineering activities in the context of change-intensive systems, choosing suitable approaches depends on what goals an organization wishes to fulfill. Therefore, the quality of these goals is vital to maximizing the potential benefits. Hence, this paper provides first insights into what drives the adoption of PLE, i. e. what goals companies typically identify to justify this transition, and attempts a first hierarchical overview of the aforementioned goals. To this end, we draw on well-known industrial cases from the Software Product Line Conference's (SPLC) Hall of Fame.

1 Introduction

Software reengineering activities are inherently associated with risks [1], which require careful consideration. Explicitly documenting the motivation that drives the need for change, for example through the use of goals, also has an impact on risk reduction, as a clearer picture is always beneficial.

Change-intensive systems are typically developed following PLE approaches, either systematically or ad-hoc. Often, achieving goals such as “reduce time to market”, “improve quality” and “reduce costs”, require significant reengineering efforts, encompassing the complete product life-cycle. Therefore, the need to understand goals becomes apparent. However, in practise, companies often misinterpret pain points and formulate them as solution-driven goals. This results in an inefficient use of resources, as the wrong goal, and therefore inadequate reengineering activity, is followed. Drawing an analogy to the medical field, a patient describes symptoms to his doctor, who then interprets them to identify the appropriate treatment, i. e. the treatment which most efficiently treats the underlying cause (disease).

In an effort to address the aforementioned problem, this paper provides a first insight into what drives the adoption of PLE, i. e. what goals companies typically

identify to justify this transition, and attempts a first hierarchical overview of the aforementioned goals.

2 Background

This publication was written in the context of a master's thesis by the first author, which is supervised by the second author. Within the scope of the aforementioned master thesis, an integrated analysis framework, which combines individual tools in order to find solutions for common scenarios which arise in industry, is to be developed. By focusing on typical goal-driven scenarios, both re-engineering and PLE practitioners will benefit from an improved guidance across the complete life-cycle.

3 Goals

Goals describe desirable results whose achievements require careful planning of actionable steps, commitment of resources, and purposeful and productive actions [2]. Moreover, goals are articulations of fundamental needs, and may be related to one or multiple needs, and vice versa.

Therefore, defining attainable goals paves the way to a successful adoption of product line engineering approaches. One key aspect of which is strategic reuse, which is crucial to achieving organizational business benefits beyond what is possible with single-product development approaches [3]. To this end, both business and engineering strategies must be aligned, allowing for a planned, i. e. strategic reuse approach which leverages cross-organizational synergies and ultimately satisfies the current demand for mass customization.

The needs and expectations of different stakeholders throughout the organizational hierarchy can be outlined by systematically defining high-level *business* goals and decomposing them successively into more concrete, lower-level goals. To this end, we analyzed a subset of the state of the art, focusing on well-known case studies to elicit a set of goals and attempt a first classification.

4 Data Collection

First, representative publications for each organization in the Hall of Fame were chosen from the list of

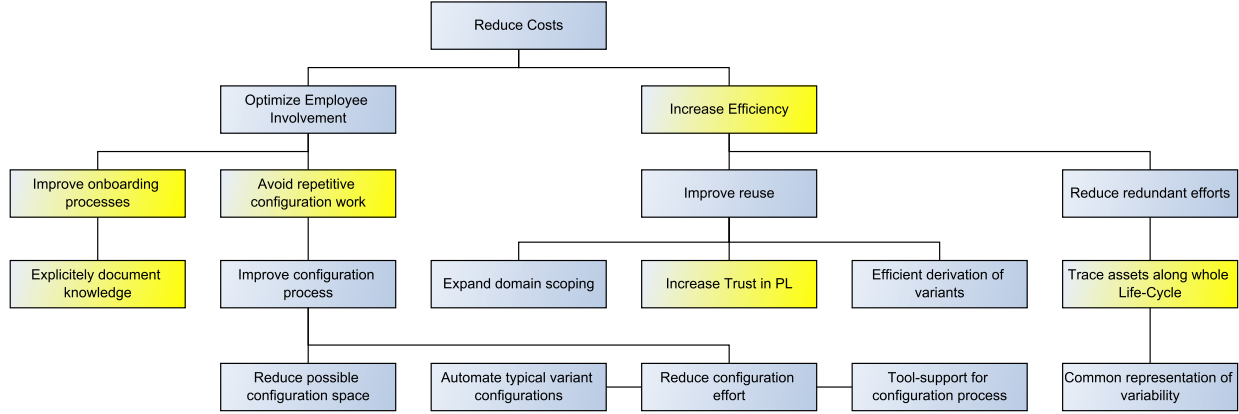


Figure 1: Goals Concerning Costs

available publication on the SPLC’s website. As these publications were sometimes not sufficient to extract the required information, the scope was expanded to include additional literature [4, 5, 6]. Then, in a manual process, goals were extracted from each publication. The resulting list of goals was then organized hierarchically. As the identified set of goals showed no hierarchy, we added additional, more abstract goals to allow a hierarchical structure to emerge. In the subsequent figures, these missing goals are highlighted in yellow.

5 Results

Most, if not all, cases mentioned “decreasing time to market”¹, “improving quality”¹, and “reducing costs” (Fig. 1) as motivating goals. In our overview, these form the top-level goals on the business level. In our set of publications, most goals were either very abstract (e.g. “improve quality”), or very specific, describing solution-oriented goals such as “remove unused code”. Unfortunately, current case studies do not typically describe challenges, instead they focus on successes and lessons learned [6].

“Reduce possible configuration space” (Fig. 1), for example, could have two different interpretations, depending on which higher-level goal is prioritized. By reverse-engineering the dependencies between switches, constraints can be added to the configuration space, therefore minimizing it. This reduces the overall complexity. Alternatively, configuration switches that are always identical for all delivered variants can be removed from the platform, and the associated variability (which is never used) can be stripped from the core assets.

This solution-driven mindset misses the underlying root cause, potentially diverting resources from high return on investment activities to lesser efficient ones. In the authors’ opinion, this first result allows both

PLE and reengineering practitioners to more easily identify the appropriate actions to address the problems described by customers.

6 Summary and Conclusion

As mentioned in Section 2, this overview is a work in progress and open to suggestions. In future work, we would like to also consider other, more recent publications, as some challenges described in the SPLC Hall of Fame study cases have already been partially, if not fully, solved in recent years [6]. Additionally, interviews with experienced practitioners would certainly yield insights into current issues, goals and their underlying challenges. Nevertheless, we believe that the considered cases are still relevant in today’s research world.

In conclusion, overly specific solution-driven goals should be complemented by also considering underlying root causes, as the resulting picture will enable a more informed decision, ultimately contributing to the success of the endeavour.

References

- [1] H. M. Sneed, “Risks involved in reengineering projects,” WCRE ’99, pp. 204–211, IEEE, 1999.
- [2] E. A. Locke and G. P. Latham, *A theory of Goal Setting & Task Performance*. Prentice-Hall, 1990.
- [3] L. M. Northrop, “Software product lines essentials,” tech. rep., Carnegie Mellon University, Pittsburgh, 2008.
- [4] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. SEI Series in Software Engineering, Addison-Wesley, 2 ed., 2003.
- [5] F. J. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag Berlin Heidelberg, 1 ed., 2007.
- [6] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, and J. Martinez, “The state of adoption and the challenges of systematic variability management in industry,” *Empirical Software Engineering*, vol. 25, pp. 1755–1797, May 2020.

¹Due to space constraints, the corresponding figures have been uploaded to <https://github.com/DavidMoraisFerreira/DerivingGoalsFromWellKnownIndustrialCasesofPLEAdoption>