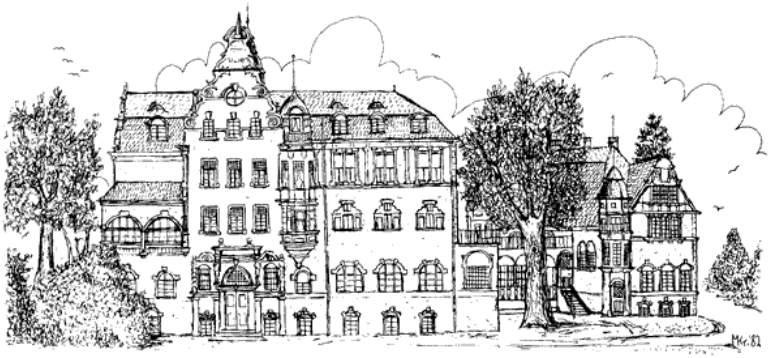
27. Workshop Software-Reengineering und -Evolution

WSRE 2025

der GI-Fachgruppe Software-Reengineering (SRE)

9.-11. April 2025

Physikzentrum Bad Honnef



Programmübersicht

Mittwoch, 9.4.2025

ab 12:30	*Mittagessen (optional)*
	ARCHITECTURE ANALYSIS
14:00	Daniela Schilling Einen Big Ball of Mud analysieren
14:30	Michel Krause, Rainer Koschke Reflexionsanalyse mit SEE (Tool-Demo)
15:00	Leon Ehrhardt, Rainer Koschke Reflexionsanalyse in SEE: ein kontrolliertes Experiment
15:30	*Kaffeepause*
	KEYNOTE
16:00	Sven Apel AI4SE: Adventures in the Promised Land
17:00	*Ende des Tagesprogramms, anschließend traditioneller Besuch der Eisdiele*

Donnerstag, 10.4.2025

	SOFTWARE QUALITY AND MODERNIZATION
09:00	Rahel Sundermann, Sebastian Krieter, Birte Glimm, Thomas Thüm Teaching Software Quality with Adaptive Source Code Feedback
09:30	Ada Slupczynski, Horst Lichter Challenges and Impact Factors on Modernization Projects - Results of an Industry Study
10:00	*Kaffeepause*
	STUDENT PAPERS
10:30	Aleksandra Pazova Factors involved in Modernization Decision Making
11:00	Sarah Augustinowski Entwicklung eines Conversational Interface zur Steuerung von SEE und Abfrage von Graphdatenbanken
11:30	Leon Ehrhardt Ein Vergleich von Attract-Funktionen in SEE
12:00	*Mittagessen*
	LARGE LANGUAGE MODELS
13:30	Jochen Quante, Jesko Hecking-Harbusch and Matthias Woehrle C to Rust Translation via Large Language Models
14:00	Christian Malek, Vasil Tenev and Martin Becker KI-basierte Erstellung von digitalen Zwillingen
14:30	STUDENT PAPER AWARDS
15:00	*Social Event - festes Schuhwerk wird empfohlen*
18:00	*Conference Dinner*

Freitag, 11.4.2025

	FACHGRUPPENSITZUNG
09:00	Fachgruppensitzung der GI Fachgruppe Software-Reengineering mit Wahl des Leitungsgremiums
	REQUIREMENTS AND TRACING
10:00	Sandra Greiner Experiences with Combining Proactive with Retroactive Feature Traces
10:30	Harry Sneed Fallstudie in Requirements Reengineering
11:00	*Kaffeepause*
	LOW CODE MIGRATION
11:30	Andreas Schmietendorf, Ben Rymar, Sandro Hartenstein Reengineering einer bestehenden High-Code KI-Applikation zu einer Low-Code Lösung
12:00	Marco Konersmann A Use Case and Method for Migrating a Low Code Business Information System to a Custom Three Tier Application
12:30	CLOSING
ab 12:35	*Mittagessen*

27. Workshop Software-Reengineering und -Evolution

der GI-Fachgruppe Software Reengineering (SRE)

Herzlich Willkommen zum 27. WSRE! Wir freuen uns, dass sich wieder viele Interessierte rund um das Thema Software-Reengineering in Bad Honnef zusammengefunden haben.

Der WSRE (damals noch WSR) wurde 1999 von Jürgen Ebert und Franz Lehner ins Leben gerufen. Ziel war es damals, ein deutschsprachiges Diskussionsforum zu allen Aspekten rund um das Thema Reengineering zu schaffen. Durch aktive und gewachsene Beteiligung vieler Personen aus Forschung und Praxis hat sich der WSRE als zentrale Reengineering-Konferenz im deutschsprachigen Raum etabliert. Viele Teilnehmer*innen haben ihn als Student*in oder Doktorand*in kennen und schätzen gelernt und bleiben ihm auch in ihrem späteren Berufsleben treu. Dabei wird der Workshop weiterhin als Low-Cost-Workshop ohne eigenes Budget durchgeführt. Bitte tragen Sie dazu bei, den WSRE weiterhin erfolgreich zu machen, indem Sie interessierte Kolleg*innen und Bekannte darauf hinweisen.

Auf Basis der erfolgreichen WSR-Treffen der ersten Jahre wurde 2004 die GI-Fachgruppe Software-Reengineering (https://fg-sre.gi.de/) gegründet, deren Leitungsgremium den WSRE seitdem organisiert und auch bei anderen Aktivitäten rund um das Thema Reengineering mitwirkt. Als GI-Mitglied laden wir Sie ein, der Fachgruppe beizutreten, um dieses Thema zu stärken.

Die Themen des WSRE erstrecken sich auf die Bereiche Software-Reengineering, Software-Wartung und -Evolution. Darunter verstehen wir prinzipiell alle Aktivitäten rund um die Analyse, Bewertung, Visualisierung, Verbesserung, Migration und Weiterentwicklung von Software-Systemen. Im Vordergrund steht der Austausch zwischen Interessierten, insbesondere auch der Austausch zwischen Forschung und Praxis. Aus dem WSRE sind in den vergangenen 27 Jahren viele Kooperationen zwischen Forscher*innen, zwischen Forscher*innen und Praktiker*innen, aber auch unter Praktiker*innen hervorgegangen. Viele Beiträge des WSRE erzählen von diesen Erfolgsgeschichten.

Beim diesjährigen WSRE sind folgende Programmpunkte vorgesehen:

- **Keynote**: Eine Keynote von Prof. Dr. Sven Apel von der Universität des Saarlands zum Thema „AI4SE: Adventures in the Promised Land“.
- **Vorträge**: Einblick in sowie Rückblick und Ausblick auf interessante Arbeiten und Ergebnisse rund ums Software-Reengineering.
- **Best Student Paper Award**: Wir prämieren den besten studentischen Beitrag (Paper und Vortrag), bewertet durch eine Jury aus Wissenschaft und Praxis.
- **Tool-Demos**: Erleben Sie neuartige Reengineering-Tools live in Aktion.
- **Fachgruppensitzung** der GI-Fachgruppe Software-Reengineering mit Neuwahl des Leitungsgremiums.
- **Networking**: Vernetzung zwischen den Teilnehmenden in den Pausen und beim gemütlichen Zusammensein am Abend.
- **Social Event**: Wir werden zusammen eine Wanderung zum Drachenfels unternehmen und hoffen auf gutes Wetter.

Die Organisatoren danken allen Beitragenden für ihr Engagement – insbesondere den Autor*innen, den Vortragenden sowie den Teilnehmer*innen und Juroren des Best Student Paper Awards und allen Workshop-Teilnehmer*innen für die lebhaften, kontroversen und interessanten Diskussionen. Vielen Dank auch an das Team des Physikzentrums Bad Honnef, das im Hintergrund dafür sorgt, dass wir hier drei angenehme Tage verbringen können. Insbesondere danken wir auch den Sponsoren des Best Student Paper Awards: Delta Software Technology GmbH, Schmallenberg und S&N Invent GmbH, Paderborn, die es durch ihre finanzielle Unterstützung ermöglichen, den Finalist*innen einen Reisekostenzuschuss zu gewähren und für den besten Beitrag ein Preisgeld auszuloben.

Für die Fachgruppe Software-Reengineering:

Dr.-Ing. Jochen Quante
Bosch Research, Renningen (Sprecher)

Dr. Marco Konersmann
ista SE, Essen

Prof. Dr. Stefan Sauer
Universität Paderborn

Dr. Daniela Schilling
Delta Software Technology, Schmallenberg

Prof. Dr.-Ing. Sandro Schulze
Hochschule Anhalt (Köthen)

Einen Big Ball of Mud analysieren

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

Abstract

Das saubere Design einer Anwendung geht über die Jahrzehnte oft verloren. Es entstehen Anwendungen bei denen Designvorgaben nicht eingehalten werden. Solche Anwendungen sind schwer zu warten und noch schwerer zu modernisieren. AMELIO Logic Discovery hilft Designverletzungen in einem iterativen Prozess zu ermitteln.

1 Vom sauberen Design zum Big Ball of Mud

Wie konnte es nur dazu kommen? Als die heute sog. Legacy-Anwendungen entwickelt wurden, gab es ein sauberes Design. Es wurde z.B. festgelegt welche Domänen oder Teilanwendungen existieren, welche Programme zu welcher Domäne gehören und wie die Kommunikation zwischen den Domänen erfolgen soll. Seitdem sind jedoch einige Jahrzehnte vergangen. Die Anwendungen wurden von Entwickler zu Entwickler weitergereicht, sie wurden angepasst, erweitert und überarbeitet, neue Teilanwendungen sind hinzugekommen und das oft unter Zeitdruck. Das saubere Design ist verloren gegangen. Die Kommunikation ist unstrukturiert und Domänen greifen direkt auf Artefakte zu, die im Zuständigkeitsbereich anderer Domänen liegen. Oft sind die Kommunikationswege nicht dokumentiert und auch nicht bekannt. Kurz es ist ein Big Ball of Mud entstanden.

Wenn aber unklar ist, wer auf bestimmte Artefakte zugreift, dann sind unerwünschte Nebeneffekte bei Änderungen vorprogrammiert. Ebenso wird es bei Modernisierungen schwer Teilanwendungen oder Domänen durch neugeschriebene oder Standard-Software zu ersetzen. Im schlimmsten Fall treten resultierende Fehler erst zur Laufzeit auf.

Für eine sichere und effiziente Wartung oder Modernisierung ist es deshalb wichtig die Domänen, ihre Schnittstellen und die zulässigen Interaktionen zu definieren und zu analysieren an welchen Stellen es zu Verletzungen des Designs kommt. Diese Analyse kann durch AMELIO Logic Discovery automatisiert durchgeführt werden.

2 Iterativ Designverletzungen erkennen

AMELIO arbeitet sowohl modell- als auch regelbasiert. Zunächst werden alle Sourcen der Anwendung in Modelle überführt. Auf diesen Modellen können dann die Regeln zur Analyse der Designverletzung ausgeführt werden. Diese Regeln können z.B. beschreiben welche Kommunikation zwischen Programmen zulässig ist, welche Einschränkung es bei der Verwendung von Copybooks gibt oder wer wie auf welche Datenbank oder welches File zugreifen darf.

Das Regelwerk ist kundenspezifisch und wird in einem iterativen Prozess gemeinsam mit den Kunden festgelegt. Dabei werden zunächst die einfachen, groben Regeln spezfiziert und dann immer weiter verfeinert.

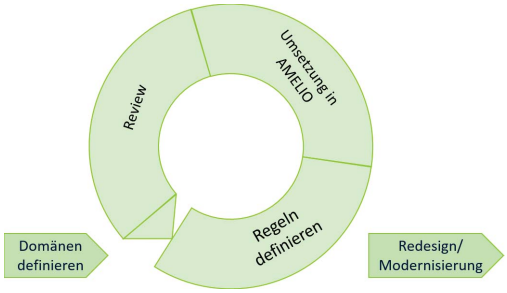


Figure 1: Iterativ Designverletzungen erkennen

2.1 Wer interagiert wann und wie mit wem?

Zunächst gilt es festzulegen, welche Domänen existieren und welches Programm zu welcher Domäne gehört. Die Zuweisung der Programme zu den Domänen kann über Regeln beschrieben werden, oft z.B. anhand von Programmnamen. Im nächsten Schritt ermittelt AMELIO dann die Programmbäume, also welches Programm ruft welches andere auf. Programme, die keinen Aufrufer haben, sind potentielle Einstiegsprogramme. Für alle ermittelten Programmaufrufe prüft AMELIO, ob die beteiligten Programme der gleichen Domäne angehören oder nicht.

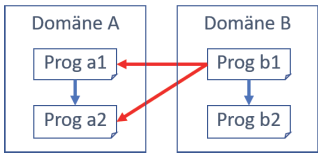


Figure 2: Kommunikation über Domänengrenzen (rot)

2.1.1 Darf der das?

AMELIO geht erstmal davon aus, dass eine Kommunikation zwischen den Domänen nicht zulässig ist. Deshalb gilt es nun zu definieren, welche Arten von Interaktion über Domänengrenzen hinweg zulässig sind

- Serviceprogramme: das Programm ist ein Service über den Programme aus anderen Domänen Daten anfordern oder schreiben können. Das Programm kann als ganzes als Service definiert werden oder einige Funktionen daraus.
- Programmpaare: es wird festgelegt, dass bestimmte Programmpaare aus verschiedenen Domänen miteinander interagieren dürfen
- Weitere Regeln können in folgenden Iterationsschritten definiert werden.

AMELIO nutzt diese Regeln, um die Interaktionen zwischen den Domänen zu bewerten und kritische Interaktionen gesondert darzustellen.

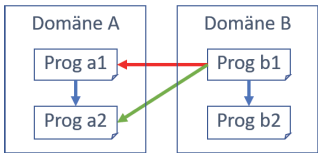


Figure 3: Kommunikation über Domänengrenzen (rot verboten, grün erlaubt)

2.2 Copybooks

Für die Modernisierung einer Anwendung kann es notwendig sein auch die Verwendung von Copybooks zu analysieren und zu reorganisieren. Dazu ermittelt AMELIO zunächst welche Copybooks von den einzelnen Programmen verwendet werden. Als nächstes wird analysiert, ob ein Copybook von Programmen aus verschiedenen Domänen verwendet wird. Ist das der Fall, wird für jede Zeile des Copybooks ermittelt in welchen Domänen sie benötigt wird.

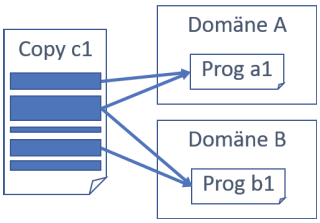


Figure 4: Copybook mit Verwendung in unterschiedlichen Domänen

Gibt es Copybooks, die von Programmen aus unterschiedlichen Domänen verwendet werden, so gilt es im REview-Schritt zu definieren, welche Verwendungen zulässig sind und welche nicht.

2.3 Datendiebe – auf frischer Tat ertappt

Im ursprünglichen sauberen Design war auch genau festgelegt wer lesend oder schreibend auf eine Datenbank zugreifen darf. Bei genauerer Betrachtung stellt man jedoch fest, dass es heute den ein oder anderen Datendieb in der Anwendung gibt. Damit sind Programme gemeint, die auf Datenbanken zugreifen, auf die sie eigentlich keinen (direkten) Zugriff haben dürften. Auch solche Datendiebe lassen sich automatisiert ermitteln.

Die Datenbanken werden den verschiedenen Domänen zugewiesen. Es gilt die Regel: auf eine Datenbank dürfen nur Programme aus der gleichen Domäne zugreifen. AMELIO ermittelt für alle Datenbanken, welche Programme lesend oder schreibend auf die Datenbanken mit ihren jeweiligen Spalten oder Segmenten zugreifen. Zugriffe aus einer anderen Domäne werden entsprechend hervorgehoben.

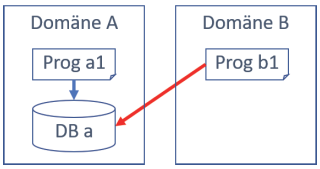


Figure 5: Datenbankzugriff über Domänengrenzen (rot)

3 Einen Big Ball of Mud analysieren

AMELIO Logic Discovery kann Interaktionen und Abhängigkeiten in einer Anwendung analysieren. Mittels eines interativen Prozesses kann zudem analysiert werden, ob diese Designvorgaben verletzen. Dies ist die Grundlage um die Anwendungen effizient und erfolgreich zu warten, refaktorisieren oder modernisieren.

Reflexionsanalyse mit SEE (Tool-Demo)

Michel Krause, Rainer Koschke, AG Softwaretechnik, Universität Bremen

I. Einführung

Die Architektur ist für eine Software von fundamentaler Bedeutung in allen Belangen. Daher ist es wichtig, regelmäßig zu prüfen, ob Implementierung und Architektur konform sind. Hierzu müssen meist verschiedene Stakeholder zusammenkommen. Da diese heutzutage zunehmend nicht alle am selben Ort arbeiten, bietet SEE *(Software Engineering Experience)* eine Plattform, um die notwendigen Analysen kollaborativ aus der Ferne und auf höherem Abstraktionsniveau durchzuführen.

In diesem Beitrag beschreiben wir den aktuellen Stand von SEE zur Architekturmodellierung und -verifikation[1]. SEE ist eine kollaborative Software-Visualisierungsplattform, die speziell für verteilte Entwicklerteams entwickelt wird und zur Analyse von Software dient. Sie ermöglicht neben der Architekturprüfung auch die Untersuchung der Evolution, der inneren Qualität und der Ausführung von Software. Die zu untersuchende Software wird als dreidimensionale Code-City in einem virtuellen Raum visualisiert. Benutzer können diesem Raum mit verschiedenen Endgeräten beitreten, wobei sie durch Avatare repräsentiert werden und mittels Sprache und Gesten kommunizieren können.

II. Reflexionsanalyse

Murphys [1] Reflexionsanalyse ist eine etablierte Methode, die Lücke zwischen abstrakten Architekturmodellen und dem tatsächlichen Quellcode von Softwaresystemen zu überbrücken. Dieser Ansatz hilft Softwareingenieuren, ihre Design-Absichten mit dem implementierten Code zu vergleichen, um Übereinstimmungen und Diskrepanzen zu identifizieren. Der Prozess umfasst folgende Schritte:

1. Architekten erstellen ein Architekturmodell, das ihr Verständnis der Systemarchitektur repräsentiert.
2. Architekten ordnen den Quellcode dem Architekturmodell zu, um zu spezifizieren, welche Elemente des Quellcodes welche Teile der Architektur implementieren.
3. Eine automatisierte Analyse erzeugt ein Reflexionsmodell, das beschreibt, wo das Archiktekturmodell und der Quellcode übereinstimmen und wo sie sich unterscheiden.

[1] `https://youtu.be/EZI4nyovbbs`

Übereinstimmungen werden hierbei als *Konvergenzen* bezeichnet. Eine Abhängigkeit in der Implementierung, die nicht durch eine spezifizierte Abhängigkeit gedeckt ist, nennt man *Divergenz*. Eine spezifizierte Abhängigkeit, die in der Implementierung keine konvergente Abhängigkeit besitzt, ist eine *Absenz*. Ein *Reflexionsmodell* ist somit die Summe der Konvergenzen, Absenzen und Divergenzen.

III. Reflexionsanalyse in SEE

Im Folgenden werden die wesentlichen Neuerungen der SEE-Visualisierung erläutert, die darauf abzielen, die Reflexionsanalyse zu unterstützen.

Sowohl die Architektur als auch die Implementierung werden nebeneinander als Code-Cities auf einem virtuellen Tisch präsentiert, um den herum sich die Teilnehmer gruppieren können. Die Daten für die Code-City der Implementierung werden mit einer statischen Analyse aus dem Quelltext gewonnen und dann durch ein automatisches hierarchisches Layout angeordnet (wahlweise als Tree-Map, Circular-Packing, Balloon-Layout, Rectangular-Packing oder als EvoStreets). Nutzer können beliebige Metriken auswählen, um die drei Dimensionen sowie die Farbe der als 3D-Objekte dargestellten Code-Komponenten zu bestimmen. Abhängigkeiten zwischen Code-Teilen werden als hierarchisch gebündelte Kanten visualisiert, die jedoch nur nach Bedarf beim Hovering über die 3D-Objekte eingeblendet werden, um die Darstellung übersichtlich zu halten.

Schritt 1: Erstellen eines Architekturmodells Das Architekturmodell wird entweder in einem externen Tool erstellt und dann in SEE importiert oder aber in SEE selbst erzeugt. In der Regel werden die Architekten die Objekte nach semantischen Kriterien selbst manuell anordnen, aber sie können auch über die in SEE zur Verfügung stehenden automatischen Layouts platziert werden. Ebenso wird die Größe und Farbe der Architekturobjekte zumeist manuell statt mittels Metriken festgelegt. In der Regel besitzen die Architekturobjekte die gleiche Höhe und sind völlig flach. Kanten verbinden diese Flächen, um die erwarteten Architekturabhängigkeiten auszudrücken. Auch die Architekturkomponenten können hierarchisch gegliedert sein. Die Teil-von-Beziehung wird dabei durch den räumlichen Einschluss ausgedrückt. Dadurch ergibt sich letztlich eine Art UML-Paketdiagramm. Abbildung 1 zeigt

einen Nutzer, der die Architektur eigenständig erstellt und dabei die Größe der Komponente *Main* anpasst. Dabei wird sichergestellt, dass sich Komponenten nicht überlappen, wenn sich ihre Größe verändert. Überlappung – nicht nur bei der Skalierung sondern auch beim Bewegen – muss vermieden werden, weil die Teil-Von-Beziehung durch den räumlichen Einschluss ausgedrückt wird. Überlappung wird von SEE zurückgewiesen, indem die manipulierte Komponente wieder ihre ursprüngliche Größe bzw. ihren ursprünglichen Ort erhält. Wenn hingegen eine Komponente vollständig von einer anderen Komponente umschlossen wird, wird im zugrunde liegenden Datenmodell die Teil-Von-Beziehung angepasst.

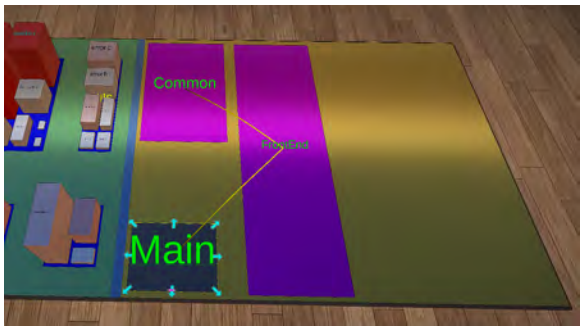


Abb. 1: Erstellung des Architekturmodells

Schritt 2: Abbildung der Implementierung auf die Architektur Die einheitliche Darstellung der Implementierung und Architektur ermöglicht die intuitive Abbildung der Implementierung auf die Architektur, indem die Nutzer per Drag-and-Drop die 3D-Objekte aus der Implementierung auf die Architekturkomponenten verschieben (siehe Abb. 2). Werden Implementierungskomponenten in Architekturkomponenten eingebettet, stellt dies eine Abbildung dar und keine Teil-Von-Beziehung wie beim Enthaltensein von Komponenten innerhalb derselben Domäne (Code bzw. Architektur). Anhand dieser Abbildung berechnet eine inkrementelle Reflexionsanalyse [2] im Hintergrund die Veränderungen des Reflexionsmodells, die sich durch die neue Abbildung ergeben.

Schritt 3: Darstellung des Reflexionsmodells Initial sind alle modellierten Architekturabhängigkeiten Absenzen, da auf die Architekturkomponenten noch keine Code-Elemente abgebildet wurden. Der Ampel-Metapher folgend werden diese als gelbe Kanten dargestellt. Wird eine Absenz zu einer Konvergenz, weil auf die beiden Enden der Kante Implementierungskomponenten abgebildet wurden, die mindestens eine passende Abhängigkeit haben, werden aus den gelben Kanten grüne Kanten. Dabei werden alle durch den zuletzt getätigten Abbildungsschritt geänderten Kanten mit einem Leuchteffekt versehen, damit sie sich besser von den auch schon zuvor existierenden Kanten unterscheiden. Damit werden Veränderungen direkt sichtbar. Werden Implementierungskomponenten abgebildet, die Abhängigkeiten mit sich bringen, die aufgrund der Architektur gar nicht erlaubt sind, werden diese rot dargestellt.

Nutzer müssen dabei gar nicht erst die von ihnen gerade bewegten Implementierungskomponenten auf den Architekturkomponenten ablegen. Es genügt, wenn sie über eine Architekturkomponente hovern, um das Reflexionsmodell zu aktualisieren. Dies erlaubt schnelle What-if-Analysen, die die Auswirkungen einer möglichen Abbildung unmittelbar anzeigen. Erst wenn die bewegten Objekte auch wirklich abgelegt werden, wird die Abbildung finalisiert.

Zur Unterstützung der Abbildung bietet SEE eine automatisierte Empfehlung [3], wohin eine Implementierungskomponente abgebildet werden könnte. Mögliche Ziele werden farbig hervorgehoben. Zur Berechnung dieser Vorschläge wurden in der Literatur bereits mehrere Vorschläge unterbreitet, die die Empfehlung anhand existierender Abhängigkeiten bereits abgebildeter Komponenten und/oder anhand der Übereinstimmung der Terme (Bezeichner, Kommentare), die in diesen Komponenten auftreten, ermitteln.

Abb. 2: Abbildungsschritt

IV. Zusammenfassung

In dieser Tool-Demo stellen wir vor, wie wir SEE erweitert haben, um die Reflexionsanalyse zu ermöglichen. Die Umsetzung haben wir in einer Nutzerstudie evaluiert [4].

Literaturverzeichnis

[1] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, 1995.

[2] Rainer Koschke. Incremental reflexion analysis. *Journal on Software Maintenance and Evolution*, 25(6):601–637, 2013.

[3] Leon Ehrhardt and Rainer Koschke. Comparing attract functions for the reflexion analysis regarding the usage of dependencies and words. In *Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon)*, 2025.

[4] Leon Ehrhardt and Rainer Koschke. A controlled experiment on the usability of automated reflexion mapping suggestions integrated in code cities. In *Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon)*, 2025.

Reflexionsanalyse in SEE: ein kontrolliertes Experiment

Leon Ehrhardt, Rainer Koschke, AG Softwaretechnik, Universität Bremen

I. Einführung

In diesem Beitrag stellen wir unser kontrolliertes Experiment vor, in dem wir die Umsetzung der Reflexionsanalyse [6] in unserer Visualisierungsplattform SEE (*Software Engineering Experience*) evaluiert haben. Die Reflexionsanalyse ermöglicht es, eine Implementierung gegen eine Architekturvorgabe zu verifizieren. Hierzu ist es notwendig, Komponenten aus der Implementierung auf Komponenten der Architektur abzubilden. Dieser Schritt wird in der Regel von Menschen durchgeführt und benötigt einiges an Zeit. Verschiedene Forscher – unter anderem auch wir selbst – haben automatisierte Verfahren entwickelt, die den Menschen Vorschläge unterbreiten, wohin eine gegebene Implementierungskomponente abgebildet werden kann [3]. In SEE sind alle Schritte der Reflexionsanalyse umgesetzt (Architekturmodellierung, Abbildung, Berechnung der Übereinstimmungen und Unterschiede zwischen Architektur und Implementierung). Die Umsetzung dieser Schritte – inklusive der Unterbreitung automatisierter Vorschläge – wird in unserem anderen Beitrag zum WSRE mit einer Tool-Demo genauer vorgestellt [5].

In diesem Beitrag beschreiben wir ein kontrolliertes Experiment, mit dem wir folgende Hypothesen im Zusammenhang der Reflexionsanalyse in SEE untersucht haben:

H_A: Der subjektiv wahrgenommene kognitive Aufwand von Anwendern im Abbildungsschritt der Reflexionsanalyse ist geringer, wenn automatisierte Empfehlungen unterbreitet werden.

H_D: Die Dauer des Abbildungsschritts ist kürzer, wenn automatisierte Empfehlungen unterbreitet werden.

Ferner evaluieren wir die Usability des Abbildungsschritts der Reflexionsanalyse in SEE.

II. Design des Experiments

Ziel unseres Experiments ist es, die Nullhypothesen zu H_A und H_D zu falsifizieren. Zunächst operationalisieren wir unsere Hypothesen durch die im Folgenden beschriebenen abhängigen Variablen, die wir im Experiment gemessen haben.

Abhängige Variablen Die Dauer für den Abbildungsschritt aus H_D lässt sich offensichtlich als die Zeit messen, die zwischen dem Beginn und dem Ende des Abbildungsschritts vergangen ist. Das Konzept des „kognitiven Aufwands“ ist ungleich schwieriger zu fassen. Wir operationalisieren dieses Konzept indirekt mit dem *Raw NASA TLX*. Der NASA Task Load Index (NASA-TLX) [7] ist ein etablierter Index in der Mensch-Computer-Interaktion [1], auch wenn er ursprünglich nicht für Software entwickelt wurde. Dieser Index wird mit Hilfe eines Fragebogen implementiert, der aus sechs Fragen besteht, in denen eine Person nach den mentalen, physischen und zeitlichen Anforderungen, die an sie gestellt wurden, und nach der Interaktion der Person mit der Aufgabe (wahrgenommene Anstrengung, Leistung und Frustration) gefragt wird. Jede Frage wird auf einer mehrstufigen Ordinalskala beantwortet, die von *sehr niedrig* (0) bis *sehr hoch* (100) reicht [7]. Niedrigere Werte sind besser. Die Werte für die verschiedenen Aspekte werden als gewichteter Durchschnitt zu einem einzigen Wert kombiniert. Der *Raw NASA-TLX* ist eine Vereinfachung des *NASA-TLX*, in der alle Aspekte das gleiche Gewicht haben.

Die wahrgenommene Usability messen wir mit dem *System Usability Scale (SUS)* [2]. SUS basiert auf einem Fragebogen mit zehn verschiedenen Fragen, die auf einer fünfstufigen Likert-Skala von „stimme überhaupt nicht zu“ (0) bis „stimme voll zu“ (4) beantwortet werden. Die Angaben werden aufaddiert und mit dem Faktor 2,5 multipliziert, so dass sich ein Wert zwischen 0 und 100 ergibt. Der obere Wert entspricht der maximalen Usability.

Eine ausführliche Begründung für die Entscheidung für den *Raw NASA-TLX* und SUS findet der geneigte Leser in einer anderen Publikation von uns [4].

Aufgaben Die Teilnehmende mussten eine partielle Abbildung eines realen Systems ergänzen. Wir haben hierzu das Java-Projekt *PetClinic* ausgewählt, das die Verwendung des Frameworks *Spring* für Web-Applikationen illustriert. Seine Implementierung besteht aus 47 Java-Dateien und ist damit weder trivial noch überfordernd. Die Anwendung folgt einer klassischen Web-Architektur, die aus Controllern, Logik, Modellobjekten und einer Persistenzschicht besteht. Die entsprechende Architektur haben wir einer Präsentation der Entwickler zu *PetClinic* entnommen. Die Entscheidung für *PetClinic* folgte auch unserer Annahme, dass den meisten Entwicklerinnen und Entwicklern solche Architekturen für Web-Applikationen geläufig sein sollten. Wir haben die Architektur noch um eine Komponente *Utils* für allgemeine, wiederverwendbare Klassen ergänzt, die im ursprünglichen Architekturmodell nicht vorgesehen war. Die Teilnehmenden hatten die Aufgabe, von uns vorgegebene

Java-Klassen auf die genannten Architekturkomponenten passend abzubilden. In der Experimentalgruppe erhielten die Teilnehmenden von SEE Vorschläge, wohin eine Klasse abgebildet werden kann. In der Kontrollgruppe gab es keine solche Vorschläge. Um einerseits mehr Datenpunkte zu erhalten und andererseits auch einen intrapersonellen Vergleich anstellen zu können, war jeder Teilnehmer Teil beider Gruppen. Um offensichtlichen Lerneffekten entgegenzuwirken, unterschieden sich die Klassen, die es abzubilden galt, vom einen zum nächsten Durchgang. Zudem haben wir die Reihenfolge variiert. Eine Hälfte der Teilnehmenden begann mit automatisierten Vorschlägen und musste dann im zweiten Durchgang ohne auskommen. In der anderen Hälfte verhielt es sich umgekehrt. In beiden Durchgängen waren es jeweils sieben Klassen, die abgebildet werden sollten. In beiden Durchgängen waren die in der Architektur vorkommenden Zielkomponenten gleich (zweimal Controller, zweimal Modellobjekte, zweimal Persistenz und einmal *Utils*). Die korrespondierenden Klassen zwischen den beiden Durchgängen waren in ihren Eigenschaften vergleichbar.

Teilnehmer An unserer Studie haben 13 Männer im Alter zwischen 24 und 36 Jahren teilgenommen, die wir mittels Convenience-Sampling gewonnen haben. Dazu zählten sowohl Studenten als auch professionelle Entwickler. Alle Teilnehmer waren mit Java vertraut. Vier hatten bereits Vorerfahrungen mit Spring. Fünf hatten SEE in der Vergangenheit schon einmal benutzt. Davon hatten vier Teilnehmer zuvor an SEE mitentwickelt, aber keiner davon an der Reflexionsanalyse.

III. Resultate

Im Folgenden vergleichen wir jeweils die Werte zweier Stichproben. Die Stichprobe E (Experimentalgruppe) umfasst alle Datenpunkte, bei denen automatisierte Vorschläge unterbreitet wurden. Die Stichprobe K (Kontrollgruppe) umfasst alle Datenpunkte, bei denen keine automatisierten Vorschläge gegeben wurden. Die beiden Stichproben sind nicht voneinander unabhängig, weil alle Teilnehmer zu beiden Stichproben beigetragen haben. Aus diesem Grund verwenden wir als statistische Tests stets solche, die einen über denselben Teilnehmer gepaarten Vergleich vornehmen. Außerdem verwenden wir ausschließlich nichtparametrische Tests, weil wir keine Annahmen über die zugrundeliegende Verteilung der Gesamtpopulation machen können und bei unserer Stichprobengröße auch der zentrale Grenzwertsatz nicht greift. Zudem sind alle Tests für ordinalskalierte Daten geeignet. Auch wenn die beiden Indizes *Raw NASA-TLX* und SUS Angaben auf den Ordinalskalen als addierbare Zahlenwerte interpretieren, sind sie letztlich aus messtheoretischer Sicht eigentlich nur ordinalskaliert.

Raw NASA-TLX Der *Raw NASA-TLX* für E lag bei 35,75 (±18,42) und für K bei 43,02 (±18,74). Meta-Analysen zu Referenzwerten für den *Raw NASA-TLX* legen einen Wert von 45 als Durchschnitt nahe. Der Wert für E liegt deutlich darunter, wobei weniger besser ist.

Für den Vergleich des *Raw NASA-TLX* haben wir den asymptotischen gepaarten Fisher-Pitman Permutationstest verwendet, der einen p-Wert von 0,05208 liefert, was sehr nahe an dem häufig in der Literatur verwendeten Signifikanzniveaus von 0,05 liegt. Wir überlassen es dem werten Leser die Entscheidung zu treffen, ob ein p-Wert von 0,05208 es rechtfertigt, die Nullhypothese abzulehnen und unsere Alternativhypothese H_A zu akzeptieren. Es wird auch in der Fachwelt kritisch darüber diskutiert, welches Signifikanzniveau für derlei Experimente angemessen ist.

Dauer Die Teilnehmer benötigten im Durchschnitt 8 min und 40 sek, wenn automatisierte Vorschläge gemacht wurden, und sonst 9 min und 33 sek. Der Test nach Wilcoxon liefert einen p-Wert von etwa 0,3, so dass die Nullhypothese nicht verworfen werden kann.

SUS Insgesamt ergab sich ein durchschnittlicher SUS-Wert von 67,5 (±21,41). Studien in der wissenschaftlichen Literatur interpretieren 68 als eine durchschnittliche Usability.

IV. Fazit

Unsere Studie legt nahe, dass die automatisierten Vorschläge einen Vorteil beim kognitiven Aufwand und in der Dauer bringen können, letzteres aber definitiv nicht statistisch signifikant. Die Usability unserer Lösung ist nur durchschnittlich. Die Teilnehmer haben uns konkrete Hinweise gegeben, wie man sie verbessern kann.

Literaturverzeichnis

[1] Aaron Bangor, Philip T Kortum, and James T Miller. An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, July 2008.

[2] John Brooke. *Usability Evaluation in Industry*, chapter SUS: A quick and dirty usability scale. Taylor and Francis, 1996.

[3] Leon Ehrhardt and Rainer Koschke. Comparing attract functions for the reflexion analysis regarding the usage of dependencies and words. In *Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon)*, 2025.

[4] Leon Ehrhardt and Rainer Koschke. A controlled experiment on the usability of automated reflexion mapping suggestions integrated in code cities. In *Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon)*, 2025.

[5] Michel Krause and Rainer Koschke. Architekturmodellierung und -verifikation mit SEE (Tool-Demo). In *Workshop Software-Reengineering & Evolution (WSRE)*, 2025.

[6] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, 1995.

[7] NASA. *Task Load Index (NASA-TLX)*. NASA Ames Research Cener, 1.0 edition, 1998.

AI4SE: Adventures in the Promised Land

Prof. Dr. Sven Apel

Chair of Software Engineering
Saarland University
Saarbrücken, Germany

AI is about to revolutionize how we develop software. In particular, large language models have shown great promise in assisting programmers in writing and reasoning about source code. In this talk, I will take a step back and reflect on the rich history of AI in software engineering and the manifold ways in which AI can benefit software developers beyond code generation. In particular, I will share stories on our own adventures in this promised land, as some see it. This includes experiences with using AI for software performance modeling and software model completion as well as discussions on what we can learn from other disciplines in this endeavor.

Teaching Software Quality with Adaptive Source Code Feedback

Rahel Sundermann[1], Sebastian Krieter[2], Birte Glimm[1], and Thomas Thüm[2]

[1]University of Ulm, {rahel.sundermann, birte.glimm}@uni-ulm.de
[2]TU Braunschweig, {sebastian.krieter, thomas.thuem}@tu-braunschweig.de

Abstract

One of the necessary skills to become a good programmer is writing code which is not only correct but also maintainable. The practice of adhering to coding conventions should be introduced early in the learning process. However, students in computer science and related areas often come with different levels of prior coding experience, but have acquired this skill only partially in passing. Thus, the students require individualized feedback on code quality, which is challenging for teachers to provide. To provide valuable feedback to students with different skill levels, we developed a methodology to give adaptive feedback, tailored to the students skill level. We implemented the methodology in a tool, which provides the adaptive feedback for a given source code. In this paper, we present the tool and evaluate its effectiveness through an experiment conducted in programming courses at two universities. Our user study leads to the conclusion that filtered feedback does not significantly help students improve the quality of their source code.

1 Motivation

For source code to be easily maintainable and extendable, it should be comprehensible [2]. To this end, coding conventions and best practices define how source code should be written and structured [3, 8]. If every programmer follows the same rules, it eases the understanding of foreign code and makes maintenance less error prone [1, 6, 4].

We argue that learning coding conventions and best practices should be part of education, as bad habits are difficult to correct [5]. Achieving this requires feedback not only on code correctness but also on code quality. Providing individual feedback tailored to each student's skill level is typically not manageable for teachers.

In this paper, we introduce a tool that provides automated and individualized feedback on code quality for regular programming tasks in teaching. The given feedback is adaptive, in terms of number of reported mistakes and level of detail. It depends on the number of mistakes and the programming expertise of the student. The feedback tested in our user study varies in the number of errors being reported.

2 Feedback on Code Quality

In order to integrate feedback on code quality as seamlessly as possible into existing teaching infrastructure for typical tasks within a programming course, we opted to extend the learning platform *Moodle*. Our implementation consists of a Java back-end and a Moodle plug-in serving as a front-end user interface. In particular, we use the Virtual Programming Lab (VPL) plug-in,[1] which provides a source code editor within a Moodle task and allows us to communicate with our analysis back-end [7]. As back-end, we use a Java program, running in a Docker[2] environment, which receives the source code from the VPL plug-in and sends analysis results back, which are displayed directly within the VPL source code editor.

In detail, the Java program executes the following steps: First, it compiles the received source code. In our current implementation, we focus on Java source code as input and we use `javac` as a compiler. If any errors occur during this step, the execution stops and the errors are displayed to the user in the editor. Second, the existing analysis tools `pmd`,[3] `spotbugs`,[4] and `checkstyle`[5] are applied to the compiled code. Third, we combine the results from the three tools (i.e., merging problems that were reported more than once), format the detected problems into a user-friendly text, and display the problems to the user in the editor. This step allows us to customize the feedback to the user, e.g., by providing more or less detail on the detected problems or by controlling the number of problems displayed simultaneously.

3 Evaluation

In our evaluation, we focus on the amount of feedback given to the users. In particular, we are interested in understanding whether displaying only the most critical problems instead of everything detected is beneficial to users, as it may help them to focus on the most problematic parts of the source code first. As our current implementation builds upon an existing Moodle plug-in, we did not analyze the evaluation of

[1] `https://moodle.org/plugins/mod_vpl`
[2] `https://www.docker.com`
[3] `https://pmd.github.io/`
[4] `https://spotbugs.github.io/`
[5] `https://checkstyle.sourceforge.io/`

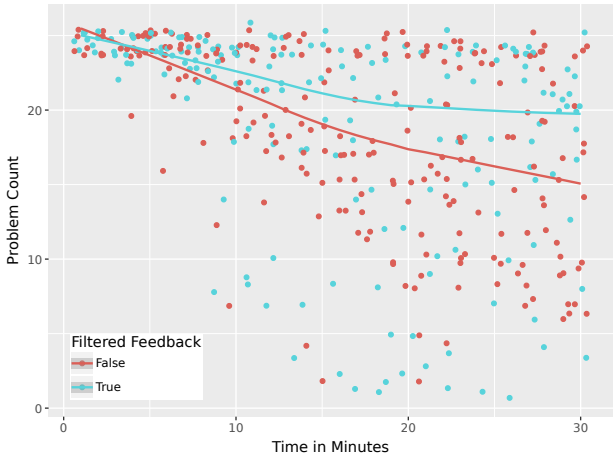


Figure 1: Number of problems in the example programs per user per minute.

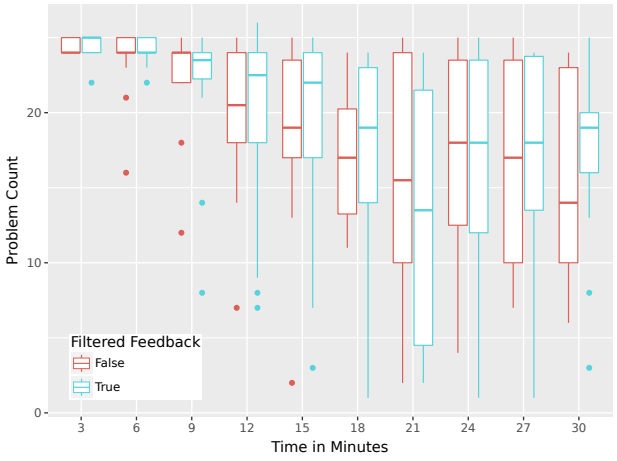


Figure 2: Number of problems in the example programs. Grouped into time intervals of 3 minutes.

the usability of the tool itself, but only of the feedback it provides. In detail, we try to answer the following research questions:

- RQ_1 Does adaptive feedback support students in improving software quality faster?
- RQ_2 Does using adaptive feedback result in better programs?
- RQ_3 How is the self-assessment of the students regarding the helpfulness of the tool?

To answer our research questions, we carried out an experiment to investigate whether filtered feedback, compared to unfiltered feedback, supports students in improving source code quality. To this end, we integrated our plug-in into Moodle and used the Virtual Programming Lab plug-in as the source code editor [7]. The material and the results of our evaluation are publicly available.[6]

Experiment Design Using our tool, the students worked on the code of two small programs. Each program had two failing unit tests and contained a number of quality problems (24 in the first and 25 in the second task) that were detected by our tool and not the cause of the failing tests. Each student had 30 minutes per task to fix the code and improve the overall code quality. At every point during the experiment each student could use a button within the editor to evaluate their code (i.e., check wether they fixed the failing test cases and receive a list of quality problems in their code with respective explanations). For each reevaluation, we recorded the number of problems in the student's code. Each student, except one who declined, received either two participant hours in Ulm or 15 Euros for participating.

We used a within-subjects design and randomly assigned the students, without their knowledge, to two groups of equal size. The first group received filtered feedback on the first task and unfiltered feedback on the second task, while for the second group it was the other way around. Unfiltered feedback shows all problems in the code, while filtered feedback only shows the top five problems (ordered by priority provided by the analysis tools).

We conducted the experiments at two German universities: University of Ulm and TU Braunschweig. In total, 46 students participated in the experiment (26 in Ulm and 20 in Braunschweig). For the result analysis, we excluded five students from Ulm and one student from Braunschweig for not adequately working on fixing the test cases. The coding experience of the students ranged from none to experts in Java.

After the experiment the students answered a questionnaire with 11 questions with a five-level Likert scale (agree, partially agree, half and half, partially disagree, disagree) and one free text field for general feedback. Only one of the 11 questions was directly related to the effect of filtered feedback, while the other 10 questions focused on the usability of the tool.

Results Regarding RQ_1 and RQ_2, we investigate whether filtered feedback leads to faster improvement in code quality. In Figure 1, we display a jitter plot showing, for each user, the minimum number of problems within their code within a one-minute time frame for both tasks. We separated the data by whether the student got filtered or unfiltered feedback for the respective task. In addition, we show a trend line for the filtered and unfiltered feedback, based on the mean values for each minute time frame. In Figure 2, we display boxplots showing the minimum number of problems within in a three-minute time frame.

In both plots, we can see an indication that the filtered feedback has a negative impact on the number of problems fixed over time. However, performing a t-test on the median of the number of problems per

[6] https://doi.org/10.5281/zenodo.14968587

student shows that the difference is not significant.

Regarding RQ_3, we rely on the answers given by the students in the questionnaire after the experiment. In their general feedback, three students explicitly mentioned a discomfort resulting from only having five problems displayed instead of all problems at once. Additionally, the students disagreed on the usefulness of feedback. While some students considered it helpful, others thought the given explanations were too difficult to understand. For more details, we provide anonymized answers to all questions online.[6]

In the questionnaire, we asked the students whether they could imagine using the system regularly. Twelve students used the extreme ends of the scale: two answered with 'agree' and ten chose 'disagree'.

Deviations At the start of the first evaluation, some students told us they did not receive any error messages. This was most likely caused by the server not being able to handle the large number of requests. The server returned no error messages in some cases even with errors present in the provided source code. These faulty feedback from the server were also shown in our data as if the student did not have any errors in his source code. After at most a couple of reevaluations, the error messages were shown again.

Discussion In each research question, we aimed to investigate whether filtered feedback helps students to improve the quality of a given source code. Executing a t-test on the results of our user study revealed no significant difference when using filtered feedback. Additionally, three students considered it noteworthy that they liked the unfiltered feedback better. Overall, this leads to the conclusion, that the filtered feedback might not help improving source code quality.

Threats to Validity During the evaluation, our servers reached their request limit and returned zero error messages, even with errors present in the code. We quickly identified the issue during the first task of the first evaluation. We informed all students that if they received zero error messages when they should have received some, they needed to reevaluate. They were told to repeat the process if necessary. This ensured all students were aware of the problem and had a workaround. We also removed all zero values from the evaluation to avoid including incorrect zeros.

The number of participants (40) was relatively small for such a user study. To increase our sample size, we approached students in two mandatory bachelor courses at two different universities. This allowed us to develop at least a preliminary trend, which we focused on in our evaluation.

For six students, the data showed no significant activity. As a result, we excluded these students from the analysis for RQ1 and RQ2. However, they completed the anonymous questionnaire, and we cannot determine their answers. Since the questionnaire did not clarify whether the students found the adaptive feedback helpful, the unexplainable answers from these six students do not affect our results.

4 Conclusion

In this paper, we presented a tool for adaptive feedback on source code quality. We conducted a user study to investigate the impact of the feedback. We compared filtered feedback (i.e., only report the five most important problems) with unfiltered feedback (i.e., all problems are reported). The goal was to assess how the two feedback types impact code quality improvement. While students preferred unfiltered feedback, the performance difference between the two conditions was not statistically significant.

The study suggests that the number of presented errors does not significantly affect the efficiency of source code quality improvement. However, due to the lack of statistical significance, we cannot draw firm conclusions. The results should be interpreted as an indication, and it is yet to be investigated whether there is, in fact, no difference between filtered and unfiltered feedback.

Acknowledgments This work was mainly funded under the 2LIKE project by the Federal Ministry of Education and Research (BMBF) and the ministry Ministry for Science, Research and Arts Baden-Württemberg within the funding line Artificial Intelligence in Higher Education. We also thank Matthias Tichy and Ruben Dunkel for their support in conducting the experiment.

References

[1] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In *ICSM*, pages 277–286. IEEE, 2008.

[2] R. P. Buse and W. R. Weimer. Learning a metric for code readability. *TSE*, 36(4):546–558, 2009.

[3] P. King, P. Naughton, M. DeMoney, J. Kanerva, K. Walrath, and S. Hommel. Code conventions for the java programming language. *Sun*, 1999.

[4] T. Lee, J. B. Lee, and H. P. In. A study of different coding styles affecting code readability. *IJSEIA*, 7(5):413–422, 2013.

[5] X. Li and C. Prasad. Effectively teaching coding standards in programming. In *SIGITE*, pages 239–244, 2005.

[6] M. E. Nordberg. Managing code ownership. *Software*, 20(2):26–33, 2003.

[7] J. C. Rodríguez-del Pino, E. Rubio Royo, and Z. Hernández Figueroa. A virtual programming lab for moodle with automatic assessment and anti-plagiarism features. 2012.

[8] D. Spinellis. Elyts edoc. *Software*, 28(2):104–104, 2011.

Challenges and Impact Factors on Modernization Projects - Results of an Industry Study

Ada Slupczynski
slupczynski@swc.rwth-aachen.de
Research Group Software Construction
RWTH Aachen University
Aachen, Germany

Horst Lichter
lichter@swc.rwth-aachen.de
Research Group Software Construction
RWTH Aachen University
Aachen, Germany

1 Introduction

For many years, modernizing existing systems has been a significant challenge for many software organizations. Due to the rapid pace of technological change and the pressure to remain competitive, modernization is still crucial for industry and science. The challenge remains as every modernization project needs to be tailored to the underlying legacy system. There are many experience reports; most of them present success stories of modernization projects (e.g., [2] [3]). In addition, few reports have examined various aspects of modernization projects in industry. For example, Khadka et al. report the results of an extensive industry interview study conducted in the Netherlands [1]. Although they present the most relevant challenges, their work is more exploratory.

The study presented in this paper aimed to supplement the published findings by identifying positive and negative influencing factors and general aspects of modernization projects. The goal was to provide a basis for practical guidelines on what actionable items can be used to improve the work of modernization projects. The study was conducted with a software development organization that has recently carried out various information system modernization projects. The study was designed to answer the following questions:

1. How do examples, motivations, and evaluations of modernization contribute to creating a shared understanding of the modernization process?
2. How can experiences with proven modernization approaches support the system perspective that shows the connections between technical characteristics and modernization strategies?

2 Study Method

The study was conducted from Nov 23 to Aug 24 using expert interviews. This method was chosen to collect experience reports on modernization projects from various stakeholders with many years of experience in different projects in a large software development organization. In total, 21 experts were interviewed. The experts had the following roles and functions in the organization: developers, software architects, system architects, and testers. All interviews took place online and lasted one hour each; all experts were interviewed separately.

The interview consisted of two sections. The first section served to obtain general statements on understanding modernization in an industrial environment. To get unbiased statements, the questions asked in this section were not distributed to the participants beforehand. The second section focused on the technical aspects of modernization and the resulting concrete challenges. These questions were communicated in advance. The interviews were recorded and systematically analyzed. All results and findings were presented and communicated to the organization.

3 Results

The results derived from the interviews' analysis can be divided into two categories: general statements on modernization and factors that influence the modernization of information systems, as presented below.

3.1 General results

Terminology matters! Although all experts clearly stated that continuous modernization is necessary, they had a different understanding of modernization and related approaches. This may result from the lack of unified modernization guidelines encompassing its intricacies. If we do not use the same language, misunderstandings will become more likely!

Modernization projects come too late and have no clear goals! Modernization needs to happen earlier than it usually does. Modernization becomes more complicated when technical debts exist in the legacy system, remaining untouched until they become problematic. Furthermore, the goals of modernization projects are often unclear or extensive, so the projects deliver no or only partial modernizations. This makes it difficult or impossible to evaluate modernizations. As a result, valuable data is lost when planning future

modernization projects. Postponing modernizations might lead to a situation where the legacy language is considered "unattractive" and is no longer taught on a broader scale.

No modernization without domain knowledge! The documentation required for modernization is often unavailable or incomplete from a technical and business point of view. This always leads to misunderstandings between the domain and the technical experts. Users, another critical stakeholder, are often not involved. This is negligent as they are the primary victims of unsatisfactory modernization and can jeopardize or even reject it.

Modernization is more than technology! Another risk lies in considering modernization as a technology thing. If we do not consider everything impacted by modernization, such as processes, products, and business, the likelihood of failure will increase!

3.2 Factors affecting modernization

We identified these factors affecting modernization projects positively or negatively.

Available knowledge and documentation: This is most important positive factor. To bundle knowledge, mixed teams with both domain and technical expertise should carry out modernizations. This reduces risks and increases the rate of successful modernization. An organization should share the knowledge gathered in their modernization projects to allow for a company-wide modernization culture. Proactive, agile, mixed teams across the enterprise further support this. This could establish a baseline not only for the technical execution of modernization but also for planning the budget and the time required for a new modernization project.

Suitable standard software: Modernization can be cheaper and more manageable if standard software can be bought and adapted. Of course, this is not always possible. When using standard software, responsibility is shifted onto the provider. On the other hand, the organization must consider the resulting dependency risk, i.e., organizations could depend on their standard software providers. In the long run, buying standard software might prove more costly, especially if further adoptions are needed.

Granularity of modernization: If a system can be divided into components, both technically and in terms of the business logic, this allows for shorter, easier-to-plan modernization projects. This can prevent modernization projects from being abandoned before completion (which is the worst case). In general, modernizing smaller and decoupled components is more effortless. However, several experts warned about starting modernization with the smallest component. On the one hand, it is advantageous because the modernization can be completed relatively quickly so that business stakeholders can see the ROI, and it is easier to obtain funding for the modernization of the remaining components. On the other hand, it can be dangerous as a small component may not reflect the complexity of modernizing larger components, making it more difficult to assess the risks. This can lead to unexpected problems that only arise in a later modernization project.

Knowledge retention or acquisition: Missing or incomplete documentation, failure to pass on relevant information and the lack of domain specialists and technical experts are the leading causes of the modernization problems. Ensuring internal communication and understanding might be difficult if an organization does not have a unified documentation culture.

3.3 Limitations

Although the study was extensive, it was conducted only with a single company. This limitation was mitigated by looking at different, often independent projects and products with separate teams and team structures.

4 Conclusions and Future Work

The study strongly underlines that modernization is a complex endeavor and needs to be considered in a broader context than a technical one. The described challenges show the importance of supporting decision-makers in determining whether and how to modernize legacy systems. Modernization must be viewed as a comprehensive and risky project affecting the entirety of the organization, requiring careful consideration of goals, relevant stakeholders, and an execution strategy suitable to the given legacy system.

Further, this study highlights a research gap regarding modernization's organizational side. Ultimately, new approaches and strategies are needed to make decisions about modernization projects systematically, based on data, and at the right time before modernization becomes more costly than necessary.

References

[1] R. Khadka et al. "How do professionals perceive legacy systems and software modernization?" In: *Proceedings of ICSE 2014*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 36–47.

[2] Y. de Lima Justino and C. E. da Silva. "Reengineering legacy systems for supporting SOA: a case study on the brazilian's secretary of state for taxation". In: *Proceedings of ICSE 2018: Companion Proceeedings*. Gothenburg, Sweden: ACM, 2018, pp. 125–126.

[3] S. C. d. Fonseca et al. "Migrating Legacy Systems: An experience report on the industrial environment". In: *Pro. of the XXIII Brazilian Symposium on Software Quality*. SBQS '24. ACM, 2024, pp. 452–459.

Factors involved in Modernization Decision Making

Aleksandra Pazova
RWTH Aachen, aleksandra.pazova@rwth-aachen.de

Abstract

In today's rapidly evolving world of technology, modernizing legacy software is more crucial than ever. Outdated systems hinder innovation, reduce efficiency, and pose risks of failure. Modernization enhances competitiveness and adaptability but is a complex process requiring careful evaluation of numerous interdependent factors. Stakeholder perspectives influence how these factors are perceived and prioritized.

A major challenge is the interconnectedness of these factors, complicating decision-making. A summarized overview of their influences is essential.

Existing literature lacks a holistic view of technical, financial, organizational, and strategic factors. This thesis conducts a Systematic Literature Review to identify key factors and their interrelations, developing a conceptual model that integrates business and technical perspectives.

The goal is to prevent oversight, provide a clear visual representation, and improve stakeholder communication and decision-making.

1 Introduction

Many software systems in use today were developed decades ago, becoming outdated and difficult to maintain. Meanwhile, rapid technological advancements and evolving market demands push organizations to modernize. Since these systems are often the backbone of businesses, modernization is crucial to remain competitive. Lehman's first law [1] states that a system must continually adapt or risk becoming unsatisfactory.

Modernization is complex and can lead to pitfalls such as delays, budget overruns, or system failures[2]. Many failures stem from poor strategies, overlooked factors, or insufficient planning. Early identification of key factors, technical, financial, organizational, and strategic, is essential for success. Stakeholder involvement from the beginning can prevent critical mistakes, as seen in failed healthcare modernization projects.

Despite the importance of modernization, scientific literature on decision-making in this area is limited. Existing studies focus on frameworks, motivations, or isolated factors but rarely explore their interdependencies. To address this gap, a Systematic Literature Review was conducted, to collect and analyze key factors influencing modernization decisions, their interrelations, and stakeholder perspectives.

This study aims to develop a conceptual model that maps key factors and their relationships, helping organizations make informed decisions.

Expert interviews complement the research by providing real-world insights, ensuring relevance to industry practices. The model aims to serve as a foundation for modernization strategies and highlight areas needing further academic exploration.

2 Results

The study identified key factors influencing modernization decisions and mapped their interrelations using a conceptual model. This model distinguishes between characteristics, which define the state and nature of a system or organization, and influential factors, which drive changes in modernization efforts. Characteristics include system complexity, flexibility, correctness, and strategic alignment, while influential factors encompass costs, risks, regulatory requirements, and stakeholder involvement.

A key finding is the intricate interdependence between these elements. Increased system complexity directly correlates with higher maintenance effort [3], making modernization more resource-intensive and prolonging implementation timelines. Conversely, greater flexibility and modularity facilitate adaptation, reducing long-term risks and ensuring smoother transitions to modern architectures. The number of software functionalities significantly impacts competitiveness by enhancing user satisfaction and market differentiation, but it also escalates operational costs and maintenance challenges, leading to trade-offs in modernization planning.

Personnel expertise plays a pivotal role, as knowledge retention mitigates risks associated with modernization delays and inefficiencies. Organizations with a high turnover rate often struggle to maintain system knowledge, leading to

additional training costs and slower adoption of new technologies. The study also highlights how external regulatory pressures necessitate continuous compliance monitoring, influencing both cost and technical decisions. Strict industry regulations may demand additional security enhancements and documentation efforts, further complicating modernization strategies.

The findings emphasize the necessity of balancing organizational, financial, technical, and strategic considerations. While cost reduction is often a primary driver, prioritizing short-term savings over long-term adaptability can result in failed modernization efforts. Organizations that invest in sustainable modernization approaches and stakeholder-inclusive decision-making, tend to achieve more resilient and scalable outcomes. Additionally, stakeholder perspectives significantly shape modernization priorities, business stakeholders emphasize return on investment and risk mitigation, whereas technical teams focus on system stability, maintainability, and innovation potential.

The conceptual model provides a structured framework for identifying dependencies between factors, ensuring a holistic approach to decision-making. By understanding these relationships, organizations can prioritize investments, optimize resource allocation, and mitigate modernization challenges effectively. This interconnected perspective allows decision-makers to anticipate obstacles, balance competing interests, and implement modernization strategies that align with both immediate needs and long-term objectives.

3 Discussion

The expert evaluation provided valuable insights into the practical applicability of the conceptual model. Experts confirmed that the model effectively captures the complexity of modernization decisions but noted that its detailed structure might be challenging for direct implementation. They suggested that further refinement should focus on making the model more accessible by segmenting it into smaller components that allow for step-by-step application.

A recurring discussion point was the challenge of balancing cost efficiency with system sustainability. Experts highlighted that while financial constraints often drive modernization strategies, excessive focus on cost reduction can lead to increased technical debt and maintenance burdens. Another critical observation was the importance of cross-departmental communication. Experts emphasized that misalignment between business and technical teams frequently leads to delays and inefficiencies, making early stakeholder involvement crucial.

The results also indicate that modernization efforts should incorporate flexibility in their strategies to accommodate evolving organizational needs. Experts recommended integrating industry-specific frameworks to better align with practical constraints and regulatory requirements. Furthermore, the discussion revealed gaps in existing literature regarding failed modernization attempts, suggesting that future research should analyze these cases to extract valuable lessons.

4 Conclusion

This study aims to contribute to the understanding of software modernization decision-making by mapping key factors and their interdependencies. The conceptual model can serves as a foundation for guiding modernization efforts, helping organizations anticipate challenges and optimize decision-making. The findings highlight the necessity of structured decision-making that balances technical feasibility, financial sustainability, and strategic goals.

While the model provides a structured approach, further refinements are needed for its practical application. Simplification of its representation, alignment with specific industry practices, and empirical validation through case studies are essential next steps. Additionally, the study underscores the importance of interdisciplinary collaboration, ensuring that modernization decisions are made with a comprehensive perspective that integrates business and technical considerations.

Future research should explore failed modernization projects to identify common risk factors and develop mitigation strategies. By refining the model and expanding empirical research, organizations can enhance their ability to implement successful modernization strategies that ensure long-term adaptability and competitiveness.

References

[1] M. Lehman. “On understanding laws, evolution, and conservation in the large-program life cycle.” In: *Journal of Systems and Software* 1 (1979), pp. 213–221.

[2] J. Bisbal et al. “Legacy information systems: issues and directions.”, *IEEE Software* (1999)

[3] R. Khadka et al. “Does software modernization deliver what it aimed for? A post modernization analysis of five software modernization case studies.” IEEE 2015

Entwicklung eines Conversational Interface zur Steuerung von SEE und Abfrage von Graphdatenbanken

Sarah Augustinowski
augustin@uni-bremen.de
Universität Bremen

1 Einleitung und Motivation

Dieser Beitrag fasst meine Bachelorarbeit zur Erweiterung der Anwendung SEE (Software Engineering Experience)[1] zusammen. SEE ist ein System zur kollaborativen Softwarevisualisierung, bei dem Anwender als Avatare eine virtuelle Welt betreten, in der Software-Projekte als sogenannte Code-Cities dargestellt werden. Diese basieren auf dreidimensionalen Treemaps und vermitteln den Eindruck einer Großstadt mit Wolkenkratzern.[1]

SEE ermöglicht es, verschiedene Informationen über die visualisierten Software-Projekte zu gewinnen. So können beispielsweise Code-Metriken wie Lines of Code oder Qualitätsmerkmale wie die Testabdeckung sichtbar gemacht werden. Zum Beginn meiner Arbeit verfügte SEE bereits über einen Assistentin-Avatar. Die Interaktion mit diesem war jedoch auf die Online-Windows-Spracherkennung beschränkt und die Assistentin konnte nur allgemeine Antworten geben, die keinen Bezug zur visualisierten Software hatten. Weder das Abfragen von Metriken per Sprachbefehl noch die Steuerung anwendungsspezifischer Funktionen innerhalb von SEE waren möglich.

2 Entwicklung

Das Conversational Interface sollte eine betriebssystemunabhängige und offline lauffähige Spracherkennung ermöglichen. Des Weiteren soll die Assistentin „intelligenter“ werden. Durch eine Repräsentation der Software-Projekte als Graph in einer Graphdatenbank sollte die Möglichkeit geschaffen werden, mit Hilfe der Assistentin dort hinterlegte Software-Metriken abzufragen. Durch Sprachbefehle sollen weitere Funktionen in SEE gesteuert werden. Die Entwicklung des Conversational Interfaces umfasst grundsätzlich vier Komponenten. Zum einen SEE als Hauptanwendung, OpenAI's Whisper als lokale Spracherkennung, Rasa Open Source als Chatbot-Framework und Neo4j als Graphdatenbank. Da SEE als Unity-Anwendung mit C#-Programmierung entwickelt wird, konnte das Projekt *whisper.unity* von Macoron als Package importiert werden.[2] So wurden von SEE aufgenommene Sprachbefehle durch whisper in Text umgewandelt. Der zuvor umgewandelte Text wird über eine HTTP-Verbindung an die laufende Rasa-Instanz gesendet. Hier wurden zuvor verschiedene Intents (Absichten) und Entitäten mit Hilfe von Trainingsdaten vorbereitet. Intent und eingehender Text werden abgeglichen. Der Intent, der mit der höchsten Konfidenz erkannt wurde, wird ausgewählt. In zuvor festgelegten *Stories* kann einem Intent eine bestimmte Action zugewiesen werden. Innerhalb der Actions werden der erkannte Intent sowie die Entitäten im Body des HTTP-Post zurück an SEE gesendet. Ein Beispiel für einen Intent (hier *isMetricIn*) und die erkannten Entitäten, mit ihren Werten (hier verdeutlicht durch die Unterstreichung und der eckigen Klammern) zeigt dieser Satz: „How many lines of code [Metric] are there in GraphReader [Place]?“ Es sei aber angemerkt, dass dies nicht der eigentlichen Syntax von Rasa entspricht. Sobald SEE die Rückmeldung von Rasa erhält, werden die nötigen Abfragen in der Cypher Query Language, unter Verwendung der zuvor erkannten Entitäten, formuliert und an die Graphdatenbank gestellt. Das Ergebnis wird dann in Antwortvorlagen integriert und durch das vorhandene Text-to-Speech-System ausgegeben. Intents, die keine Datenabfrage enthalten, sondern Funktionen triggern sollen, werden von SEE direkt ausgeführt.

3 Studie

Nach Fertigstellung eines Prototypen mit vermindertem Funktionsumfang wurde eine Nutzerstudie durchgeführt. Die Nutzerstudie soll dabei helfen, die folgenden drei Forschungsfragen zu beantworten: 1. *Wie beeinflusst die Nutzung eines Conversational Interfaces im Vergleich zur Maus/Tastatur-Steuerung die Bearbeitungszeit und Genauigkeit der Nutzer bei Aufgaben?* 2. *Wie unterscheidet sich die Benutzerfreundlichkeit (Usability) bei der Nutzung eines Conversational Interfaces im Vergleich zur Maus/Tastatur-Steuerung?* 3. *Wie beeinflusst die Nutzung eines Conversational Interfaces im Vergleich zur Maus/Tastatur-Steuerung die Erinnerungsfähigkeit der Nutzer an frühere Informationen?*

Die Nutzerstudie wurde im Within-Subject-Design durchgeführt, sodass die Probanden, die zuvor nach dem Convenience Sampling rekrutiert wurden, sowohl das Conversational Interface als auch die übliche

[1] GitHub: https://github.com/uni-bremen-agst/SEE
[2] GitHub: https://github.com/Macoron/whisper.unity

Maus/Tastatur-Steuerung von SEE testen. Die Studie beginnt mit der Abfrage von demografischen sowie spezifischen Daten, um später mögliche Korrelationen messen zu können. Die erfassten demografischen Daten sind Geschlecht, Alter, höchster Schul- oder Hochschulabschluss, Beruf/Tätigkeit. Bei den spezifischen Daten handelt es sich um die Bekanntheit von SEE sowie die Erfahrung mit Sprachsteuerung. Nachdem diese Daten in das Online-Formular KoboToolBox[3] eingegeben wurden, erhielten die Probanden eine kurze Einweisung in die Funktionen und Steuerung von SEE mit anschließender Testmöglichkeit.

Um mögliche Lerneffekte vollständig ausschließen zu können, begann die eine Hälfte der Probanden mit der Maus-/Tastatur-Steuerung und die andere Hälfte mit dem Conversational Interface. Die gestellten Aufgaben wurden dabei an einer Visualisierung von SEE selbst durchgeführt. Die Aufgaben waren dabei: 1. *Finde die Klasse „X“ in der Code-City. Welchen Wert hat die Metric „Metric.Lines.LOC“ (auch „Lines of Code“ genannt)?* 2. *Finde den Namespace „Y“. Wie viele Klassen gehören zu diesem Namespace?* 3. *Öffne den Browser. Welche Webseite wird aktuell angezeigt? Schließe danach die gesamte Anwendung.* und 4. *Wie war der Wert von Lines of Code in der Klasse „X“?* X und Y variieren dabei zwischen den beiden Interaktionsmöglichkeiten, haben aber ähnliche Eigenschaften. Aufgabe 4 findet für beide Interaktionen ganz am Ende der Studie statt, da es sich hierbei um eine Aufgabe zur Erinnerungsfähigkeit handelt. Nach jeder Aufgabe wird der After Scenario Questionnaire (ASQ) abgefragt, um die Zufriedenheit der Probanden mit der Komplexität, dem Zeitbedarf und den unterstützenden Informationen zu messen. Vor dem Wechsel der Interaktion bzw. der vollständigen Beendigung der Studie wird der System Usability Scale-Fragebogen verwendet, um die grundsätzliche Benutzerzufriedenheit zu erfassen. Neben der Erfassung durch die Fragebögen werden als weitere abhängige Variablen die Bearbeitungszeit und Korrektheit der Aufgaben ermittelt.

4 Ergebnisse

Insgesamt konnten 16 Probanden für die Studie gewonnen werden. 11 Probanden hatten dabei einen Bezug zur Informatik. Nur zwei der Probanden haben an SEE selbst entwickelt. Bei den folgenden Auswertungen wurde ein Signifikanzniveau von 0,05 verwendet. Für die Auswertung der abhängigen Variablen wurde der Wilcoxon-Test in seiner gepaarten Version verwendet. Bei Auftreten von mehreren Zeros und Ties wurde der Test durch den Fisher-Pitman Permutation-Test ergänzt.[4]

Im Hinblick auf die Korrektheit der Antworten konnte nur im Falle von Aufgabe 2 ein signifikant besseres Ergebnis mit dem Conversational Interface erzielt werden ($p \approx 0,0004556$). Da bei Aufgabe 4 kein signifikanter Unterschied zwischen Conversational Interface und Maus-/Tastatur-Steuerung festgestellt werden konnte, können wir hier bereits die dritte Forschungsfrage beantworten und davon ausgehen, dass die Art der Interaktion keinen Einfluss auf die Erinnerungsfähigkeit der Nutzer hat. Betrachten wir die Bearbeitungsdauer, konnten die Probanden sowohl die Aufgabe 2 als auch die Aufgabe 3 mit dem Conversational Interface signifikant schneller bearbeiten (Aufgabe 2: $p \approx 0,0005126$, Aufgabe 3: $p \approx 0,002412$). Aufgabe 4 wurde an dieser Stelle nicht betrachtet, da hier die Bearbeitungsdauer nicht gemessen wurde.

Der Vergleich der beiden System-Usability-Scales zeigte, dass die Probanden die Benutzerfreundlichkeit des Conversational Interfaces als signifikant besser empfanden ($p \approx 0,03879$). Ein Mittelwert von 70,16 spricht dabei laut Bangor et al. [2] für eine Benutzerfreundlichkeit von OK bis Good und verpasst die Einschätzung Good bis Excellent nur knapp. Die Auswertung des ASQ signalisierte für die Aufgaben 2 und 3 ein signifikant positiveres Empfinden bezüglich der Komplexität und der Bearbeitungszeit der Aufgaben mit Hilfe des Conversational Interfaces.

Die Überprüfung der Korrelationen zwischen den demografischen/spezifischen Daten und den abhängigen Variablen durch das Kendall's Tau und Pearson deutet auf eine mittlere positive lineare Korrelation zwischen dem Alter der Probanden und der Bearbeitungsdauer der Aufgaben mit der Maus/Tastatur-Steuerung ($r \approx 0,54$ $p \approx 0,3$). Weitere Korrelationen konnten nicht nachgewiesen werden.

5 Fazit

Durch die Entwicklung konnte SEE um eine weitere Interaktionsmöglichkeit erweitert werden. Die Studie hat gezeigt, dass die Verwendung des Conversational Interfaces je nach Aufgabe einen positiven Effekt auf die Bearbeitungszeit haben kann. Eine deutliche Verbesserung der Korrektheit der Aufgaben konnte jedoch nicht gewährleistet werden. Erfreulich ist die erhöhte Benutzerfreundlichkeit gegenüber der Maus/Tastatur-Steuerung. Ein Effekt auf die Erinnerungsfähigkeit an zuvor erlangte Informationen konnte nicht nachgewiesen werden. Nötige Verbesserungen der Entwicklung sind die Optimierung der Zeit, die whisper benötigt, um die Sprachbefehle zu transkribieren, sowie die Ergänzung eines Gedächtnis des Assistenten.

Literaturverzeichnis

[1] Brian Johnson und Ben Shneiderman. Treemaps: a space-filling approach to the visualization of hierarchical information structures. *Proceeding Visualization '91*, pages 284–291, 1991.

[2] Aaron Bangor, Philip Kortum und James Miller. Determining what individual sus scores mean: Adding an adjective rating scale - jux, 2009.

[3] https://www.kobotoolbox.org/

[4] Verursacht durch Null-Einträge und gleiche Ränge.

Ein Vergleich von Attract-Funktionen in SEE

Leon Erhardt, Leeeon333@gmail.com

I. Einführung

In diesem Beitrag stelle ich einen Vergleich von Methoden zur Vorschlagsauswahl für Zuordnungen während der Reflexionsanalyse[4] vor, der anhand automatischer Experimente auf sechs Java Open-Source-Projekten in der Visualisierungsplattform SEE durchgeführt wurde. Der Vergleich, die Visualisierung sowie eine Benutzerstudie[2] über den Einfluss der Vorschläge gingen aus meiner Masterarbeit hervor. Die Visualisierung solcher Vorschläge in SEE wird in einem weiteren Beitrag zum WSRE mit einer Tool-Demo genauer vorgestellt [3].

Reflexionsanalyse Die Reflexionsanalyse ist ein iterativer Prozess bei dem Implementierungskomponenten auf Architekturkomponenten abgebildet werden, um Abweichungen und Übereinstimmungen der Implementierung zu Architekturregeln zu lokalisieren. Dabei werden Implementierung und Architektur als Abhängigkeitsgraphen repräsentiert. Eine manuelle Zuordnung ist oft zeitaufwändig und fehleranfällig.

Attract-Funktionen Um diesen Prozess zu unterstützen, existieren sogenannte Attract-Funktionen, die Anziehungswerte zwischen Teilen der Implementierung und Architektur berechnen. Die Funktionen benutzen bereits bestehende Zuordnungen als Grundlage zur Berechnung weiterer Vorschläge. Dabei werden strukturelle Merkmale des Source-Codes (z. B. Use- oder Call-Beziehungen) oder das Auftreten von Termen darin (z. B. Bezeichnernamen oder Kommentare) als Information genutzt. In meiner Arbeit wurde die neuartige Funktion *ADC-Attract* - welche Struktur und Text-Merkmale kombiniert - erarbeitet und mit zwei weiteren Attract-Funktionen verglichen. Die erste Attract-Funktion *Count-Attract*[1] ist Struktur-basiert und versucht bei der Auswahl von Zuordnungen Kopplung zu minimieren. Allerdings ist die Funktion parametrisiert durch einen zusätzlichen Parameter ϕ, welche durch Architekturregeln erwünschte Kopplung miteinbezieht. Die zweite betrachtete Attract-Funktion *NB-Attract*[5] ist Wort-basiert und verwendet Naive-Bayes-Klassifikation als Grundlage ihrer Berechnung. Die Architekturkomponenten repräsentieren Klassen und die Worte, welche in den Zuordnungen vorkommen, dienen als Trainingsdaten. Dabei wird nur das Auftreten von Worten in Implementierungskomponenten berücksichtigt, nicht ihre Häufigkeit.
Um sowohl Struktur- als auch Text-Informationen zu berücksichtigen, werden aus Implementierungskanten gewonnene Worte in Architekturkanten zusammengefasst. Für eine Architekturkante werden dabei die Implementierungskanten berücksichtigt, welche mit ihr übereinstimmen. Die Worte einer Implementierungskante ist die Intersektion der Wörter beider adjazenter Komponenten. Dadurch werden bei zunehmender Größe der Zuordnug jene Worte in Architekturkanten zusammgefasst, welche typisch für den Zusammenhang in der Architektur sind. Dieses Prinzip habe ich *Abstract Dependency Concretization (ADC)* genannt. Es lassen sich dadurch die Wörter von Implementierungskanten, welche bei einer Zuordnung übereinstimmen würden, mit den Worten der erlaubendenen Architekturkanten durch Distanzfunktionen vergleichen und als Anziehungswert aufsummieren. Für die Funktion *ADC-Attract* wurde als Distanzfunktion der *overlap*-Koeffizient gewählt, da er in vorausgehenden Experimenten die besten Ergebnisse erzielt hat. Eine detailliertere Definition mitsamt graphischer Darstellung lässt sich im SEE-Wiki finden[1].

HugMe-Methode Auf Basis der berechneten Anziehungswerte zwischen Architektur- und Implementierungskomponenten kann die sogenannte *HugMe*-Methode[1] eine Auswahl geeigneter Architekturkomponenten vorschlagen. Dabei berücksichtigt sie nur Vorschläge, die sich statistisch von den übrigen Werten der gesamten Matrix aus Anziehungswerten abheben. Die *HugMe*-Methode wird häufig in automatische Zuordnungs-Experimente eingebettet, da sie eindeutige Vorschläge direkt zuordnet.

II. Design der Experimente

Ziel der Experimente war es die Performanz der bisher vorgestellten Attract-Funktionen mithilfe von Orakelzuordnungen zu vergleichen. Dabei wurde zudem nach Einfluss von Text- und Strukturmerkmalen gefragt. Es wurden die sechs Java Open-Source-Projekte SweetHome3D (SH3D), CommonsImaging (CI), JabRef (JR), TeamMates (TM), Lucene (LC) und Ant in Abhängigkeitsgraphen übersetzt. Architektur und Orakelzuordnungen wurden aus dem S4rdM3x-Projekt abgeleitet, welches in verwandten Arbeiten verwendet wurde[5][2]. Implementierungsknoten aus Systembibliotheken und solche ohne übertragbare

[1] `https://raw.githubusercontent.com/wiki/uni-bremen-agst/SEE/literature/SAerCon_2025_Reflexion_Mapping.pdf`

[2] `https://github.com/h0bb3/s4rdm3x`

Orakelzuordnung wurden entfernt. Um Worte aus Implementierungsknoten zu gewinnen, wurde der Quellcode gemäß der verwandten Literatur aufbereitet[5].
Ein Experiment beginnt mit einer zufälligen, Orakelbasierten Zuordnung der Größe θ. Die Größe gibt den Anteil der initial zugeordneten von den verfügbaren Implementierungsknoten an. Anschließend wurde die *HugMe*-Methode angewandt, um eindeutige Vorschläge iterativ zu identifizieren und zuzuweisen, bis entweder keine eindeutigen Vorschläge mehr gefunden wurden oder die Zuordnung vollständig war. Die erfolgten Zuordnungen wurden gespeichert und als Klassifikationsproblem betrachtet. Eine Implementierungskomponente wird entweder richtig (TP), falsch (FP) oder gar nicht (FN) zugeordnet. Daher wurde Precision, Recall und der F-Score verwendet, um die Auswahl zu bewerten.
Es wurden vier Gruppen zu den jeweils vier θ-Werten 0.3, 0.5, 0.7 und 0.9 betrachtet, um verschiedene Zeitpunkte der Modellierung zu simulieren. Die vier Gruppen sind Count-Attract mit $\phi = 0$ (CA_0) und $\phi = 1$ (CA_1) sowie ADC-Attract (ADC) und NB-Attract (NB). CA_0 bezieht Architekturregeln mit ein, wohingegen CA_1 diese ignoriert. Für jede Gruppe wurden pro System $n = 100$ Experimente berechnet und anschließend wurden Durchschnitt und Standardabweichung der Qualitätsmetriken gebildet. Alle zugeordneten Implementierungskomponenten entsprachen Klassen des Quellcodes.

III. Resultate

Wir betrachten die Ergebnisse für jede Gruppe und diskutieren sie anschließend zusammenfassend. Die beschriebenen Ergebnisse in graphischer Form lassen sich im SEE-Wiki finden[3]. Die Gruppe CA_1 weist die niedrigste oder zweitniedrigste Precision in vier Systemen (TM, SH3D, JR, Ant) auf und erreicht erst bei höheren θ-Werten bessere Werte, während der Recall in vier Systemen (TM, LC, CI, JR) führend ist. Allerdings kann CA_1 auf dem System CI durchgehend sehr hohe F-Scores erzielen (0.8-0.9). CA_0 erzielt dagegen zwar eine hohe Precision (0.8-1.0 auf ANT, JR, LC, SH3D und TM), leidet jedoch unter einem unverhältnismäßig starken Einbruch des Recall- und F-Score-Werts. *NB* zeigt insgesamt die höchsten F-Scores (meistens 0.8) in vier Systemen (JR, LC, SH3D, TM) und ist weitgehend unabhängig von θ. Allerdings wurden auf zwei Systemen (CI, ANT) für alle θ-Werte gar keine Zuordnungen gefunden. *ADC* erreicht maximale und durchschnittliche F-Scores, die denen von NB-Attract nahekommen oder entsprechen. Zudem performt die Funktion vergleichweise am schlechtesten, wenn *NB* gar keine Ergebnisse findet. Die Höhe der Standardabweichungen scheinen systemabhängig anstatt funktionsabhängig zu sein. Das Einbeziehen der Architekturregeln in Count-Attract bewirkt zwar hohe Precision, allerdings leidet der Recall-Wert. Ob Struktur oder Wörter besser zur Identifikation korrekter Zuordnungen geeignet sind, scheint ebenfalls vom System abzuhängen. Dennoch schneiden Funktionen, die Wörter einbeziehen, in vielen Fällen besser ab.
Es wurde zudem die Kendall-Tau-Korrelation zwischen einer Kopplungsrate (errechnet bei vollständiger Zuordnung nach Orakel) und der Performanz der Attract-Funktionen pro θ-Wert berechnet. Es scheint, dass die Performanz aller Attract-Funktionen bei den niedrigen θ-Werten negativ mit hoher Kopplung korreliert, wobei NB-Attract am robustesten wirkt. Bei höheren θ-Werten scheint sich hohe Kopplung positiv auf die Präsizion von CA_0 und ADC auszuwirken. Dies liegt möglicherweise am Einbezug der Architekturregeln. Hier sei anzumerken, dass die Auswahl der Systeme klein ist und nur wenige signifikante p-Werte pro Korrelationstest erreicht wurden.

[3] `https://raw.githubusercontent.com/wiki/uni-bremen-agst/SEE/literature/SAerCon_2025_Reflexion_Mapping.pdf`

IV. Fazit

Die Qualität der Auswahl von Vorschlägen durch eine Attract-Funktion scheint Systemabhängig zu sein, wobei das Einbeziehen von Worten die Vorschlagsauswahl auf vielen Systemen verbessert. Die Ergebnisse legen aber dennoch nahe, dass sich bei einer geringen Anzahl an Zuordnungen *Count-Attract* mit $\phi = 0$ oft eignet, um erste präzise Vorschläge zu finden. Bei fortlaufender Modellierung kann die Funktion von *ADC-Attract* oder *NB-Attract* abgelöst werden, da bei höheren θ-Werten oft mehr, aber ähnlich präzise, Vorschläge erfolgen. *ADC-Attract* konnte nahe der bisher stärksten Funktion *NB-Attract* performen, allerdings keine eindeutigen Vorteile bieten. Sie erlaubt aber potentielle Verbesserung durch das Einbeziehen von Kantengewichten. *ADC-Attract* und *NB-Attract* teilen die Eigenschaft, nur das Auftreten von Worten in Implementierungsknoten zu berücksichtigen, was auf ein allgemeines Prinzip für die Auswahl von Zuordnungen hindeuten könnte.

Literaturverzeichnis

[1] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Automated clustering to support the reflexion method. *Information and Software Technology*, 49(3):255–274, 2007.

[2] Leon Ehrhardt and Rainer Koschke. Reflexionsanalyse in SEE: ein kontrolliertes Experiment. In *Workshop Software-Reengineering & Evolution (WSRE)*, 2025.

[3] Michel Krause and Rainer Koschke. Architekturmodellierung und -verifikation mit SEE (Tool-Demo). In *Workshop Software-Reengineering & Evolution (WSRE)*, 2025.

[4] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, 1995.

[5] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. To automatically map source code entities to architectural modules with naive bayes. *Journal of Systems and Software*, 183:111095, 2022.

C to Rust Translation via Large Language Models

Jochen Quante, Jesko Hecking-Harbusch, and Matthias Woehrle
Bosch Research, Renningen, Germany

Generative AI shows great potential for various software engineering tasks. Last year, we discussed chances and challenges of LLM-based Software Reengineering [3]. In this paper, we focus on translation of embedded C code to Rust using Large Language Models (LLMs). We detail on specific issues and show concrete remedies in task formulation with code. These remedies are general and can be leveraged across various software engineering use cases.

1 Why Rust?

The programming language Rust has gained significant popularity over the past decade. The main advantage of Rust is its built-in memory safety guarantee. Based on the concepts of "ownership" and the "borrow mechanism", Rust ensures that memory access is always safe. This is done as much as possible at compile time, and otherwise at runtime, which may lead to a "panic" and termination. Rust can also enable safe concurrency. Therefore, Rust is particularly suitable for development of safety-relevant software, where faulty behavior must be avoided by all means. Google, Mozilla, and others consistently reported that around 65% of security vulnerabilities have been caused by memory safety issues in the past – and a considerable drop when switching to Rust. Therefore, there is a high interest in switching to Rust.

2 Basic C to Rust

Previous work has discussed the shortcomings of "classical" rule-based approaches for C to Rust translation and proposed using Large Language Models (LLMs) for idiomatic translation [3, 1]. While LLMs may be better at producing more typical Rust code, this comes at the price of hallucination, which means that results are not always correct and complete. In case of Rust, the generated code is often not even compilable, as the Rust compiler can be picky about the code it accepts: The code has to fulfill the rules of the ownership concept. Fortunately, the Rust compiler also provides meaningful error messages, which often give hints how problems can be solved. When we feed back this information to the LLM and ask it to fix the code, we often get compilable code. This way, it is possible to translate pieces of C code to Rust. Of course, the correctness of its behavior still has to be checked, e.g., by testing, formal checks [4], or differential fuzzing [1].

3 Translating Larger Code

When larger amounts of code shall be translated, this cannot be done in a single shot. LLMs have context limitations, but these limitations have reduced over time. More importantly, even when the input context is large, the quality of the generated output tends to deteriorate the longer the output gets. Therefore, we have to split the code into smaller fragments and translate them individually. We thus translate the code function by function. Our example module is a complex driver of around 1,000 lines of C code that is written in traditional embedded style, i.e., using global variables ("messages") for communication. This is how large shares of legacy embedded code work. In the following, we generically talk about LLMs because we did not observe significant differences between the state-of-the-art models we tried.

A viable approach for large code is to give the C functions to the LLM individually and to then iterate by feeding back error messages until successful compilation. However, the LLM has a lot of freedom how to translate all the interfaces and data structures. These decisions are not consistent over all functions: The same global variable may be transformed in different ways for different functions (e.g., local variable vs. static variable), and data types may be inconsistent. It also transforms identifier names in varying ways, like `camelCase` vs. `snake_case`. This leads to the result that many of the translated functions are incompatible to others. As a results, the module as a whole neither compiles nor works, which would result in a huge (manual) effort to fix all the inconsistencies.

Therefore, we additionally provide all declarations that are required to compile a specific function to the LLM along with the functions code. That means all #defines, type declarations, and function declarations that are referenced in the function are prepended to the code. Collecting these dependencies is a classical software analysis task. It turns out that besides the C declarations, we also have to provide the corresponding Rust code, in case it has already been translated, to reach consistency. This works for small functions with few dependencies. For most functions, the required declarations become so large that the LLM fails to translate the whole code, because the context just becomes too large. For example, in some cases, we got results as shown in Figure 1a.

As we can see, exactly the part that we asked for –

```
fn calc_position(ct_cas: NumCaST) {
    // Your implementation here
    unimplemented!()
}
```

(a) Long function – incomplete translation.

```
fn calc_position(ct_cas: NumCaST) {
    unsafe {
        ... // the translated function body
    }
}
```

(b) Unsafe body to access globals.

```
struct Globals {
    phi_cas_pos: [i16; NUMCASMAX],
    phi_curr_pos: i32,
    // ...further variables...
}

fn calc_position(globals: &mut Globals,
                 ct_cas: NumCaST) {
    // ...body...
}
```

(c) Globals passed via struct.

Figure 1: Problems and solutions when translating C to Rust via LLM.

the implementation of the function – is missing. This means that splitting to function level is not sufficient in this case, and we have to split large functions into several parts. For this task, ideas from software clustering and refactoring come into mind. However, the resulting new interfaces that have to be considered typically increase in complexity.

4 Global Variables

Another issue that we faced is the translation of global variables. In most cases, they are translated to "static" variables, which means that accesses can only occur within "unsafe" blocks. The LLM then puts the entire function body into an "unsafe" block, as shown in Figure 1b. However, if the whole function is "unsafe", the memory safety guarantee of Rust is weakened – certain things are no longer enforced by the compiler (see the Rust Book[1] for details). As the memory safety guarantee is considered the main advantage of Rust, such a solution is not acceptable.

An alternative is to put all former global variables into a struct and pass it to the function in Rust. When telling the LLM to do so ("Put all global variables into a struct"), we get a result as shown in Figure 1c, which is a more adequate approach to generically dealing with globals. However, the globals need to be explicitly passed into each function, which could also become messy. Another option would be to put all global variables into a separate module with get and set methods for each variable. This way, the unsafe code is limited to the getters and setters and does not cover the entire code. Unfortunately, telling the LLM to use getters and setters for accessing global variables was silently ignored in our experiments.

Ultimately, it is unavoidable to refactor the code to get rid of global variables, using a more adequate design. Nevertheless, we can reduce tedious and error-prone manual effort by generating design with LLMs that are closer to the desired target.

[1] `https://doc.rust-lang.org/book/`

5 Conclusion

We have discussed two concrete challenges for a particular use case for LLM-based software engineering. The general lessons learned are in our experience applicable across different tasks: LLMs perform better when we give them more (*i*) targeted and (*ii*) well-defined tasks. The targeting may be in the form of preprocessing as shown in our example or providing the LLM with tools (through function calling) that facilitates solving the engineering task. How to handle global variables is a better definition of the task. For both improvements, we can leverage software engineering tools and knowledge.

Moreover, we only reported on successful compilation. Note that further and more rigorous checks, e.g., formal checks [2, 4], or differential fuzzing [1], may uncover even more challenging problems. This increases the need for domain-specific information in task specification and also the feedback cycle we mentioned.

Finally, note that this feedback cycle is an integral part of LLM-based software engineering tasks. While we see large differences between models when we consider single shot performance without a feedback loop, these differences are largely diminished by domain-specific feedback mechanisms, e.g., based on the Rust compiler error messages.

References

[1] H. F. Eniser, H. Zhang, C. David, M. Wang, M. Christakis, B. Paulsen, J. Dodds, and D. Kroening. Towards translating real-world code with LLMs: A study of translating to Rust, 2024. arXiv:2405.11514.

[2] J. Hecking-Harbusch, J. Quante, and M. Schlund. Formal runtime error detection during development in the automotive industry. In *Proc. of VMCAI*, volume 14499 of *LNCS*, pages 3–26. Springer, 2024.

[3] J. Quante and M. Woehrle. Chances and challenges of LLM-based software reengineering. *Softwaretechnik-Trends*, 44(2):46–47, 2024.

[4] A. Z. H. Yang, Y. Takashima, B. Paulsen, J. Dodds, and D. Kroening. Vert: Verified equivalent Rust transpilation with Large Language Models as few-shot learners, 2024. arXiv:2404.18852.

KI-basierte Erstellung von digitalen Zwillingen

Christian Malek Vasil L. Tenev Martin Becker
Fraunhofer-Institut für Experimentelles Software Engineering IESE, Kaiserslautern, Germany
{christian.malek, vasil.tenev, martin.becker}@iese.fraunhofer.de

Zusammenfassung

Dieses Paper beschreibt einen Ansatz zur automatisierten Generierung von Verwaltungsschalen (Asset Administration Shells) für digitale Zwillinge mithilfe großer Sprachmodelle (LLMs). Ziel ist, aus unstrukturierten PDF-Produktdatenblättern relevante Informationen zu extrahieren und diese in standardisierte Teilmodelle zu überführen. Dabei werden individuelle Prompt-Strategien, iterative Validierungen sowie Datenschutzaspekte berücksichtigt. Erste Ergebnisse zeigen, dass sich durch systematische Prompt-Anpassungen und heuristische Plausibilisierungen wichtige Produktdaten automatisch identifizieren lassen. Die Methode erleichtert Unternehmen die Digitalisierung ihrer Bestandsanlagen und unterstützt die Standardisierung industrieller Prozesse.

Keywords: Verwaltungsschale, Digitale Zwillinge, Asset Administration Shell, GenAI, Large Language Model, Datenextraktion

1 Einleitung und Motivation

Die Verwaltungsschale (engl. Asset Administration Shell, AAS) gilt als Schlüsselelement der Industrie 4.0, indem sie physische Assets digital repräsentiert und einen standardisierten Datenaustausch ermöglicht [1]. Eine Verwaltungsschale dient als zentraler Container, der alle relevanten Informationen zu einem physischen Asset bündelt. Innerhalb der Verwaltungsschale werden sogenannte Teilmodelle (Submodels) organisiert, die jeweils spezifische Aspekte des Assets abbilden – beispielsweise technische Daten, Betriebsinformationen oder Sicherheitsparameter. Jedes Teilmodell wiederum besteht aus einzelnen Produkteigenschaften, die konkrete Merkmale und Werte des Produkts darstellen, wie etwa Nennspannung, Abmessungen oder Materialbeschaffenheit.

Viele Unternehmen möchten ihre Bestandsanlagen als digitale Zwillinge abbilden, jedoch liegen relevante Informationen häufig nur in unstrukturierten und nicht maschinenlesbaren PDF-Dokumenten vor. Deren manuelle Auswertung ist zeitaufwändig und fehleranfällig.

Ziel dieses Beitrags ist es, eine *automatisierte* Methode zur Erstellung von Verwaltungsschalen zu entwickeln, die generative KI, insbesondere große Sprachmodelle (LLMs), einsetzt. Dies umfasst:

- Extraktion technischer Daten aus PDF-Produktdatenblättern,
- Strukturierung der Informationen in standardisierten Teilmodellen,
- Sicherung von Datenschutz durch den Einsatz lokaler Modelle.

2 Herausforderungen

Bei der automatisierten Extraktion aus PDF-Dateien ergeben sich mehrere zentrale Problemfelder:

1. **Nicht maschinenlesbare PDFs**: Komplexe Layouts, eingebettete Tabellen und mehrspaltige Formate erschweren eine verlustfreie Konvertierung.
2. **Individuelle Prompt-Gestaltung**: LLMs benötigen gezielt formulierte Eingaben, um relevante Daten korrekt zu erkennen. Ein generischer Prompt genügt oft nicht, da technische Daten stark variieren.
3. **Plausibilisierung**: LLM-Ausgaben können fehlerhaft oder inkonsistent sein. Mehrfache Durchläufe und kritische Validierungen sind erforderlich, um Halluzinationen zu reduzieren.
4. **Datenschutzvorgaben**: Da technische Dokumente vertrauliche Informationen enthalten, sind oft On-Premise-Modelle mit kleinerem Kontextfenster nötig, was die Extraktionsqualität mindern kann.
5. **Verlust kontextueller Zusammenhänge**: Bei sehr langen Dokumenten muss der Text in Segmente geteilt werden. Dies kann dazu führen, dass bestimmte Abhängigkeiten zwischen verschiedenen Dokumentpassagen nicht mehr eindeutig erkannt werden.

3 Lösungsansatz

Der hier vorgestellte Lösungsansatz zielt darauf ab, Produkteigenschaften automatisiert aus unstrukturierten technischen Dokumenten (insbesondere PDF-Produktdatenblättern) zu extrahieren und in das strukturierte Format des digitalen Produktpasses (engl. digital nameplate) [2] zu überführen. Dabei kommen maßgeschneiderte Prompts zum Einsatz, die für jede Eigenschaft eines Teilmodells spezifische positive und negative Beispiele, klare Anweisungen sowie relevante Spezifikationen beinhalten. Ergänzt wird

der Ansatz durch iterative Validierungsverfahren und Datenschutzmechanismen, die – auch unter den Einschränkungen lokaler Sprachmodelle – eine konsistente und zuverlässige Datenaufbereitung ermöglichen.

3.1 Methodik der Datenextraktion

Zunächst werden die PDF-Dokumente mithilfe von Konvertierungswerkzeuge [3, 4] in ein durchsuchbares Textformat überführt. Aufgrund der Heterogenität von Produktdatenblättern ist eine verlustfreie Konvertierung jedoch nicht immer garantiert, insbesondere bei Tabellen oder speziellen Layouts. Kommerzielle Tools liefern hier oft stabilere Ergebnisse als Open-Source-Lösungen [5].

Maßgeschneiderte Prompts pro Teilmodell-Eigenschaft. Um die Qualität der LLM-Ausgabe zu erhöhen, wird *für jede Eigenschaft eines Teilmodells* ein eigener Prompt eingesetzt. Dieser enthält:

- **Positive Beispiele**: Typische Werte (z. B. *„Nennspannung: 230 V“*).
- **Negativbeispiele**: Falsche Angaben, die ignoriert werden sollen (z. B. Leistungswerte als Spannung).
- **Konkrete Anweisungen**: Heuristiken zu erwarteten Einheiten, Wertbereichen und Formaten.
- **Auszüge aus Spezifikationen**: Falls verfügbar, z. B. relevante Normen oder Richtlinien.

Diese *individualisierten Prompts* fokussieren das LLM gezielt auf die jeweilige Eigenschaft und reduzieren die Wahrscheinlichkeit irrelevanter oder erfundener Informationen.

Iterative Validierung und Re-Prompting. Da LLM-Ausgaben trotz guter Prompts fehlerhaft sein können, werden Kontrollmechanismen integriert:

- **Mehrfache Extraktion**: Mehrere Durchläufe mit unterschiedlichen Parametern (Temperature, Sampling) und anschließendem Ergebnisvergleich.
- **Kritische Plausibilisierung**: Ein zweites LLM oder ein spezieller „Kritik“-Prompt überprüft, ob die extrahierten Werte plausibel sind (z. B. Einheiten-Check).
- **Menschliche Kontrolle (Human-in-the-Loop)**: Bei widersprüchlichen Resultaten kann ein Experte gezielt eingreifen und den Prompt anpassen.

Auf diese Weise entsteht eine Schleife aus Extraktion, Validierung und Feedback, die sukzessive zu konsistenteren Daten führt.

3.2 Datenschutz

In vielen Fällen erfordern Unternehmensrichtlinien, dass keinerlei externe Cloud-Dienste genutzt werden dürfen. Daher kommen verkleinerte On-Premise-Sprachmodelle zum Einsatz, die jedoch oft über kleinere Kontextfenster verfügen und weniger leistungsfähig sind als große Cloud-basierte Modelle. Dieses Trade-off zwischen Datensouveränität und Extraktionsqualität wird teilweise durch die iterative Prompt-Optimierung kompensiert.

4 Diskussion und Ausblick

Erste Versuche zeigen, dass ein individualisierter Prompt-Ansatz die Genauigkeit der Datenextraktion erheblich steigert. Dennoch ist eine automatisierte Lösung ohne menschliche Kontrolle derzeit schwierig, da Informationen in PDF-Dokumenten oft lückenhaft sind oder bei der Dokumentenkonvertierung in ein für die LLMs geeignetes Format zu Fehlinterpretationen und On-Premise-Modelle rasch an kontextuelle Grenzen stoßen.

Zukünftige Arbeiten konzentrieren sich darauf, noch flexiblere Prompt-Schablonen zu entwickeln und die Validierung mithilfe externer Datenbanken (z.B. Referenzwerte oder Materialdaten) zu verbessern. Außerdem ist die Integration aktuellerer LLM-Modelle geplant, um die Extraktionsgenauigkeit weiter zu erhöhen. Langfristig könnte der Ansatz auf andere Arten unstrukturierter Dokumente übertragen werden und so die Digitalisierung auch anderer Inhalte vorantreiben und ermöglichen.

5 Fazit

Die vorgestellte Methode zeigt, dass eine gezielte Kombination aus PDF-Konvertierung, individualisierten Prompts und iterativer Validierung den manuellen Aufwand zur Erstellung von Verwaltungsschalen reduzieren kann. Insbesondere der „Tailormade“-Ansatz pro Teilmodell-Eigenschaft steigert die Genauigkeit der Datenextraktion. Auf diese Weise lassen sich unternehmensweit Bestandsanlagen schrittweise digitalisieren und in standardisierte digitale Zwillinge überführen, was langfristig die Effizienz und Interoperabilität in der Industrie 4.0 steigert.

Literatur

[1] *The Exchange of Information between Partners in the Value Chain of Industrie 4.0.* No. 1, Federal Ministry for Economic Affairs and Energy (BMWi).

[2] “IDTA 02006-2-0 Digital Nameplate for Industrial Equipment.”

[3] “Py-pdf/pypdf.”

[4] Docling Team, “Docling.”

[5] H. Bast and C. Korzen, “A Benchmark and Evaluation for Text Extraction from PDF,” pp. 1–10.

Experiences with Combining Proactive with Retroactive Feature Tracing

Sandra Greiner
University of Southern Denmark, Demark

Abstract. Reverse engineering feature traces is crucial to systematically support organized reuse in evolving highly configurable software systems. A feature represents a user-visible characteristic of the software which allows for its configuration and may be mapped onto almost all artifacts in a software system. Consequently, variable parts can optionally be included or excluded from variants of the software system. Methods to build feature traces, which establish a mapping between features and software artifacts, range from purely manual to fully automated techniques. Feature traces can be built proactively during development or retroactively recovered mainly based on heuristics. In a published study, we examined how minimal seeds of proactive traces can increase the accuracy of a retroactive feature tracing method. In this article, we summarize the key findings and outline actions and research directions following up on the published experiment.

Keywords. feature tracing, reverse engineering, highly configurable software systems, software product lines

1 Motivation and Background.

Reverse engineering software product lines aims at migrating single software systems to the systematic management of variability of a family of related products [5]. Thus, reverse engineering represents the starting point for optimizing and modernizing the development and maintenance of highly configurable software. First, features which represent mostly configurable user-visible characteristics need to be identified in a given software project or a set of related software products. Next, the extracted features are organized in a variability model, which defines constraints among the features. Crucially, then a mapping of features onto artifacts in the project is determined. *Feature traceability*, which identifies all artifacts needed to realize a feature, results from this mapping.

Several methods exist to locate and extract features and to build feature traces for software projects. All of them come with their pros and cons [1]. *Proactive* feature tracing builds feature traces during development time and mainly boils down to a manual task requiring discipline, organizational guidelines, or (semi-)automated supportive techniques. *Retroactive* feature tracing tries to establish feature traces after features have been developed and mainly relies on automated, heuristic methods. While the accuracy and reliability of proactively acquired feature traces can be considered higher, they come with a manual annotation burden.

2 Adding Proactive Feature Traces to Comparison-Based Feature Location

In a previous study, we examined how proactively collected traces can increase the accuracy of comparison-based feature location, a retroactive, static feature tracing technique [2]. Comparison-based feature location assumes several co-existing variants and compares them gradually to identify common and distinguishing artifacts [4]. Eventually, it combines the respective features sets in a Boolean expression and maps them onto the code artifacts. It relies on heuristics (e.g., combine the collected feature sets in a conjunction or disjunction) which affect the precision and recall of the built feature traces.

Still, in many software artifacts, proactive knowledge is encoded. Particularly, in the annotation-based software systems examined in our study (C/C++ and Java), preprocessor directives enclose variable source code. Thereby, they map a presence condition onto the source code which can be used to derive accurate feature trace information.

In a controlled experiment, we simulated in publicly available Github repositories how increasing seeds of proactive traces improve the overall accuracy of a feature trace in comparison-based feature location methods when being propagated and overruling retroactive traces. We find that the median F1 scores increase at minimum by 20% when adding 10% proactive feature traces in all examined subject systems. Moreover, we find that the effect of adding proactive traces decreases, the more variants we compare.

On the one hand, the gained results demonstrate that minimal proactive knowledge can boost the overall accuracy of the feature traces. On the other hand, achieving better accuracy with more compared variants is partly expected. The more (heterogeneous) the variants are, the more overlapping and distinguishing artifacts can be found. This is also in line with dy-

namic feature tracing which in general improves with more execution runs and data [3]. Still, the confirmed insight that only few reliable manual input and few compared variants suffice to achieve high accuracy promises fruitful to be exploited optimally in static retroactive tracing; e.g., to migrate a small set of products to a shared platform.

3 Research Directions

Our experiences give rise to several research directions: we may examine the effects of errors in the proactive traces, and the effect in different feature trace acquisition methods including qualitative analyses of the proactive trace information. We may also examine which degree of accuracy is needed to modernize software automatically.

Effect of erroneous proactive traces. In our study, we assumed that the proactive features are of perfect correctness. While it can be assumed that proactive traces are of high reliability, they are mainly established through humans (or proposed by semi-automated heuristic-based methods). Still, for instance, developers may not always be aware of the full complexity of the software system and wrongly assign too broad or too restrictive feature information. A logical next step is to examine how and which errors affect the accuracy of combined proactive and retroactive feature tracing, answering questions, such as: Are errors of a diminishing effect? Are there types of errors that have a more severe effect and should be prevented or fixed by supportive tools?

Effect of acquisition method and artifact types. In our study, we examined annotation-based software systems and extracted the proactive traces from annotations (e.g., preprocessor directives) applied by the developers. Furthermore, we examined a comparison-based algorithm as retroactive feature tracing method. Still, a plethora of other techniques exist to establish proactive and retroactive traces. Thus, it is questionable whether the boosting effect remains the same when applying proactive traces in different retroactive tracing methods and in complex systems with scattered and tangled features.

Feature traces in action. Lastly, feature traces collect information about features and which artifacts realize them. One usage of this information is to consistently evolve highly configurable system. Thus, by employing the traces in different application areas (e.g., to change or remove features), we can find out whether and which degree of feature trace accuracy satisfies the aim of its usage. Similarly, we may examine which feature tracing technique sufficiently scale to large configurable software systems, such as the Linux kernel[1] or the Marlin[2] printer software which exhibit thousands, of configuration options challenging the feature retrieval techniques in itself.

4 Conclusion

Feature tracing is a crucial method in maintaining evolving highly configurable software. Its accuracy is key to properly utilize the knowledge encoded in the trace. Thus, future research needs to examine in more detail how and to what degree accuracy can be improved to serve specific applications.

References.

[1] Sandra Greiner and Timo Kehrer. Is the feature traceability problem already solved? *Softwaretechnik-Trends*, 44(2), 2024.

[2] Sandra Greiner, Alexander Schultheiß, Paul Maximilian Bittner, Thomas Thüm, and Timo Kehrer. Give an inch and take a mile? effects of adding reliable knowledge to heuristic feature tracing. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference (SPLC '24)*. ACM, 2024.

[3] Rainer Koschke and Jochen Quante. On dynamic feature location. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 86–95. ACM, 2005.

[4] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability extraction and modeling for product variants. *Softw. Syst. Model.*, 16(4):1179–1199, 2017.

[5] Roberto E. Lopez-Herrejon, Jabier Martinez, Wesley Klewerton Guez Assunção, Tewfik Ziadi, Mathieu Acher, and Silvia Regina Vergilio, editors. *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer, 2023.

Acknowledgments We thank Xhevahire Tërnava, Alexander Schultheiss, Paul M. Bittner, Thomas Thüm, Timo Kehrer, Bernhard Gumplmair, and Paul Grünbacher for fruitful discussions that partly influenced this article.

[1] `https://github.com/torvalds/linux`

[2] `https://github.com/MarlinFirmware/Marlin`

Fallstudie in Requirements Reengineering

Harry M. Sneed & Stephan Sneed
Softing Kft. H-1091 Budapest
Harry.Sneed@T-Online.de

Eine änderungsfreundliche Anforderungsdokumentation ist eine wichtige Voraussetzung für eine evolutionäre Softwareentwicklung. Es muss möglich sein, die Anforderungsdokumente elektronisch zu speichern und jederzeit ohne Qualitätsverlust anzupassen. Dazu müssen sie wohlstrukturiert und indiziert sein. Es komme darauf an, die zu verändernden Stellen im Dokument möglichst schnell und genau wiederzufinden. Es sollte auch möglich sein den veränderten Text gegen den veränderten Code zu prüfen, um Differenzen festzustellen. Schließlich sollte es möglich sein, System-Testfälle aus der Anforderungsspezifikation abzuleiten. Die Anforderungsdokumente müssten deshalb neben dem Code fortgeschrieben werden. Sonst driften Code und Dokumentation immer weiter auseinander. Um die Anforderungsdokumentation auf dem gleichen Stand wie der Code zu erhalten ist es notwendig sie regelmäßig zu überarbeiten, bzw. zu reengineeren.

Der **Requirements Reengineering Prozess** vollzieht sich in vier Schritten:

1. Messen des aktuellen Anforderungsdokuments
2. Restrukturieren des Dokuments
3. Markieren des restrukturierten Dokuments
4. Parsen des markierten Dokuments

Messung der Anforderungen

Im *ersten Schritt* werden nicht nur die Anforderungen, sondern auch die Seiten, die Zeilen, die Abbildungen, die Abschnitte und alle erkennbaren Anforderungsentitäten gezählt. Außerdem werden die neu gewonnenen Meßwerte mit den alten Meßwerten verglichen und die Abweichungen festgehalten. Der Bericht fasst die einzelnen Messwerte in einem einzigen allumfassenden Metrik-Bericht zusammen. Damit gewinnt der Anforderungsanalytiker einen Überblick über den Umfang des Dokuments und kann den Aufwand für dessen "Reengineering" mit Function-Points, Data-Points und Object-Points genauso schätzen wie mit den Code-Messwerten, nur hier findet die Schätzung zu einem früheren Zeitpunkt auf der Basis der Anforderungsspezifikation statt anstatt später auf der Basis des Codes. Dafür, dass die Aufwandsschätzung zu Beginn des Projektes stattfinden kann, muss der Stakeholder in Kauf nehmen, dass die Schätzgrößen etwas fuzzy sind. Deshalb brauchen wir automatisch verarbeitbare Dokumente wie jene, die hier in diesem Beitrag geschildert werden [Sneed07].

Restrukturierung der Anforderungen

Der *2. Schritt* erfolgt manuell durch den zuständigen Analytiker. Das Ziel ist es, eine passende Gliederung der Anforderungstexte zu finden und die Texte durch „cut and paste" neu zu ordnen. Die Textabschnitte in einem herkömmlichen Anforderungsdokument sind unterschiedlich groß. Die Größe hängt vom Inhalt ab. Bei der Restrukturierung sollten die kleinen zusammengelegt und die großen aufgeteilt werden. Die maximale Absatzgröße soll der Analytiker selbst festsetzen. Auf keinen Fall darf ein Textabschnitt länger als eine Seite sein, d.h. maximal 25 Zeilen.

Was die Reihenfolge der Textabschnitte anbetrifft, empfiehlt es sich mit dem Kontext der Applikation zu beginnen und dann zu den Zielen des Projektes überzugehen. Auf die Ziele des Projektes sollten die für die Zielerfüllung erforderlichen Geschäftsprozesse und die von diesen Geschäftsprozessen benutzten Geschäftsobjekte und Geschäftsschnittstellen folgen. Über die Geschäftsobjekte und Geschäftsprozesse kommen wir zum Produkt und über die Benutzeranforderungen zum Projekt. Eine gute Praxis ist es, die Datenwelt zuerst darzustellen, weil sie die Grundlage für die Vorgänge ist. Nach dem Objekt- und Vorgangsmodell folgen die Geschäftsregeln. Hier zieht sich die Grenze vom allgemeinen Geschäftsmodell zu dem projektspezifischen IT-Modell. Nach den allgemeingültigen Regeln, die nur für alle Vorgänge im Betrieb gelten, kommen die funktionalen Anforderungen. Das sind jene Funktionen, die nur durch dieses Projekt zu erfüllen sind. Sollte es Verweise von einer Anforderung auf eine andere geben, sollte der Anforderungstext, auf den verwiesen wird, in die verweisende Anforderung hineinkopiert werden. Auf diese Weise werden Abhängigkeiten zwischen Dokumenten abgebaut und die Lesbarkeit erhöht. Die wichtigsten Beziehungen für die Restrukturierung der Anforderungen sind die Daten- und Funktionsbäume. Sie werden in Teilbäume aufgeschnitten und die Teilbäume in separaten Textdokumenten präsentiert. Beide Bäume werden mittels XML-Syntax dargestellt.

Dokument-Markup

Der *dritte Schritt* in dem Reengineering Prozess ist das Mark-up der Anforderungsdokumente. In den Dokumenten kommen verschiedene Entitätstypen vor, physische Entitäten wie z.B. Bildschirmoberflächen, Dateien, Nachrichten und Listen, sowie abstrakte Entitäten wie Datenmengen Ereignisse, Funktionen und Auslöser. Datenelemente gehören zu den physischen Entitäten. Sie sind Attribute der Mengenentitäten, denen sie zugeordnet sind. Ein Scanner-Tool durchsucht die Anforderungstexte nach Hauptwörtern. Jedes Hauptwort wird als potentielle Datenentität behandelt und in eine Entitätentabelle eingetragen. Anschließend erfolgt eine Bereinigung der Tabelle durch den Requirements-Engineer. Er stößt ein Programm an, um die Hauptwörter der Sätze unter den vordefinierten Entitäten zu erkennen. Wenn sie gefunden werden, wird das Kennwort für diese Entität am Anfang der Zeile, in

der die Entität vorkommt, als Tag eingetragen. Sollten zwei oder mehr Entitäten in der gleichen Zeile vorkommen, wird die Zeile dupliziert. Die Tags werden vom Tool zunächst generiert, aber sie können vom Analytiker nachher beliebig umbenannt werden [Briand17].
Als Beispiel einer Anforderungsmarkierung wird folgender Ausschnitt aus dem Anforderungsdokument einer Gesundheitsakte herangezogen.

&FREQ_3_4 Beifügen_Nachhol_Impftermine
Auf Grundlage des gültigen Österreichischen Impfplanes muss bei der Abfrage des e-Impfpasses eines Teilnehmers das Datum der nächste(n) fälligen Impfungen und etwaige Nachhol-Impftermine beigefügt werden. Von GDA manuell eingefügte Impftermine müssen unterstützt werden und diese dürfen von der Fachlogik nicht überschrieben werden.

Das Mark-up Tool beschränkt sich auf das notwendigste, nämlich ein Entitäten-Typ Kennzeichen und einen Entitäten-Bezeichner. Damit werden die Anforderungs-Entitäten für den Text-Parser erkenntlich gemacht. Der Rest ist dem Textanalyse-Tool überlassen. Das Textdokument wird zunächst automatisch verarbeitet und anschließend manuell korrigiert. Die Hauptwörter in den vorhandenen deutschen Sätzen werden auch ohne Markierung durch den Text-Scan erkannt und ausgewiesen. Die vereinigte Menge von original deutschen Hauptwörtern und künstlich eingefügten Key-Wörtern bildet die Key-Word Tabelle, die vom Parser benutzt wird, um die Sätze zu interpretieren [Sneed22].
Der Text, der auf ein Schlüsselwort folgt, wird bis zum nächsten Schlüsselwort bzw. bis zum Ende des betreffenden Absatzes herausgeschnitten und dem aktuellen Schlüsselwort zugewiesen. Dieser Vorgang ist gleich einer Zerschneidung des Textes. Es ist so, als ob das Analyse-Programm mit einer Schere den Text in viele Schnipsel zerschneiden würde. Die Schnittstellen sind die Schlüsselwörter. Überall, wo ein Schlüsselwort auftaucht, wird der Text zerschnitten. Zum Schluss gibt es eine Menge Textausschnitte, einen für jedes Hauptwort im Text und einen für jedes Schlüsselwort. Die Textausschnitte sind durch den Entitäten-Typ und den Entitäten-Bezeichner eindeutig identifiziert und können nach Entitäten-Typ, z.B. Datenobjekt oder Anwendungsfall, zugeordnet werden.

```
0001 07 ACT  &Akteur
0009 05 CASE &Anwendungsfall
0010 01 DATA #Datenelement
0011 05 GOAL &Ziel
0012 08 INPT  &Eingabe
0014 07 OBJT  &Objekt
0015 01 INTR @Schnittstelle
```

Die zwei Ordnungsmerkmale (Hauptwörter und Schlüsselwörter) werden durch die ursprüngliche Zerlegung des Textes durch Titel und Untertitel ergänzt, so dass die Analytikerin jetzt drei Sichten auf den Text hat:

- eine Sicht nach Hauptwörtern bzw. Datenobjekten,
- eine Sicht nach Schlüsselwort bzw. Entitäten-Typ,
- eine Sicht nach Titel und Untertitel.

Parsing der marked-up Anforderungstexte
Der *4. Schritt* erfolgt vollautomatisch mit einem Textverarbeitungswerkzeug. Die automatisierte Anforderungsanalyse ist eine Art KI-Leicht [Sult24]. Das gespeicherte Fachwissen liegt in einem Pool diverser Dokumente wie Excel Tabellen, marked-up Textdateien, XML Dateien und einem Requirements-Repository. Sie beinhalten alle relevanten Details aus dem gespeicherten Fachwissen. Diese Dateien werden durch sprachspezifische Textverarbeitungstools verarbeitet, um die Grammatikelemente auszugraben. Anschließend werden jene Elemente miteinander über die Namen und Typen in den Schlüsselwort-Tabellen durch die Suchalgorithmen verknüpft. Es entstehen dadurch neue Tabellen, mit denen die einzelnen Textattribute verknüpft sind. Aus den verschiedenen Modellen werden die Entitäten und Aktionen in einem weiteren Schritt zusammengeführt. Das Ergebnis ist ein zusammengesetztes Modell aller Entitäten und Beziehungen aus den Lasten- und Pflichtenheften.
Die Bedingungen im Text werden ohnehin durch die Verwendung bestimmter Prädikate, z.B. „wenn“, „falls“, „solange“, „oder“, erkannt. Das Tool pflegt eine Tabelle der Bedingungswörter in deutscher Sprache. Wenn eines erkannt wird, wird der Satz, in dem es vorkommt, als Bedingungssatz markiert und kommt in die Bedingungstabelle.
Das Endergebnis ist ein semi-formales Anforderungsmodell, das sich vollautomatisch weiterverarbeiten lässt. Es dient erstens dazu den Weiterentwicklungsaufwand zu schätzen und zweitens dazu fachlogische Testfälle zu generieren.

Erfahrung mit Requirement-Reengineering
Die Anwendung der Requirements-Reengineering Technologie ist relativ jung. Sie wurde bisher vom Autor in vier Projekten benutzt. Das erste Projekt war für die Evolution einer staatlichen Sozialversicherung mit 242 Anforderungen, *87531 Haupntwörtern, 11440 Datenentitäten, 1232 Geschäftsregel und 266 Anwendungsfällen. Das zweite Projekt war die Evolution eines neuen Krankenversicherungssystem mit 229* funktionalen Anforderungen 12.957 Hauptwörter, 949 Datenentitäten, 1.454 Geschäftsregeln und 267 Anwendungsfällen. Das dritte Projekt war die Erweiterung einer elektronischen Gesundheitsakte mit 125 Anforderungen, 13388 H*auptwörtern, 133 Datenentitäten, 892 Geschäftsregeln und 101 Anwendungsfällen.* Das vierte Projekt war die nachträgliche Modellierung eines elektronischen Kassen-Abrechnungssystem mit 50 funktionalen Anforderungen, 2.285 Hauptwörter, 12 Datenentitäten, 68 Geschäftsregeln und 6 Anwendungsfällen. In allen vier Projekten wurde eine vollautomatisierte Nachdokumentation innerhalb einer Woche produziert.

Reengineering einer bestehenden High-Code KI-Applikation zu einer Low-Code Lösung

Ben Rymar, Sandro Hartenstein, Andreas Schmietendorf
Hochschule für Wirtschaft und Recht Berlin
s_rymar23@stud.hwr-berlin.de, sandro.hartenstein@hwr-berlin.de, andreas.schmietendorf@hwr-berlin.de

1. Motivation und Ziele

Im Mittelpunkt des Beitrags steht das Reengineering bzw. die ggf. notwendige Portierung einer bestehenden Python-Applikation (erstellt mit Hilfe des Jupyter Notebooks) zu einer Low-Code-Lösung. Im Kern beschäftigt sich diese mit Häufigkeits- und Sentiment-Analysen transkribierter Mediationssitzungen. Unterstützt wird auf dieser Grundlage die Professionsforschung zum Berufsbild des Mediators. Mit Hilfe der Low-Code-Application Plattform (kurz LCAP) sollte eine bessere Integration der Domänenexperten (Mediatoren, Soziologen, …) unterstützt werden, so dass diese aktiv in die Entwicklung einbezogen werden können. Ebenso sollte dem prototypischen Charakter der KI-Applikation im Sinne der benötigten Entwicklungsgeschwindigkeit, Flexibilität und Anpassungsfähigkeit stärker Rechnung getragen werden. Während das Quelldatenmanagement, benötigte KI-Funktionen und Ergebnisvisualisierungen in ähnlicher Weise bereitzustellen waren, sollte es nicht erforderlich sein Python-Quellcode zu erzeugen.

2. Existierende Arbeiten

Eine interessante Arbeit zu den Funktionen von LCAP findet sich unter [1]. Hinsichtlich des hier im Mittelpunkt stehenden Reengineerings zeigen sich allerdings nur implizite Bezüge im Zusammenhang mit den folgenden Aspekten:

- Funktionen der Entwicklungsumgebung (u.a. Integration handgeschriebenen Quellcodes)
- Unterstützte Datenbanktechnologien (u.a. SQL/RDBMS, Key/Value Stores)
- Typen unterstützter Nutzerschnittstellen (Web, Native Apps, Desktop)
- Funktionen der generierten Applikationen (u.a. Collaboration, Integration von REST-APIs)
- Deployment & Operation (u.a. Serverless, Cloud und Monitoring)

[2] beschäftigt sich mit ausgewählten Herausforderungen des Einsatzes von LCAP. Das angesprochene Versionsmanagement und die benötigten Software-Tests stellen ein wesentliches Erfolgskriterium für überführte Anwendungen dar. Darüber hinaus stellen gerade KI-Lösungen besondere Anforderungen an die Vertrauenswürdigkeit im Sinne der Nachvollziehbarkeit und Erklärbarkeit konkreter Ergebnisse, was durch den Einsatz einer LCAP nicht obsolet wird.

Aktuell finden sich wenige Arbeiten, die sich mit der Überführung einer High-Code zu einer Low-Code Lösung beschätigen. Aus Sicht der Autoren wäre es sinnvoll, sich mit dem Vorgehen, der ggf. automatisierten Überfügung von Artefakten des Altsystems, Aspekten der Qualitätssicherung oder auch dem benötigten Versions- und Konfigurationsmanagement zu beschäftigen.

3. Auswahl einer LCAP

Neben einer exakten Analyse der zu portierenden Anwendung hinsichtlich der abzubildenden Funktionen und der eingesetzten Technologien gilt es eine LCAP nachvollziehbar (vgl. Abb. 1) auszuwählen.

Abbildung 1: Auswahlaspekte einer Low-Code-Plattform (Quelle: [3])

Für eine Operationalisierung der Auswahlaspekte gilt es, die projektspezifischen Anforderungen heranzuziehen. Im konkreten Fall bezogen sich diese auf:

- Benötigte (KI-) APIs (Statistik und Sentiment)
- Umgang mit Quelldaten (Transkripte)
- Konfiguration und Prompting
- Ergebnisbereitstellung (Visualisierung)

Zur Vermeidung einer Herstellerabhängigkeit (vgl. [2]) sollte eine hinsichtlich des verwendeten Softwarestacks offen gelegte Lösung (Open Source) zum Einsatz kommen.

Die Auswahl einer geeigneten Low-Code-Plattform (LCAP) erfolgte anhand projektspezifischer Anforderungen. Im hier betrachteten Fall standen insbesondere angebotene KI-APIs und die Offenheit der Lösung im Vordergrund. Zu diesem Zweck wurde zunächst ein mehrstufiges Bewertungsverfahren durchgeführt, in dem 21 verschiedene LCAPs anhand zentraler Kriterien (u.a. KI-Funktionalität, Lizenzmodell, Integrationsfähigkeit) analysiert wurden. Aus dieser Bewertung ging eine engere Auswahl hervor, in der Mendix und appsmith (hier verwendet) aufgrund ihrer unterschiedlichen Ausrichtungen in Bezug auf KI-Funktionen und Offenheit genauer betrachtet wurden.

4. Aspekte des Reengineerings

Im Zuge des Reengineerings wurden die funktionalen Bestandteile der zuvor in Python und Jupyter-Notebooks realisierten High-Code-Implementierung in eine Low-Code-Lösung überführt. Die ursprüngliche Anwendung nutzte zentrale Komponenten wie das zeilenweise Laden der Transkriptionsdateien, die Konfiguration von KI-Parametern (z. B. *Large Language Model (LLM)-temperature* oder *REST-Endpoints*) sowie die Prompt-Steuerung, welche verschiedene Sprachmodelle wie GPT 3.5, LLAMA 2 oder BERT adressierten.

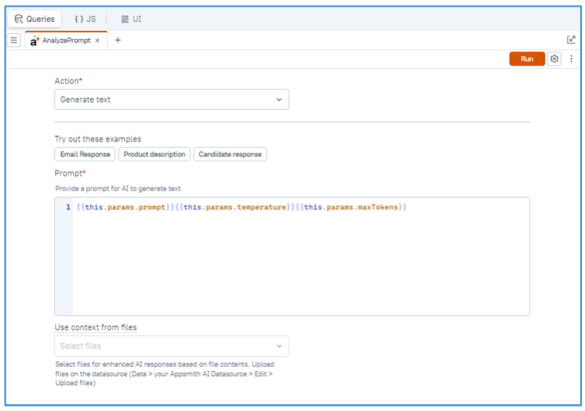


Abbildung 2: appsmith Prompt-Konfiguration (https://www.appsmith.com)

Diese Abläufe konnten in der neuen Umgebung ähnlich abgebildet werden, indem die bereits bewährten Parameter und Verfahren unverändert übernommen wurden. Insbesondere die Logik für das Einlesen und Vorverarbeiten der .txt-Dateien (Transkripte), das dynamische Zusammenstellen von Prompts (vgl. Abb. 2) sowie die API-Requests an externe Dienste wurden lediglich an die Low-Code-typischen Workflows angepasst, ohne dass tiefgreifende Änderungen an der Analysekonzeption erforderlich waren. Während in der High-Code-Lösung Python-Skripte zum Vorbereiten der Quelldaten eingesetzt wurden und die Ergebnisse mit Bibliotheken wie Plotly oder Mat-plotlib visualisiert wurden, ließ sich dieser Prozess in der Low-Code-Plattform über Queries zum Einbinden externer Daten- und Funktionsservices (u.a. REST-APIs), manuell erstellte JavaScript-Funktionen und Drag-and-Drop-Widgets abbilden. Da die LCAP bereits integrierte LLM-Funktionalitäten bereitstellt, war eine zusätzliche Anbindung externer KI-Modelle nur bedarfsweise notwendig, zum Beispiel für Szenarien, bei denen ein Zero-Shot-Ansatz nicht ausreichte oder eine Feinjustierung auf bestimmte NLP-Aufgaben (Natural Language Processing) erforderlich war. Auf diese Weise konnte die grundlegende Architektur der High-Code-Anwendung bezüglich des Zusammenspiels aus Analyse von Transkripten, Sentimentauswertung und Gesprächsvisualisierung erhalten bleiben, während der Implementierungsaufwand für Erweiterungen oder Anpassungen signifikant sank.

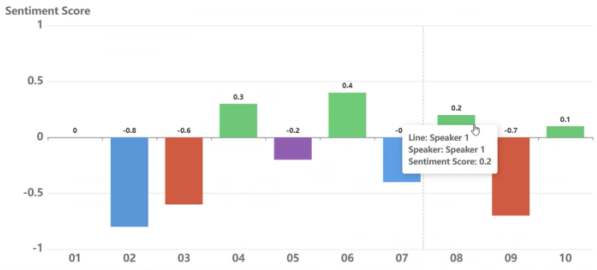


Abbildung 3: Low-Code-Output, Sentiments per row (je Zeile des Transkripts)

Beide Lösungen erfassen die Transkripte zeilenweise, ordnen sie den jeweiligen Sprechern zu und werten sie mit Hilfe einer Sentimentanalyse aus. Anschließend werden die numerischen Stimmungswerte genutzt, um Redeanteile, emotionale Verläufe und auffällige Aussagen zu visualisieren (vgl. Abb. 3). In den meisten Fällen stimmen die Resultate weitgehend überein, insbesondere bei der Erkennung von deutlich positiven oder negativen Aussagen. Lediglich bei komplexeren, vielschichtigen Formulierungen weist die High-Code-Lösung zum Teil differenziertere Einschätzungen auf, da dort spezifischere Vorverarbeitungen bzw. maßgeschneiderte Modellanpassungen zum Einsatz kommen.

5. Reflektion und Ausblick

Die Übertragung der High-Code-Anwendung in eine Low-Code-Umgebung hat gezeigt, dass sich grundlegende Analysemechanismen vergleichsweise nahtlos abbilden lassen. Zugleich treten jedoch spezifische Herausforderungen zutage, wenn es etwa um die Feinjustierung komplexer KI-Verfahren oder die tiefergehende Kontrolle der Datenvorverarbeitung geht, da Low-Code-Plattformen von Natur aus eher standardisierte Workflows unterstützen. Gerade in hochspezialisierten Anwendungsszenarien kann dies zu Einschränkungen führen, zum Beispiel bei der Integration eigener LLM-Modellentwicklungen oder der fortlaufenden Anpassung an domänenspezifische Besonderheiten. Trotz dieser Limitierungen bietet der Low-Code-Ansatz den entscheidenden Vorteil, dass Fachanwender leichter die Entwicklung unterstützen können, was die Iterationszyklen deutlich verkürzt und die Akzeptanz der Lösung fördert. Insgesamt ist zu erwarten, dass eine kontinuierliche Weiterentwicklung sowohl auf Plattform- als auch auf Anwendungsebene notwendig sein wird. Die Erfahrungen aus der bestehenden High-Code-Lösung, etwa im Hinblick auf individuell angepasste Vorverarbeitungsroutinen oder komplexe Modellarchitekturen, sollten dabei helfen, auch in der Low-Code-Variante optimierte KI-Funktionen zu realisieren. Gleichzeitig ist es ratsam, ein konzeptionelles Versions- und Konfigurationsmanagement einzuführen, um bei steigender Komplexität die Nachvollziehbarkeit von Änderungen zu wahren und mögliche Fehlerquellen frühzeitig zu erkennen. Da das im Projekt verwendete Zero-Shot-Verfahren für viele grundlegende Analysen ausreichend war, könnten künftige Studien untersuchen, inwieweit speziell trainierte Modelle die Genauigkeit weiter steigern, ohne den Low-Code-Charakter zu unterlaufen. Letztlich bietet die Verlagerung in eine Low-Code-Plattform die Chance, die Zahl der Anwenderinnen und Anwender zu erweitern, die mit solchen KI-gestützten Werkzeugen arbeiten können.

Quellenverzeichnis

[1] Kirchhof, J. C.; Jansen, N.; Rumpe, B.; Wortmann, A.: Navigating the Low-Code Landscape: A Comparison of Development Platforms. In: Proceedings of MODELS. Workshop LowCode, pp. 854 - 862, ACM/IEEE, Oct. 2023.

[2] Konersmann, M.: Challenges of Low Code PaaS Environments for Future Software Reengineering, in Softwaretechnik-Trends Band 44, Heft 2, Gesellschaft für Informatik e.V., 2024

[3] Schmietendorf, A.; Knuth, M.: Aspekte des Software Engineerings im Diskurs einer Low-Code orientierten Softwareentwicklung, Ausgewählte Ergebnisse des Projekts TAHAI, Logos-Verlag, Berlin

A Use Case and Method for Migrating a Low Code Business Information System to a Custom Three Tier Application

Marco Konersmann
ista SE, https://www.ista.com/
marco.konersmann@ista.com,

Abstract

Non-trained software engineers are able to rapidly develop productive software applications using low code platforms. Those platforms offer ease of use, compared to classical software development and operation environments. However, when software needs to be heavily customized, the advantage of standardization is replaced by the complexity of customizing. In this paper, we present an industry use case and our method for transforming a low code information system into a custom three-tier architecture.

1 Introduction

Low code development [2] is a method for rapidly developing software applications, even by non-trained developers. It uses modeling languages, domain-specific functionality, and a platform or code generator[1]. Platform-as-a-Service (PaaS) providers [1] offer a platform for developing, deploying, and running software. They usually provide an integrated development environment and monitoring capabilities. Low-Code on PaaS is often used in practice, especially for information systems, which often drive products and businesses. In a former paper [4] we describe the challenges that we see arising from Low-Code PaaS.

In our experience, low code platforms are beneficial in the development of business information systems, in the exploration of potential solutions, and for early steps towards a solution. But with rising expectations of the users regrading the functionality and quality of the software system, the effort for customization and optimization of low code applications tends to exceed the effort of classical approaches. In this paper, we present an industry use case for an application that started as a prototype and a rapid solution to a business opportunity, which then required extensive customization and optimization while facing growing numbers of users, more functional requirements, and higher quality requirements. We then present a method for migrating this low code application to a classical three tier architecture. The method shows potential—but also challenges—for automation, to be integrated in a software engineering method that embraces low code platform as a service for rapid prototyping and incremental development in early phases of a software lifecycle, before switching to a classical approach.

[1] compare https://www.se-rwth.de/essay/Low-Code/, accessed 2025, Mar 03

2 Use Case

Our use case is a business information system, that is used by domain experts for the analysis of large commercial and industrial properties for submetering services. The users enter information about the property, including metering points, and the areas supplied, e.g. with heat, cold, or water. The data is validated and processed to provide different views upon the data, including detailed reports about defects and issues to be resolved by specialized craftsmen, the location of devices within the property, bills of material, and submetering billing information. The data stored and processed within the application is highly interdependent. E.g., different devices can only be installed in specific pipe dimensions. Multiple laws, directives, and guidelines constrain the solution space for correct submetering.

The application has initially been built and operated with a low code platform to rapidly react to the business requirements in an agile, interdisciplinary team with only few software developers. On the low code platform it uses a database, standard UI components for data tables and detailed views of the data, customized UI components for efficiently entering the data, and logic components that trigger the generation of documents or synchronization with external systems. The low code platform has a custom Java backend for complex tasks like integration with internal tools and the generation of complex documents.

Implementing the functional requirements for the user interaction requires deep knowledge of the platform's functionality. While some properties are easily implemented in the low code platform, such as security, or user interfaces with standard views, many tools that are best-practice in software engineering and operation are limited by the platform approach, such as version control, systematic automated testing, or static code analysis. The increasingly complex requirements for data structures from the technical point of view, and user experience from the business

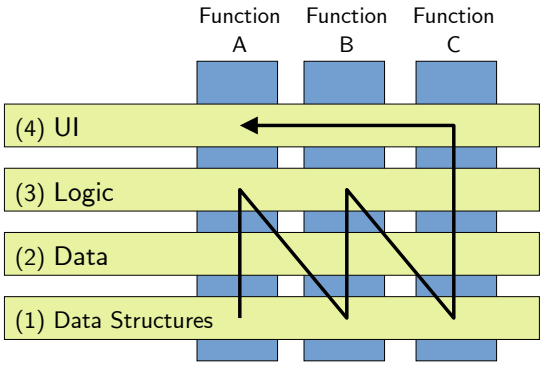


Figure 1: Migration of functional slices, layer by layer, user interface at last

perspective impose enough risks and impediments for the development and operation, that a migration to a non-low-code approach seems beneficial in the long run. This includes, besides performance requirements, the lack of control imposed by the constant platform evolution.

3 Incremental Migration Method

We embed the migration method into an incremental software reengineering approach with two repeated parts (see Figure 1): 1. vertical functional slices, and 2. horizontal layers for classical tier architectures.

We create the vertical, functional slices, prioritized based on risk and benefits. This means that we prioritize more complex, and most-used slices higher to mitigate risks as early as possible, and to benefit from short feedback cycles with our users.

For each vertical slice, we migrate layer by layer. For each functional slice, we first **(1)** migrate the data structures and **(2)** existing data to a cloud-based SQL database. Both data structure and data are provided via an OData [3] interface of the platform provider. We implement a tool using Java and the Spring framework to consume the OData data structures and create SQL data structures. Sequentially, we develop a tool in the same stack to consume the data and stream it into the SQL database. We connect the new database to the low-code platform to provide short outage times.

Then **(3)**, we migrate business logic to a spring boot application. In our use case, we use the same spring boot application for all logic. It is connected to the UI via REST interfaces. Since logic components are implemented using activity models, we can partly automate the migration. The activity diagram can only be exported in a proprietary base64 format, therefore we have to reimplement them in a modeling or programming language. The JavaScript code can be exported and reused for the components. Platform-dependent code needs to be adapted or connected to an emulation layer. We decided to implement the activity models manually in a Spring Boot application, since the logic components in the low code application were simple enough so that a manual transformation is beneficial over an automation.

At last, **(4)** we migrate the UI components. We distinguish between standardized UI components, for which automation is feasible, and custom UI components, which are more difficult to migrate automatically. The platform does not provide means to export UI specifications. Therefore, we have to reimplement them in a UI modeling language or in a programming language. We chose angular as target language for the UI, since it provides suitable replacements for the UI components used in the low code platform. The JavaScript code implemented as low code in the UI cannot be copied to the angular UI, since it is highly dependent on the platform. We decided to execute a manual transformation, since the amount of UI views is relatively low.

4 Outlook

The approach presented above is applied to a single use case of a limited size. The use case is small enough to be transformed by a small team of software engineers. We plan to apply the presented method to the use case above, and gain experience with the transformation. In future work, we will investigate methods for further automation, and to provide it organization-wide. Automation is limited due to the lack of openness of the low code platform. We can neither export information about the UI structure or components, nor can we export a readable format of the activities.

5 Summary

In this paper, we presented an industry use case and method for migrating a low code application to a custom three tier business information system. We highlighted potential and challenges for automation and described how to embed the method in an agile software reengineering approach. In future work, we will apply this method in use cases, adding and generalizing automation where it seems most-beneficial.

References

[1] L. F. Albuquerque, F. Ferraz, R. Oliveira, and S. M. L. Galdino. Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS. In *International Conference on Software Engineering Advances (ICSEA) 2017*, 2017.

[2] A. C. Bock and U. Frank. Low-code platform. *Business & Information Systems Engineering*, 63(6):733–740, November 2021.

[3] ISO/IEC. Information technology – Open Data Protocol (OData) v4.0 – Part 1: Core. International Standard ISO/IEC 20802-1:2016, International Organization for Standardization, 2016.

[4] Marco Konersmann. Challenges of Low Code PAAS Environments for Future Software Reengineering. *Softwaretechnik Trends, 26. Workshop Software-Reengineering und -Evolution 2024*, 44(2):26–27, May 2024.