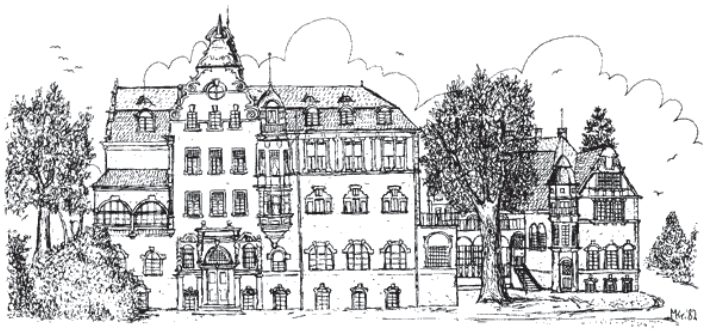
26. Workshop Software-Reengineering und -Evolution

WSRE 2024

der GI-Fachgruppe Software-Reengineering (SRE)

29.-30. April 2024

Physikzentrum Bad Honnef



26. Workshop Software-Reengineering und -Evolution

der GI-Fachgruppe Software Reengineering (SRE)

Herzlich Willkommen zum 26. WSRE! Diesmal treffen wir uns im ungewohntem 2-tägigen Format, was weiterhin den Nachwirkungen der Corona-Pandemie zu verdanken ist. Trotz aller Unruhe und organisatorischer Herausforderungen ist es (hoffentlich) wieder gelungen, ein interessantes Programm zusammenzustellen.

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 26 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:

- **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.
- **Networking**: Vernetzung zwischen den Teilnehmenden in den Pausen und beim gemütlichen Zusammensein am Abend.
- **Social Event**: Wir veranstalten diesmal ein Kneipenspiel rund um WSRE und Bad Honnef.

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:
Jochen Quante, Bosch Research (Sprecher)
Marco Konersmann, RWTH Aachen
Stefan Sauer, Universität Paderborn
Daniela Schilling, Delta Software Technology
Sandro Schulze, TU Braunschweig

Challenges of Low Code PAAS Environments for Future Software Reengineering

Marco Konersmann
ista SE

Low Code software on Platform as a Service (PAAS) environments allow for rapid development of productive business software systems by people that have no software development background. They enable rapid and relatively easy way of developing, deploying, and running business software.
In this talk, we identify and describe the challenges that Low Code software in PAAS environments pose on their future reengineering. We distinguish between the challenges imposed by the Platform as a Service approach, and those imposed by Low Code development. We also outline potential solutions to the challenges, some of which have to be addressed by the platform owners, while some can be addressed by the organizations using Low Code PAAS environments.

Is the Feature Traceability Problem Already Solved?

Sandra Greiner, Timo Kehrer
University of Bern, Switzerland

Abstract *Reverse engineering feature information from a family of software products or configurable software projects is crucial to systematically support organized reuse. A feature represents a user-visible characteristic of the software which allows for its configuration; the resulting variable source code can optionally be included or may also have to be present in each variant of the software. Thus, tracing features to artifacts in the software project is essential to support systematic reuse. Existing solutions to the problem of identifying and mapping feature information in configurable software, either enforce specific development processes, rely on extensive executions of the software, remain coarse-grained at the level of files, or assume language-specific information. These issues raise the question whether the problem of identifying and mapping features to artifacts in configurable software is already sufficiently solved?*

1 Motivation and Background

Reverse engineering software product lines [4] facilitates migrating from single-system development to the systematic management of variability in highly configurable software projects. Thus, it serves as starting point for optimizing and modernizing the development and maintenance of highly configurable software. In this context, reverse engineering centers around identifying features, which represent mostly configurable user-visible characteristics, in a given software project or a set of distinct software products. Once the features and potential constraints among them are identified, they are organized in a variability model, which defines constraints among the features, and a mapping of features onto artifacts in the project can be determined. *Feature traceability*, which identifies all artifacts needed to realize a feature, can result from this mapping.

Several methods were proposed in the past to locate and extract features in software projects. However, they either require dedicated knowledge-based development processes, assume specific implementation techniques, partly only available in certain programming languages (e.g., embedded annotations in form of preprocessor directives) or additional technological support. In summary, generic solutions without strong assumptions hardly exist. With the rise of software to control the highly configurable machine learning and the prevalence of polyglot repositories, however, we require generic methods to reverse engineer feature traces in those software projects. Thus, we raise the question whether feature traceability is already sufficiently solved and discuss existing methods in the sequel.

2 Feature Traceability Methods

Firstly, we can distinguish proactive feature tracing from retroactively computed traces. While the former imply a high reliability, as the developers should know which features they implement and provide this information manually. Retroactively computed traces can be derived statically or dynamically by employing heuristics [2]. While dynamic methods are mostly language-agnostic, they require more information, such as input data to exercise test cases, and execute the variable software in costly runs. Therefore, we do not regard methods which rely on dynamic retroactive feature tracing. For a representative set of static methods, we discuss their current state with respect to their granularity, genericity, and accuracy.

Version Control Systems (VCS) can exploit the development processes or proactively provided information to identify features. For instance, features can be developed on branches, variants can be developed on forked clones, and commit messages may state changed features. All this information can be used to extract feature traces.
Discussion While VCS development as resource may represent reliable information (developers should know which features they implement), the accuracy and granularity of the identified feature traces heavily depends on the developers' manual efforts and discipline. If conventions are missing, tracing features guaranteed correctly may become hardly possible. On the upside, the methods are per se language-agnostic as they consider line-wise text in state-of-the-art VCS.

Variation Control Systems base on the idea of managing variability-intensive software through a variability-aware VCS [1, 3]. Developers must commit against features and, hidden from the user, the repository maintains the development history and feature information. Thus, users can checkout a (potentially yet unseen) product by providing a feature selection.
Discussion While proactively collecting (partial) traces of high reliability, variation control systems imply disciplined development processes and a certain

knowledge about all features. On the upside, the resulting traces are maintained in a language-agnostic way and may be fine-grained. Due to computing either potential sets of matching artifacts, maintaining a superimposed set of variants and computing novel products on checkout, the performance may not scale to large systems. Even though generic, the process cannot be applied to existing configurable software which typically have not been committed against features and depends on the discipline of the developers.

Embedded Annotations represent a straightforward way to identify variability in software [2]. Code comments or preprocessor directives, similar as feature toggles, mark source code regions which can be conditionally included or excluded from a variant.
Discussion By providing embedded annotations (e.g., preprocessor directives in C/C++ source code), fine-grained feature traces at statement level can be extracted. Assuming comments encode the features in a similar way, it is possible to automatically compute feature traces for almost any source code artifact. Again, the accuracy of this automated feature tracing heavily relies on the developers' discipline to provide the feature traces proactively while allowing for fine granularity. Furthermore, current tools are language-specific and if not, the approach still assumes that features can be easily mapped onto software artifacts and parsed, a strong assumption when it comes to models or documentation.

Information Retrieval Techniques try to identify features retroactively through natural language processing. Latent Semantic Indexing, and Linear Discriminant Analysis are common techniques but perform significantly bad, for instance, compared to reinforcement learning [5]. While the former are per se retroactive methods, the machine learning approaches typically require manually provided correct traces for the learning, too.
Discussion While machine learning promises an accurate and fine-grained retroactive, automated tracing, it requires a specific seed of proactively provided feature mappings for training. The type of supported artifacts heavily depends on the training and, thus, is classified as language-specific.

3 Conclusion

To conclude, the majority of feature tracing techniques represent a trade-off between granularity, genericity, and accuracy. While previous literature surveys [6] reach similar conclusions, the state-of-the-art has not improved inasmuch as to solving feature traceability in its entirety.

Maintaining the configurability in modern and legacy highly configurable software, requires reverse engineering feature traces of high accuracy and fine granularity to systematically and efficiently support their variability. Potential solutions should start with optimizing existing techniques, for instance, by exploiting the benefits of small manual efforts in form of proactively provided traces to inform retroactive tracing. This knowledge is essential to develop novel techniques, such as methods employing machine learning, to exploit all information provided (implicitly) by the developers. Eventually, it is necessary to examine to what extent the existing and optimized techniques suffice to trace features in artifacts of highly configurable software, such as machine learning projects, which have been mostly neglected in the software product line community.

References.

[1] Sofia Ananieva, Sandra Greiner, Jacob Krueger, Lukas Linsbauer, Sten Gruener, Timo Kehrer, Thomas Kuehn, Christoph Seidl, and Ralf Reussner. Unified operations for variability in space and time. In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '22. ACM, 2022.

[2] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining feature traceability with embedded annotations. In *Proceedings of the 19th International Conference on Software Product Line*, page 61–70. ACM, 2015.

[3] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. Concepts of variation control systems. *JSS*, 171:110796, 2021.

[4] R. 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.

[5] Mukelabai Mukelabai, Kevin Hermann, Thorsten Berger, and Jan-Philipp Steghöfer. Featracer: Locating features through assisted traceability. *IEEE TSE*, 49(12):5060–5083, 2023.

[6] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin, editors, *Domain Engineering, Product Lines, Languages, and Conceptual Models*, pages 29–58. Springer, 2013.

Nachhaltigkeit im Fokus des Software-Reengineering

Stefan Sauer
Universität Paderborn

Das Thema Nachhaltigkeit spielt – verdientermaßen – eine zunehmend wichtige Rolle in der Softwareentwicklung. Dabei können die ökologischen, ökonomischen und sozialen Facetten der Nachhaltigkeitsbetrachtung gleichermaßen angegangen werden. Software kann selbst nachhaltig sein oder die Nachhaltigkeit der durch sie beeinflussten Produkte, Prozesse und Services positiv beeinflussen. Außerdem kann der Nachhaltigkeitsfokus auch auf den Softwareentwicklungsprozess und den Softwarelebenszyklus gerichtet werden. Aber was bedeutet dies alles im Zusammenhang des Software-Reengineering?

Dieser Beitrag liefert eine Positionsbestimmung und soll die Diskussion hin zu Forschungsfragen und einer Roadmap zur Behandlung des Themas Nachhaltigkeit im Kontext des Reengineering und der Evolution von Software leiten.

Transforming Code-Review Practices: An Action Research Study with Collaborative Software Visualization in SEE

Sarah Augustinowski, Azim Cheema, Lysander Gawenda
Dept. of Mathematics and Computer Science, University of Bremen
augustin@uni-bremen.de, azim@uni-bremen.de, lysander@uni-bremen.de

I. Introduction

SEEIA stands for Software Engineering Experience In Action. It is an continuation of an ongoing bachelor project (SEE), where the authors of this paper are part of. The main goal of SEE is to visualize software and presenting software projects as code cities in virtual rooms for distributed development teams and the goal in this cycle of the bachelor project is to test our software in action, hence the name SEEIA. The current emphasis on improving SEE centers around the code-review process, employing action research methods and collaborating with an industrial partner. This paper summarizes the current state of our ongoing project.

II. SEE

Before we introduce our new features in section IV, we take a look at what SEE where already capable of. SEE offers a virtual office where developers can join and navigate with their avatars, akin to a multiplayer game. Users can interact with SEE using conventional methods such as a keyboard and mouse, or opt for a more immersive experience using VR. For communication, SEE provides both voice and text chat functionalities. In the virtual office scene multiple tables are present, each containing different kinds of code cities representing software. Code cities are three-dimensional treemaps [1] that are reminiscent of large cities with skyscrapers due to their height and arrangement. The dimensions of blocks can represent different code metrics, such as the lines of code in the associated software part. The code-city approach allows for a first-hand understanding of software and its metrics just by visually comparing size, different materials or colors of the blocks. SEE comprised Implementation, Evolution, Dynamic and Architecture tables. With the help of the Implementation table users can get a comprehensive overview of their project and its metrics. The Evolution table visualizes project changes over time through animation. Portions of code that have been modified frequently may require closer examination or improved testing, with SEE expediting the identification of such areas. The Dynamic table displays interactions between software parts at runtime. Projects should adhere to a specific software architecture and the Architecture table facilicates uncovering possible architectural violations. Regardless of the table used, SEE displays the underlying code (Figure 1) and metrics for selected parts. SEE also includes a built-in web browser to search the web without having to leave the virtual room.

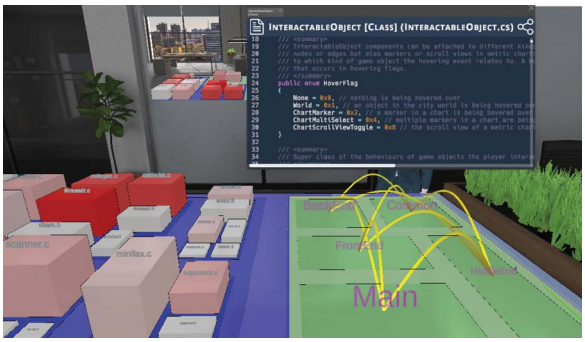


Fig. 1. Architecture table with Code Window in SEE

III. Action Research

The utilization of action research methodologies in software engineering contexts has only recently gained traction, despite being introduced in social sciences over 40 years ago [2]. Traditionally, methodologies based on the principles of experimentation or observation have been common practice in software engineering. According to Staron [2], an experimental approach struggles with finding participants with industry experience, identifying experiment objects that scale to an industrial context, and isolating treatments, resulting in non-representative contexts. While an observational approach, such as case studies, does not grapple with curated problems losing industrial relevance or a lack of expertise in the observed reference group, its primary goals are to observe, classify, and analyze software engineering practices rather than improving them.
Action research, with its collaborative long-term study environment and iterative process, addresses this issue and is thus chosen as our approach. Each iteration of the action research is divided into the following steps:

I. Diagnosing: In the initial step, a question is formulated, striving to be sufficiently answered by the time the last step, Evaluating Action, is reached. After-

wards, we got in touch with our industry partner and gathered information regarding our formulated question by asking questions directly through an interview and observations. In the context of SEE, our formulated question is as follows:
Which functions should be added to SEE to enhance the code-review process, in a way that the code reviewer gains a quick and comprehensive overview of the entire project context?

II. Planning Action: In this step, we decide on data-gathering methods and how to visualize our data sets concisely. Knowledge gaps are to be resolved, and this step concludes with the development of a prototype based on our initial ideas. As in our case, we decided on a vertical prototype.

III. Taking Action: The prototype is used by our industry partner in their everyday environment, while data is being collected on its behalf and other decided methods.

IV. Evaluating Action: We aim to address our proposed question by reflecting on our actions taken and evaluating the data we have collected.

Fig. 2. Action research process

IV. Code-Review

When it comes to working on a larger code base with a team, it has become common practice to review code, before merging it into the main project [3]. This practice is not only good for improving the quality of the code and therefore the quality of the whole project, but also to improve the skills of developers [3]. In order to use SEE for code reviewing, our industry partner requested three major enhancements, which focus on the code inspection part. In this part the reviewer inspects the code regarding coding standards, style violations, design/architecture choices and overall issues. Afterwards, comments associated to the code sections and overall feedback are being made. The first two enhancements refer to the difference between the main code base and the changed code, also known as ”diff”. To get a better overview on what changed, and what stayed the same in the submitted code, we introduced a new kind of code city, named ”Diff City”. This city visualizes the diff between two revisions of a project. The nodes (buildings) in the Diff City are respectively marked in case they got deleted, added or modified, so that the reviewer can get a better grasp on the changes being made. Additionally, we introduced a new ”Code Window”, that shows the diff in the source code of each code element represented as a node in the city that is selected by the reviewer. This way the reviewer can get a deeper understanding of the changes. We gain the two versions of the source code through an API called ”LibGit2Sharp”, which is letting us perform ”git show”. The actual diff on the source code level is then being made through the ”diff-match-patch” by Google. This way we have the option to get the source code from other version control systems like SVN and not fully rely on git. The third enhancement relates to test coverage, which is a metric to determine what portion of the source code got tested. In case of reviewing code changes, it is of great interest for the reviewer, how the test coverage changes with it. To actually see how the test coverage evolved, we implemented the ability to add coverage data for Gradle-built Java projects to the nodes of a code city. The associated data is received by a JaCoCo-XML file. Our goal is, that through these enhancements the reviewer gains a more comprehensive overview and deeper understanding of the changes being made, while maintaining the context of the whole project.

V. Conclusion

This ongoing study tries to answer the question ”Which functions should be added to SEE to enhance the code-review process, in a way that the code reviewer gains a quick and comprehensive overview of the entire project context?”. At this point it is important to note that we are currently in the first cycle of Action Research and are about to enter the Taking Action phase. In this next step, we will observe if the reviewer actually benefits from our implementations or if it is not yet sufficient enough. Currently we can only show how SEE could be integrated in code-reviews and how action research can potentially be leveraged to improve software to fit user needs. Utilizing action research and input from team members, additional feature ideas to help enhance the code-review process include:

- *Create code-cities directly from data of version control systems like Git*: This will make generating of code-cities more user-friendly.
- *A log function for recording important findings*: Allowing users to document their meetings in SEE.
- *Annotation of code*: Write and attach comments to specific parts of code.

To see the on-going implementation progress, please visit SEE’s publication on GitHub under the MIT license: https://github.com/uni-bremen-agst/SEE or visit https://see.uni-bremen.de for further information.

References

[1] B. Johnson and B. Shneiderman, “Tree-maps: a space-filling approach to the visualization of hierarchical information structures,” in *Proceeding Visualization ’91*, 1991, pp. 284–291.

[2] M. Staron, *Action Research in Software Engineering: Theory and Applications*, 1st ed. Springer Cham, 2020.

[3] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, “Modern code review: A case study at google,” in *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

Wissensmodell zur Identifikation unwirtschaftlicher Variabilität im Produktportfolio von Industrie 4.0-Unternehmen

Raphael Martin
Fraunhofer IESE, Kaiserslautern
raphael.martin@iese.fraunhofer.de

1 Thema

Diese Bachelorarbeit [1] befasst sich mit der Analyse und Optimierung der Wirtschaftlichkeit von Reengineering-Maßnahmen. Industrie 4.0-Unternehmen sind hier besonders im Fokus, da viele relevante Daten, wie Bestelldaten, Stücklisten und Konfigurationsinformationen, bereits erhoben werden oder ohne großen Mehraufwand erfasst werden können.

Ziel dieser Arbeit ist es, einen Ansatz zur Verarbeitung dieser Daten zu entwickeln, um eine Auswertung der Wirtschaftlichkeit mit möglichst wenig Daten zu ermöglichen und Bereiche mit besonders hohem Optimierungspotential zu identifizieren. Damit lassen sich Reengineering-Maßnahmen kosten-effektiver planen. Durch einen inkrementellen Ansatz soll es ermöglicht werden, mit einem möglichst kleinen Datensatz anzufangen und die Analyse für zukünftige Anwendungsfälle erweiterbar zu gestalten. Dies ist ein wichtiger Schritt zur Effizienzsteigerung von Unternehmen, da die Identifizierung und Reduzierung von unwirtschaftlicher Variabilität zu Kosteneinsparungen und Produktivitätssteigerungen führen kann.

2 Motivation

Um die ständig steigende Komplexität und die damit verbundenen Kostensteigerungen zu bewältigen, wird das bestehende Sortiment immer häufiger in Produktlinien [2] und Produktfamilien umgewandelt. Ein Aspekt dieser Umgestaltung ist es, auszuwählen, welche Maßnahmen zuerst angegangen werden. Hierbei müssen Unternehmen priorisieren, welchen Maßnahmen sie mehr Ressourcen zum Reengineering zur Verfügung stellen, oder welche Maßnahmen früher angegangen werden sollen. Dabei wird unter anderem zuerst auf die Wirtschaftlichkeit der Maßnahmen geachtet. Hierfür wird die Wirtschaftlichkeit der verschiedenen Merkmale genauer betrachtet und danach entschieden, welche Bereiche als Basis für die Architektur der Produktlinie ausgewählt werden und damit zuerst Mittel zur Umstrukturierung erhalten. Diese Planung der Reengineeringmaßnahmen soll mithilfe von fundierten Daten unterstützt werden, in dem die Wirtschaftlichkeit von einzelnen Merkmalen unabhängig voneinander untersucht wird.

3 Zielsetzung

Der Fokus der Arbeit ist hierbei die Erstellung eines erweiterbaren Wissensmodells, mit dem die Untersuchung der Wirtschaftlichkeit von Merkmalen datenbasiert unterstützt wird. Hierbei soll mit möglichst wenig Daten angefangen werden, damit erste Abschätzungen mit relativ wenig Aufwand schnell erhoben werden können. Durch die Erweiterbarkeit soll die Möglichkeit geschaffen werden, später mehr oder genauere Daten zu integrieren und bei Bedarf die Analysen zu modifizieren oder zu verbessern. Hinzu kommt, dass Teile der Daten für die Auswertung häufig aus unterschiedlichen Bereichen einer Firma stammen. Es muss also darauf geachtet werden, unterschiedliche Formatierungen zusammenführen zu können.

Zusätzlich soll das Modell transparent sein. Das heißt, dass Ergebnisse über jeden Verarbeitungsschritt bis hin zu ihrer Quelle zurückverfolgbar sein sollen.

4 Durchführung

Die wirtschaftliche Auswertung von Merkmalen benötigt eine Vielzahl an Daten. Diese beinhalten Bestelldaten sowie Stücklisten und Konfigurationsinformationen, mit denen wir ein zentrales Wissensmodell aufbauen können. Mit den Bestelldaten können Ähnlichkeitsanalysen durchgeführt werden, um eine Übersicht über die tatsächlich bestellten Merkmale der Produktvarianten aufzubauen. Mithilfe der Stücklisten und Konfigurationsinformationen lassen sich Hierarchiestrukturen innerhalb der Varianten herleiten und daraus ein Feature-Modell [3] erstellen. Zusammen mit den Kundenpreisen sollen nun mittels einer Kosten-Nutzen-Analyse unwirtschaftliche Merkmale identifiziert werden.

Das grobe Vorgehen ist wie folgt: Extrahieren der Roh-Daten (Bestelldaten, Konfigurationsinformationen, Preislisten, etc.) in fein-granularer Form, diese dabei aber möglichst unverändert in einer Datenbank abbilden. Dieses Vorgehen ermöglicht Transparenz, mit dem dem Anspruch der Nachvollziehbarkeit von Ergebnissen bis hin zu den ursprünglichen Daten.

Konfigurationsinformationen und Feature-Modell: Nach dem Importieren der Rohdaten wird

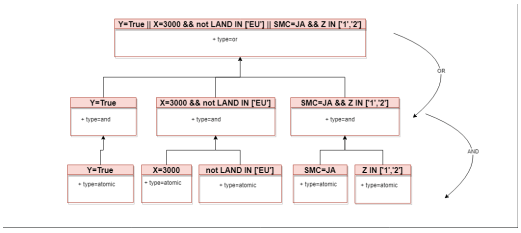


Abbildung 1: Visualisierung des Aufbaus eines abstrakten Syntaxbaumes für eine fiktive Bedingung

daraus ein Feature-Modell extrahiert und aufgebaut. Dies dient dazu, eine Abbildung der internen Struktur der Produktlinien zu generieren und zur Verfügung zu stellen. Zuerst werden alle Merkmale der Produktfamilie herausgesucht und anschließend wird die vollständige Belegung für jedes zuvor gefundene Merkmal gespeichert. Zusätzlich werden alle vom Unternehmen verkaufte Konfiguration der Produktfamilie aus den Rohdaten isoliert und mit den darin verwendeten Merkmalen des Feature-Modells verlinkt, damit später schnell überprüft werden kann, welche tatsächlich verkaufte Konfiguration welche Merkmale verwendet hat.

Preise und Kosten: Danach werden die Preise der verschiedenen Merkmale aus den Rohdaten ausgelesen und in Zusammenhang mit den Merkmalen im Feature-Modell gebracht. Hier muss auf etwaige Formatierungsfehler zwischen den Preislisten und den extrahierten Merkmalen geachtet werden.

Die Kosten sind klassischerweise nicht direkt auf Merkmale verteilbar, da sie häufig für komplexere Materialien oder Bestandteile anfallen. Um dieses Problem zu lösen, werden die Bedingungen, um das Material einzubauen oder den Bestandteil zu verwenden, genauer untersucht. Um herauszufinden, welche Merkmale zur Erfüllung der Bedingung benötigt werden, muss diese erst einmal auf ihre Bedeutung analysiert werden. Hierfür bauen wir einen Abstrakten-Syntax-Baum (AST) auf, und zerlegen ihn anhand seiner logischen Komponenten. Die atomaren Bestandteile dieser Bedingungen werden dann auf die Merkmale des vorher erstellten Feature-Modells abgebildet. Der Aufbau eines solchen AST ist beispielhaft in Abbildung 1 dargestellt.

Analysen und Auswertung: Im nächsten Schritt wird über eine semantische Analyse des AST ermittelt, wie viele Konfigurationen die gesamte Bedingung erfüllt haben. Dann wird für jedes Merkmal einzeln bestimmt, wie viele Konfigurationen die beteiligten Merkmale verbaut haben. Über dieses Verhältnis kann eine Abschätzung getroffen werden, wie ausschlaggebend ein bestimmtes Merkmal für den Einbau des kompletten Materials ist. Damit kann man eine Gewichtung der Materialkosten zwischen den Merkmalen erstellen und somit die Kosten eines Materials auf seine beteiligten Merkmale verteilen.

Ein abstraktes Beispiel ist z.B. eine GPS-Funktion, die als Bedingung die Merkmale 'Touchscreen' **und** '4G Konnektivität' benötigt. Dazu wurde ermittelt, dass von X Konfigurationen 10 einen Touchscreen haben und 6 die 4G Konnektivität. Nach der Analyse finden wir nur 5 Konfigurationen, die beides gleichzeitig erfüllen. Die Gewichtung ist damit 5/10 für den Touchscreen und 5/6 für 4G. Nach Normierung ergibt sich daraus 37,5% für den Touchscreen und 62,5% für 4G. Diese Gewichtung repräsentiert die kundenseitige Signifikanz einzelner Merkmale und dem folgend werden die Kosten des Materials mit dieser Gewichtung auf die Merkmale verteilt.

Damit liegen dann Kosten und Preise für jedes Merkmal alleine stehend vor und deren Wirtschaftlichkeit kann bestimmt werden.

5 Ergebnisse

In dieser Bachelorarbeit wurde ein Ansatz zur Beurteilung der Wirtschaftlichkeit einzelner Merkmale einer Produktlinie entwickelt. Es wurden Kundenpreise mit Merkmalen assoziiert und Materialkosten auf die beteiligten Merkmale gewichtet verteilt. Mithilfe dieser Information lässt sich nun die Planung und Priorisierung von Reengineering Maßnahmen unterstützen, indem Merkmale mit der größten Profitabilität in den Focus der Umstrukturierung gerückt werden können.

Ein wichtiger Aspekt ist die Nachvollziehbarkeit von Daten bis hin zur Quelle, aus der sie erhoben wurden. Dadurch kann jederzeit festgestellt werden, aus welchen Analysen die Ergebnisse abgeleitet wurden, auf welchen Informationen diese Analysen beruhen und aus welchen Rohdaten diese Informationen gewonnen wurden.

Ein Vorteil der Verwendung eines zentralen Wissensmodells besteht in der Möglichkeit, alle einzelnen Wissensaspekte (Bestelldaten, Konfigurationsinformationen, Preislisten, etc.) miteinander zu verbinden. Auf diese Weise wird ein System geschaffen, mit dem die Wirtschaftlichkeit einzelner Merkmale untersucht werden kann.

Literatur

[1] R. Martin, *Wissensmodell zur Identifikation unwirtschaftlicher Variabilität im Produktportfolio von Industrie 4.0-Unternehmen.* Bachelor Thesis, RPTU, Kaiserslautern, Feb. 2023.

[2] S. Thiel and A. Hein, "Systematic integration of variability into product line architecture design," in *Software Product Lines* (G. J. Chastek, ed.), Lecture Notes in Computer Science, pp. 130–153, Springer, 2002.

[3] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Tech. Rep. CMU/SEI-90-TR-021, Nov 1990.

Integration eines interaktiven Whiteboards für SEE

Michel Krause (krausem@uni-bremen.de)
Universität Bremen

1 Einführung

Dieser Beitrag fasst meine Masterarbeit mit dem gleichnamigen Thema zusammen. Die Arbeit umfasst die Integration eines Drawable[1]-Systems für Software Engineering Experience (SEE). SEE ist eine kollaborative Softwarevisualisierung zum Analysieren von Software. Dabei wird die Software als dreidimensionale Code-City in einem virtuellen Raum dargestellt. Diese virtuellen Räume können von den Benutzern mit verschiedenen Endgeräten betreten werden, wobei sie darin als Avatare repräsentiert werden und miteinander verbal und mit Gesten interagieren können.

2 Motivation

Planung ist für einen Softwareentwicklungsprozess essenziell. Viele verschiedene Experten sind der Ansicht, dass es für die Planung hilfreich ist, sich Notizen zu machen. Ein bekannter Grundsatz von David Allen hierzu lautet: "Your mind is for having ideas, not holding them." Dieser Grundsatz verdeutlicht, dass der Verstand zum Generieren von Ideen da ist und nicht zum Festhalten von Details und Anforderungen. Daher sollten Ideen in Notizen übertragen werden, um den Kopf frei für neue Ideen zu bekommen [1].

In den Kognitionswissenschaften wurde gezeigt, dass Menschen aus einer Kombination von verbaler und nonverbaler Darstellungsform effektiver lernen, verstehen und erinnern können [2]. Aus diesem Grund ist es wirkungsvoller, Notizen in einer solchen Kombination anzufertigen.

3 Zielsetzung

Um die kollaborativen Meetings in SEE zu unterstützen, indem die Teilnehmer besser planen oder andere Notizen festhalten können, sollte SEE um interaktive Drawables (Whiteboards und Haftnotizen) erweitert werden. Die Drawables sollten dem Nutzer ermöglichen, frei zu zeichnen, geometrische Formen hinzuzufügen, Texte zu schreiben, Bilder einzufügen sowie Mind Maps zu erstellen. Diese hinzufügbaren Objekte werden als Drawable-Typen bezeichnet und mussten jeweils hoch konfigurierbar sein. Der Typ für die Linien stellt verschiedene Linienarten, Farbarten und Liniendicken sowie eine Loop-Funktion und die Wahl der Primär- und Sekundärfarbe bereit. Der Text-Typ bietet verschiedene Schriftgrößen sowie Textstile, eine Textfarbe und eine Außenfarbe an. Mit dem Bilder-Typ können lokale Bilder vom Endgerät sowie Webbilder bereitgestellt werden. Der Typ für die Mind Maps setzt sich aus den Linien- und Text-Typ zusammen und stellt drei Knotenarten (zentrales Thema, Unterthema, Blatt) zur Verfügung. Zudem sollte es auch möglich sein, die Drawable-Typen zu transformieren[2] und gegebenenfalls zu löschen. Beim Löschen wird zwischen dem vollständigen Löschen und dem teilweisen Löschen für Linien unterschieden. Zusätzlich sollte die Möglichkeit bestehen, weitere Drawables in Form von Haftnotizen während der Laufzeit der Visualisierung hinzuzufügen. Diese mussten des Weiteren löschbar, bearbeitbar und bewegbar sein. Diese Haftnotizen können auf jedem Objekt mit einem Collider platziert werden. Zudem musste es möglich sein, die Drawables zu speichern und zu laden. Ein Demonstrationsvideo wurde für diesen Beitrag bereitgestellt[3].

Durch diese Integration soll es den Benutzern unter anderem ermöglicht werden, neue Erweiterungen besser zu planen, indem sie kollaborativ innerhalb der Visualisierung entsprechende UML-Diagramme zeichnen können. Ein weiteres Anwendungsszenario wäre das Erstellen einer Mind Map, um alle Eigenschaften einer zu implementierenden Komponente zu erfassen.

Die folgenden Forschungsfragen wurden im Rahmen der Arbeit untersucht: 1. Welche Usability bietet die Integration insgesamt? 2. Bewerten Informatiker die Usability besser und führen sie gestellte Aufgaben schneller durch als Nicht-Informatiker? 3. Wie verändert sich die Wahrnehmung der Usability und die benötigte Zeit bei erneuter Verwendung der Integration? Die zweite Forschungsfrage dient dabei der Untersuchung, ob die Integration so intuitiv ist, dass sie selbst von Nicht-Informatikern problemlos bedient werden könnte. Diese Überprüfung erfolgte, da das Ziel war, die Integration so einfach wie möglich zu gestalten.

4 Durchführung

Um diese Forschungsfragen zu beantworten, wurde eine Studie mit vierzehn Probanden durchgeführt. Von den vierzehn Probanden stammten sechs aus dem Informatikbereich und acht nicht. Zuerst wurden einige demografische Fragen nach Alter, Geschlecht, höchster Bildungsabschluss und IT-Zugehörigkeit sowie die bisherige Erfahrung mit SEE, Bildbearbeitungsprogrammen und Videospielen gestellt. Anschließend erhielten alle Probanden dieselben sieben Aufgaben zur Bearbeitung. Die Aufgabgen wurden so konstruiert, dass alle bereitgestellten Aktionen mindestens einmal verwendet werden mussten. So mussten die Probanden zunächst ein dreistufiges Podest mit einer Linie zeichnen und dann den Bereich für

[1] Ist ein Objekt, auf dem gezeichnet, geschrieben, Bilder hinzugefügt oder Mind Maps erstellt werden können.

[2] Umfasst: Edit, Move, Rotate, Scale, Change The Sorting Layer, Cut/Copy/Paste, Move a Point und Line Split.

[3] `https://www.youtube.com/watch?v=n-p9MzYXkNU`

den ersten Platz abtrennen und modifizieren. In einer weiteren Aufgabe musste das Podest entsprechend beschriftet werden. Mittels einer anderen Aufgabe wurde eine geometrische Form erstellt, modifiziert, kopiert und anschließend Teile von ihr gelöscht. Die vierte Aufgabe umfasste das Hinzufügen und Modifizieren eines Webbildes. In der fünften Aufgabe wurde eine Haftnotiz, auf der das Bild eingefügt wurde, auf einem Tisch platziert. Die sechte Aufgabe umfasste das Erstellen einer Mind Map und die siebte Aufgabe befasste sich mit dem Speichern und Laden der Drawables. Nach jeder Aufgabe wurde ein kurzer Post-Task-Fragebogen in Form eines After Scenario Questionnaire (ASQ) ausgefüllt und am Ende der Studie ein Post-Study-Fragebogen in Form eines System Usability Scale (SUS). Mittels des SUS wurde die Usability der Drawable-Integration und durch den ASQ die wahrgenommene Komplexität (Einfachheit), der Aufwand und die Zufriedenheit mit den angezeigten Informationen wie Benachrichtigungen der ausgeführten Aktion und der visuellen Darstellung der Änderungen gemessen.

5 Ergebnisse

Die Auswertung dieser Studie ergab, dass der SUS einen Durchschnitt von $\bar{x} = 82{,}32$ und einen Median von $\tilde{x} = 85$ mit der Standardabweichung von $s = 11{,}07$ aufwies. Die Ergebnisse deuten laut Sauros Tabelle[4] auf eine exzellente Usability hin. Der ASQ ergab, dass die Komplexität und der Aufwand generell als leicht empfunden wurden und die Informationszufriedenheit als sehr gut bis gut bewertet wurde. Die durchschnittliche Bearbeitungszeit der Aufgaben betrug 39,37 min.

Informatiker bewerteten die Usability mit einem Durchschnitt von $\bar{x} = 81{,}25$ und einem Median von $\tilde{x} = 85$, wobei die Standardabweichung $s = 14{,}03$ beträgt. Die Nicht-Informatiker bewerten sie mit einem Durchschnitt von $\bar{x} = 83{,}13$ und einem Median von $\tilde{x} = 83{,}75$, wobei die Standardabweichung $s = 9{,}23$ beträgt. Der Unterschied zwischen den beiden Gruppen ist nicht signifikant und wurde mittels des Mann-Whitney-U-Testes überprüft. Im Folgenden wird für jede Unterschiedsprüfung der Mann-Whitney-U-Test mit einem Signifikanzniveau von $p = 0.05$ verwendet. Bezüglich der Geschwindigkeiten benötigten Informatiker eine durchschnittliche Bearbeitungszeit von $\bar{x} = 33{,}74$ min und Nicht-Informatiker eine von $\bar{x} = 43{,}6$ min. Damit waren die Informatiker beim Bearbeiten der Aufgaben um 22,61% schneller als die Nicht-Informatiker. Dieser Unterschied war mit einem p-Wert von $p = 0.05395$ knapp nicht signifikant. Zudem bewerten Informatiker die Komplexität geringer, den Aufwand als einfacher und die Informationszufriedenheit als besser als die Nicht-Informatiker. Die Unterschiede für die Komplexität und den Aufwand waren dabei signifikant.

[4] `https://measuringu.com/interpret-sus-score/`

Sieben Probanden konnten für einen erneuten Durchlauf gewonnen werden, um die dritte Forschungsfrage zu klären. Dabei führten die Probanden exakt dieselbe Studie erneut durch, wobei der Zeitpunkt für die erneute Durchführung maximal eine Woche später lag. Die gestellten Aufgaben und das verwendete Tool blieben dabei gleich. Dadurch können Rückschlüsse auf das erinnerte Verständnis sowie die Wiedererkennung der Integration gezogen werden. Dabei wurde eine durchschnittliche Zeitersparnis von 12,52 min (27,68%) erzielt. Die Usability wurde um 6,07 SUS-Punkte, die Komplexität um 4,86 der Aufwand um 5,85 und die Informationszufriedenheit um 2,14 durchschnittlich besser bewertet. Dabei wurde ein signifikanter Unterschied für die erhobenen abhängigen Variablen mittels des Wilkoxons Vorzeichentests festgestellt. Diese Erkenntnis deutet darauf hin, dass es sich generell für Usability-Studien lohnen könnte, eine Messung zu wiederholen. Eine zweite Messung könnte ebenfalls repräsentativer sein, da sich Benutzer zunächst in neue Anwendungen einarbeiten müssen, bevor sie problemlos genutzt werden können.

Es wurden folgende signifikante Korrelationen zwischen den abhängigen und unabhängigen Variablen festgestellt: 1. Eine negative Korrelation zwischen der Zeitersparnis und dem Alter, was bedeutet, dass jüngere Probanden tendenziell mehr Zeit durch einen weiteren Durchlauf einsparen konnten. 2. Eine positive Korrelation zwischen der IT-Zugehörigkeit und dem Aufwand sowie zwischen der IT-Zugehörigkeit und der Komplexität. Daraus resultiert, dass Informatikern die Aufgaben leichter gefallen sind und sie weniger Aufwand investieren mussten um diese abzuschließen. 3. Eine negative Korrelation zwischen dem Aufwand und der Erfahrung mit Videospielen sowie zwischen der Komplexität und der Erfahrung festgestellt. Das bedeutet, dass Personen mit mehr Erfahrung die Aufgaben tendenziell leichter gefallen sind und sie weniger Aufwand dafür aufbringen mussten.

6 Fazit

Das Ergebnis der Studie zeigt, dass sowohl Informatiker als auch Nicht-Informatiker die Integration als intuitiv betrachten, sie zufriedenstellend bedienen und problemlos nutzen können. Zudem wurde festgestellt, dass der SUS-Score auf eine exzellente Usability hinweist und Informatiker die gestellten Aufgaben schneller und leichter ausführen können als Nicht-Informatiker. Der zweite Studiendurchlauf zeigte eine signifikante Verbesserung der erhobenen abhängigen Variablen der Studie.

Literaturverzeichnis

[1] David Allen. Getting Things Done: The Art of Stress-Free Producitivity. pages 15–17. Penguin Books, 2003.

[2] Richard E. Mayer. Multimedia Learning: Second Edition. pages 1–3, 19. Cambridge University Press, 2009.

Wenn das Englischwörterbuch nicht ausreicht
Mit AMELIO Logic Discovery COBOL-Anwendungen verstehen

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

Abstract

Um eine große, komplexe und über Jahrzehnte gewachsene COBOL-Anwendung (wieder-)zuverstehen, reicht es nicht aus, den Code lesen zu können. Stattdessen ist es notwendig, von technischen Details zu abstrahieren und Wissen auf einen Blick zu erhalten. Für verschiedene Aufgaben werden zudem passende Sichten auf die Anwendung benötigt.

Wir stellen mit AMELIO Logic Discovery ein Tool zur automatisierten Wissens(wieder)gewinnung vor.

1 Versteht das jemand?

Ein alter Witz unter Entwicklern lautet "Um ein COBOL-Programm lesen zu können, benötigt man lediglich ein Englischwörterbuch". Das mag stimmen, allerdings reicht lesen können alleine nicht aus, um das Programm auch tatsächlich zu verstehen.

Die meisten Legacy-Programme sind groß, komplex und über Jahrzehnte gewachsen. Die Architektur- und Sprachparadigmen, nach denen sie entwickelt wurden, weichen stark von heutigen Paradigmen ab. Es wurden Technolgien verwendet, die heute nicht mehr gelehrt werden, die aber ebenfalls Einfluss auf die Architektur der Anwendung haben. Dokumentation, falls überhaupt vollständig vorhanden, reicht nicht aus, um die Anwendungen zu verstehen.

AMELIO Logic Discovery

- analysiert auch große und komplexe Legacy-Anwendungen vollständig automatisiert,
- bietet unterschiedliche, thematisch passende Sichten auf die Anwendung,
- fasst Informationen zusammen, auch wenn sie über den Code verteilt sind,
- abstrahiert von technischen Details und
- leitet aus Informationen Erkentnisse ab und stellt diese sprachneutral dar

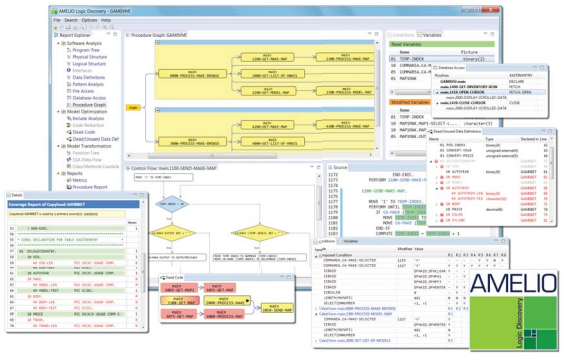


Figure 1: AMELIO Logic Discovery - Der Logic Browser

2 COBOL nicht nur für Babyboomer

Es gibt viele Gründe, eine COBOL-Anwendung (wieder-)verstehen zu wollen oder zu müssen, die beiden häufigsten sind

- die Babyboomer sind oder gehen in Rente. Weiterentwicklung und Wartung müssen durch ihre "Erben" übernommen werden.
- die Anwendung oder ein Teil daraus soll in einer anderen Sprache neuentwickelt werden.

In beiden Fällen ist es wichtig, zunächst einmal in Erfahrung zu bringen, was die Anwendung tut. Wie sie das tut ist dagegen nicht von Interesse. Deshalb führt AMELIO Logic Discovery sowohl eine formale als auch eine logische Abstraktion durch. Auf diese Weise können sprachspezifische Details ausgeblendet werden. Stattdessen werden abstrakte/logische Konstrukte eingeführt, die das Verständnis der Anwendung vereinfachen.

Zur Darstellung von Analyseergebnissen wurde eine möglichst sprachneutrale Darstellung gewählt. So werden Kontrollflüsse zum Beispiel in Anlehnung an Acitivity Diagrams angezeigt oder komplexe Bedinungen als Bedingungstabellen. Ziel ist eine

Darstellung, die es auch nicht COBOL-Experten erlaubt, die Anwendung zu verstehen.

3 Eine Frage der Perspektive

Je nach Aufgabenstellung benötigt man einen anderen Blickwinkel auf die Anwendung mit unterschiedlichen Abstraktionsgraden. Aus diesem Grund bietet AMELIO Logic Discovery verschiedene Perspektiven, die es ermöglichen, aus der Vogelperspektive auf die Anwendung zu schauen, aber auch in ihre Details einzutauchen. Jede Perspektive beleuchtet die Anwendung aus einem anderen Blickwinkel und liefert aufgabenspezifische Antworten.

4 Prozeduren in COBOL?!

Ein besonderes Highlight von AMELIO Logic Discovery ist die Prozeduranalyse für COBOL.

Prozeduren gibt es in COBOL nicht, jedenfalls nicht explizit (in älteren COBOL-Versionen). Stattdessen gibt es in der Procedure Division eine Sammlung von Sections und darin enthaltene Paragraphen, die mal einzeln und mal gemeinsam ausgeführt werden und sich gegenseitig aufrufen.

AMELIO Logic Discovery bestimmt Prozeduren und logische Gruppierungen, sowie deren Schnittstellen anhand von Aufrufbeziehungen und macht sie sichtbar.

Zusätzlich werden sog. Composites ermittelt. Dabei handelt es sich um Prozeduren, die logisch gruppiert werden können.

Prozeduren und Composites werden in ihrem jeweiligen Kontext dargestellt.

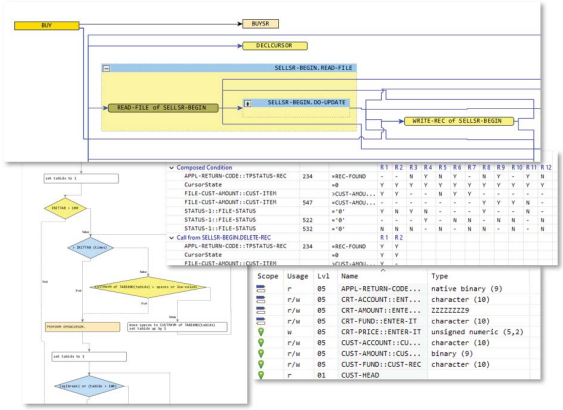


Figure 2: Prozeduren und Composites mit Kontext

5 Aus global wird lokal

Variablen sind in COBOL grundsätzlich global definiert, auch dann, wenn sie nur lokal verwendet werden. AMELIO Logic Discovery kann den Scope einer Variablen jedoch anhand ihrer Verwendung festlegen und somit bestimmen, ob eine Variable tatsächlich global ist oder eigentlich lokal.

Mit dem Ergebnis der Scope-Analyse stellt AMELIO Logic Discovery für jede Prozedur und jedes Composite fest, welche Parameter als Input erwartet und welche als Output zurück geliefert werden.

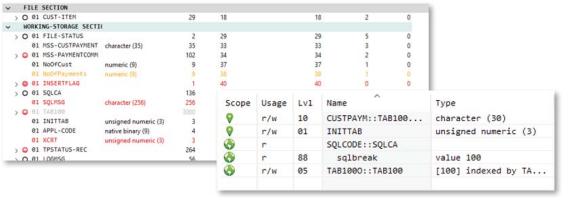


Figure 3: Datendefinitionen

6 COBOL-Anwendungen (wieder-) verstehen

AMELIO Logic Discovery analysiert COBOL-Anwendungen vollständig automatisiert. Verschiedene Perspektiven helfen, einen Überblick über die Anwendung zu gewinnen. Mittels Abstraktion wird ermittelt, was die Anwendung tut, wie sie es tut, wird ausgeblendet. Und sprachneutrale Darstellungen erleichtern die Lesbarkeit der Ergebnisse. So kann man auch ohne COBOL-Knowhow große und komplexe COBOL-Anwendungen (wieder-)verstehen.

Kollaborative Software-Visualisierung für verteilte Entwicklungs-Teams mit SEE (Demo)

Rainer Koschke, *AG Softwaretechnik*, *Universität Bremen*,
Dennis Küster, *Cognitive Systems Lab*, *Universität Bremen*

I. Einführung

In diesem Beitrag einer Tool-Demo zeigen wir den aktuellen Stand unserer Visualisierungsplattform SEE (für *Software Engineering Experience*), an der wir seit vielen Jahren entwickeln. SEE visualisiert Software als Code-City in 3D. Der Fokus hierbei liegt auf der Unterstützung verteilt arbeitender Teams. SEE schafft virtuelle Räume, in denen sich Entwicklerinnen und Entwickler treffen können, um sich über ihre Software auszutauschen. Die virtuellen Räume können mit normalen Desktop-Computern mit Monitor, Tastatur und Maus, aber auch mit moderner VR-Hardware betreten werden.

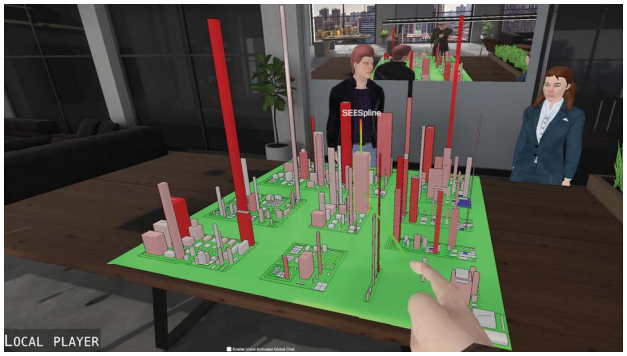


Abb. 1. Virtueller Raum in SEE. Gezeigt wird eine Code-City für SEE selbst.

II. Der Kontext

Der Trend zur verteilten Entwicklung, der längst schon vor der Pandemie eingesetzt hatte, wird weiter anhalten. Große Organisationen sind meist über verschiedene Standorte verteilt. Zudem arbeiten viele Entwicklerinnen und Entwickler gerne im Home-Office, nicht nur wegen pandemiebedingter Einschränkungen, sondern auch weil damit lange tägliche Anfahrtswege wegfallen und sich die Arbeit flexibler mit familiären Aufgaben verbinden lässt.

III. Das Problem

Verteilte Arbeit hat aber den großen Nachteil, dass man anderen während der Entwicklung nicht eben einmal helfend über die Schulter schauen kann, wenn weitere Informationen oder Feedback benötigt werden. Video-Konferenzsysteme und Chats, die eingesetzt werden, um räumliche Distanzen zu überbrücken, helfen hier nur bedingt weiter. Das wissen alle, die es selbst einmal erlebt haben. Zwar kann man den Bildschirm teilen, aber immer nur eine Person hat die volle Kontrolle über die Anwendung und alle anderen müssen diese quasi dirigieren. Zwar gibt es seit einiger Zeit auch IDEs, wie die IntelliJ-basierten IDEs der Firma *JetBrains* mit der Erweiterung *Code With Me*, die kollaboratives Editieren ermöglichen, aber das unterstützt die direkte Zusammenarbeit nur auf der Ebene des oft zu detaillierten Quelltexts und zeigt auch nur, was gerade ist, aber nicht, was früher einmal war. Der Blick für das große Ganze geht hier schnell verloren. Was spielt denn dieser Quellcode für eine Rolle in der Software-Architektur? Welche Auswirkungen auf andere Teile des Systems hat es, wenn wir ihn ändern? Wann und warum wurde er zuletzt verändert und wie hat er sich über die Zeit entwickelt? All das sagt einem der Quelltext alleine nicht.

IV. Die Ziele

Wir wollen Abhilfe bei bekannten Problemen in der verteilten Software-Entwicklung schaffen. In unserem Projekt SEE untersuchen wir die Frage, wie kooperatives Programmverstehen in verteilten Teams in der Software-Entwicklung mit Hilfe von modernen konvergenten Visualisierungstechnologien besser unterstützt werden kann. Unter technologischer Konvergenz verstehen wir an dieser Stelle die enge Integration zuvor unzusammenhängender Technologien. Konkret integrieren wir sowohl herkömmliche Desktop-Hardware (Computer, Tastatur und Maus) und Tablet-Geräte als auch fortschrittlichere Hardware für virtuelle (VR) und erweiterte (AR, engl. *augmented*) Realität so, dass Software-Entwicklerinnen und -Entwickler ein zweckdienliches, einheitliches und zutreffendes Bild ihrer Software über räumliche Distanzen hinweg bekommen. Es wird ein gemeinsamer virtueller Raum erschaffen, in dem sich die Beteiligten in exemplarischen Anwendungsszenarien in der Weiterentwicklung von Software verständigen können. Diesen virtuellen Raum können die Beteiligten mit Geräten ihrer Wahl (Desktop, Tablet, VR oder AR) „betreten“. Darin können Entwicklerinnen und Entwickler nicht nur einen gemeinsamen Blick auf ihre Software, die als Code-City dargestellt ist, und auch die Quelltexte werfen, sondern es werden auch Visualisierungen des Aufbaus und der Abhängigkeiten eines Programms, eingesammelter Laufzeitdaten sowie historischer Änderungen an der Software angeboten. Die Visualisierung bietet eine abstraktere Grundlage, wenn das große Ganze für das Verständnis notwendig ist. Bei den exemplarischen

Anwendungsszenarien streben wir danach, ein großes Spektrum realistischer Anwendungsfälle in der Weiterentwicklung von Software abzudecken. Hierzu gehören das Debugging, die Performance-Analyse, die Analyse der inneren Qualität, das Nachvollziehen früherer Entwicklungen und die (semi)automatische Prüfung der Software-Architektur. Im Projekt arbeiten wir auch eng mit software-entwickelnden Organisationen zusammen, mit Hilfe derer wir partizipativ und iterativ konkrete Anforderungen erheben und deren Umsetzung kontinuierlich entlang der Ideen der Aktionsforschung evaluieren.

V. Die Umsetzung

SEE basiert auf der Spiele-Engine Unity 3D und ist in C# implementiert. Unsere Implementierung ist unter der MIT-Lizenz öffentlich verfügbar[1]. Für Unity haben wir uns entschieden, weil es viele Funktionalitäten mitbringt, die wir benötigen. Wir hatten anfangs die Unreal Engine verwendet, die es jedoch erforderlich macht, Code in C++ zu entwickeln oder aber so genannte Blueprints zu verwenden. Letztere sind eine graphische Notation, die jedoch schnell zu Merge-Konflikten bei textbasierten Versionskontrollsystemen wie Git führt. Neben der leichteren Zugänglichkeit von C# für unsere Studierenden ist auch die höhere Verfügbarkeit von Dokumentation und anderen Formen von Hilfestellungen (wie YouTube-Videos und Entwicklerforen) ein Vorteil von Unity.

Als Visualisierungsform haben wir die Metapher der Code-Cities ausgewählt, weil sie sich für verschiedene Zwecke gut eignet und eine intuitive höhere Abstraktion bietet. Auch viele andere Forschungsgruppen im deutschsprachigen Raum verwenden Code-Cities.

Die Teilnehmenden im virtuellen Raum werden mit menschlichen Avataren dargestellt und sind somit Teil der Szene. Dadurch können wir neben unserem Voice-Chat auch non-verbale Kommunikation in Form von Gesten übermitteln. Zum Beispiel ist es möglich, mit Hilfe eines Lichtstrahls auf Dinge zu zeigen. Diese Geste wird auf alle verbundene Instanzen von SEE übertragen und ist somit von allen von Ferne zugeschalteten Teilnehmenden nachvollziehbar. Wird VR-Hardware benutzt, können wir die Position der VR-Controller, die von den Teilnehmenden in den Händen gehalten werden, nachverfolgen und auf die Handpositionen der Avatare übertragen. Durch inverse Kinematik werden die Körperhaltungen der Avatare so angepasst, dass daraus eine stimmige Bewegung wird.

Gegenwärtig arbeiten wir daran, auch die Mimik zu übertragen. Im Falle von Desktop-Computern mit einer Web-Cam erkennen wir das Gesicht der Person und übertragen dies vor dem Kopf des Avatars, der diese Person in der Szene repräsentiert. Im Falle von VR-Hardware, bei der das Gesicht der Person durch die VR-Brille bedeckt ist, verwenden wir den Facial Tracker von Vive, der die untere Partie des Gesichts sehr zuverlässig erkennt. Damit ist es möglich, Lachen, Sprechbewegungen und auch das Herausstrecken der Zunge zu übertragen. Letzteres ist in formalen Meetings eher selten anzutreffen, wird aber gerne ausprobiert, wenn unsere Probanden das Gerät zum ersten Mal nutzen. In einem Forschungsprojekt mit dem *Cognitive Systems Lab* der Universität Bremen, das sich auf Biosignale sowie die Erkennung und Interpretation von Sprache, Muskel- und Hirnaktivitäten spezialisiert hat, verwenden wir EMG-Sensoren, die auf dem Gesicht unter der Brille angebracht sind, um die Muskelaktivitäten zu identifizieren. Aus diesen Signalen schließen wir auf die Mimik, um sie zutreffend im Gesicht der Avatare abzubilden.

Wir haben verschiedene Methoden implementiert, um Daten über die Software, die es zu visualisieren gilt, zu generieren. Zum einen haben wir eine Anbindung an die Analyseergebnisse geschaffen, die mittels statischer Code-Analyse von der Axivion Suite[2] gewonnen werden. Das hierbei ausgewählte Austauschformat ist GXL. Wir sind auch in der Lage, beliebige Graphen in GXL zu importieren, die von anderen GXL-fähigen Tools generiert werden. Zudem arbeiten wir an Anbindungen an IDEs mit Hilfe des *Language Server Protocol*[3] und an Debugger mittels des *Debug Adapter Protocol*[4] sowie an das Versionskontrollsystem Git. Auf diese Weise werden wir in der Lage sein, sowohl statische als auch dynamische Daten für eine Vielzahl von Programmiersprachen zu gewinnen und diese mit historischen Daten aus einem Versionskontrollsystem anzureichern.

VI. Zusammenfassung

SEE visualisiert Software auf Basis der Code-City-Metapher in virtuellen Räumen, die von Teilnehmenden von verschiedenen Arten von Endgeräten von jedem beliebigen Punkt des Internets betreten werden können. Die Teilnehmenden können unabhängig voneinander mit der dargestellten Software interagieren und jeweils ihre eigene Perspektive einnehmen. Sie selbst werden als Avatare repräsentiert und können sowohl verbal als auch zu einem großen Grad non-verbal miteinander kommunizieren. SEE soll auf diese Weise räumliche Distanzen in verteilt arbeitenden Teams überwinden, die bis dato von existierenden Videokonferenzsystemen sehr eingeschränkt werden.

Wir entwickeln und evaluieren SEE im Kontext eines von der DFG geförderten Forschungsprojekts mit Anwendungspartnern aus der Industrie weiter.

[1] https://github.com/uni-bremen-agst/SEE

[2] https://www.axivion.com

[3] https://microsoft.github.io/language-server-protocol/

[4] https://microsoft.github.io/debug-adapter-protocol/

Aktionsforschung in der Software-Visualisierung

Rainer Koschke, *AG Softwaretechnik*, *Universität Bremen*

Dieser Beitrag widmet sich der empirischen Forschung in der Software-Visualisierung (SV). Wir setzen uns kritisch mit den bis dato eingesetzten empirischen Methoden auseinander und argumentieren, dass die Aktionsforschung ein komplementärer Ansatz ist, der gerade für das Design und die Evaluation von Software-Visualisierungen vielversprechend ist.

I. Empirische Forschung in der SV

Merino et al. [1] haben 209 Artikel, die bei der *IEEE Working Conference on Software Visualization (VISSOFT)* oder dem *ACM SOFTVIS Symposium on Software Visualization (SOFTVIS)* als *Full Paper* veröffentlicht wurden, ausgewertet. Von den 181 Artikel, die sich einer Visualisierung im engeren Sinne widmen, haben die Autoren diese hinsichtlich der in den Artikeln beschriebenen Form der Evaluation weiter kategorisiert in empirisch (113), rein theoriegestützt (2) oder gar nicht evaluiert (24). Bei den 113 als empirische Evaluation eingestuften Arbeiten traten die folgenden Formen einer Evaluation auf: anekdotische Evidenz (6), Nutzungsszenarien, in denen eine Visualisierung anhand eines Beispiels demonstriert wird (38), Literaturübersichten (4), Fallstudien (12) und Experimente (53). In allen Fallstudien waren professionelle Entwicklerinnen und Entwickler beteiligt. Bei den Experimenten waren die Teilnehmer hingegen meist Studierende. Der Median der Teilnehmerzahlen dieser Experimente lag bei 13. Die meisten Experimente hatten zwischen eins und fünf Teilnehmende.

Literaturübersichten sind keine direkten Evaluationen von Visualisierungen. Anekdotische Evidenz stellt keine wirklich empirisch fundierte Evaluation dar. Die Demonstration einer Visualisierung in Form eines Nutzungsszenarios mag eine Visualisierung an einem konkreten Beispiel gut illustrieren, jedoch sind es in der Regel die Autoren, die ihre eigene Visualisierung vorführen, und nicht davon unabhängige Entwickler und Entwicklerinnen, die die Visualisierung an ihrem eigenen Code erproben. Dieser Form der Evaluation – wenn man sie überhaupt als solche bezeichnen mag – fehlt es in der Regel auch an Systematik und objektiver Beobachtung.

Kontrollierte Experimente sind im Gegensatz zu allen anderen empirischen Methoden in der Lage, Wirkungszusammenhängen nachzugehen, indem sie bestimme Einflussvariablen fixieren und andere systematisch variieren, um deren Auswirkung zu untersuchen. Wie Merino et al. [1] jedoch festgestellt haben, sind Experimente in der Software-Visualisierung in Bezug auf ihre Teilnehmerzahl eher klein. Meist wird hier bei der Teilnehmerauswahl auch nur Convenience-Sampling angewandt und bei Forschenden an Universitäten führt dies überwiegend dazu, dass auf Studierende zurückgegriffen wird. Der Anspruch an die Standardisierung (Kontrolle von Variablen) führt fast notwendigerweise dazu, dass die Teilnehmenden mit der Software, die visualisiert wird, gar nicht vertraut sind. Die Auseinandersetzung der Teilnehmenden mit der Software und ihrer Visualisierung ist meist einmalig und für sie folgenlos. In der realen Welt haben solche Dinge jedoch potentiell einen Einfluss auf die weitere Entwicklung der Software. Die Teilnehmenden verwenden üblicherweise auch die Visualisierungform zum ersten Mal. Deshalb werden sie im Experiment erst einmal einem Training in der Visualisierung und der damit verbundenen Interaktionen unterzogen. Der Ökonomie geschuldet, ist die Zeit hierfür jedoch meist sehr begrenzt. Die Teilnehmenden sind schon gar nicht beim Design involviert gewesen, so dass ihre individuellen Bedürfnisse in der Visualisierung allenfalls zufällig berücksichtigt sind. Eine weitere Schwäche von Experimenten ist es, dass sie recht punktuelle Untersuchungsformen sind. Nach einem Experiment vergeht eine ganze Zeit bis zum Nachfolgeexperiment, bei dem dann aber die Teilnehmenden in der Regel wieder andere sind. Dieser Untersuchungsform fehlt dann Kontinuität und Langfristigkeit.

Bei Fallstudien beobachten Forschende passiv, wie eine Software-Visualisierung von Entwicklern im eigenen Arbeitskontext verwendet wird. Der Begriff *Fallstudie* drückt es aus: man studiert hier einen Fall in der realen Welt, ohne Kontrolle auszuüben. Insofern haben Fallstudien einen höheren Grad an Realismus als kontrollierte Experimente, die in einer Laborumgebung stattfinden. Während dieses Vorgehen in Kontexten adäquat sein kann, in denen es vornehmlich darum geht, ein Phänomen zu verstehen, geht es jedoch in der Software-Visualisierung in den meisten Fällen gerade darum, eine Visualisierung so zu gestalten, dass sie möglichst nutzbringend ist. Wir wollen als Forschende hier tatsächlich in die reale Welt eingreifen.

II. Aktionsforschung

Im Software-Engineering steckt der Begriff *Engineering*, also das Ziel, ein reales Problem zu lösen, in unserem Falle eines im Zusammenhang der Software-Entwicklung. Würden wir als Forschende im Software-Engineering so vorgehen, wie wir es auch in der Lehre unseren angehenden Software-Engineers vermitteln, dann würden wir zunächst einmal eine genaue Bedarfsanalyse durchführen, um zu ermitteln, wer welche Information zu welchem Zweck benötigt. Im Sinne einer partizipativen Entwicklung würden wir die späteren Nutzer möglichst früh einbinden und unsere Konzepte mittels Prototypen

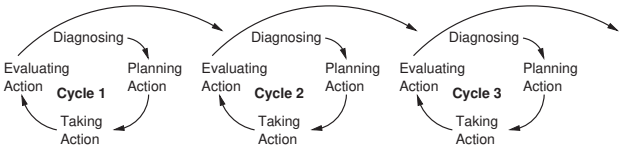


Fig. 1. Vorgehen in der Aktionsforschung

erproben. Als evolutionäre Prototypen würden wir diese iterativ und inkrementell weiterentwickeln, bis sie die Anforderungen erfüllen. Auch danach im Betrieb würden wir weitere Erfahrungen sammeln, die uns helfen, die Software-Visualisierung kontinuierlich weiterzuentwickeln.

Partizipative, agile, iterative und inkrementelle Software-Entwicklung hat viele Parallelen mit der Aktionsforschung. Die Aktionsforschung (engl. *Action Research*) stammt ursprünglich aus den Sozialwissenschaften. Sie wurde für Kontexte eingeführt, in denen die Forschenden nicht nur ein soziales Phänomen beobachten, sondern auch positiv auf es einwirken wollen. Die Aktionsforschung ist mittlerweile aber auch eine anerkannte Methodik in der empirischen Softwaretechnik [2], wenngleich sie in der Darstellung „kanonischer“ Methoden in der empirischen Softwaretechnik gewöhnlich nicht auftaucht.

Die Aktionsforschung übertragen auf die Softwaretechnik verbindet die guten Praktiken des Software-Engineerings mit dem empirischen Erkenntnisgewinn. Im Gegensatz zu Experimenten ist dies eine Untersuchung über einen längeren Zeitraum mit hohem Realismus, bei dem die Teilnehmenden von Anfang an eingebunden sind und die Visualisierung im eigenen Arbeitskontext am eigenen Code evaluieren. Im Gegensatz zu reinen Fallstudien greifen Forschende hier ein, indem sie die Visualisierung kontinuierlich und bedarfsgerecht anpassen. Die Aktionsforschung ist aber keineswegs ein Ersatz für andere Untersuchungsformen, vielmehr lassen sich die klassischen Methoden wie Befragung, Beobachtungsstudien, Fallstudien und kontrollierte Experimente darin gut integrieren.

Das prinzipielle Vorgehen bei der Aktionsforschung ist in Abbildung 1 verbildlicht. Im Schritt *Diagnosing* gehen die Forschenden von den real existierenden Fragestellungen der Entwickler und Entwicklerinnen (im Folgenden *Partner* genannt) in deren eigenem Umfeld aus, die mit Hilfe einer Visualisierung potentiell beantwortet werden können und die auch für die Forschenden von wissenschaftlicher Relevanz sind. Hierzu erheben die Forschenden mit den Partnern – etwa durch Befragung oder Beobachtungen – die Anforderungen in Bezug auf die notwendige Art von Information für die Partner, damit diese ihre realen Aufgaben erledigen können.

Im Schritt *Planning Action* erarbeiten die Forschenden mit ihren Partnern genauer, wie man die im Schritt *Diagnosing* als notwendig identifizierte Information erheben und insbesondere visualisieren kann. Die im gemeinsamen Design-Prozess entwickelten initialen Ideen werden noch in diesem Schritt mit Hilfe von evolutionären Prototypen umgesetzt und erprobt.

In der Phase *Taking Action* setzen die Partner die Visualisierung im Rahmen ihrer normalen Entwicklungstätigkeit an ihrer eigenen Software und im eigenen Arbeitsumfeld ein. In dieser Phase werden Daten gesammelt, die geeignet sind, die im Zyklus zu beantwortende Fragestellung zu adressieren. Die genaue Erhebung der Daten hängt von der Fragestellung des Zyklus ab. Sie kann von automatisiert erhobenen Gebrauchsdaten oder Beobachtungen, über Fragebögen bis hin zu kontrollierten Experimenten reichen, abhängig von der Art der Frage und des aktuellen Kenntnisstands.

Im letzten Schritt eines Zyklus werden die gesammelten Daten von den Forschenden ausgewertet und mit den Partnern diskutiert. Hierzu kann auf eine reichhaltige und detaillierte Menge sowohl quantitativer als auch qualitativer Daten aus dem vorherigen Schritt zurückgegriffen werden. Da es die Aktionsforschung im Unterschied zu klassischen kontrollierten Experimenten zur Evaluation von Software-Visualisierung nicht bei punktuellen Erhebungen unter doch eher künstlichen Laborbedingungen belässt, sondern vielmehr sowohl quantitative als auch qualitative Daten über einen langen Zeitraum im realen Umfeld und Arbeitskontext erhebt, kann man sich einen höheren Erkenntnisgewinn erhoffen. Insbesondere ist es auf diese Weise auch möglich, Nachhaltigkeit und Veränderungen über die Zeit zu betrachten, weil die Partner über einen langen Zeitraum begleitet werden. Graduell werden die gewonnenen Erkenntnisse zur Theoriebildung beitragen und damit die Fragestellungen nachfolgender Zyklen mitbestimmen. Teil des letzten Schritts eines Zyklus ist auch eine Reflexion über die gesammelten Erfahrungen in Bezug auf den Forschungsprozess selbst, wie etwa die eingesetzten Instrumente zur Datenerhebung, die Güte und Aussagekraft der gesammelten Daten oder die Zusammenarbeit mit den Partnern, um auf diese Weise zukünftig auch den Forschungsprozess weiter zu verbessern.

III. Fazit

Dieser Beitrag möchte die Aktionsforschung weiter bekanntmachen und zur Diskussion anregen, inwieweit sie sich als empirische Methode in der Forschung zur Software-Visualisierung eignet. Eine Herausforderungen ist es, wie auch bei Fallstudien und Experimenten, die Repräsentativität und Verallgemeinerbarkeit sicher zu stellen. In späteren Phasen müssen deshalb auch Untersuchungen mit anderen Organisationen durchgeführt werden, die nicht primär an der Entwicklung beteiligt waren. Die Chance ist aber höher, dass solche anderen Organisationen zur Teilnahme bewegt werden können, wenn die Visualisierung bereits eine hohe Reife hat.

Referenzen

[1] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, “A systematic literature review of software visualization evaluation,” *Journal of Systems and Software*, vol. 144, 06 2018.

[2] M. Staron, *Action Research in Software Engineering: Theory and Applications*. Springer International Publishing, 2020.

Gamification für Software Reengineering

Dr. Baris Güldali
S&N CQM
Zukunftsmeile 2, 33102 Paderborn

Dr.-Ing. Dehla Sokenou
WPS – Workplace Solutions
Hans-Henny-Jahnn-Weg 29, 22085 Hamburg

Zusammenfassung

Vor einer Software-Migration oder Reengineering steht oft die Frage, ob die Software 1-zu-1 migriert werden soll oder ob die Migration auch für eine Modernisierung genutzt werden kann. Dabei fehlt Softwareentwicklungsteams oft die Zeit und manchmal auch die notwendige Expertise, das System zu refraktorieren und technische Schulden zu beseitigen. Wir stellen spielerische Methoden für die genannten Probleme vor, die auf unseren Erfahrungen mit anderen Gamification-Ansätzen basieren, die wir hier auf den Bereich der Software-Reengineering angewendet haben.

Einleitung

In unserem früheren Artikel „Architektur nicht versenken“ [1] haben wir Szenarien definiert, wie Software-Entwickler-Teams Gamification nutzen können, um Probleme mit der Software-Architektur in den Griff zu bekommen und ihre Architektur kontinuierlich zu verbessern. Als wichtige Richtlinie gilt: *Erkenne zuerst das Problem, dann plane die Lösung, dann setze sie um*.

Probleme im Software Reengineering und in der Software-Migration sind ähnlicher Natur und haben ihre Ursachen oft in der Software-Architektur. Oft ist die Architektur mit der Zeit gewachsen und wird – wenn überhaupt – von einigen wenigen Team-Mitgliedern durchschaut, die am längsten dabei sind.

Ein Team, das ihre Software modernisieren oder in eine neue Technologie oder in eine neue Plattform migrieren möchte, muss den Ist-Stand der Architektur verstehen, den Einfluss der Reengineering-Maßnahmen bewerten und Entscheidungen für zukünftige Architektur-Eigenschaften treffen. Wir sind der Meinung, dass diese Entscheidungen im Team gemeinschaftlich getroffen werden sollten, um eine bessere Identifikation im Team mit den anstehenden Aktivitäten sicherzustellen. Gamification kann dabei die Kommunikation und den systematischen Austausch fördern.

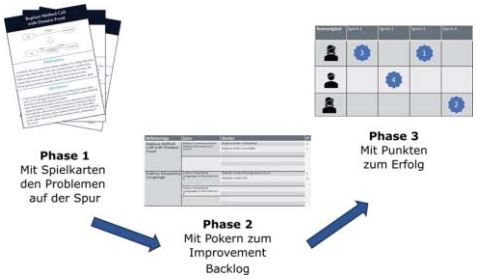


Abb. 1: Drei Phasen des Spiels

Der Gamification-Ansatz

Der Begriff *Gamification* meint den Einsatz von Elementen aus Spielen in einem nicht spielerischen Kontext [2]. Innerhalb des Arbeitsalltags sollen spielerische Elemente helfen, Prozesse zu optimieren, Alltagstätigkeiten zu erleichtern und Produkte zu verbessern. Das Ziel ist also, einen direkten, positiven Einfluss auf das Arbeitsergebnis zu haben. Unser Ansatz von Gamification für Software Reengineering besteht aus drei Phasen:

1) Mit Spielkarten den Problemen auf der Spur
2) Mit Pokern zum Improvement Backlog
3) Mit Punkten zum Erfolg

In der Phase 1 setzen wir einen Spielkartenset, das ähnlich wie die Karten beim Risk Storming [3] das Team bei der Problemanalyse über die Architektur und der Lösungsfindung leitet. Die Spielkarten beschreiben die Domain-Driven Refactorings, die von Henning Schwentner [4] zusammengetragen wurden. Sie umfassen jeweils eine Motivation – also das zu lösende Problem – und schlagen eine Lösung vor, die auf den Best-Practices aus dem Domain-Driven Design basiert (siehe Abb. 2).

Einige für Software Reengineering relevante Refactorings finden sich in den folgenden Spielkarten wieder:

- Extract Bounded Context: Das Team erkennt, dass sich die Fachlichkeit nicht adäquat im System abbildet. Mehrere unterschiedliche Karten adressieren das Problem. Als Lösung schlägt dieses Refactoring das Herausschälen eines Bounded Contexts aus dem Monolithen

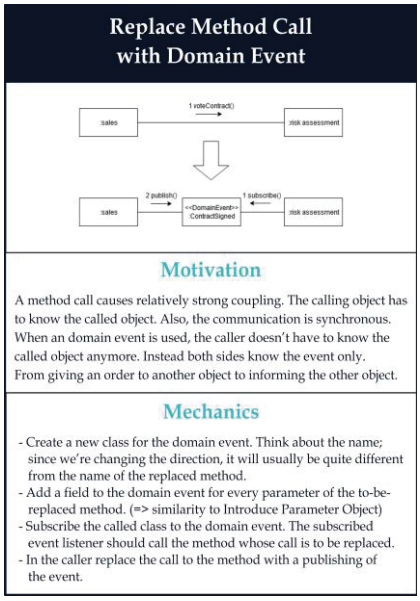


Abb. 2: Spielkarte aus den Domain-Driven Refactorings

vor. Das Team entscheidet sich nach einiger Diskussion für diese Variante.

●Enforce Ubiquitous Language: Diese Karte wird von einem neuen Mitglied im Team ausgespielt. Die alten Hasen sind so an die technischen Begriffe im Code gewöhnt, dass sie die Diskrepanz zur Fachlichkeit nicht mehr erkennen, das neue Teammitglied ist aber immer unsicher, an welche Stellen im Code auf die angeforderten Änderungen von der Fachseite durchzuführen sind. Als Lösung wird das Umbenennen im Code, aber auch in der Datenbank anhand der fachlichen Begriffe gefordert.

●Move Logic from Service to Entity: Die Logik wird so weit wie möglich in den Entities gekapselt.

●Replace Method Call with Domain Event: Zudem stellt das Team anhand dieser Karte fest, dass die Kopplung zwischen den Microservices viel zu eng ist, da diese sich gegenseitig direkt aufrufen. Das Team beschließt deshalb, stattdessen Domain-Events zu schicken, um die Microservices unabhängiger voneinander weiterentwickeln zu können.

Die Team-Mitglieder ziehen aus dem Spielkartenset Karten heraus und diskutieren über die beschriebenen Refactorings und ihre Priorisierung. Wo steht die eigene Architektur im Vergleich zu dem Refactorings? Welche Reengineering- oder Modernisierungs-Ziele können mit welchen Refactorings erreicht werden? Was ist der Einfluss eines ausgewählten Refactorings auf die anstehenden Entwicklungsarbeiten? Aus diesen Diskussionen entsteht in der zweiten Phase ein Improvement-Backlog.

Nach unserer Beobachtung findet Software Reengineering oft parallel zum laufenden Betrieb und Weiterentwicklung statt. Daher ist es

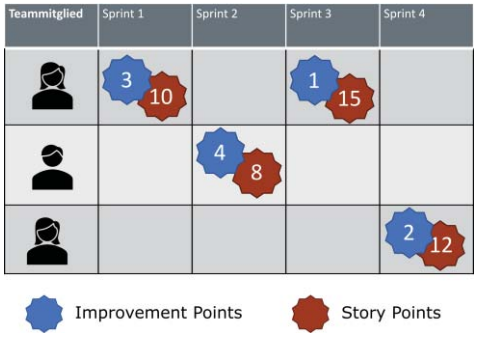


Abb. 3: Auswertung von Improvement- und Story-Points im Team

unausweichlich, die Refactorings in das tägliche Projektgeschäft zu integrieren. Je nach Organisation eines Reengineering-Projektes kann ein dediziertes Improvement Backlog erstellt werden oder die Refactorings werden in das vorhandene Backlog integriert.

Um ein gesundes Gleichgewicht zwischen den Refactorings und der Weiterentwicklung zu erzielen und ihre Abhängigkeiten zu berücksichtigen, stimmen die Teammitglieder die entsprechenden Entwicklungs-Aktivitäten während der Planungs-Meetings ab. Das Team macht daraus ein Challenge, wer wie viel zum Refactoring und zur Weiterentwicklung beigetragen hat. Bei Review-Meetings wird reflektiert, wie viele Improvements und Stories die Teammitglieder geschafft haben.

Das Team definiert, wie der gute Beitrag zur Verbesserung und Weiterentwicklung belohnt werden soll. Neben individuellen Belohnungen ist auch die Team-Belohnung eine gute Wahl zur Stärkung des Team-Spirits.

Literatur

[1] D. Sokenou, B. Güldali: Architektur nicht(!) versenken - Software-Architektur spielerisch evaluieren und verbessern, Java Magazin Whitepaper Architektur, 2024

[2] S. Deterding, D. Dixon, R. Khaled, L. Nacke. From game design elements to gamefulness: defining "gamification". MindTrek '11: Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, September 2011.

[3] Dehla Sokenou, Baris Güldali. Gamification in der Qualitätssicherung – Teil 3: Schach dem Risiko! JavaMagazin, 7.2023, S. 92-97. https://entwickler.de/agile/gamification-risiko-qualitaetssicherung

[4] Henning Schwentner. Domain-Driven Refactorings. https://hschwentner.io/domain-driven-refactorings/

Visualization of Evolving Architecture Smells

Sandro Schulze
Anhalt University of Applied Sciences
sandro.schulze@hs-anhalt.de

Armin Prlja
TU Braunschweig
arminprlja@hotmail.de

Abstract

Architecture Smells (AS) have gained importance in recent past as indicator of bad practices related to the design of software systems. While AS and their symptoms show up on a rather abstract level compared to code smells, both share the characteristic that they evolve over time. This, in turn, may lead to even more severe smells that manifest themselves in the system. However, understanding the evolution of AS and when or how such smells tend to degrade in an undesirable way is not trivial given just tons of data from an analysis tool. In this paper, we introduce a visualization based on network graphs that support developers in understanding the evolution of three common architecture smells: cyclic dependencies, hub-like dependencies, and unstable dependencies. We also discuss scenarios when this is beneficial and what are current limitations of our visualization.
keywords software visualization, software evolution, architecture smells, software quality, software metrics

1 Introduction

As software evolves, it is likely that changes are made without adhering to design or coding principles, and thus, a software system exhibit *smells* causing quality degradation over time [4]. As a result, this may lead to undesired effects (e.g., reduced maintainability, increased bug proneness), commonly described as the metaphor of *technical debt* [3]. While there are different kinds of smells (e.g., code smells, test smells), in this work we focus on *architecture smells (AS)* that constitute violations against best practices for designing software architecture.

To gain comprehensive understanding of when and why architecture smells appear. As well as how they manifest in the system, the evolution of such smells is of superior interest [2, 5]. However, the plain data of such evolutionary analysis is of limited use for developers or software architect for understanding the origin and impact of architecture smells.

In this paper, we propose different visualizations that allow developers to investigate such AS in more depth and in an interactive way, and thus, allow to reason about birth, life, and death of such smells. While our visualization currently supports three common AS, namely cyclic dependencies (CD), hub-like dependencies (HD), and unstable dependencies (UD) [6], we only focus on CDs due to space limits.Particularly, we present the visualisations and briefly discuss why we have chosen them and which information they convey. Moreover, a preliminary study based on cognitive walkthrough indicates the usefulness of our visualization, even though a more profound empirical evaluation is required and planned for future.

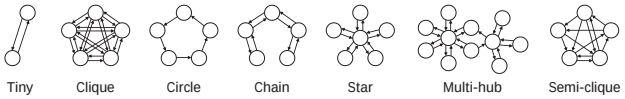


Figure 1: Different topologies for CDs based on Al-Mutawa et al. [1].

2 Background

We briefly introduce CDs and concepts that we developed to investigate their evolution [2].
Cyclic Dependencies exist between two or more components, if these components mutually depend on each other, and thus, increase the risk of ripple effects [6]. Such smells can exist on class-level as well as package-level (assuming Java and programming language). Moreover, CDs can differ in their shape, with more complex shapes indicating a higher negative impact on the overall system's quality [2]. In Figure 1, we we show the different shapes as proposed by Al-Mutawa et al. [1].

We introduced the notion of *intra-version* smells and *inter-version* smells to distinguish between a smell of one particular version and smells that exist over a certain period of time. More precisely, an intra-version smell is a smell in one particular version, whereas an inter-version smell is a set of related intra-version smells over multiple versions of a system.

Moreover, we consider *merging* and *splitting* in the evolution of CDs. In a nutshell, due to adding or removing dependency edges, CDs can merge together, resulting in a larger CD, or split, resulting in two smaller CDs. This concept allows us to perform novel analyses and identify possible situations that are critical wrt AS evolution.

3 Architecture Smell Visualization

For visualizing inter-version CD smells (see Figure 2 for an example), we have chosen a graph-based visual-

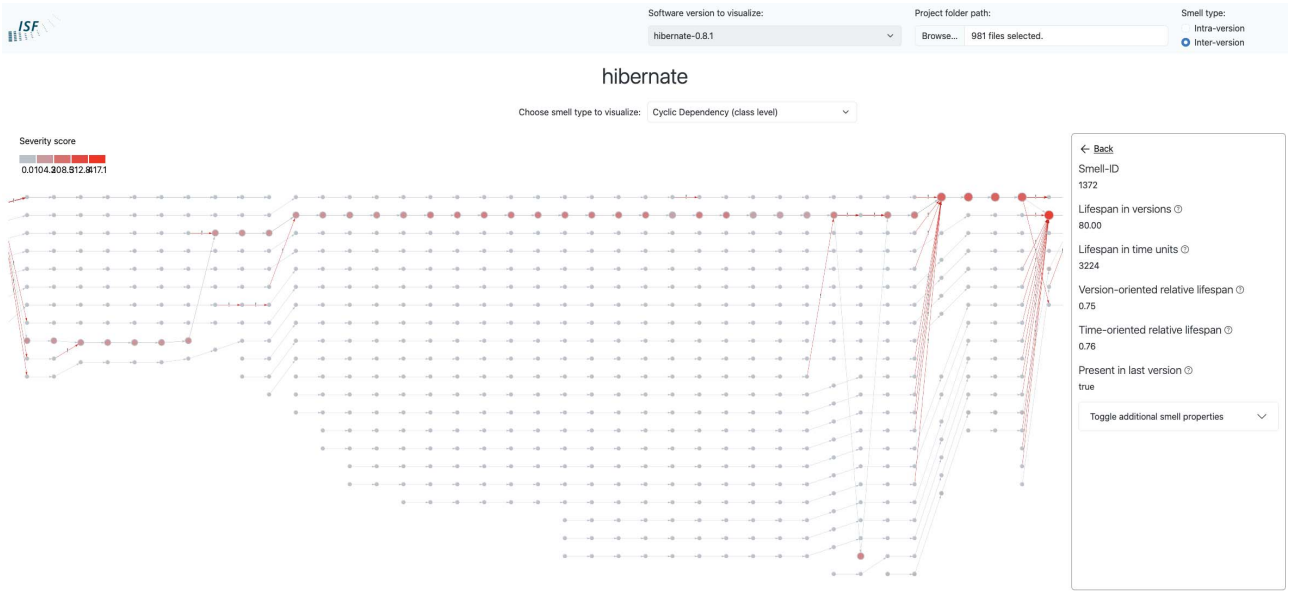


Figure 2: An example for an inter-version smell in hibernate with merging, splitting, and changes of shapes.

ization using network graphs with vertices representing an intra-version smell in a particular version and edges representing transitions between. This graph-based visualization allows us to visualize the different concepts such as merging/splitting, change of shapes, detailed information about participating intra-version smells, but also the time aspect to visualize the evolution over several versions. Next, we briefly explain how different kind of information is visualized.

Merging & Splitting. We use parallel (sub)graphs for indicating CD families that are subject to merging/splitting. In case that such CDs merge, multiple vertices exhibit an outgoing edge to the same successor vertex. In contrast, if a split occurs, this is visualized by one vertex from which multiple outgoing edges have different successor vertices.

Change of Shapes. During evolution, inter-version smells can change their shape, e.g., from a star-like shape to a multi-hub. In our visualization, we indicate such a change by a collared edge that additionally is annotated with an exclamation mark. When hovering over this exclamation mark, a tool tip provides information about the the source and the target shape.

Details of Intra-Version Smells. AS inter-version smells consist of a set of related intra-version smells, it may be of interest to inspect a particular intra-version smell. We support this kind of exploration; by just clicking on an intra-version smells, information shows up by means of metrics such as size, order, or severity score. As future work, we also want to allow users to seamlessly witch to the detailed visualization of the intra-version smell.

Metrics. Eventually, we provide various characteristics of inter-version smells either explicitly or implicitly to the user. For instance, the size of the vertices corresponds to the size of an intra-version smell ,that is, the number of classes/modules that form the intra-version smell of a particular version. Additionally, the color of a vertex corresponds to its severity score with a red note indicating a higher severity score. Finally, we provide additional metrics for the shown inter-version smell, such as lifetime, in a menu on the right-hand side.

References

[1] H. A. Al-Mutawa, J. Dietrich, S. Marsland, and C. McCartin. On the shape of circular dependencies in Java programs. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 48–57. IEEE, 2014.

[2] P. Gnoyke, S. Schulze, and J. Krüger. An evolutionary analysis of software-architecture smells. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 413–424. IEEE, 2021.

[3] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[4] D. L. Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 279–287. IEEE, 1994.

[5] D. Sas, P. Avgeriou, and U. Uyumaz. On the evolution and impact of architectural smells: An industrial case study. *Empirical Software Engineering*, 27(4), 2022.

[6] G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier, 1 edition, 2014.

Auf dem Weg zum automatischen Reengineering von digitalen Zwillingen mit faktenbasierten großen Sprachmodellen

Vasil L. Tenev
Fraunhofer IESE, Kaiserslautern
vasil.tenev@iese.fraunhofer.de

Zusammenfassung

Industrie 4.0 hat große Datenmengen aus verschiedenen Unternehmenssystemen zugänglich gemacht, aber die Umsetzung des Konzepts der Digitalen Zwillinge (DZe) bleibt eine Herausforderung. Dieser Beitrag schlägt vor, große Sprachmodelle (Large Language Models, LLMs), in Verbindung mit Wissensgraphen zu nutzen, um das Reengineering von DZe zu automatisieren. Ein zentraler Ansatz besteht darin, eine Plattform zur Erstellung von Wissensgraphen zu entwickeln, die Daten aus verschiedenen Quellen integriert und in ein einheitliches Format übersetzt. Durch den Einsatz von LLMs können komplexe Probleme gelöst werden, ohne dass spezifisches Wissen in die Modelle eingearbeitet werden muss. Dieser Ansatz bietet Vorteile wie die Vermeidung von Fehlinformationen und die Schaffung eines einzigen Wahrheitspunktes.

1 Einführung

Moderne Unternehmen erzeugen durch die Verbreitung vernetzter Geräte, die Automatisierung und die datengesteuerter Entscheidungsfindung beispiellose Datenmengen. Die Daten können aus verschiedenen Quellen stammen, wie zum Beispiel ERP-Systemen (Enterprise Resource Planning), PLM-Software (Product Lifecycle Management), IoT-Geräten (Internet of Things) und in Produktionsmaschinen eingebetteten Sensoren. Obwohl diese Datenflut Möglichkeiten zur Optimierung und Innovation bietet, stellt sie auch eine große Herausforderung dar, insbesondere im Hinblick auf die effektive Nutzung dieser Informationsfülle.

Das Problem. Inmitten dieser Datenflut stellt die Darstellung physischer Entitäten eine entscheidende Herausforderung dar, insbesondere im Bereich der DZe. DZe sind virtuelle Darstellungen von physischen Anlagen, Prozessen oder Systemen und haben sich zu einem Eckpfeiler der Industrie 4.0 entwickelt. Sie versprechen eine verbesserte Überwachung, vorausschauende Wartung und Optimierung. Trotz der Fortschritte bei der Datenerfassung und -analyse gibt es jedoch nach wie vor eine bemerkenswerte Lücke bei der Erreichung einer ganzheitlichen und genauen Darstellung physischer Entitäten innerhalb der DZe. Bestehende Ansätze haben oft Schwierigkeiten, Daten aus unterschiedlichen Quellen zu integrieren, widersprüchliche Informationen abzugleichen und eine einheitliche Sicht auf komplexe Fertigungssysteme zu bieten.

Die Vorteile. Angesichts dieser Herausforderungen soll der vorgeschlagene Ansatz die Synergien zwischen zwei Technologien nutzen: LLMs und Wissensgraphen. LLMs haben ihre Fähigkeiten im Verstehen und Generieren von natürlichsprachlichem Text bewiesen, während Wissensgraphen eine strukturierte Rahmenbedingung für die Organisation und Verbindung von verschiedenen Datenelementen bieten. Durch die Kombination der Stärken von LLMs und Wissensgraphen soll die grundlegende Lücke bei der umfassenden Darstellung physischer Entitäten in DZen geschlossen werden. Die Motivation beruht auf der Überzeugung, dass die Nutzung dieser fortschrittlichen Technologien die automatisierte Umgestaltung von DZen ermöglicht und einen nahtlosen Übergang von disparaten Datenquellen zu kohärenten, umsetzbaren Erkenntnissen erleichtert. Mit diesem Ansatz sollen Hersteller befähigt werden, das volle Potenzial der Digitalisierung auszuschöpfen, um Effizienz, Innovation und Wettbewerbsvorteile in einer zunehmend digitalisierten Landschaft zu fördern.

2 Über Daten, Modelle und Wissen

Im Kern beschreibt die DIKW-Hierarchie (Data-Information-Knowledge-Wisdom) die Transformation von Rohdaten in strukturierte Informationen, kontextbezogenes Wissen und schließlich handlungsrelevantes Wissen. Es ist nicht nur wichtig, Daten zu sammeln, sondern ihre Umwandlung in aussagekräftiges Wissen zu orchestrieren, das eine fundierte Entscheidungsfindung ermöglicht.

Digitale Zwillinge und Verwaltungsschalen. DZe sind virtuelle Gegenstücke zu physischen Anlagen, Prozessen oder Systemen und bieten ein digitales Abbild in Echtzeit zur Überwachung, Analyse und Optimierung. Das Metamodell der Verwaltungsschale (Asset Administration Shells, AAS) dient als standardisiertes Darstellungsschema, um die wesentlichen Eigenschaften und Verhaltensweisen von Industriean-

lagen innerhalb eines cyber-physischen Ökosystems zu kapseln. Die Verwendung des AAS Metamodells allein bietet keine Garantie für Interoperabilität, Konsistenz und Skalierbarkeit und ist als Grundlage für ein automatisiertes Reengineering von DZen unzureichend.

Wissensgraphen. Wissensgraphen sind ein leistungsfähiges Paradigma für die Organisation, Integration und semantische Anreicherung heterogener Datenquellen innerhalb eines einheitlichen Rahmens. Im Gegensatz zu herkömmlichen relationalen Datenbanken verwenden Wissensgraphen ein graphenbasiertes Datenmodell, das die inhärenten Beziehungen und die Semantik zwischen Entitäten, Attributen und Konzepten erfasst. Diese vernetzte Struktur erleichtert flexible Abfragen, Schlussfolgerungen und Wissensentdeckungen und ermöglicht es den Beteiligten, aus komplexen und vernetzten Datensätzen verwertbare Erkenntnisse zu gewinnen. In unserem Ansatz dienen Wissensgraphen als Eckpfeiler für die Darstellung der umfassenden Wissensbasis, die aus unterschiedlichen Datenquellen abgeleitet wird. Durch den Einsatz von Wissensgraphen-Technologien können wir unterschiedliche Sichtweisen auf Daten miteinander in Einklang bringen, eine einzige Quelle der Wahrheit schaffen und ganzheitliche Analysen und Entscheidungen im Kontext von DZen und cyber-physischen Systemen ermöglichen.

3 Vision

Unsere Vision basiert auf der Transformation von Daten aus verschiedenen Unternehmenssystemen wie ERP, PLM und anderen operativen Datenbanken. Diese Daten dienen als grundlegende Bausteine für den Aufbau des Wissensgraphen und liefern den notwendigen Input für die Synthese strukturierter Darstellungen physischer Entitäten und ihrer zugehörigen Attribute, Beziehungen und Verhaltensweisen.

Von Informationen zu Wissensgraphen. Relationale Informationen aus verschiedenen Unternehmenssystemen bilden die Grundlage für den Aufbau strukturierter Darstellungen physischer Einheiten und ihrer zugehörigen Attribute, Beziehungen und Verhaltensweisen. Um die Lücke zwischen Informationen und umsetzbarem Wissen zu schließen, schlagen wir vor, die Fähigkeiten von LLMs zu nutzen. Unser Ansatz besteht darin, relationale Informationen aus verschiedenen Datenquellen in ein einheitliches Wissensgraphenformat zu übersetzen und so die Herausforderung der Halluzination zu bewältigen. LLMs mindern das Risiko der Erzeugung falscher oder irreführender Informationen, indem sie menschliche Eingabeaufforderungen in die formale Abfragesprache der Wissensgraphen-Datenbank umwandeln. Dadurch wird das Phänomen der Halluzination vermieden. Darüber hinaus integrieren wir selbstkorrigierende Mechanismen, die auf Rückkopplungsschleifen basieren, um LLMs im Laufe der Zeit kontinuierlich zu verfeinern und zu verbessern. Dadurch wird ihre Genauigkeit, Zuverlässigkeit und Anpassungsfähigkeit erhöht.

Kohärenz und Vereinbarkeit. Der resultierende Wissensgraph stellt ein reichhaltiges Netzwerk von miteinander verbundenen Knoten und Kanten dar, das das kollektive Wissen aus verschiedenen Datenperspektiven zusammenfasst. Durch die Integration von Regeln für die Konsistenz und den Abgleich verschiedener Datenquellen dient der Wissensgraph als zentraler Ort der Wahrheit und harmonisiert fragmentierte Perspektiven in einer einheitlichen Darstellung. Dieser "Single Point of Truth" ermöglicht den Beteiligten ein ganzheitliches Verständnis der zugrundeliegenden physischen Entitäten und erleichtert eine fundierte Entscheidungsfindung, strategische Planung und operative Optimierung.

Vom Wissensgraph zu den digitalen Zwillingen. Um eine nahtlose Interaktion und einen intuitiven Zugang zu dem in den Wissensgraphen kodierten Wissen zu ermöglichen, schlagen wir vor, LLMs für die bidirektionale Kommunikation zwischen Stakeholdern und DZen zu verwenden. Durch die Übersetzung von Anfragen in natürlicher Sprache in Datenbankabfragen ermöglichen LLMs den Nutzern, ihre Informationsbedürfnisse in einer ihnen vertrauten Sprache zu artikulieren, was die Zugänglichkeit und Benutzerfreundlichkeit verbessert. Darüber hinaus erleichtern LLMs die Übersetzung von Datenbankabfrageergebnissen in Antworten in natürlicher Sprache, so dass die Beteiligten die von den DZen gewonnenen Erkenntnisse leicht interpretieren und entsprechend handeln können.

4 Fazit

Zusammenfassend bietet der von uns vorgeschlagene Ansatz einen vielversprechenden Weg zur Automatisierung des Reengineerings von DZe durch die Nutzung der Fähigkeiten von faktenbasierten LLMs und Wissensgraphen. Indem wir die Lücke zwischen Rohdaten und umfassenden Repräsentationen physischer Entitäten schließen, ermöglichen wir eine effizientere und genauere Entscheidungsfindung in komplexen industriellen Umgebungen. Mit der weiteren Erforschung seiner Anwendungen und der Verfeinerung seiner Implementierung könnte unser Ansatz die Art und Weise, wie digitale Repräsentationen in verschiedenen Bereichen erstellt und genutzt werden, verbessern.

Die Anwendung unseres Ansatzes geht über das Reengineering von DZen hinaus. Er kann beispielsweise für die automatische Konfiguration von Simulationsmodellen verwendet werden, was den Prozess rationalisiert und die Genauigkeit erhöht. Darüber hinaus zeigt unsere Methodik Potenzial bei der Lastprognose in Stromnetzen, wo prädiktive Analysen eine entscheidende Rolle bei der Aufrechterhaltung von Stabilität und Effizienz spielen.

KI-gestützte Modernisierung von Altanwendungen: Anwendungsfelder von LLMs im Software Reengineering

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

1. Motivation

Die Integration von Large Language Models, kurz LLMs, in den Modernisierungsprozess von Altanwendungen bietet nicht nur eine Vielzahl technologischer Möglichkeiten, sondern dient auch als starke Motivation für Unternehmen, ihre bestehenden Systeme zu verbessern. LLMs repräsentieren einen bedeutsamen Fortschritt in der künstlichen Intelligenz (KI), veraltete Anwendungen können mit, aber auch durch LLMs aufgewertet werden. Durch die Nutzung von LLMs können Unternehmen nicht nur ihre Effizienz steigern, sondern auch ihre Wettbewerbsfähigkeit auf dem Markt stärken. Diese Ausarbeitung adressiert die folgenden Fragen zur Implementierung von LLMs im Modernisierungsprozess:

FF1 Wie können LLMs die Modernisierung von Altanwendungen im Anforderungsmanagement unterstützen?

FF2 Inwiefern ermöglichen LLMs effiziente Software Reengineering Prozesse?

2. LLMs im Software Reengineering

LLMs wie OpenAI's ChatGPT bieten vielfältige Möglichkeiten im Software Reengineering. Neben der Klassifizierung des Aufwands von Anforderungen können sie auch zur automatischen Dokumentationserstellung, natürlichsprachlichen Anforderungsspezifikation, Anforderungsanalyse und -verarbeitung sowie zur Codegenerierung und -migration eingesetzt werden. Darüber hinaus unterstützen LLMs die Qualitätssicherung und das Testing, indem sie bei der automatischen Generierung von Testfällen (ggf. auch Testdaten) und -szenarien helfen.

3. Existierende Arbeiten

Eine Analyse existierender Arbeiten zum Einsatz Generativer KI, insbesondere LLMs im Software Reengineering Kontext brachte die folgenden Schwerpunkte:

LLMs in der Softwaretechnik haben die Fähigkeit, Codeschnipsel oder komplette Programme zu generieren, Code zum besseren Verständnis zusammenzufassen und Schwachstellen und Fehler im Code zu erkennen. Sie dienen als wertvolle Werkzeuge für Aufgaben wie automatisierte Codegenerierung, Codedokumentation und Codeanalyse und sind daher in allen Phasen des Software Engineerings nützlich [1].

Die Implementierung der Koordination von LLMs und Plattformwissen für die Software-Modernisierung beinhaltet die Gestaltung von Arbeitsabläufen, die LLMs mit Prompt-Engineering, Qualitätskontrolle und Fusions-/Aggregationsstrategien einbeziehen. Auf der Ebene des Ensembles können parallele Muster mit mehreren Verzweigungen und Aggregation sowie Sequenzmuster mit Modellausgaben als Eingaben verwendet werden. Auf der Ebene der einzelnen LLM können Parameterabstimmung und RAG-Strategien (Red, Amber, Green) eingesetzt werden. Darüber hinaus können Low-Code und automatische Koordinationslogik entwickelt werden, um den Aufwand für Ingenieure zu reduzieren [2].

Zu den ermittelten Schwerpunkten wurden Studien identifiziert, welche die Leistungsfähigkeit unterschiedlicher LLMs in den Anwendungsfeldern des Software Reengineering untersuchten:

Eine Studie von Federico Cruciani et al. (2023) untersuchte die Leistungsfähigkeit von LLMs bei der Klassifikation von Anforderungen unter Verwendung eingebetteter Wörter. Die Analyse umfasste die Klassifizierung von Anforderungen in funktionale und nicht-funktionale Kategorien sowie die Identifizierung der häufigsten Klassen. Darüber hinaus wurde die Leistung der LLMs bei der Klassifizierung in alle vorhandenen Klassen untersucht. [3]

In der Softwaretechnikforschung wird versucht, Entwicklungsprozesse basierend auf informell formulierten Absichten der Entwickler durch automatisierte Ansätze zu unterstützen. Eine Reihe von Methoden zielen darauf ab, natürlichsprachliche Beschreibungen in Quellcode umzuwandeln, wobei jedoch noch kein umfassendes Verständnis für deren Effektivität und Ergänzung untereinander besteht. Eine groß angelegte empirische Studie zeigt, dass kombinierte Techniken zur natürlichsprachlichen Codesuche und -generierung die Leistung um 35% im Vergleich zur besten eigenständigen Methode verbessern können. [4]

4. Prototypische Klassifikation von Anforderungen

Um die Forschungsfragen zu beantworten, haben wir uns für experimentelles Prototyping entschieden. Diese Methode ermöglicht praxisnahe und qualitätsorientierte Ergebnisse. Wir verwenden die vorklassifizierten Datensätze aus der Evaluation von AutoML von 2023 [5]. Diese 147 Softwareanforderungen haben folgenden Struktur:

„The software should have a community forum feature, Low“.

Der Prototyp ist in Jupiter Notebook mit phyton implementiert. Es wird als ML Vertreter scikit-learn [6] trainiert und getestet. Hierfür werden die Datensätze in Trainings- (80%) und Testdaten (20%) aufgeteilt, wobei ein Random Seed von 42 verwendet wurde. Die LLM Vertreter sind ChatGPT 3.5 Turbo und ChatGPT 4 von OpenAI [7]. Diese werden per restful WeoAPI verwendet mit jeweils einen Prompt pro Datensatz, ohne die Vorklassifizierung mitzusenden. Somit erfolgt kein Training (ZeroShot).
In Abbildung 1 wird der Ablauf der Experimente kurz dargestellt.

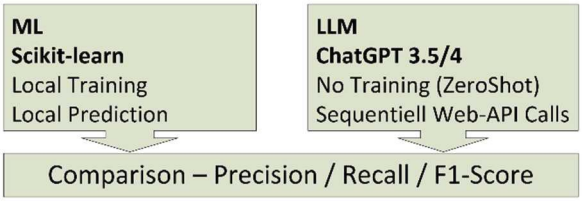


Abbildung 1 Schematischer Ablauf der Experimente im Prototypen

Der Vergleich zwischen den beiden Ansätzen erfolgt anhand verschiedener Metriken: Für die ML-Modelle wurden gewichtete Durchschnittswerte verwendet, während bei den LLM-Modellen die Übereinstimmungen mit den manuellen Klassifikationen aus den Rückgaben der Web-APIs verglichen wurden. Die ersten Ergebnisse sind in Tabelle 1 aufgeführt.

Tabelle 1 Vergleich von LLMs und legacy KI

	scikit-learn	ChatGPT 3.5	ChatGPT 4
time	1sec	15min	50min
Precision	0.60	0.55	0.29
Recall	0.63	0.35	0.01
F1 Score	0.54	0.29	0.01

5. Reflektion und Ausblick

Die Ergebnisse belegen, dass trainierte ML-Modelle für die Klassifizierung des Aufwands von Anforderungen eindeutig besser geeignet sind als LLMs. Die Präzision, Recall und F1-Score der ML-Modelle waren deutlich höher als die der LLMs. Dies lässt sich zum einen auf die effektive Vorverarbeitung der Daten zurückführen, die für die ML-Modelle durchgeführt wurde, und zum anderen auf das gezielte Finetuning, das auf die spezifische Aufgabe der Aufwandsklassifizierung abgestimmt war.

Die Laufzeiten der verschiedenen Ansätze variieren erheblich. Während ML-Modelle schnellere Ergebnisse liefern, erfordern LLMs deutlich mehr Zeit. Dies ist bedingt durch die Netzwerkkommunikation und die generative Funktionsweise, die wesentlich mehr Ressourcen benötigt.

Die Ergebnisse lassen den Schluss zu, dass es Raum für Verbesserungen bei der Nutzung von LLMs für die Klassifizierung von Anforderungen gibt. Mögliche Ansätze sind eine feinere Abstimmung der Modelle, Training, Finetuning und die Optimierung des Anfragepromptes. Die Initialfragen können beantwortet werden:

FF1 LLMs wie ChatGPT 3.5 und ChatGPT 4 können die Modernisierung von Altanwendungen unterstützen, indem sie natürlichsprachliche Anforderungen analysieren und klassifizieren. Allerdings ist die Ergebnisqualität untrainiert im Vergleich zu traditionellen ML-Modellen möglicherweise begrenzt.

FF2 LLMs können effiziente Software-Reengineering-Prozesse unterstützen, indem sie Anforderungen automatisiert analysieren und verarbeiten. Jedoch sind ihre Ressourcenanforderungen im Vergleich zu traditionellen ML-Modellen höher.

Insgesamt bieten die Experimente Einblicke in die Leistungsfähigkeit verschiedener Ansätze zur Klassifizierung des Aufwands von Anforderungen und zeigen potenzielle Bereiche für zukünftige Forschung und Verbesserungen auf.

6. Quellenverzeichnis

[1] ZHENG, Z. ; NING, K. ; CHEN, J. ; WANG, Y. ; CHEN, W. ; GUO, L. ; WANG, W.: *Towards an Understanding of Large Language Models in Software Engineering Tasks*. URL http://arxiv.org/pdf/2308.11396.pdf. – Aktualisierungsdatum: 2023-08-22

[2] LINH TRUONG ; MAJA VUKOVIC ; RAJU PAVULURI: *On Coordinating LLMs and Platform Knowledge for Software Modernization and New Developments*. WorkingPaper. URL https://acris.aalto.fi/ws/portalfiles/portal/133671133/collms_position.pdf. – Aktualisierungsdatum: 2023-11-22

[3] FEDERICO CRUCIANI ; SAMUEL MOORE ; CD NUGENT: Comparing general purpose pre-trained Word and Sentence embeddings for Requirements Classification, Bd. 3378. In: *Joint Proceedings of REFSQ-2023 Workshops, Doctoral Symposium, Posters & Tools Track and Journal Early Feedback (REFSQ-JP 2023)* : CEUR-WS, 2023

[4] WANG, S. ; GENG, M. ; LIN, B. ; SUN, Z. ; WEN, M. ; LIU, Y. ; LI, L. ; BISSYANDÉ, T.F. ; MAO, X.: Natural Language to Code: How Far Are We? In: CHANDRA, Satish; BLINCOE, Kelly; TONELLA, Paolo (Hrsg.): *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA : ACM, 2023, S. 375–387

[5] SCHMIETENDORF, A. ; HARTENSTEIN, S. ; JOHNSON, S.L.: KI-gestützte Modernisierung von Altanwendungen: (Sentiment-) Analysen im Diskurs des Anforderungsmanagements. In: *Workshop Software-Reengineering und -Evolution WSRE 2023*, 2023, S. 20–21

[6] PEDREGOSA, F. ; VAROQUAUX, G. ; GRAMFORT, A. ; MICHEL, V. ; THIRION, B. ; GRISEL, O. ; BLONDEL, M. ; PRETTENHOFER, P. ; WEISS, R. ; DUBOURG, V. ; VANDERPLAS, J. ; PASSOS, A. ; COURNAPEAU, D. ; BRUCHER, M. ; PERROT, M. ; DUCHESNAY, E.: *Scikit-learn: Machine Learning in Python*. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830

[7] OPENAI: *ChatGPT API*. 2022. URL https://platform.openai.com/docs/overview

Chances and Challenges of LLM-based Software Reengineering

Jochen Quante and Matthias Woehrle

Bosch Research
Renningen, Germany

Large Language Models (LLMs) have opened up unforeseen new possibilities. They deliver amazing results for complex text-based tasks for which no satisfactory automated solution was available before. This is even more astonishing as they just calculate the most probable subsequent token, given a sequence of tokens. This is also true for software development support: There are various software engineering task for which LLMs show potential [1, 5]. In this paper, we discuss the potential and current shortcomings of LLM-based approaches for selected Software Reengineering tasks with a focus on language translation. All experiments reported in the following were performed with GPT-4.

1 Language Translation

Language translation has been performed based on abstract syntax trees (AST) for decades. A typical use case for language translation are legacy applications from banking or insurance domains that often need to be translated from COBOL to Java and migrated to a new platform. This is necessary as the old hardware systems and compilers can no longer be maintained, and as it is increasingly hard to find people who have the required competencies to maintain that kind of code. The problem of these AST-based approaches is that the resulting code does not look and feel like real Java code, as it does not use the idioms of the target language, but rather stays close to the style of the original language. For example, when transforming from COBOL to Java, classes will be used. However, they will not be used in a natural way, but rather as containers for the respective code.

In contrast to that, LLMs have the capability to do a different way of translation, as they take more context into account. These models can leverage knowledge of similar code of the target language that they have seen during training. In the best case, they "recognize" an algorithm and translate it to the same algorithm in the target language. This works particularly well for often-used algorithms like sorting and searching. However, things look different for custom code. And from our experience, this heavily depends on the languages used: The more code in a language was contained in the training data, the better it works. As an example, the Python capabilities of current LLMs often excel other languages.

As a concrete example, let us discuss the results of some experiments on C to Rust translation using an LLM. Let us first look at the advantages of the LLM-based approach: The code from GPT-4 looked much better (more Rust-like) than code that was translated using c2rust[1], a classical AST-based tool for this task. This is because c2rust does a 1:1 translation. That means that all code is wrapped in unsafe blocks and then the original C implementation is used, e.g., the C functions as well as C strings. In contrast to that, the LLM translation mostly uses the adequate Rust-specific language features. It creates "safe" code whenever possible, which enables much stronger memory safety checks by the compiler. It can also produce code in different flavors, e.g., you can ask the LLM to generate the code in a more functional style. However, there are also some disadvantages. The LLM-based approach sometimes fails to use the right types, e.g., built-in types of Rust. Moreover, it has problems with correctly translating global variables from C, which basically do not exist in Rust. This means that in many cases, the proposed code is not even compilable. Thus, the LLM-based translation often requires significant manual clean-up work or additional interaction with the LLM to get to a correct translation.

Another issue for current models arises with longer functions or even whole code bases that shall be translated. Due to the limited context size of current LLMs, they can only deal with text of limited size. For example, the version of GPT-4 we used has a context size of 8,192 tokens, which often means for the translation tasks that only functions on the order of 100 lines of code can be translated at once. Furthermore, LLMs have a random component: you get different results for the same prompt each time, even with the lowest possible temperature. This means when translating pieces of code iteratively (to deal with limited context size), these fragments do not necessarily fit together. Again, this results in manual work, as identifiers may be renamed differently, different types may be used, etc. Therefore, it is hardly possible to use LLMs for translating entire code bases without additional measures[2]. You must at least provide detailed instructions on how to deal with certain constructs, like global variables, to get a more consistent and syntactically

[1] `https://c2rust.com/`

[2] See Outlook below for comments on current developments.

and semantically correct set of translated functions and data structures.

Overall, LLMs deliver very promising translation results for small code snippets or individual functions. However, current models with context limits are hardly usable for translation of larger code bases.

2 Other Reengineering Tasks

Besides language translation, LLMs can support in many other code-related tasks [1, 5]. When asking GPT to propose **software refactorings** for a given piece of code, it usually comes up with a very generic list of things that could be done: Choose better identifier names, split the function into smaller ones, etc. However, when asking it to perform these refactorings on a concrete piece of code, its capabilities are quite limited. While simple things such as the introduction of better identifier names often works quite well if the code is sufficiently self-explanatory, more sophisticated refactorings like extracting methods fail completely or deliver incomplete code. For performing such refactorings, the classical approach still seems to be the better choice.

LLMs can also be asked to explain code, so they can potentially help in **program comprehension**. This works perfectly fine for well-known algorithms, even if names of the functions and identifiers are obfuscated. For custom code, you often get an explanation that sounds reasonable, yet often the explanation is either wrong or not helpful, as it then explains the code line by line. Hence, this is not a big advancement compared to state-of-the-art code summarization approaches on that level. Nevertheless, we should mention that there were recently remarkable advances in answering sophisticated questions about a whole code base with long context models [3]. These results are very encouraging and contrast our current experiments on smaller models.

Finally, let use mention several further promising tasks for reengineering support. These include automatic program repair [2], performance improvement [1], as well as code search [5]. For a comprehensive overview of the potential of LLMs for software engineering tasks in general, we refer to Fan *et al.* [1].

3 Outlook

Predictions are hard, especially about the future. This is particularly true in the fast-moving field of LLMs. As such, current failing applications of LLMs in Software Reengineering tasks may only be a current snapshot as technology evolves. As an example, we want to highlight the rapid progress with respect to context size. While the context size of current models may limit the capabilities in program comprehension as discussed above, recent demonstration of LLMs that allow to feed a complete codebase into the LLMs have shown remarkable capabilities [3]. It will be exciting to see whether improvements in LLMs and systems including LLMs can remedy current limitations. Nevertheless, we need to consider how developers will interact with such a system depending on the success rate of queries and what we can learn from other engineering fields that have seen similar increasing levels of automation [4].

4 Conclusion

LLMs offer completely new possibilities for Software Reengineering. Notably, they take a different approach compared to classical techniques and thus provide different advantages, e.g., adapting to coding style. Moreover, LLM-based approaches outperform classical techniques on several benchmark tasks. However, there are some open problems with these approaches. For example, we mentioned scalability above. More substantially, there is in general a lack of any guarantee about the result quality. This lack of guarantees is however an inherent property of any AI approach, and thus we – as a software (re-)engineering community, probably have to identify additional (non-AI) measures to address it. For now, LLMs are a programmer's companion that can be used to generate proposals. The developers must be in the loop and check if the result is really what they wanted. This interaction between the developer and the companion and corresponding increasing levels of automation [4] is an important challenge and may require even deeper skills from the developer.

References

[1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. Large language models for software engineering: Survey and open problems. In *Proc. of 45th Int'l Conf. on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, 2023.

[2] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large language models. In *Proc. of 45th Int'l Conf. on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[3] M. Reid, N. Savinov, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[4] A. Sellen and E. Horvitz. The rise of the AI Co-Pilot: Lessons for design from aviation and beyond. *arXiv preprint arXiv:2311.14713*, 2023.

[5] G. Sridhara, R. H. G., and S. Mazumdar. ChatGPT: A study on its utility for ubiquitous software engineering tasks. *arXiv preprint arXiv:2305.16837*, 2023.