A Vision on Intentions in Software Engineering

Jacob Krüger
Eindhoven University of Technology
Eindhoven, The Netherlands
j.kruger@tue.nl

Yi Li
Nanyang Technological University
Singapore, Singapore
yi_li@ntu.edu.sg

Chenguang Zhu
The University of Texas at Austin
Austin, USA
cgzhu@utexas.edu

Marsha Chechik
University of Toronto
Toronto, Canada
chechik@cs.toronto.edu

Thorsten Berger
Ruhr-University Bochum &
Chalmers | University of Gothenburg
Bochum & Gothenburg
Germany & Sweden
thorsten.berger@rub.de

Julia Rubin
University of British Columbia
Vancouver, Canada
mjulia@ece.ubc.ca

ABSTRACT

Intentions are fundamental in software engineering, but they are typically only implicitly considered through different abstractions, such as requirements, use cases, features, or issues. Specifically, software engineers develop and evolve (i.e., change) a software system based on such abstractions of a stakeholder's intention—something a stakeholder wants the system to be able to do. Unfortunately, existing abstractions are (inherently) limited when it comes to representing stakeholder intentions and are mostly used for documenting only. So, whether a change in a system fulfills its underlying intention (and only this one) is an essential problem in practice that motivates many research areas (e.g., testing to ensure intended behavior, untangling intentions in commits). We argue that none of the existing abstractions is ideal for capturing intentions and controlling software evolution, which is why intentions are often vague and must be recovered, untangled, or understood in retrospect. In this paper, we reflect on the role of intentions (represented by changes) in software engineering and sketch how improving their management may support developers. Particularly, we argue that continuously managing and controlling intentions as well as their fulfillment has the potential to improve the reasoning about which stakeholder requests have been addressed, avoid misunderstandings, and prevent expensive retrospective analyses. To guide future research for achieving such benefits for researchers and practitioners, we discuss the relationships between different abstractions and intentions, and propose steps towards managing intentions.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**.

KEYWORDS

software evolution, intention, quality assurance

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613087

ACM Reference Format:
Jacob Krüger, Yi Li, Chenguang Zhu, Marsha Chechik, Thorsten Berger, and Julia Rubin. 2023. A Vision on Intentions in Software Engineering. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613087

1 INTRODUCTION

Software developers rely on various abstractions (e.g., features, requirements, issues) to manage their software systems along the dimensions of time (i.e., evolution) and space (i.e., functionalities) [2, 5]. Essentially, these abstractions aim to capture the *intention* of an involved stakeholder. For example, the feature request of a stakeholder represents an intended functionality at an abstract level, requirements specify the boundaries of this intention, a change to the system implements the intended behavior, and a bug report documents a violation of this intention (cf. Figure 1). Essentially, we argue that many of the widely established abstractions used in software engineering implicitly describe stakeholder intentions.

While such abstractions are widely used in practice, we argue that we should reflect on the abstractions' relations to stakeholder intentions as a potential means for better connecting, explaining, and managing these intentions. In fact, we have found that, in practice, developers use issue and pull-request templates[1] to describe the intended evolution of their system. Moreover, researchers have proposed various techniques to analyze, manage, and reverse engineer information on different abstractions and the underlying intentions. Most prominently, researchers have worked on recovering and untangling [6–8, 11, 12] intentions, testing as a means to ensure intended behavior [19], or verifying changes against a specified intention [22, 29].

Apparently, developers as well as researchers care about knowing and specifying intentions. Particularly, we consider *software-change intentions* (SCIs) as an interesting notion that can help describe a developer's underlying intention for changing a system (e.g., fixing a bug, refactoring, implementing a requested feature) and that can connect other, more established abstractions intuitively to each other. However, existing attempts for managing SCIs focus on documenting (e.g., pull-request templates) or recovering

[1] https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/about-issue-and-pull-request-templates

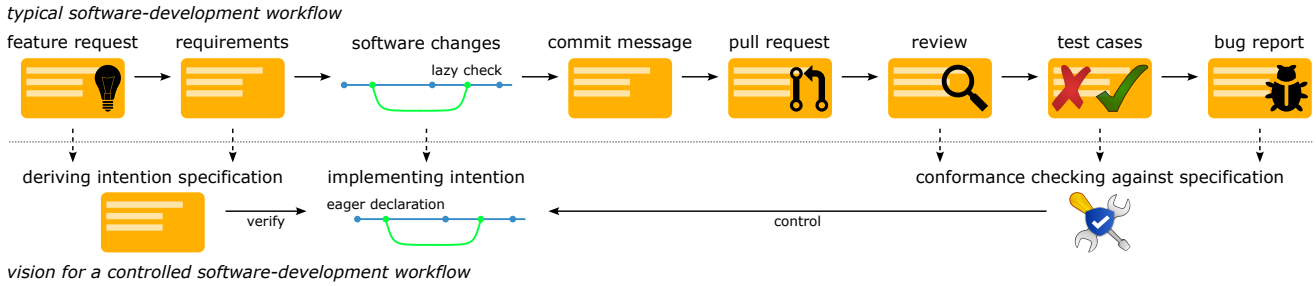

Figure 1: Sketch of a typical software-development workflow (top) and its relations to our vision for a more controlled workflow based on explicitly specified SCIs (bottom).

them after the fact. In the former case, we cannot ensure that the documented SCI aligns with the actual change; in the later case, a tangled or wrong SCI may already be in the system. Both cases are lazy strategies for managing SCIs, and can easily lead to bugs, cluttered version histories [9], or additional efforts when reviewing and quality assuring a system [10]. Moreover, recovering and potentially untangling SCIs after they have been implemented is costly and cognitively challenging [25, 27], especially because only the actual developer knows what their concrete SCI was. Consequently, research on software engineering and evolution also faces challenges, for instance, when the goal is to identify and investigate bug-inducing or refactoring changes.

In this paper, we reflect on existing abstractions in software engineering and their connections to intentions as represented in changes (i.e., SCIs). We argue that research into this direction can help researchers obtain a better conceptual understanding of software development that, in turn, can guide the design of novel techniques. As a concrete instantiation for tackling the aforementioned problems, we sketch the idea of employing an eager strategy for managing SCIs (cf. Figure 1). Specifically, we outline how empirically collected and formally declared SCIs can help (i) control that only a specific SCI can be employed, (ii) verify that a change matches its SCI, and (iii) ensure a reliable documentation throughout a system's evolution. Since this is a long-term vision that is challenging to achieve, we propose intermediate steps for moving into this direction and obtaining novel insights in the mean time.

2 WHY MANAGE INTENTIONS

We reflect on the notion of SCIs as a helpful means for software engineering by discussing the use of different software-engineering abstractions, reviewing related work, and considering observations we made in open-source as well as industrial projects.

Reflection on Research. In research, many techniques and tools aim to recover some form of SCIs from version histories, for instance, to identify refactorings [28] or refactor the version history itself [23]. Other researchers have focused on measuring efforts based on SCIs or predicting and understanding the impact of certain SCIs [18, 21]. Consequently, having more control and a better mapping of SCIs to the actual changes is highly interesting for researchers. This promotes our vision of managing and controlling SCIs via eagerly specifying them (cf. Figure 1), allowing to control the fulfillment of SCIs, to automatically and reliably document changes, and thus to improve data quality, reliability, as well as analyses for researchers and practitioners.

Already in 1976, Swanson [26] started to classify SCIs by defining high-level maintenance activities, namely:

- **Adaptive** changes are intended to update or modify a system to keep it compatible, for instance, with the underlying hardware or other systems.
- **Corrective** changes are intended to fix bugs, design flaws, or security issues in the system.
- **Enhancive** (added later) changes are intended to add new functionality (or tests) to a system, providing new features with specific requirements to the stakeholders.
- **Perfective** changes are intended to improve the system, for instance, by optimizing its properties, refactoring code, or deprecating functionality.

Such SCIs have been used extensively to classify changes, but they are rather abstract and more fine-grained SCIs are likely more helpful for researchers and developers—also to untangle changes with multiple SCIs. A lot of research has built upon this idea and achieved several advancements [6, 7, 9–12, 25, 27, 29, 30], even though recovering SCIs is less reliable than controlling them from the beginning.

Reflection on Practice. A more interesting perspective is the practical point of view. On the top of Figure 1, we sketch an abstracted development workflow that may occur in open-source or industrial projects: Some stakeholder raises a feature request (their intention for the system), which is refined based on requirements (the boundaries of the intention), and then implemented in a change. Already at this point, misunderstandings between different stakeholders (e.g., customer, developer) can cause severe gaps and faults in an intention's specification—which is why extensive research has focused on managing features and requirements [3, 13–15, 17, 24].

Typically, developers document their implemented SCI via commit messages and pull requests, which, however, are no check whether this (and only this) SCI has been properly implemented. For instance, GitHub allows developers to create templates for issues and pull requests. The templates define a rough corpus of what information a developer or user should provide when they create issues or pull requests. Over time, such templates have become somewhat common, with various standard templates being collected in different repositories.[2] In fact, searching for "add pull request template" returns roughly 5 million issues and 250 thousand commits on GitHub, with large projects using such templates to structure their documentation and communication, for instance, by

[2] https://github.com/stevemao/github-issue-templates
https://github.com/devspace/awesome-github-templates

updating their templates to align them to their needs.[3] Such templates involve different types of information, such as a descriptions of what the issue or pull request is about; links to related documentation (e.g., pull requests linking to a respective issue); checklists for confirming to project styles or testing practices; and regularly a selection of the type of change employed, which represents the SCI. Even more, developers have derived tools for checking that pull requests align to the specified templates, for instance, the RedHat-powered Tyr.[4]

However, such templates are only used to document changes, and the tools can only enforce that the developer fills in information according to the template. Whether that information is correct and whether it reflects the changes implemented in the pull request cannot be verified using such templates. For instance, a developer may state in the template that their SCI is to fix a certain bug and that the pull request passed all corresponding test cases. But, the developer may have also refactored the system or fixed another bug. So, the information in the template may not fully represent the actual intention underlying the pull request—it is up to the developer or reviewer to ensure that. Also, reviewers and testers have to check that the implemented change does not conflict with the actual SCI. Specifically, we can consider test cases as specifications for controlling the fulfillment of intentions, but whether the test cases properly reflect that intention is also an open problem. We argue that it can be helpful to specify SCIs eagerly in the development to control its fulfillment (i.e., allowing only changes to fix a bug if that SCI is declared), rather then reviewing and testing it lazily after the implementation.

Summary. Reflecting on these two perspectives, we perceive our vision of managing software evolution via declaring SCIs as highly valuable for researchers and practitioners alike. To use a simple sketch based on Figure 1: We can imagine that an organization wants to control that (at least) bug fixes are completely separated from any other changes to reduce interferences caused by, for instance, tangled refactorings. So, the organization derives a specification for bug-fixing or even more fine-grained SCIs (i.e., corrective). The specification may then build on a well-defined test suite (as one example) and declare the conditions for the different types (as far as possible) of SCIs. As a simple and incomplete example, such a declaration could be: A corrective SCI is fulfilled when the corresponding test cases before the change failed (i.e., identified a bug) and then completely pass afterwards. Here, executing the test suite against the system version before and after a change could suffice, and could be implemented as one technique to check that the SCI conforms with its specification. As a consequence, if a change would fix the bug and align to the defined rules, the conformance check would approve that the SCI of the change is fulfilled.

Note that this is a simple and limited example. We are convinced that a single technique is unlikely to ensure the fulfillment of a SCI, which is already the case for our test-based example. Specifically, test cases may also tangle intentions or misrepresent them. Moreover, the exact meaning, differences, and possible specifications for the various types of SCIs are rather vague. As a consequence, we see the need to better explore different types and granularities of SCIs, aiming to understand how to distinguish them. Then, starting with a few concrete SCIs, we can explore how well specifications and techniques can help to clearly identify, specify, and control SCIs in a change. This is clearly a challenging and long-term research vision, but we are convinced that it can lead to immense benefits regarding the management and evolution of software systems as well as their consequent quality. For example, to explore the feasibility and real-world potential of our idea in the short-term, we envision that, instead of controlling SCIs eagerly, we collect and develop concepts for lazily checking SCIs (cf. Figure 1).

[3] https://github.com/ionic-team/ionic-framework/pull/25286
[4] https://github.com/jboss/tyr

3 A VISION OF MANAGING INTENTIONS

At the bottom of Figure 1, we sketch a coarse overview of our vision for improving the management of SCIs. For example, imagine an organization that wants to control the evolution of its software system using branches. Specifically, the developers may be allowed to create a branch, but they have to declare what they intend to implement in that branch (e.g., fixing a bug). Our idea is to design a technique that could then control that only the specified SCI can be executed and integrated back into the main system. This would result in a more understandable version history by avoiding tangled SCIs, limit quality problems, facilitate comprehension of changes and version histories, simplify code transplantation, improve documentation, and simplify automated analyses—among many more benefits. Next, we describe this idea in more detail. Note that an organization or developer community must define the extent to which they employ and control SCIs, but we envision developing techniques and tools that provide a reusable foundation.

First, for each of the SCIs, we have to **derive an intention specification**. This specification shall define the properties and values of each SCI, such as a unique identifier to track every change, what properties have to hold to verify that the SCI is fulfilled, or what types of checks will be executed. We consider the specification to serve as the foundation against which a change will be verified. As a consequence, an intention specification must be associated with a certain type of change (e.g., commit or branch level). Note that the intention specifications can be employed already when a change starts to eagerly control its fulfillment, or the changes can be lazily checked against the specification to assess a change after the fact. For managing the different specifications, we can envision an intention meta model that describes what types of SCIs are allowed, how these are specified, and on what level of abstraction they are executed. Building on our insights from a literature review and own work on such specifications [1, 22], we understand that SCIs can be on various levels of abstraction, such as the high-level intentions of *adaptive*, *corrective*, *enhancive*, and *perfective* that build on the taxonomy of Swanson [26] down to low-level intentions of *adding a feature* or *fixing a bug*. The SCIs that are relevant for an organization, their boundaries, and their granularity can be defined or selected by the organization or researchers.

Second, we aim to implement **conformance checks** that ensure that an implemented change fulfills its specified SCI. Specifically, we envision that several techniques are integrated into an intention library to distinguish and check SCIs based on the defined specifications as well as meta model. For our long-term vision, we envision to incorporate these checks continuously, enabling novel

techniques to check that SCIs are fulfilled throughout the entire system evolution. Depending on which strategy an organization employs (i.e., lazily checking, eagerly controlling), such a library can warn about violations of SCIs or not even allow developers to execute such violations. We argue that various different techniques can and must be used to balance each technique's strengths and weaknesses, for instance, test cases, software verification, or formalized operations on the system. Notably, defining the intention specifications and how to execute corresponding conformance checks lazily or eagerly is the most challenging aspect of our idea.

4 STEPS FOR FUTURE WORK

To advance towards our vision of improving software evolution, we see several open research gaps. In the following, we describe these gaps, sketch some first steps into each direction, and what (intermediate) benefits addressing these gaps can yield.

Understanding Intentions. A primary step to specify intentions is to understand how developers are expressing, using, and reflecting on what types of SCIs. This way, we are able to identify in what context SCIs are relevant to them and for what purposes SCIs are used. As a step towards this direction, we need empirical studies, for instance, on pull-request templates (e.g., whether they are used and enforced, what they exhibit) or developers' cognition (e.g., whether they are thinking about and in terms of SCIs [16]). Before eliciting a complete model of SCIs, we see this as beneficial to understand how developers interact and behave with their software systems and its evolution. Understanding such aspects would greatly help design novel concepts for program comprehension and adapt existing solutions to new foundational insights on developers' cognition.

Intended versus Actual Change. Stakeholders and developers usually have a particular SCI in mind, for instance, adding a new feature (i.e., enhancive). However, this SCI may not be implemented as planned, leading to misbehavior or bugs, due to the actual change. As a result, corrective or perfective changes are needed, aiming to get the actual behavior in line with the intended one. So, in addition to one of such four classes, each change may also be distinguished based on whether it fulfills its SCI. Identifying changes that do not fulfill their SCI can help developers identify bugs faster and avoid the propagation of faulty code. Also, understanding the root causes for a mismatch in intended and actual behavior is key to define specifications for SCIs, since they represent under what circumstances such specifications are violated.

Behavior-Changing versus Behavior-Preserving Intention. While most changes are intended to change system behavior (e.g., fixing a bug, adding new functionality), others are not (e.g., refactoring, ensuring compatibility with hardware). So, some SCIs can also be considered as *behavior preserving*. This distinction can help identify changes relevant for certain software-engineering activities, but most importantly, it is arguably much more challenging to specify behavior-preserving SCIs. For example, refactorings should be behavior preserving, and knowing the corresponding changes can facilitate library adaptations, merging, or cherry-picking by indicating that no behavioral adaptations are required. In contrast, a bug fix should change (i.e., correct) behavior, which is why such changes could be identified with tests, and could help evaluate the quality and completeness of a test suite.

Tangled SCIs. In our ideal scenario, every change (e.g., in the form of a commit) would represent a single, correctly implemented SCI. However, in practice, changes often comprise multiple tangled SCIs, for instance, fixing a bug (i.e., corrective) and refactoring (i.e., perfective) [4, 7]. Such SCI-tangling changes are problematic, since they are harder to comprehend for developers, complicate integration, and challenge analysis tools. For example, developers may want to cherry-pick and propagate a particular intention that is tangled with other intentions. If these SCIs cannot be identified and separated, the developers need to propagate additional, unwanted changes to ensure correct behavior [20]. Achieving the ideal of a one-to-one mapping between intentions and changes would help facilitate support for such use cases. So, we argue that methods and techniques for untangling SCIs (e.g., splitting a commit into multiple) are important.

Declaring SCIs. Building on insights for the previous points, we will be able to define specifications for SCIs, that is, some form of declaration developers can use to ensure and control that only a specific SCI is fulfilled. While there have been some attempts, such a declaration is highly challenging (i.e., defining clear boundaries between specific SCIs), and poses immediate new questions. For instance, we have to define whether missing declarations for some changes, SCIs, or parts of a software system will mean that other declarations become invalid. Moreover, we need to identify what we need to declare on what level of detail in what format (e.g., via annotations or integrated tooling), and how to maintain the declarations themselves. The declarations then serve as the foundation for actually controlling software evolution.

Developing an SCI-Management Framework. Recovering SCIs for existing systems is important to analyze and introduce SCIs in real-world settings. To enable such an analysis, we seek a framework that incorporates different techniques for recovering SCIs. The framework must enable its users to make sense of unique, incomplete, inconclusive, and diverging classifications of SCIs recovered from each technique. Building on the insights gained for the previous directions, we aim to advance towards a management framework that enables developers to eagerly declare and control SCIs for software engineering. So, instead of recovering SCIs when needed, they could be used as the primary notion for managing a system. Ideally, this can help address the problems we highlighted, and thus facilitate developers' tasks.

5 CONCLUSION

In this paper, we reflected on the notion of intentions in software engineering and sketched a vision for using SCIs to manage software evolution. Our long-term vision is to declare and specify SCIs to define what developers are allowed to implement for a certain (set of) changes, for instance, in a pull request. As guidance, we sketched future research directions, which are primarily connected to:

- Empirically analyzing SCIs, including their relevance for developers, their tangling, and how to distinguish them.
- Designing techniques for specifying and checking SCIs.
- Engineering tools for fully and eagerly controlling software evolution based on SCIs.

This is an ambitious research agenda, but already even steps will yield novel insights to help improve research and practice.

REFERENCES

[1] Sofia Ananieva, Sandra Greiner, Jacob Krüger, Lukas Linsbauer, Sten Gruener, Timo Kehrer, Thomas Kuehn, Christoph Seidl, and Ralf Reussner. 2022. Unified Operations for Variability in Space and Time. In *International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM. https://doi.org/10.1145/3510466.3510483

[2] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf Reussner, and Bernhard Westfechtel. 2020. A Conceptual Model for Unifying Variability in Space and Time. In *International Systems and Software Product Line Conference (SPLC)*. ACM. https://doi.org/10.1145/3382025.3414955

[3] Vaibhav Anu, Wenhua Hu, Jeffrey C. Carver, Gursimran S. Walia, and Gary Bradshaw. 2018. Development of a Human Error Taxonomy for Software Requirements: A Systematic Literature Review. *Information and Software Technology* 103 (2018). https://doi.org/10.1016/j.infsof.2018.06.011

[4] Wesley K.G. Assunção, Jacob Krüger, Sébastien Mosser, and Sofiane Selaoui. 2023. How Do Microservices Evolve? An Empirical Analysis of Changes in Open-Source Microservice Repositories. *Journal of Systems and Software* 204 (2023). https://doi.org/10.1016/j.jss.2023.111788

[5] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *ACM Computing Surveys* 30, 2 (1998). https://doi.org/10.1145/280277.280280

[6] Andrea Di Sorbo, Sebastiano Panichella, Corrado A. Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C. Gall. 2015. Development Emails Content Analyzer: Intention Mining in Developer Discussions. In *International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ASE.2015.12

[7] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling Fine-Grained Code Changes. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. https://doi.org/10.1109/saner.2015.7081844

[8] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2018. Communicative Intention in Code Review Questions. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. https://doi.org/10.1109/ICSME.2018.00061

[9] Shinpei Hayashi, Takayuki Omori, Teruyoshi Zenmyo, Katsuhisa Maruyama, and Motoshi Saeki. 2012. Refactoring Edit History of Source Code. In *International Conference on Software Maintenance (ICSM)*. IEEE. https://doi.org/10.1109/icsm.2012.6405336

[10] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The Impact of Tangled Code Changes on Defect Prediction Models. *Empirical Software Engineering* 21, 2 (2016). https://doi.org/10.1007/s10664-015-9376-6

[11] Sebastian Hönel, Morgan Ericsson, Welf Löwe, and Anna Wingkvist. 2020. Using Source Code Density to Improve the Accuracy of Automatic Commit Classification Into Maintenance Activities. *Journal of Systems and Software* 168 (2020). https://doi.org/10.1016/j.jss.2020.110673

[12] Qiao Huang, Xin Xia, David Lo, and Gail C. Murphy. 2018. Automating Intention Mining. *IEEE Transactions on Software Engineering* 46, 10 (2018). https://doi.org/10.1109/TSE.2018.2876340

[13] Irum Inayat, Siti S. Salim, Sabrina Marczak, Maya Daneva, and Shahaboddin Shamshirband. 2015. A Systematic Literature Review on Agile Requirements Engineering Practices and Challenges. *Computers in Human Behavior* 51 (2015). https://doi.org/10.1016/j.chb.2014.10.046

[14] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University.

[15] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering* 5 (1998). https://doi.org/10.1023/a:1018980625587

[16] Jacob Krüger and Regina Hebig. 2020. What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. https://doi.org/10.1109/ICSME46990.2020.00015

[17] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019). https://doi.org/10.1016/j.jss.2019.01.057

[18] Stanislav Levin and Amiram Yehudai. 2019. Visually Exploring Software Maintenance Activities. In *International Working Conference on Software Visualization (VISSOFT)*. IEEE. https://doi.org/10.1109/VISSOFT.2019.00021

[19] Yuejian Li and Nancy J. Wahl. 1999. An Overview of Regression Testing. *ACM SIGSOFT Software Engineering Notes* 24, 1 (1999). https://doi.org/10.1145/308769.308790

[20] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2018. Semantic Slicing of Software Version Histories. *IEEE Transactions on Software Engineering* 44, 2 (2018). https://doi.org/10.1109/tse.2017.2664824

[21] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. 1978. Characteristics of Application Software Maintenance. *Communications of the ACM* 21, 6 (1978). https://doi.org/10.1145/359511.359522

[22] Max Lillack, Ștefan Stănciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wąsowski. 2019. Intention-Based Integration of Software Variants. In *International Conference on Software Engineering (ICSE)*. IEEE. https://doi.org/10.1109/icse.2019.00090

[23] Katsuhisa Maruyama, Eijiro Kitsu, Takayuki Omori, and Shinpei Hayashi. 2012. Slicing and Replaying Code Change History. In *International Conference on Automated Software Engineering (ASE)*. ACM. https://doi.org/10.1145/2351676.2351713

[24] Damir Nešić, Jacob Krüger, Ștefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. https://doi.org/10.1145/3338906.3338974

[25] Sarocha Sothornprapakorn, Shinpei Hayashi, and Motoshi Saeki. 2018. Visualizing a Tangled Change for Supporting Its Decomposition and Commit Construction. In *Annual Computer Software and Applications Conference (COMPSAC)*. IEEE. https://doi.org/10.1109/compsac.2018.00018

[26] E. Burton Swanson. 1976. The Dimensions of Maintenance. In *International Conference on Software Engineering (ICSE)*. IEEE.

[27] Song Wang, Chetan Bansal, and Nachiappan Nagappan. 2020. Large-Scale Intent Analysis for Identifying Large-Review-Effort Code Changes. *Information and Software Technology* (2020). https://doi.org/10.1016/j.infsof.2020.106408

[28] Peter Weißgerber and Stephan Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ASE.2006.41

[29] Jooyong Yi, Dawei Qi, Shin H. Tan, and Abhik Roychoudhury. 2015. Software Change Contracts. *ACM Transactions on Software Engineering and Methodology* 24, 3 (2015). https://doi.org/10.1145/2729973

[30] Shurui Zhou, Ștefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *International Conference on Software Engineering (ICSE)*. ACM. https://doi.org/10.1145/3180155.3180205