End-to-End Rationale Reconstruction

Mouna Dhauadi
Université de Montréal
Montréal, Canada
mouna.dhauadi@umontreal.ca

Bentley James Oakes
Université de Montréal
Montréal, Canada
bentley.oakes@umontreal.ca

Michalis Famelis
Université de Montréal
Montréal, Canada
famelis@iro.umontreal.ca

ABSTRACT

The logic behind design decisions, called design rationale, is very valuable. In the past, researchers have tried to automatically extract and exploit this information, but prior techniques are only applicable to specific contexts and there is insufficient progress on an end-to-end rationale information extraction pipeline. Here we outline a path towards such a pipeline that leverages several Machine Learning (ML) and Natural Language Processing (NLP) techniques. Our proposed context-independent approach, called Kantara, produces a knowledge graph representation of decisions and of their rationales, which considers their historical evolution and traceability. We also propose inconsistency checking mechanisms to ensure the correctness of the extracted information and the coherence of the development process. We conducted a preliminary evaluation of our proposed approach on a small example sourced from the Linux Kernel, which shows promising results.

KEYWORDS

Software rationale, Natural Language Processing

ACM Reference Format:
Mouna Dhauadi, Bentley James Oakes, and Michalis Famelis. 2018. End-to-End Rationale Reconstruction. In Proceedings of The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE ’22). ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software development is a process that requires making several decisions. The logic behind these decisions, or the rationale, represents beneficial information as it can serve as a valuable repository of past experiences that could be used to solve similar problems, and to learn best practices [6]. Thus, many researchers [4, 7, 12, 22] have tried to capture and represent this logic in order to exploit it. In fact, making rationale available in a well-structured and exploitable format will better support decision-making [15], and would foster collaboration, coordination, and project management [6]. But this line of research has not seen wide adoption because of the large effort required to capture the rationale manually (i.e. the capture problem), and of its inability to provide immediate results (i.e. the cost-benefit problem) [5]. The solution is thus to extract and structure rationale automatically, in a non-intrusive way and with no extra-effort, solving both the capture and the cost-benefit problems. However, this is a difficult task, as rationale appears in an implicit, poorly organized way and scattered across different artifacts [3].

Recently, several researchers tried to extract rationale automatically with Natural Language Processing (NLP) techniques [11, 13, 28, 29]. However, their techniques are only applicable to specific contexts and, to the best of our knowledge, there are no techniques allowing their adaptation to other contexts, which hinders their adoption in practice. Additionally, there is no prior work that has proposed a complete end-to-end pipeline that encompasses the extraction, the structuring and the management of rationale, which limits their usefulness. Finally, to the best of our knowledge, none of the previous work have considered capturing the temporal evolution of decisions and of their rationales.

Our vision is therefore to assist decision makers, whether they are engineers, developers, or others, in any context, with an automated design rationale extraction and management system. Specifically, we propose to adopt the community vision of an automated on-demand developer documentation (OD3) system [26]. An OD3 system has two components: an information inference component that finds, extracts, links and structures the useful information from different places, and a response generation component that produces high-quality responses to the users queries.

This paper presents our initial ideas about how to build the information inference component OD3 system, that we call “Kantara”. The Kantara approach aims to automatically extract the decisions and their rationales, from different textual sources and artifacts, then to structure them as a graph-based representation. We thus assume that written artifacts produced by stakeholders during their communications, or while executing their tasks, contain rationale information. We propose to leverage advances in NLP and Machine Learning (ML) to create a pipeline for rationale reconstruction [34]. The resulting knowledge graph can then be used to a) understand the historical development process of the decisions, b) get a shared understanding of the logic behind the made decisions, across both time and space, and c) ensure the consistency and the coherence of the development process by looking for conflicts and by detecting whether a proposed decision interferes with previously agreed upon decisions – or merely repeats what was tried in the past. In the future, we envision the response generation component of our envisioned OD3 system to be a bot that leverages the knowledge graph created by the Kantara approach.

Contributions and Structure. This paper is structured as follows: 1) In Section 2 we give a running example sourced from the Linux Kernel. 2) The associated Rationale and Decision Graph for this example is presented in Section 3.1. This is a knowledge graph-based representation of design decisions and rationales that takes their historical evolution into consideration and that provides traceability to artifacts. 3) In Sections 3.2 and 3.3 we present and discuss the
Kantara approach: an end-to-end generic NLP-based information extraction pipeline to extract and structure design rationale in a knowledge graph. 4) In Section 4 we propose NLP-based mechanisms to detect inconsistencies in the knowledge graph. We survey related work in Section 5 and conclude in Section 6.

2 MOTIVATING EXAMPLE

We have selected the Out-Of-Memory Killer (OOM-Killer) component in the Linux kernel and its related commits/discussions as the running example and the object of our preliminary evaluation. The OOM-Killer is a last-ditch effort by the kernel to free up memory when tasks have requested all available memory. That is, when there is not enough system memory then future requests for memory will fail and the system may crash. To prevent this, the OOM-Killer will a) select a task to kill, then b) signal (or force) that task (the OOM victim) to release its memory and exit. This has been a controversial component from its first suggestion in 1998, as some developers do not agree with the strategy that a task is killed without the user’s intervention. Nevertheless, this component can be useful to return a system to stability.

We chose to study the OOM-Killer component for three reasons. First, it has a well-defined scope which is easy to comprehend yet semantically-rich. Second, it has a number of heuristics and interesting decisions, such as the assigning of a badness score when deciding which task to kill and the mechanisms for reclaiming memory from the OOM victim. Finally, the component is small in terms of commits and code. For the presentation and the initial evaluation of our approach, we manually selected interesting commits from the Git history of OOM-Killer as our example decisions. Specifically, we selected commits on the topic of reclaiming the used memory from the OOM victim.

In Linux kernel development, patches must contain a description of the motivation/rationale behind them, and of the impact they will have on the kernel. Traceability information is provided in multiple ways, such as a) patches are encouraged to add explicit links to the Linux Kernel Mailing List (LKML) discussions in their descriptions, and b) patches must have a summary phrase (such as "oom: give the dying task a higher priority") when deciding which task to kill and the mechanisms for reclaiming memory from the OOM victim. Finally, the component is small in terms of commits and code. For the presentation and the initial evaluation of our approach, we manually selected interesting commits from the Git history of OOM-Killer as our example decisions. Specifically, we selected commits on the topic of reclaiming the used memory from the OOM victim.

In Linux kernel development, patches must contain a description of the motivation/rationale behind them, and of the impact they will have on the kernel. Traceability information is provided in multiple ways, such as a) patches are encouraged to add explicit links to the Linux Kernel Mailing List (LKML) discussions in their descriptions, and b) patches must have a summary phrase (such as "oom: give the dying task a higher priority"). Thus, the Linux kernel and the LKML form a comprehensive repository of decision/rationale information for each commit.

Table 1 presents the details for the five relevant OOM-Killer commits, including the summary phrase, the commit date, and our description of the commit. The first two commits in Table 1 increase the priority of dying tasks so they can quickly release their memory. The third commit reverts the first two, as conflicts were found between the priority escalation and another mechanism in the kernel. The fourth commit is an approach where a dedicated kernel thread performs the memory reclamation, avoiding the priority escalation issues. The last commit implements a new system call such that a user process can ask the kernel to reclaim the memory of a dying task. This is useful for applications such as the Android process manager for fine-grained control over which task should be killed and have its memory reclaimed.

1 https://marc.info/?l=linux-kernel&m=90222140830612&w=2
2 The primary file/mem/oom_k111.c is 1205 lines of code, with 410 commits since the start of the Linux kernel Git repository in 2005.

3 END-TO-END RATIONALE RECOVERY

In this section, we present and discuss the proposed Kantara approach to automatically create a Rationale and Decision Graph from the Linux-sourced example commits. However, we note that our approach is not Linux-specific.

3.1 Rationale and Decision Graph

A Rationale and Decision Graph represents design decisions and the rationales behind them as nodes, and the relationships between those entities as edges. For example, the decisions and their rationale for the commits in Table 1 are represented by the graph in Fig. 1.

The main components of our knowledge graph are the triples (decision, "rationale", rationale). For instance, the decision D1: give the dying task a higher priority has the associated rationale: so that it can exit() soon, freeing memory. Since rationale is very subjective, engineers formulate it differently [27]. Thus, our graph may be very diverse in the content of its rationale nodes.

The graph also introduces different relationship types between decisions: 1) a history relationship that means that a decision is an evolution of another (D3/D1), 2) a similar relationship that means that the decisions are semantically similar (D2/D1), and 3) a contradicts relationship that means that the decisions might be in conflict (D3/D5). The contradicts relationship can also be combined with the history one, in the sense that the new decision contradicts its previous version (D3/D1, D3/D2). This captures the fact that the third decision D3 (taken in the third commit) reverts the first two D1/D2 (taken in the first two commits) as explained in Section 2.

The goal of the Rationale and Decision Graph representation is to help stakeholders develop a shared understanding of how decisions are interconnected, in order to counter design evaporation [25]. Our graph does not only capture the reason behind a decision, but also contextualizes it in time, which helps recover the holistic view of the development process and thus provides stakeholders with the big picture and helps avoid wasted effort. This representation is very simple and generic and can easily be translated to more specific rationale metamodels, such as IBIS [7], which makes it easily adaptable for different contexts.

3.2 Approach Overview

Here we describe and discuss the Kantara approach to automatically create such a Rationale and Decision Graph using ML and NLP. Given textual artifacts, our approach aims to extract the design decisions taken, their historical evolution, their corresponding rationales, and how these decisions are interconnected. We envision Kantara to be context-independent (as we will explain in detail later) for broader applicability. The Kantara information extraction pipeline has two stages described below: 1) extracting decision-rationale relationships, and 2) extracting decision-decision relationships.
### 3.2.1 Extracting Decision-Rationale Relationships

**Decisions extraction.** We plan to identify decision sentences by classification. For the example in Fig. 1, we redeveloped the decisions classifier from [11] and used it to label the decision-containing sentences in the different commits in Table 1. The classifier correctly labeled D3, D4 and D5 as relevant sentences for decisions. However, it incorrectly labeled D1 and D2, as non-relevant decision sentences. The source element contains the link to the mail from which the decision has been extracted.

Since our goal is to offer a context-independent extraction system, our next steps include building an improved and generic classifier for decisions. Previous work have proposed context-specific design decisions classifiers [1, 11, 14]. However, recent research [19] showed that these systems do not offer conclusion stability (i.e. the learners are unable to transfer to other contexts). To overcome this, Mahadi et al. [18] created software engineering word embeddings, and found that they improved the classifiers performance. Thus, in our next steps, we plan to reuse their available word embeddings. Furthermore, the performance of the decisions classifier depends heavily on its training data. To mitigate this potential limitation, we could label more data, use automatic labeling approaches or enhance the classifier’s performance by combining it with a heuristic-based approach.

**Rationale extraction.** Once we have the decision-containing sentence, we try to extract the reason behind the decision at the sentence level using Semantic Role Labeling (SRL) [23]. This NLP technique is able to extract the cause or the purpose of a certain event, based on the grammatical construction of a sentence, thus it is context-independent. In Fig. 1, this technique has successfully extracted the rationales behind D1 and D2. However, since it only works at the sentence level, it would fail if rationale is scattered over multiple sentences (as it would capture incomplete rationale), or if the rationale is contained in a different sentence (e.g. D3). We try to mitigate this limitation by adding the source elements so developers can double check the extracted rationale from the original source. In the future, we plan to investigate approaches for paragraph-level rationale extraction.

For D4 and D5, our observations have led us to consider investigating using the manner information as potential useful information for rationale extraction. For instance, in the case of D5 the rationale is found in the sentence: ‘This way the memory is freed in a more controllable way with CPU affinity and priority of the caller’, and is phrased as the “how?” (i.e. “in a more controllable ...”). The SRL technique labeled that part of the sentence as manner information. In the future, we intend to investigate further this particular way of phrasing rationale.

Finally, we note that our approach only recovers recorded rationale (albeit from sources that were not originally intended to be rationale documents), and not the whole actual rationale as that is often never recorded, even unintentionally. This limits its usefulness, and could even have negative effects if wrong decisions were taken based on partial rationale [10].

### 3.2.2 Extracting Decision-Decision Relationships

Once the decision/rationale nodes are created, our pipeline tries to label the relationships between decisions in a context-independent way. These relationships will be used later as basis for our consistency checking mechanisms.

**Relatedness relationship (Topic).** Topics are introduced as means to capture relatedness between decisions. Our initial idea is training a ML classifier on a related/non-related sentences dataset. We obtained this dataset after considering indirect, direct and duplicate
links as related, and isolated links as non-related, for entries where at least one of the sentences has the design tag from the StackOver-
flow (SO) relatedness dataset [30]. Since this dataset contains 300K entries spanning across all the SO community, we argue that our result-
ing classifier is context-independent. The classifier performed poorly on our example because of imbalanced classes, thus, we plan to investi-
gate the use of techniques such as SMOTE [9]. We plan to investi-
gate using keyphrase generation systems [21] to automatically gen-
erating the topic title (such as "reclaiming the used memory from the OOM victim").

Similar relationship. There are many off-the-shelf approaches for
detecting semantic similarity. The edge between D1/D2 was added af-
after detecting a similarity of 0.86 between them using SpaCy [32].

History relationship. For two related decisions (i.e. ones with the
same topic), we check whether one is a previous version of the other.
We plan on detecting this historical information using heuristics (e.g.
the decisions concern the same files, they were done by the same
people, one of them refers to the other, timestamps, etc). For in-
stance, D3 happened after D1, the mail containing D3 referred to
D1 and the author of D1 "acked" (‘acknowledged’) D3. Identifying
the right heuristics requires extensive empirical evaluation.

Contradicts relationship. We propose to capture this relationship
using the contradiction label of the Natural Language Inference (NLI)
task [17]. We fine-tuned the BERT model [8] on SNLI corpus [2].
Although the resulting model worked well in other cases (e.g. it
detected a contradiction of 0.94 between "We need to implement this
feature to be able to satisfy the requirements" and "There is no need
to do anymore changes"), it performed poorly on our example (i.e.
D3/D1 and D3/D2) because the contradiction is not obvious. We
plan to mitigate this by introducing heuristics (e.g. if the commit of
the decision is a revert of another commit, or to exploit the existence
of keywords such as revert or disable).

3.3 Preliminary Evaluation Summary
Table 2 summarizes the evaluation results of the different steps of the
Kantara pipeline as discussed throughout Section 3.2. These
promising results prompt us to carry on our work.

Table 2: Evaluation Summary

| Step in the Pipeline                | Preliminary Evaluation Results |
|------------------------------------|--------------------------------|
| Decisions extraction               | partially successful          |
| Rationale extraction               | partially successful          |
| Relatedness relationship (Topic)   | unsuccessful                 |
| Similar relationship               | successful                   |
| History relationship               | not applicable (future work)  |
| Contradicts relationship           | partially successful          |

4 GRAPH CONSISTENCY CHECKING

The consistency checking mechanisms aims to a) verify the consist-
tency of the extracted graph and of the development process, and
b) look for potential conflicts when a new decision is proposed.

To satisfy the first goal, we propose a mechanism for detecting
inconsistencies between the rationales of the decisions, and the
decisions themselves. For instance, decisions D1 and D2 have a similar
relationship. Our mechanism should thus verify the relationship
between their rationales. We computed a semantic similarity of
0.92, which is a strong indicator of consistency. Moreover, this high
value suggests incorporating duplicate detection in our future work,
as it could be useful to detect the case when the same rationale has
resulted in different decisions. If our mechanisms have found a con-
tradiction between the rationales, then this would have indicated
a problem in the graph construction, or even a deeper problem in
the reasoning.

To satisfy the second goal, we propose a verification mechanism
that should happen at the introduction of every new decision. Let’s
consider Suren, a developer of the Memory Management (mm)
sub-system in the Linux kernel. In 2021, Suren proposes to introduce a
process (commit 5 in Table 1) to reclaim the memory of a dying task.
At the introduction of D5, our mechanism should detect a similar
relationship between D5 and D1 (we computed a semantic similarity
of 0.87), and considering the graph structure (i.e. D2 is similar to D1,
and D3 contradicts D1 and D2), this should inform the developer
that a similar approach has been tried out more than a decade ago
regarding the Out-Of-Memory (oom) component (D1/D2), that it
has been abandoned after a short while because of a corner case
identified in 2011 (D3), and that their proposed decision (D5) may
cause conflict with this previously made decision (i.e. D3). Thus, the
verification mechanism would help stakeholders avoid collisions
and make the right design choices, which is particularly important
in the case of long-term, geographically-distributed or large-scale
projects. Consequently, this will result in a better design quality and
would help solve the design erosion problem [31]. In this particular
case, we should note that Suren was aware of the earlier approaches
because he discussed the commit logs with the maintainers. In fact,
the proposed approach in D5 introduces a system call, which avoids
the priority escalation problem of D1/D2.

5 RELATED WORK

In [15], the authors focus on the design of an algorithm to extract
and structure design rationale from design documents based on the
ISAL model [16]. They use artifact information identification, issue
summarization and solution–reason pair discovery, and leverage
semantic graph-based algorithms. Unlike our work, they do not
consider the historical evolution of decisions. In [33], the authors
attempted to recover design information from developer discus-
sions. Similar to our work, they try to extract design information;
however they do not attempt to structure it. In [24], the authors
tried to generate concise descriptions of the reasons resulting in
a code change. To do so, they leveraged advanced summarization
techniques. Thus, they mainly differ from us in the used approach,
and they only consider rationale in the context of code change.
In [20], the authors propose ASGAR, a semantic grammar-based
approach to automatically capture and structure design rationale.
This work differs from ours by the use of semantic grammars.

6 CONCLUSION

In this paper, we present our initial ideas for an automated end-to-
end graph-based rationale reconstruction approach that we call Kan-
tara. We use a small example of commits for the Out-Of-Memory
Killer of the Linux kernel to illustrate and evaluate the two main
steps of 1) extracting decision-rationale relationships, and 2) extracting decision-decision relationships. We also use the example to explain our proposed mechanisms to ensure the consistency of the graph. Preliminary evaluation results indicate promise, and we describe our plans concerning the observed limitations.

ACKNOWLEDGMENTS
This work was partially funded by the Tunisian Ministry of Higher Education and Scientific Research.
REFERENCES

[1] Manoj Bhat, Klym Shumaiev, Andreas Bieders, Uwe Hohenstein, and Florian Matthes. 2017. Automatic extraction of design decisions from issue management systems: a machine learning based approach. In European Conference on Software Architecture. Springer, 138–154.
[2] Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. 2015. The SNLI corpus. (2015).
[3] João Brunet, Gail C. Murphy, Ricardo Terra, Jorge Figueiredo, and Dalton Serey. 2014. Do developers discuss design? In Proceedings of the 11th Working Conference on Mining Software Repositories. 340–343.
[4] Janet E Burge. 2005. Software engineering using design RATionale. Ph. D. Dissertation. Worcester Polytechnic Institute.
[5] Janet E Burge. 2008. Design rationale: Researching under uncertainty. AI EDAM 22, 4 (2008), 311–324.
[6] Janet E Burge, John M Carroll, Raymond McCall, and Ivan Mistrik. 2008. Rationale-based software engineering. Springer.
[7] Jeff Conklin and Michael I Berman. 1988. gBIS: A hypertext tool for exploratory policy discussion. ACM Transactions on Information Systems (TOIS) 6, 4 (1988), 303–331.
[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).
[9] Alberto Fernández, Salvador García, Francisco Herrera, and Nitesh V Chawla. 2018. SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary. Journal of artificial intelligence research 61 (2018), 863–905.
[10] John Horner and Michael E Atwood. 2006. Effective design rationale: understanding the barriers. In Rationale management in software engineering. Springer, 73–90.
[11] Anja Kleebaum, Barbara Paech, Jan Ole Johansen, and Bernd Bruegge. 2021. Continuous Rationale Identification in Issue Tracking and Version Control Systems. (2021).
[12] Jintae Lee. 1991. Extending the Potts and Bruns model for recording design rationale. In Proceedings-13th International Conference on Software Engineering. IEEE Computer Society. 114–115.
[13] Miriam Lester, Miguel Guerrero, and Janet Burge. 2020. Using evolutionary algorithms to select text features for mining design rationale. AI EDAM 34, 2 (2020), 132–146.
[14] Xueying Li, Peng Liang, and Zengyang Li. 2020. Automatic identification of decision rationales from the Hibernate developer mailing list. In Proceedings of the Evaluation and Assessment in Software Engineering. 51–60.
[15] Yan Liang, Ying Liu, Chun Kit Kwong, and Wing Bun Lee. 2012. Learning the "Whys": Discovering design rationale using text mining—An algorithm perspective. Computer-Aided Design 44, 10 (2012), 916–930.
[16] Ying Liu, Yan Liang, Chun Kit Kwong, and Wing Bun Lee. 2010. A new design rationale representation model for rationale mining. Journal of Computing and Information Science in Engineering 10, 3 (2010).
[17] Bill MacCartney. 2009. Natural language inference. Stanford University.
[18] Alvi Mahadi, Neil A Ernst, and Karan Tongay. 2022. Conclusion stability for natural language based mining of design discussions. Empirical Software Engineering 27, 1 (2022), 1–42.
[19] Alvi Mahadi, Karan Tongay, and Neil A Ernst. 2020. Cross-dataset design discussion mining. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 149–160.
[20] Raymond McCall. 2018. Using argumentative, semantic grammar for capture of design rationale. In International Conference on-Design Computing and Cognition. Springer, 519–535.
[21] Rui Meng, Sanqiang Zhao, Shuguang Han, Daqing He, Peter Brusilovsky, and Yu Chi. 2017. Deep keyphrase generation. arXiv preprint arXiv:1704.06879 (2017).
[22] Douglas Noble and Horst WJ Rittel. 1988. Issue-based information systems for design. (1988).
[23] Martha Palmer, Daniel Gildea, and Nianwen Xue. 2010. Semantic role labeling. Synthesis Lectures on Human Language Technologies 3, 1 (2010), 1–103.
[24] Sarah Rastkar and Gail C Murphy. 2013. Why did this code change? In 2013 35th International Conference on Software Engineering (ICSE). IEEE, 1193–1196.
[25] Martin P Robillard. 2016. Sustainable software design. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 920–923.
[26] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Ned Ernst, Marco Aurélio Gerossa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, et al. 2017. On-demand developer documentation. In 2017 IEEE International conference on software maintenance and evolution (ICSME). IEEE, 479–483.
[27] Martin P Robillard and Nenad Medvidovic. 2016. Disseminating Architectural Knowledge on Open-Source Projects: A Case Study of the Book “Architecture of Open-Source Applications”. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 476–487.
[28] Benjamin Rogers, James Gung, Yechen Qiao, and Janet E Burge. 2012. Exploring techniques for rationale extraction from existing documents. In 2012 34th international conference on software engineering (ICSE). IEEE, 1313–1316.
[29] Pankajshwara Nand Sharma, Bastin Tony Roy Savarimuthu, and Nigel Stanger. 2021. Extracting Rationale for Open Source Software Development Decisions—A Study of Python Email Archives. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 1008–1019.
[30] Amirezea Shirani, Bowen Xu, David Lo, Thamar Soboliro, and Amin Alipour. 2019. Question relatedness on stack overflow: The task, dataset, and corpus-inspired models. arXiv preprint arXiv:1905.01966 (2019).
[31] Jilles Van Gurp and Jan Bosch. 2002. Design erosion: problems and causes. Journal of systems and software 61, 2 (2002), 105–119.
[32] Yuli Vasiliev. 2020. Natural Language Processing with Python and SpaCy: A Practical Introduction. No Starch Press.
[33] Giovanni Visciani, Michalis Famelis, Xin Xia, Calahan Janik-Jones, and Gail C Murphy. 2019. Locating latent design information in developer discussions: A study on pull requests. IEEE Transactions on Software Engineering 45, 7 (2019), 1402–1415.
[34] Gaofeng Yue, Jihong Liu, and Yongzhu Hou. 2018. Design Rationale Knowledge Management: A Survey. In International Conference on Cooperative Design, Visualization and Engineering. Springer, 245–253.