2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) | 978-1-6654-9598-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-Companion55297.2022.9793824

CIDER: Concept-based Interactive Design Recovery

Hongzhou Fang
Drexel University
Philadelphia, PA, USA
hf92@drexel.edu

Yuanfang Cai
Drexel University
Philadelphia, PA, USA
yc349@drexel.edu

Rick Kazman
University of Hawaii
Honolulu, HI, USA
kazman@hawaii.edu

Jason Lefever
Drexel University
Philadelphia, PA, USA
jtl86@drexel.edu

ABSTRACT

In this paper, we introduce CIDER, a *Concept-based Interactive DEsign Recovery* tool that recovers a software design in the form of hierarchically organized concepts. In addition to facilitating design comprehension, it also enables designers to assess design quality and identify design problems. It integrates multiple clustering algorithms to reduce the complexity of the recovered design structure, leverages information retrieval techniques to name each cluster using the most relevant topic terms to ease design comprehension, and identifies and labels highly-coupled file clusters to reveal possible design problems. It enables interactive selection of concepts of interest and recovers partial design structures accordingly. The user can also interactively change the levels of recovered hierarchical structure to visualize the design at different granularities.

ACM Reference Format:
Hongzhou Fang, Yuanfang Cai, Rick Kazman, and Jason Lefever. 2022. CIDER: Concept-based Interactive Design Recovery. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516861

1 INTRODUCTION

It is challenging to learn the structure of almost any complex software systems, with many dependencies and invariably out-of-date (or no) documentation. Most commercial tools, such as Structure 101[1] and Understand[2], visualize source code based on package structures. Several architecture recovery methods have been created, including Bunch [8], Algorithm for Comprehension-Driven Clustering (ACDC) [15], scaLable InforMation BOttleneck (LIMBO) [1], Weighted Combined Algorithm (WCA) [7], and Architecture Recovery using Concerns (ARC) [5]. These techniques split source files into mutually exclusive clusters, based on a guiding principle such as coupling-and-cohesion [8] or patterns [15].

The problem is that these approaches are not designed to directly support a developer's most pressing maintenance tasks: implementing features and fixing bugs. Feature location [3, 4, 18] has also been widely studied, but these techniques have not been applied to inform the structure of a design. Moreover, the information recovered by these techniques provides little insight into underlying design quality, such as the existence of highly coupled file groups, unexpected coupling among features, or whether the system is appropriately structured to ease the addition, modification, and debugging of features.

To address these problems, we have created CIDER, *Concept-based, Interactive, DEsign Recovery*. CIDER integrates multiple clustering algorithms to reduce the complexity of the recovered structure, leverages information retrieval methods to label each cluster, and extracts highly-coupled file groups, e.g., cliques [10], as part of the recovery process. We will demonstrate our tool's capabilities 1) to recover a high-level design structure in the form of a *concept* hierarchy that can be used to understand a system's functions, features, and their relations, 2) to reveal design problems, and 3) to support maintenance by recovering a portion of a design relevant to a feature or concern that is being modified.

We have conducted exploratory case studies using two open-source projects, fEMR[3] and Depends[4], including surveying 10 of their developers and architects. Most interestingly, all of the developers and architects agreed that the concept hierarchy recovered by CIDER is meaningful and useful to them and helpful for future maintenance tasks, such as assessing the impact of a change or understanding how features are implemented. Both projects have since been refactored based on the findings from CIDER, providing early evidence of the benefit of this novel architecture recovery approach. The project data used in this paper, including the examples elaborated in Section 3, can be found at: http://149.28.157.117/.

2 APPROACH

https://www.overleaf.com/project/6198201904e33064530fae7e Figure 1 depicts the major components of our approach: *source code processing*, *function clustering*, *concept labeling*, and *interactive recovery*. The *source code processing* component transforms the source code into dependencies among entities. The *function clustering* component first clusters files into *function groups*, defined as the set of files used to implement a function or a feature, and applies

[1] https://structure101.com/
[2] https://scitools.com/

[3] https://teamfemr.org/
[4] https://github.com/multilang-depends/depends/releases/

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-6654-9598-1/22/05…$15.00
https://doi.org/10.1145/3510454.3516861

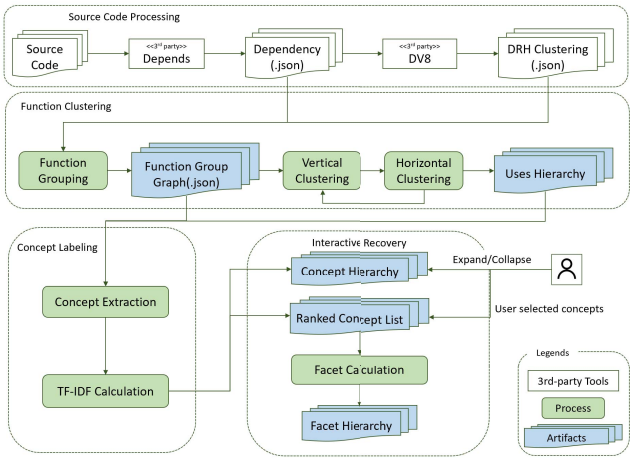


Figure 1: Approach Overview

multiple clustering algorithms to form high-level clusters so that the recovered design structure directly reflects how functions in a system *use* each other [12]. This component also detects the existence of cliques in the meantime. The *concept labeling* component labels each function group and cluster using representative concepts calculated via information retrieval algorithms, transforming the recovered design as a *concept hierarchy*. It also explicitly marks and labels identified cliques. The *interactive recovery* component enables partial design recovery based on user-selected concepts, as well as interactive expansion and contraction of the design structure to allow the user to choose the level of abstraction that best suits their needs. Our rationale for this interactivity is that each feature, pattern, or concern may have its own design space [17] and that only the user knows what to explore.

2.1 Source Code Processing

We use two 3rd-party tools to pre-process the source code. We first use Depends to extract static dependencies among files. Depends saves the dependency information in a dependency JSON file. Depends currently only processes programs written in a single language. We need to create additional preprocessors or introduce other tools to extract dependencies from projects written in multiple programming languages. Using this file as input, we then use DV8 [11] to generate a design rule hierarchy (DRH) clustering [16] and we save the clustering information in a JSON file.

The DRH clustering algorithm in DV8 arranges files into a directed acyclic graph (DAG) with multiple layers, each containing independent modules. For modules containing multiple files, the DRH clustering recursively breaks them into sub-modules. We use the two JSON files—one for dependency information and the other for clustering information—as the inputs to the *function clustering* component to calculate uses and facet hierarchies.

2.2 Function Clustering

To obtain a high-level design, we first apply function grouping to the input DRH clustering file. Unlike existing architecture recovery approaches that cluster individual files [6, 14], CIDER uses *function groups* as the basic unit of clustering, so that we can model software design as a collection of functions or features as a DAG We define a *function group* as a set of files implementing a function. The function grouping component identifies minimal function groups with cohesive functions from the DRH. The hierarchical structure of DRH guarantees that the result will be a DAG.

This component will also detect all the cliques in the system.

To further form a higher-level structure, we aggregate function groups using both *vertical clustering*, and *horizontal clustering*. Both clustering algorithms shrink the DAG with different focuses. Similar to Bunch [8], vertical clustering merges function groups based on their coupling and cohesion relations. It will merge cohesive function group pairs with minimal dependency loss. Horizontal clustering leverages the package naming structure of entry files in a function group. It merges clusters in the same layer of the graph that belong to the same package. To further reduce complexity and enable hierarchical exploration, our last step is to reduce the tree height with minimum edge loss in each iteration and stops when the DAG height reaches 4, a threshold that can be adjusted.

2.3 Concept Labeling

To make sure that the recovered design DAG reflects the most relevant concepts of the system, for each function group and each cluster in the concept hierarchy, we first extract the most relevant concepts by applying the tf-idf [13] algorithm to the file names and the source code. We choose tf-idf since it is the most straightforward algorithms for ranking but we are also experimenting with other algorithms such as Latent Dirichlet Allocation (LDA) [2]. After that, we label each node using the top 3 most relevant concepts by ranking the tf-idf scores. In addition, if the Function Clustering component has identified any cliques, this component explicitly marks them, and lists them on the side panel of the GUI, so that the user can explore each clique in more details.

2.4 Interactive Recovery

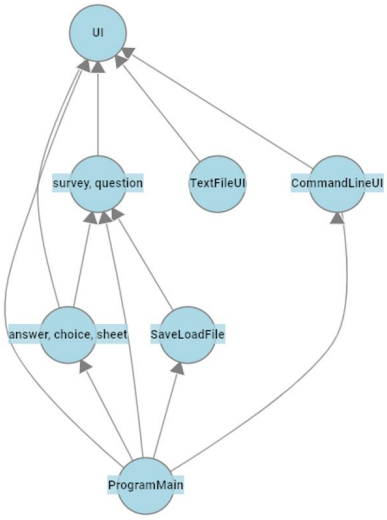
Based on the output of the previous two components, the following information is presented to the user to enable design exploration through our interactive website:

(1) A fully collapsed concept hierarchy that can be expanded or collapsed using the +/- buttons. (2) A list of all concepts extracted from all files in the "*Facet Hierarchy*" tab (Figure 3b). The user can choose one or more of them, and CIDER will present a partial concept hierarchy—that is, a facet hierarchy— accordingly (Figure 3b and Figure 4). (3) the list of function groups (Figure 3a) for the user to explore detailed cohesive functions and how they interact.

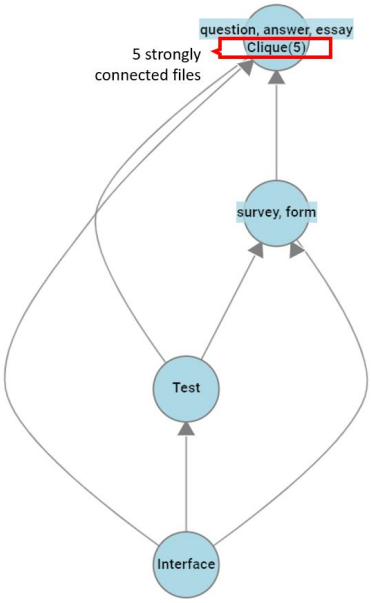
3 CIDER'S FEATURES

In this section, we use a simple system to illustrate the key features of CIDER. We will also illustrate how CIDER can reveal design quality issues and help with maintenance tasks.

This simple system allows a user to create or fill out a questionnaire, which could be a survey or a test. The system supports three types of questions—multiple-choice, matching, and essay—with the expectation that additional types of questions will be added. The user interface can be console-based or file-based, and is expected to be extensible to support additional types of user input, such as



(a) Design-1



(b) Design-2

Figure 2: Concept Hierarchies for Two Different Designs

a graphical user interface. There could of course be many designs to achieve these requirements. We will use two different implementations of the system—Design-1[5] and Design-2[6]—to illustrate CIDER's capabilities.

3.1 Concept Hierarchy

Figure 2 depicts the recovered design structures from the two implementations, in the form of *concept hierarchies*. Each circle represents a cluster aggregated from one or more *function groups*, and each edge represents a *uses* relation between two clusters. For example, the $Interface$ cluster *uses* the $Test$ cluster in Figure 2b, meaning that the files in *test* are a subset of *interface*. The *test* cluster in turn uses the cluster labeled with "$question, answer, essay, clique(5)$". The label of a node reflects (1) the most relevant topics calculated from the underlying source files, (2) strongly connected file groups, if they exist. If a cluster is labeled with $clique(\#)$, it means that there are a number of files in this cluster forming a strongly connected graph. In this case, there are 5 files in Design-2 that form a strongly connected graph. The recovered concept hierarchies in Figure 2 reveal the fundamental differences between these two designs and their quality differences, as we will show. For example, the *interface* cluster in Design-2 has one function group, fg_07, containing $Interface.java$, the entry file of the system that directly or indirectly uses all the other files.

The *Function Groups* tab (Figure 3a) in CIDER's UI lists all the function groups calculated from the source code, and their top-ranked concepts. For example, the fg_05 in Figure 3a is labeled with $answer, essay, written$, suggesting that the topic of this function is processing written questions and their answers. Here a user can select multiple function groups and observe how they interact.

Next, the *Facet Hierarchy* tab (Figure 3b) lists all the concepts extracted from the source files. For example, as shown in Figure 3b the concept of *answer* is related to multiple function groups, and appears to be a crosscutting concern. The user can select one of more concepts listed and CIDER will recover a partial design structure accordingly.

3.2 Interactive Design Structure Exploration

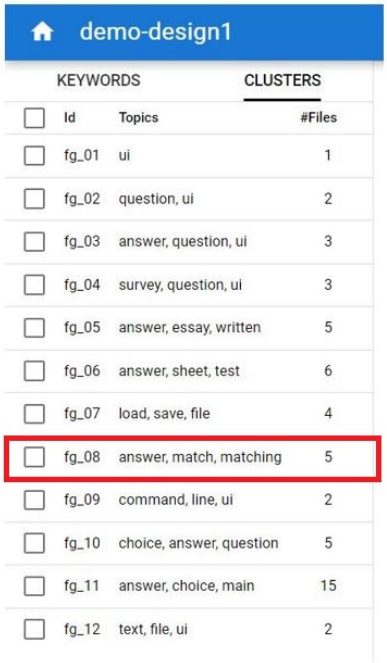
For a complex system, the number of extracted concepts could be very large. The better modularized the system is, the better concerns are separated, and the more concepts can be independently extracted [9]. From a developer's perspective, understanding the design structures related to their task at hand is more useful than visualizing the entire design.

Inspired by the work of Xiao et al.[17], which proposed that each feature and each pattern can have its own design space, and the fact that only the system stakeholders will know what the important features are, or which cross-cutting facets they would like to investigate, CIDER lists the topics extracted from source files, and a user can interactively select among them. Once a user chooses one or more topics, CIDER will present the part of the concept hierarchy that is related to those topics, which we call a *Facet Hierarchy*.

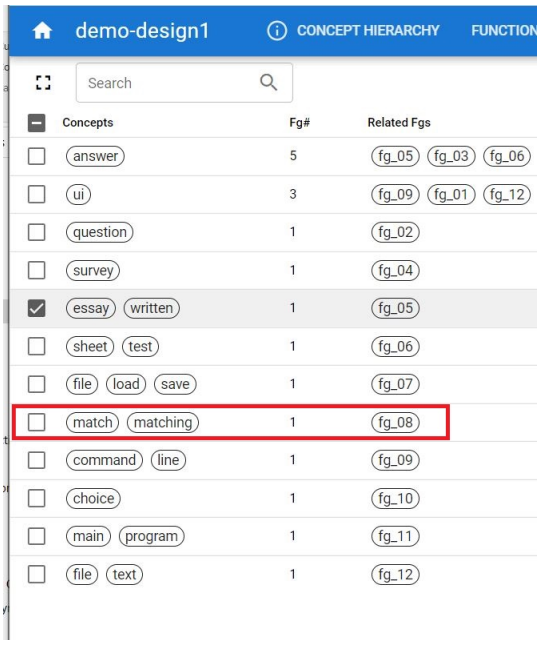
Figure 4 depicts two facet hierarchies extracted from Design-1 and Design-2 respectively, where the user selected the *essay, written* concept as shown in Figure 3b. The green circles are the files directly implementing the selected concepts, and the concepts in blue circles use the green ones. In Figure 4a, the "$Essay, Answer, Written$" cluster is directly related to the selected concepts, and *ProgramMain* uses them. By contrast, in Design-2, many more clusters depend on the concepts related to essay questions.

In addition to extracting facet hierarchies, the CIDER website provides a number of other interactive capabilities. For a complex system, the user can also shrink or expand a concept hierarchy using the +/- controls to explore the recovered design structure at different

[5]http://149.28.157.117/#/demo-design1
[6]http://149.28.157.117/#/demo-design2



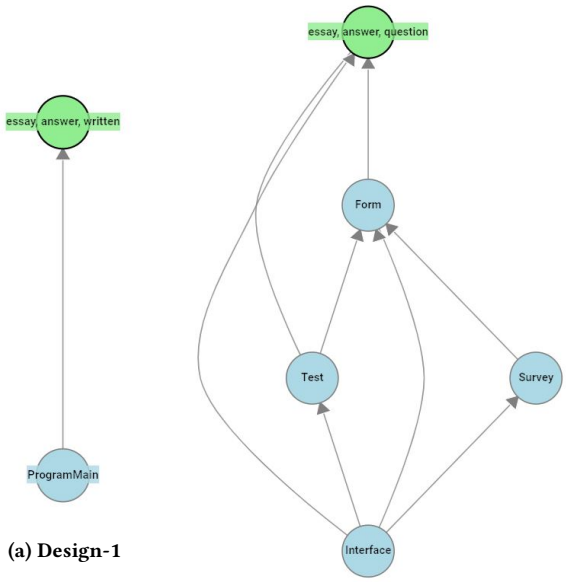
(a) Concepts of each FG



(b) Concepts cross FGs

Figure 3: Function Gruop and Concepts

levels of granularity. The user can also use the "*Function Group*" tab to explore how function groups interact, or select multiple concepts in the "Facet hierarchy" tab to observe how these concepts interact.



(a) Design-1

(b) Design-2

Figure 4: Essay Facet Hierarchies in two designs

4 EXPLORATORY EVALUATION

To evaluate the capability of understanding a design structure, and the possibility to assess design quality and detect design debts using CIDER, we conducted exploratory case studies on two open-source projects. We collected source code from these projects, reached out and presented the concept hierarchies using the CIDER interface to our collaborators in these projects and asked them to distribute the survey to their community. We conducted a survey that can be found at: http://149.28.157.117/. We received ten responses in total. Nine of them are from developers of Depends, and the last one came from the architect of fEMR. Based on these responses we assessed CIDER from the following aspects:

Understanding Design Structure. Understanding the design structure of a system involves first understanding the key abstractions and the data model. As illustrated by the questionnaire system, the hierarchical structures used by CIDER always present the most influential files at the top. It is clear that the two designs, although implementing exactly the same requirements, are designed based on very different abstractions. If a new developer wished to extend the system by adding a new type of question, the concept hierarchy shown in Figure 2a indicates that the new question class must inherit or use the three foundational concepts implemented by the three base classes, *UI.java*, *Question.java* and *Answer.java*. *ProgramMain* will need to be changed to accommodate the new question type.

A facet hierarchy aims to facilitate feature localization and change impact analysis. For example, if there is a bug found related to essay questions, a developer could use facet hierarchies to locate the problem, dramatically narrowing down the search space.

To evaluate the effectiveness of our tool on understanding design structure, we asked "*Are the concept and facet hierarchies useful and meaningful to users.*" in the survey of our case study participants. Both the architects of FEMR and all the developers of Depends considered the concept/facet hierarchy to be "*Very Useful*" or "*Quite Useful*", and provided positive comments.

Assessing Design Quality. Using CIDER, a user can observe excessive coupling, and assess how well features or concerns are separated. In particular, CIDER explicitly marks strongly connected file groups—cliques, one of the most prominent anti-patterns—and the user can open a new page to explore how a clique is formed. The number of cliques and the number of files involved in these cliques are indicators of design quality. For example, we found a node with a 32-file clique in Depends, and the architect confirmed that it is technical debt and should be removed. The architects also found unexpected coupling among features that are not anti-patterns or code smells. For example, CIDER shows that the "*Ruby*" facet depends on the "*Cpp*" facet, for which the architect confirmed to be the result of a long-forgotten shortcut. Based on the insights obtained from CIDER, both fEMR and Depends have since refactored their code bases to reduce the amount of technical debt.

Maintenance support. The survey additionally revealed that all 9 Depends developers agreed that CIDER is useful in maintenance activities. Even the two users who only marked "*Moderately useful*" provided highly positive comments, such as: *"I think it will be very useful for understanding how features are implemented in the system."*. The fEMR architect commented: *"Our developers are often working on codes they did not author, this gives a clear framework for the inter-dependencies that need to be tested"*. The Depends and fEMR teams are now using CIDER to help new on-board new developers, and to help existing developers understand and extend their systems.

5 CONCLUSION

In this paper we have presented CIDER, an architecture recovery tool that integrates multiple clustering techniques to extract a high-level design model in the form of hierarchical function clustering, and presents the recovered architecture to a user as a tree of concepts. CIDER also detects and marks the clique anti-pattern during the clustering process so that a user can examine it in detail.

CIDER also enables a user to select concepts and recover partial designs based on their selections. In this way they can become aware of unexpected couplings among features. Our exploratory case studies confirmed the significant potential benefit of CIDER in terms of early design debt detection and facilitating maintenance activities. The evaluations conducted with our case study teams have already motivated refactorings of their projects, providing initial evidence of the practical utility of CIDER.

6 ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants CCF-1816594/1817267, OAC-1835292, and CNS-1823177/1823214.

REFERENCES

[1] Periklis Andritsos and Vassilios Tzerpos. 2005. Information-Theoretic Software Clustering. *IEEE Transactions on Software Engineering* 31, 2 (Feb. 2005), 150–165.
[2] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.
[3] K. Chen and V. Rajlich. 2000. Case study of feature location using dependence graph. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*. 241–247.
[4] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2011. Feature Location in Source Code: A Taxonomy and Survey. In *Journal of Software Maintenance and Evolution: Research and Practice*.
[5] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Yuanfang Cai. 2011. Enhancing architectural recovery using concerns. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 552–555.
[6] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. 1999. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. 50–59.
[7] Onaiza Maqbool and Haroon A. Babri. 2007. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering* 33, 11 (Nov. 2007), 759–780.
[8] Brian S. Mitchell and Spiros Mancoridis. 2001. Comparing the Cecompositions Produced by Software Clustering Algorithms Using Similarity Measurements. In *Proc. IEEE International Conference on Software Maintenance*. 744–753.
[9] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2016. Decoupling Level: A New Metric for Architectural Maintenance Complexity. In *Proc. 38rd International Conference on Software Engineering*.
[10] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. 2019. Architecture Anti-patterns: Automatically Detectable Violations of Design Principles. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2910856
[11] Ran Mo, Will Snipes, Yuanfang Cai, Srini Ramaswamy, Rick Kazman, and Martin Naedele. 2018. Experiences Applying Automated Architecture Analysis Tool Suites. In *Proc. ASE*. ACM, 779–789. https://doi.org/10.1145/3238147.3240467
[12] David L. Parnas. 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–8.
[13] Karen Sparck Jones. 2004. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 60, 5 (2004), 493–502.
[14] V. Tzerpos and R. C. Holt. 2000. ACCD: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*. 258–267.
[15] Vassilios Tzerpos and Richard C. Holt. 2000. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Proc. 7th Working Conference on Reverse Engineering*. 258–267.
[16] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. 2009. Design Rule Hierarchies and Parallelism in Software Development Tasks. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 197–208. https://doi.org/10.1109/ASE.2009.53
[17] Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Design rule spaces: a new form of architecture insight.. In *International Conference on Software Engineering*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 967–977. http://dblp.uni-trier.de/db/conf/icse/icse2014.html#XiaoCK14
[18] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. 2006. SNIAFL: Towards a Static Noninteractive Approach to Feature Location. *ACM Trans. Softw. Eng. Methodol.* 15, 2 (April 2006), 195–226. https://doi.org/10.1145/1131421.1131424