Programming Tools for Working with Design Decisions in Code

Sahar Mehrpour * **and Thomas D. LaToza** †

George Mason University, Fairfax, VA

Abstract

When writing code, developers make design decisions by choosing between alternatives. Subsequent work with code requires reasoning about these design decisions, ensuring their code is consistent and answering rationale questions about why they were made. While documentation might help, it is rarely updated and often incomplete and untrustworthy. We propose a new form of documentation which is *active*, making design decisions checkable and offering immediate feedback on violations as they occur. Active documentation helps developers reason about design decisions by offering explanations and linked code snippets illustrating how to follow a design decision. To ensure active documentation is easy to create and maintain, new ways to create and edit checkable design decisions are needed. We offer a vision for active documentation, offer evidence for its potential, describe several techniques to achieve it, and suggest future directions for programming tools to better support it.

Keywords: Design decisions. Programming tools. Design rationale. Documentation. Defect detectors.

1 Introduction

Developers make design decisions whenever they choose between alternatives [1], whether between two architectural styles or two approaches for persisting data into a datastore. Design decisions collectively define how functional and non-functional requirements are satisfied by an implementation. Ignoring and violating design decisions have substantial negative consequences, such as creating defects, code decay, and architectural erosion [2], [3].

Due to their centrality to programming, developers find themselves constantly needing to understand design decisions. Developers have long been instructed to make this easier by documenting their design decisions [4]. But in practice this documentation is rarely updated, making it incomplete and untrustworthy [5]. Instead, developers reverse engineer design decisions from code, and report that rationale questions about why code was built as it was to be among the most challenging and hard to answer [6], [7].

We believe there is an important opportunity to re-invent the nature of documentation and make it *active* [8]. Rather than interact with plain text, documentation can instead be viewed as a specification, which can be checked against code. At the same time, it needs to still fulfill the role of documentation and be written in a language that is understandable by developers and explains why design decisions were made. Moreover, as code is constantly changing, it needs to be a representation which is easily and quickly editable, reflecting developer's constantly changing understanding of their design in the moment. And, rather then being viewed as a burden imposed on developers to create, documentation should help developers in reasoning about their decisions as they make them, helping quickly gather both positive examples of code following it and negative examples of violations and understand how these are related to other decisions.

To explore this vision, we first studied the potential for helping developers work more effectively with design decisions. We conducted a detailed study of code reviews, investigating the decisions that developers violate and the potential for tools to help developers identify these violations (Section 3). We found many of these issues could be written in the form of checkable rules, many of which can even be checked by relatively simple AST-based program analysis tools. Based on these findings, we have been creating a series of tools for making documentation active, helping developers more rapidly

PLATEAU
12th Annual Workshop at the Intersection of PL and HCI

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

*Email: smehrpou@gmu.edu
†Email: tlatoza@gmu.edu

and easily create checkable design rules and ensuring that these tools retain the value and benefits that effective documentation can bring to developers. We built ActiveDocumentation [8] (Section 4) to help developers follow design decisions in code and RulePad [9] (Section 5) to reduce the effort required to write design decisions in a checkable form. We are currently exploring new techniques for identifying implicit design decisions in code (Section 6).

2 Background

A design decision is a chosen technical alternative [1] which commits the project to a particular design or otherwise restricts the design space [10]. Design decisions vary in their scope and complexity, ranging from low-level decisions impacting a few lines of code to high-level architectural decisions impacting the entire codebase [11]. Design decisions may be associated with a variety of attributes, such as a description, author, impact [12], rationale [13], and constraints. Design rationale describes the alternatives which were considered as well as the reasons for choosing the selected alternative [14], [15]. The constraints of a design decision describe how code should be implemented to be consistent with the decision [16]. These form a *design rule* imposed by the decision.

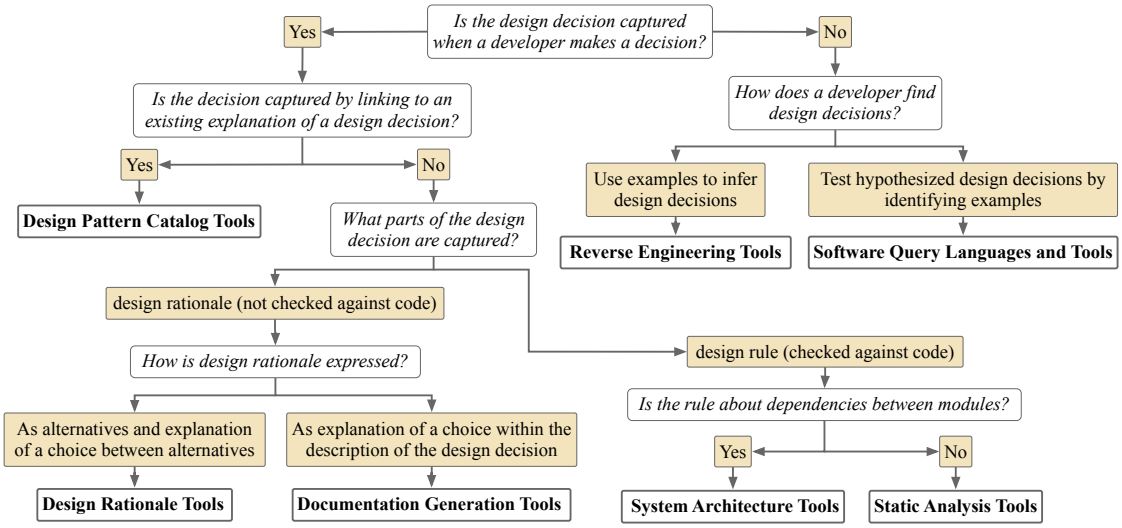

Figure 1. Tools supporting working with design decisions in code make many choices about how decisions are captured and represented (italic text), resulting in a variety of approaches (bold text).

A wide body of tools have envisioned ways in which developers might be better supported when interacting with design decisions, from capturing and documenting decisions to recovering uncaptured decisions from code (Figure 1). To capture decisions, tools may record and link design decisions to existing artifacts like design pattern catalog tools (e.g., DRIM [17]). Or support documenting checkable constraints, such as system architecture tools (e.g., SAVE [18]) and static analysis tools (e.g., FindBugs [19]). Other attributes of design decisions, such as their description, rationale, and alternatives, may be captured using design rationale tools (e.g., SEURAT [20]) or documentation tools such as on-demand documentation tools [21]). If design decisions are not captured when the decision is first made, tools may still offer support, helping developers find decisions by inferring them from examples in code or testing hypothesized decisions by identifying examples in code. Reverse engineering tools (e.g., DECKARD [22]) and software query languages and tools (e.g., EG [23]) are designed to help developers to infer decisions or test hypothesized decisions.

One way to consider how tools support developers in their work with design decisions is to consider related tasks (e.g., make a design decision, change a design decision) and goals developers face in accomplishing these tasks (Figure 2). We identify six key goals in working with design decisions: identifying [24], selecting [25], and documenting chosen alternatives [4], [26]; explaining and following design decisions [27]; and checking hypothesized decisions [28], [29] (Table 1). Tools often support more than one of these goals, and may vary in the effectiveness of their support for each (Table 2). For example, static analysis tools enable developers to find and follow relevant design decisions in code

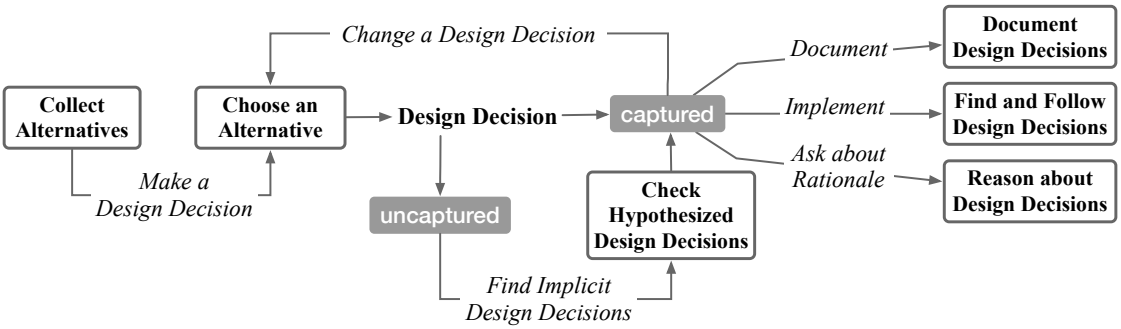

Figure 2. When working with design decisions, developers face many tasks (italic), leading to a variety of goals (rectangles) depending on the status of design decisions (dark rectangles).

Table 1. When working with design decisions, developers seek to accomplish a number of distinct goals.

	Goal	Example
Goal 1	**Identify potential alternatives**	*How should functionality be decomposed into classes to achieve extensibility and maintainability?*
Goal 2	**Select an alternative as a design decision**	*Is the best alternative for this situation the Command Pattern or Publish/Subscribe?*
Goal 3	**Document the chosen alternative**	*Communicate the design decision of selecting the Command pattern to future developers through documentation.*
Goal 4	**Check hypothesized design decisions against code**	*After reading the code, a developer hypothesizes that the Command pattern is being used and seeks additional evidence to test this hypothesis.*
Goal 5	**Find and follow relevant design decisions**	*While creating a new class to implement a new user action, a developer tries to determine how it should be connected to existing functionality that captures user toolbar actions.*
Goal 6	**Determine why an alternative was selected**	*After seeing that communication is mediated through Command patterns, the developer tries to determine why it was selected instead of a Publish/Subscribe approach.*

and offer partial support for documenting the chosen alternatives, checking hypothesized decisions, and reasoning about decisions. But they offer no support for identifying alternative decisions and choosing between alternatives.

3 Design Decisions in Code Reviews

To better understand the types of decisions developers fail to follow in their everyday work, we studied the defects found by code reviews and evaluated the potential of programming tools to detect these defects.

A number of studies have examined the ability of defect detectors to identify defects. For example studies found that FindBugs, JLint, and PMD might detect 35% to 95% of defects reported through issue trackers [30], Error Prone, Infer, and SpotBugs could detect 4.5% of defects in Defects4J dataset [31], and PMD could detect 16% of issues in code review comments [32]. However, our analysis differed in that, rather than run the tools that exist today, we instead qualitatively investigated the nature of each code review issue. Through this process, we were able to systematically examine not only what defects can definitely be found today, through the tools as they exist and are currently configured, but also which might be found by using the same underlying program analysis approach, but with a perfect database containing every design decision in the project.

We systematically collected and qualitatively analyzed more than 1300 defects found in code reviews. For this analysis, we used all available information, including submitted code, updated code, reviewers' comments, and followup discussions, to formulate each defect as a violation of a rule. While we did not count the frequency each rule applied within a codebase, we observed many to be violated several times in the same repository. We then compared these rules against the underlying techniques used by various program analysis tools. We created a taxonomy of tools (Table 3), focusing on characterizing the representation of code used to check for defects (e.g., abstract syntax tree, program execution, string literals), the origin of the rule (e.g., programming language syntax and semantics, project-specific design decisions), and the consequences of its violation (behavioral changes or code

Table 2. Existing developer tools help developers achieving different goals involving design decisions with different levels of support; ● denotes full support, ◐ denotes partial support, and ○ denotes no support.

	Goal 1	Goal 2	Goal 3	Goal 4	Goal 5	Goal 6
Documentation Generating Tools	◐	◐	○	◐	◐	◐
Static Analysis Tools	○	○	◐	◐	●	◐
Design Rational Tools	◐	●	●	◐	◐	●
Design Pattern Catalogs	●	●	●	◐	◐	●
System Architecture Tools	○	○	◐	◐	●	◐
Reverse Engineering Tools	◐	○	○	○	◐	○
Software Query Languages and Tools	◐	○	○	●	◐	○

Table 3. Program analysis tools differ in their representation of code (A: AST, CE: Code Execution, ST: Strings), origin of defects (L: Language, SP: Specifications, BP: Best Practices), and consequences (CQ: Code Quality, B: Behavioral). The ∼ symbol indicates indirect influence.

	Representation			Origin			Consequence	
PAT Categories	A	CE	ST	L	SP	BP	CQ	B
Style Checkers	✓		✓			✓	✓	
Continuous Integration Tools	✓	✓	✓		✓			✓
Data Flow Analyzers	✓	✓			✓		✓	✓
Architectural Style Checkers	✓	✓			✓		✓	
Test Suite Quality Checkers	✓	✓	✓		✓		∼	∼
Dead Code Detectors	✓	✓				✓	✓	
Code Clone Detectors	✓	✓				✓	✓	
Compilers	✓			✓				✓
String Compilers			✓		✓	✓	✓	✓
Code Smell Detectors	✓	✓				✓	✓	
Memory Leak Detectors		✓		✓				✓
AST Pattern Checkers	✓				✓		✓	✓

quality).

Our analysis revealed that existing program analysis techniques may be capable of detecting three-quarters of the defects found in code reviews. This is considerably higher than the 16% of issues found by studies of existing defect detectors [32], suggesting the potential for program analysis techniques to be used to find more defects. In particular, we found that style checkers and AST-based defect detectors might potentially detect half of the defects found in code reviews. However, many of these defects differ considerably from the typical defects found today by these tools. Rather than encode rules such as variables should be defined before use, these rules instead captured project specific rules. That is, they capture design decisions which developers had violated.

4 Making Documentation Active

To help developers more effectively work with design decisions in code, we proposed a new form of documentation which is *active* [8]. (Figure 3). ActiveDocumentation makes the design rule constraints imposed by decisions checkable as AST patterns, which we found to be a commonly applicable representation (Section 3). ActiveDocumentation helps developers follow design decisions by maintaining an active link between code and captured design decisions, actively checking the decisions against code, and informing developers about divergences between code and design decisions. The active links enable developers to find examples of code following decisions, which is often sought by developers [33], and to update design decisions to be consistent with code. To help developers easily find relevant design decisions, ActiveDocumentation surfaces design decisions related to the current files and includes a tagging mechanism for organizing and browsing design decisions.

We evaluated ActiveDocumentation by conducting a user study asking participants to implement a feature in an unfamiliar codebase. We found that ActiveDocumentation enabled developers to work more quickly and successfully with design decisions in code. Developers used the examples identified

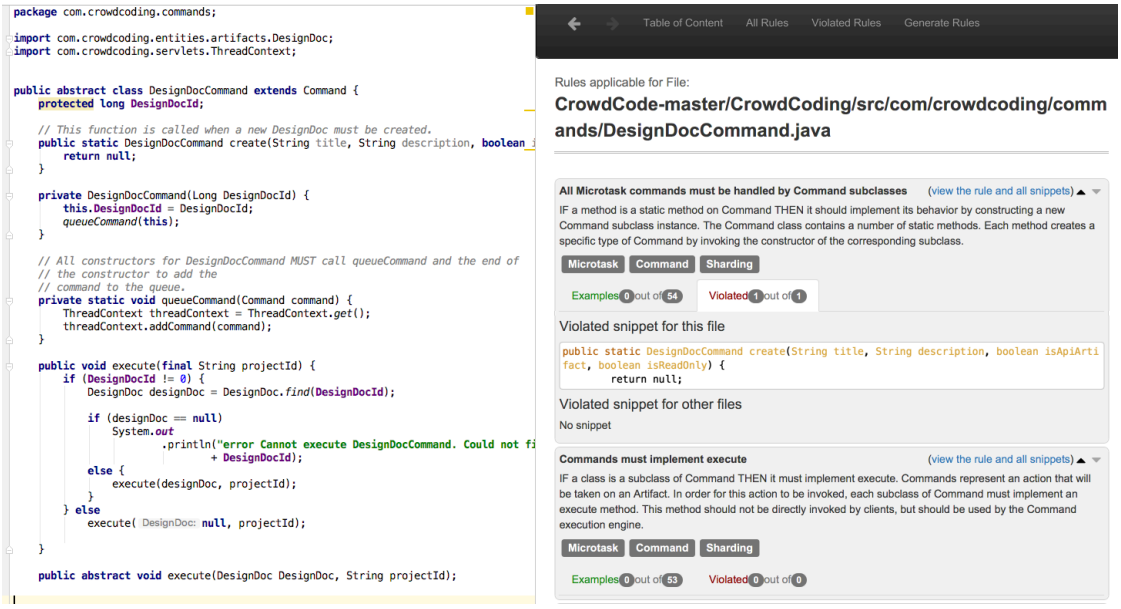

Figure 3. ActiveDocumentation enables developers to find and follow relevant design decisions, surfacing design decisions related to the current file.

by ActiveDocumentation to learn how to follow design decisions, and used identified violations to find incorrect and incomplete code. Developers reported that the instant feedback in ActiveDocumentation helped them detect defects early.

5 Helping Developers Write Checkable Design Decisions

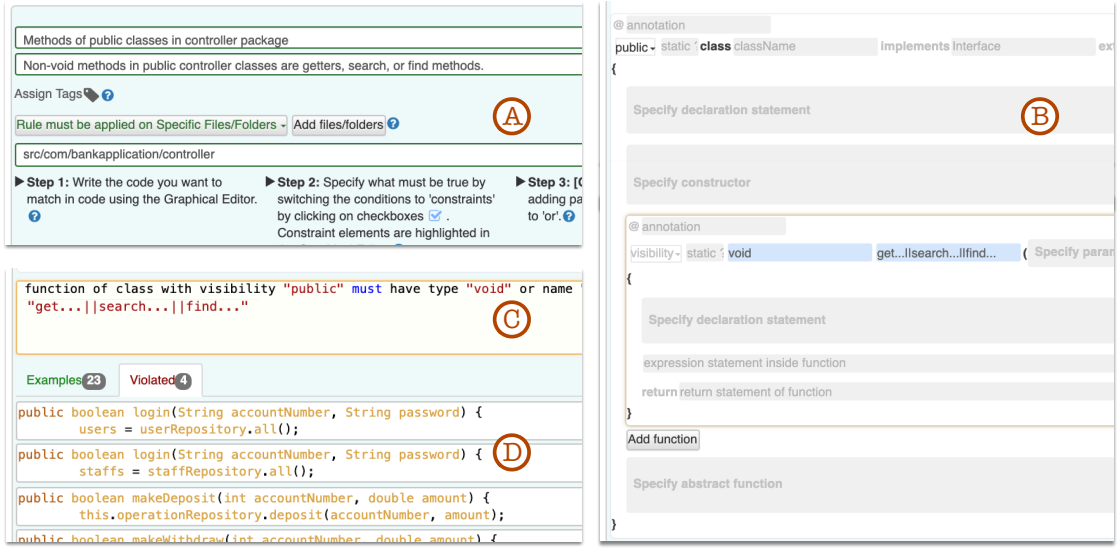

Figure 4. Developers may use RulePad to author checkable design decisions by either using (B) snippet-based authoring to express design decisions using simple code-based templates in a structured editor or (A) using semi-natural language to express design decisions in a notation close to prose. While editing decisions, developers may examine the behavior of the decision by (D) viewing a list of example code snippets which satisfy or violate the decision.

In today's tools, there is a wide gulf between explaining a design decision in a prose document and writing a defect pattern in PMD. Yet, if documentation is to be made checkable, it is necessary for a developer to express their design decision in a way that can translated into a specification which can be used by a program analysis tool. Many static analysis tools such as Error Prone [34] and Infer [35] are extensible, enabling developers to write new defect patterns. However, they require substantial knowledge of program analysis to write, either in a general purpose programming language

(e.g. FindBugs [19]) or through complex query notations (e.g. XPath in PMD [36]). These notations impose a substantial barrier in both expertise and effort. Asking developers to constantly document their design decisions in checkable form clearly requires reducing these barriers.

We created two complimentary techniques for documenting checkable design decisions: *snippet-based authoring* and *semi-natural language authoring* [9]. These approaches are complimentary in their balance between simplicity and expressiveness. Developers may first use snippet-based authoring to express design decisions through simple code-based templates, using their knowledge of code to author simple rules in a structured editor that looks like code. But more complex design decisions, requiring more control over conjunction and disjunction, cannot be expressed. In semi-natural language authoring, developers can use a language that looks like a natural language description of a decision to author a wide range of decisions. But developers must first learn the semi-natural language.

We implemented these two approaches in RulePad, an extension to our ActiveDocumentation tool, in a Graphical Editor and Textual Editor. As developers author design decisions, they receive immediate feedback in the form of positive examples following the decision and code snippets which violate the decision (Figure 4-D). The Graphical Editor and Textual Editor are bidirectionally synchronized. As developers edit rules in the Graphical Editor, they can understand their design decision by reading the constructed semi-natural language representation in the Textual Editor (Figure 4-C).

In a user study comparing authoring checkable design decisions in RulePad against existing support in PMD, we asked developers to author a series of design decisions. We found that participants were able to successfully author 13 times more query elements using RulePad than PMD and were more willing to used RulePad in their everyday work.

6 Suggesting Design Decisions From Code

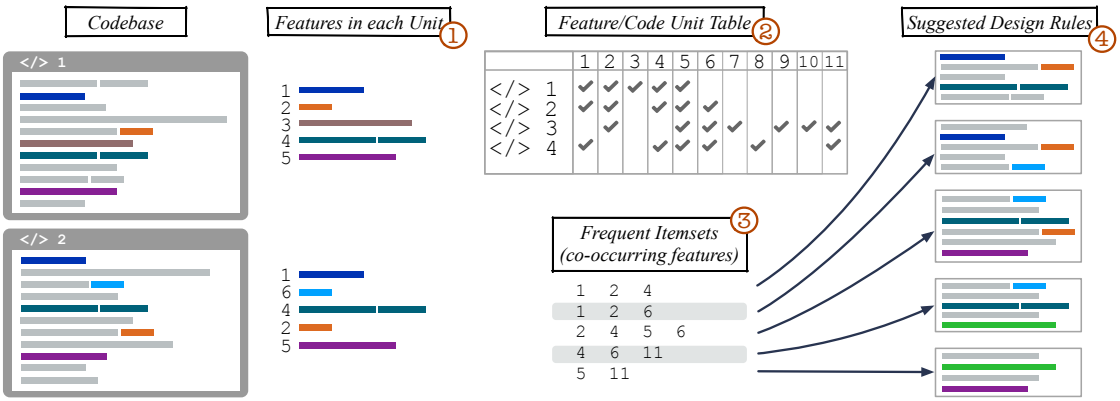

Figure 5. To suggest design decisions to developers, features are first extracted from code and stored in a data table. From this, frequent co-occurring features are identified and displayed as suggested design rules.

When design decisions are not written down, developers use source code as their primary resource and reverse engineer design decisions from code [7]. Developers look for examples in code, try to infer rules from examples, and test hypothesized rules against code.

We believe programming tools can offer better support here. We envision a tool which helps developers by suggesting potential design decisions relevant to their current work (Figure 5). To find design decisions, features are first extracted from each code element (e.g., class, method, field) (Figure 5-1). These features might include the identifier of the superclass, the parameter types for a method, or the initial value of fields. Each code element can then be represented by its features (Figure 5-2). Through frequent itemset mining algorithms, co-occurring features can next be identified (Figure 5-3). Finally, frequent itemsets can themselves be clustered and rendered in a readable form as suggested design rules (Figure 5-4).

One key challenge with mining approaches is helping to surface interesting and relevant design decisions from the vast quantity of coincidental occurrences. Widely used algorithms for mining frequent itemsets (e.g., FP-Growth [37]) focus mostly on the number of features co-occurring and less on their importance or relevance. Many itemsets may be highly similar, creating more noise to

sift through. Meeting these challenges requires approaches for clustering itemsets and finding ways to identify potential design decisions which are most relevant and interesting.

Another key challenge is selecting the features with which to represent code. The potential number of features is exponential in the size of the code snippet, making it rapidly intractable to examine all possible features. To address this challenge, a set of standard features might be curated from examples of design decision in codebases. But it is crucial to offer extensbility, letting developers themselves create new features.

To evaluate the effectiveness of an approach, it is important to investigate its ability to suggest the design decisions developers might care about. This might be assessed by finding a corpus of design decisions for a codebase and comparing it to those extracted. But it is important to also conduct user studies to evaluate the ability of such techniques to help developers more quickly and successfully find and follow relevant design decisions.

7 Conclusion

Understanding the rationale behind code has long been particularly painful and time consuming due to the failure of programming tools to adequately support the process of creating and maintaining documentation. By thinking more broadly about what documentation might and should be to effectively support work with design decisions, many opportunities for better tools can be found. Rather than conceive of documentation as a passive document, to be manually created and maintained, it can instead be thought of as an active view of code, co-created together by the developer and their programming tools. Our studies show the potential value of this approach to improving the ability of developers to work effectively. But there a wide range of goals which developers pursue when working with design decisions, each of which requires careful consideration of how to offer effective tool support.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972. DOI: 10.1007/978-3-642-48354-7_20.

[2] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001. DOI: 10.1109/32.895984.

[3] M. Lindvall, R. T. Tvedt, and P. Costa, "Avoiding architectural degeneration: An evaluation process for software architecture," in *International Software Metrics Symposium (METRICS)*, 2002, pp. 77–86. DOI: 10.1109/METRIC.2002.1011327.

[4] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251–257, 1986. DOI: 10.1109/tse.1986.6312940.

[5] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Software*, no. 6, pp. 35–39, 2003. DOI: 10.1109/ms.2003.1241364.

[6] T. D. LaToza and B. A. Myers, "Hard-to-answer Questions About Code," in *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010, pp. 1–6. DOI: 10.1145/1937117.1937125.

[7] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *International Conference on Software Engineering (ICSE)*, 2006, pp. 492–501. DOI: 10.1145/1134285.1134355.

[8] S. Mehrpour, T. D. LaToza, and R. K. Kindi, "Active Documentation: Helping Developers Follow Design Decisions," in *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2019, pp. 87–96. DOI: 10.1109/vlhcc.2019.8818816.

[9] S. Mehrpour, T. D. LaToza, and H. Sarvari, "Rulepad: Interactive authoring of checkable design rules," in *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 386–397. DOI: 10.1145/3368089.3409751.

[10] G. Fairbanks, *Just enough software architecture: a risk-driven approach*. Marshall & Brainerd, 2010.

[11] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic, "Recovering architectural design decisions," in *International Conference on Software Architecture (ICSA)*, 2018, pp. 95–9509. DOI: 10.1109/ICSA.2018.00019.

[12] P. Kruchten, "An ontology of architectural design decisions in software-intensive systems," in *Groningen Workshop on Software Variability Management*, 2004.

[13] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992, ISSN: 0163-5948. DOI: 10.1145/141874.141884.

[14] T. P. Moran and J. M. Carroll, *Design rationale: Concepts, techniques, and use*. CRC Press, 1996.

[15] J. Lee, "Design rationale systems: Understanding the issues," *IEEE Expert*, vol. 12, pp. 78–85, 3 1997. DOI: 10.1109/64.592267.

[16] C. Y. Baldwin and K. B. Clark, *Design rules: The power of modularity*. MIT Press, 2000, vol. 1.

[17] F. Peña-Mora and S. Vadhavkar, "Augmenting design patterns with design rationale," *AI EDAM*, vol. 11, no. 2, pp. 93–108, 1997. DOI: 10.1017/S089006040000189X.

[18] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007, pp. 12–12. DOI: 10.1109/wicsa.2007.1.

[19] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004, pp. 132–136. DOI: 10.1145/1028664.1028717.

[20] J. E. Burge and D. C. Brown, "Software engineering using rationale," *Journal of Systems and Software*, vol. 81, no. 3, pp. 395–413, 2008. DOI: 10.1016/j.jss.2007.05.004.

[21] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, *et al.*, "On-demand developer documentation," in *International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 479–483. DOI: 10.1109/icsme.2017.17.

[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *International Conference on Software Engineering (ICSE)*, 2007, pp. 96–105. DOI: 10.1109/icse.2007.30.

[23] C. Barnaby, K. Sen, T. Zhang, E. Glassman, and S. Chandra, "Exempla gratis (EG): Code examples for free," in *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1353–1364. DOI: 10.1145/3368089.3417052.

[24] M. X. Liu, J. Hsieh, N. Hahn, A. Zhou, E. Deng, S. Burley, C. B. Taylor, A. Kittur, and B. A. Myers, "Unakite: Scaffolding developers' decision-making using the web," in *Symposium on User Interface Software and Technology (UIST)*, 2019, pp. 67–80. DOI: 10.1145/3332165.3347908.

[25] R. Beheshti, "Design decisions and uncertainty," *Design Studies*, vol. 14, no. 1, pp. 85–95, 1993, ISSN: 0142-694X. DOI: https://doi.org/10.1016/S0142-694X(05)80007-9.

[26] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, and M. A. Babar, "10 years of software architecture knowledge management: Practice and future," *Journal of Systems and Software*, vol. 116, pp. 191–205, 2016. DOI: 10.1016/j.jss.2015.08.054.

[27] S. Rugaber, S. B. Ornburn, and R. J. LeBlanc, "Recognizing design decisions in programs," *IEEE Software*, vol. 7, no. 1, pp. 46–54, 1990. DOI: 10.1109/52.43049.

[28] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *International Conference on Software Engineering (ICSE)*, 2007, pp. 344–353. DOI: 10.1109/ICSE.2007.45.

[29] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008. DOI: 10.1109/TSE.2008.26.

[30] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *International Conference on Automated Software Engineering (ASE)*, 2012, pp. 50–59. DOI: 10.1145/2351676.2351685.

[31] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *International Conference on Automated Software Engineering (ASE)*, 2018, pp. 317–328. DOI: 10.1145/3238147.3238213.

[32] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017, pp. 101–105. DOI: 10.1109/vlhcc.2017.8103456.

[33] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681. DOI: 10.1109/icse.2013.6606613.

[34] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2012, pp. 14–23. DOI: 10.1109/SCAM.2012.28.

[35] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods Symposium*, ser. Lecture Notes in Computer Science, vol. 9058, 2015, pp. 3–11. DOI: 10.1007/978-3-319-17524-9_1.

[36] T. Copeland, *PMD Applied*. Centennial Books, 2005.

[37] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *International Conference on Management of Data (SIGMOD)*, 2000, pp. 1–12. DOI: 10.1145/342009.335372.