A Survey of Tool Support for Working with Design Decisions in Code

SAHAR MEHRPOUR*, George Mason University, USA
THOMAS D. LATOZA*, George Mason University, USA

Whenever developers choose among alternative technical approaches, they make a design decision. Collectively, design decisions shape how software implements its requirements and shape non-functional quality attributes such as maintainability, extensibility, and performance. Developers work with design decisions both when identifying, choosing, and documenting alternatives and when later work requires following and understanding previously made design decisions. Design decisions encompass design rationale, describing the alternatives and justification for a design choice, as well as design rules, describing the constraints imposed by specific alternatives. This article summarizes and classifies research on these activities, examining different approaches through which tools may support developers in working with design decisions in code. We focus both on the technical aspects of tools as well as the human aspects of how tools support developers. Our survey identifies goals developers have in working with design decisions throughout the lifecycle of design decisions. We also examine the potential support tools may offer developers in achieving these goals and the challenges in offering better support.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **Software maintenance tools**; • **Information systems** → *Information retrieval*; • **Human-centered computing**;

Additional Key Words and Phrases: design decisions, design rules, design rationale, developer tools

ACM Reference Format:
Sahar Mehrpour and Thomas D. LaToza. 2023. A Survey of Tool Support for Working with Design Decisions in Code. *ACM Comput. Surv.* 1, 1 (July 2023), 37 pages. https://doi.org/10.1145/xxxxx

1 INTRODUCTION

When building software, developers work with design decisions every day. For example, a developer might employ the Proxy pattern [56] in a document editor in order to reduce the cost of expensive object creations by providing a placeholder for the objects and only creating them on-demand. Or a developer might decide that, in order to reduce latency, map data will be lazily loaded and only transferred when needed. Or a developer might decide that, in order to make it easier to potentially change the library used for persistence, all interactions with the persistence framework will be localized in a specific module.

Design decisions describe a choice and the potential alternatives for a specific design problem [147]. They describe how the system commits to a particular design and how the design space is restricted [46]. They shape how software achieves its requirements, impacting the behavior

*This work was supported in part by the National Science Foundation under grant NSF CCF-1703734 and CCF-1845508.

Authors' addresses: Sahar Mehrpour, George Mason University, 4400 University Drive, Fairfax, VA, USA, smehrpou@gmu.edu; Thomas D. LaToza, George Mason University, 4400 University Drive, Fairfax, VA, USA, tlatoza@gmu.edu.

0360-0300/2023/7-ART $15.00
https://doi.org/10.1145/xxxxx

of a program. At the same time, design decisions shape how software achieves a wide range of non-functional requirements, such as maintainability, extensibility, and performance.

Developers work with design decisions in order to accomplish specific goals, both when initially choosing between alternatives to make a design decision as well as when returning to a previously made decision to understand how to follow it. For example, a developer may have a goal to understand the rationale behind a design decision and the reasons for choosing one alternative instead of others. Accomplishing this goal can help to build a correct mental model of a codebase and to ensure that future changes and decision-making follow the intended design. A developer might also have a goal to write code which follows an existing design decision. In this case, the developer needs a clear and concise understanding of the design decision, the constraints it imposes, and how code elsewhere is implemented consistent with these constraints.

To work with design decisions, developers are traditionally encouraged to use design documents, which are intended to contain information on important design decisions. However, in practice today, developers work with design documents that are rarely updated, leaving them outdated, incomplete, and untrustworthy [91]. Even when updated, it can be hard to follow design decisions [104]. As design documents become distrusted or abandoned, developers may manually reverse engineer design decisions and their rationale from code [82], which is often challenging. Questions about design rationale are some of the most frequently reported hard-to-answer questions [81] and create one of the most serious problems developers report facing [82].

The difficulties developers experience in working with design decisions may have profound consequences on software projects, leading to code decay, architectural erosion, and defects [45, 94]. Incorrect, unfit, or ignored design decisions alter code's behavior, result in the code diverging from developers' own mental models of it, decrease code comprehensibility, and cause code decay. Difficulties may lead the implementation of the program to diverge from the original design (architectural drift) and for new design decisions to violate the intended architecture (software erosion) [57, 58, 146, 155].

To address developers' difficulties, researchers have proposed a variety of tools which offer developers better support for accomplishing their goals in tasks in which they work with design decisions. Documentation tools enable developers to make design decisions explicit by documenting them in structured formats. Static analysis and system architecture tools enable developers to check documented design decisions against code. Design rationale tools capture, organize, and maintain design rationale and the process of decision making. Design pattern catalog tools link code to existing documentation. Reverse engineering tools and software query languages enable developers to find unwritten design decisions by extracting frequent code snippets or to test hypothesized design decisions against code.

Given the vital role of software tools in facilitating developers' work, it is imperative to investigate their usage in practice and identify areas for improvement to better support developers' needs. Previous surveys have primarily focused on the technical performance of tools, such as their features and precision. Surveys have examined tools and techniques for detecting patterns in code, such as code clones [126, 134] and design patterns [1, 42], and static analysis [90] and design rationale tools [122]. However, it is equally important to understand tools from the developers' perspective to assess if and when current tools may address the real needs of developers when working with design decisions and motivate the design of future more effective tools [109].

In this survey, we address these gaps by taking a developer-centered approach where we examine the goals developers have when working with design decisions and how tools may meet these needs. Specifically, we answer questions such as: What are the primary goals of developers when working with design decisions? Do existing tools meet these goals, and if not, how can they be modified to better support developers? Can existing tools be used to fulfill multiple goals?

This survey makes three contributions. First, we offer a taxonomy of tools which assist developers in working with design decisions. Second, we provide a list of goals that developers aim to accomplish, reflecting the needs developers have in working with design decisions. Third, we provide an evaluations of the level of support offered by each of these types of developer tools in helping developers to achieve these goals.

The structure of this survey is as follows. We first overview the background and related work on developer goals and software tools for working with design decisions (Section 2). Next, we describe the process and the methodology we followed to create this survey (Section 3). We define design decisions, including their constituent parts, various forms, and related concepts (Section 4). We then present a taxonomy of tools for working with design decisions and review tools of each type (Section 5). We identify six goals developers have when working with design decisions (Section 6) and critically examine the support offered by tools in helping developers achieve each of these goals (Section 7). We conclude with a discussion of new directions where tools may offer more effective support to developers in their work with design decisions in code (Sections 8 and 9).

2 RELATED WORK

Design decisions play a crucial role software development. A variety of tools have been designed to help developers work with design decisions, which we survey in this paper. Other work has examined how developers work with design decisions in their everyday work with programming and identified challenges with using tools to work with design decisions. This survey also reviews this work, using it to identify goals developers have when working with design decisions as well as critically examine how effectively each type of tool may support achieving these goals.

Prior survey articles have reviewed many of the tools we examine in this article. One survey examined techniques used to document design decisions and identified reasons why these documentation methods are not commonly adopted in practice [3]. Another surveyed the tools and techniques which have been designed for managing design rationale [122]. Work has also examined the use of program analysis tools for software maintenance that are available and can be used by practitioners [90]. Other surveys have classified tools and techniques for mining codebases to detect code clones [126, 134], code patterns [5], and design patterns [1, 42]. In addition to these, this article also surveys design pattern catalog tools, system architecture tools, and software query languages and tools, for which we are aware of no prior surveys.

Building on these surveys, this survey differs by going beyond a technical perspective of the techniques used to also examine tools from the point of view of a developer working with design decisions. We identify developer goals in working with design decisions and critically examine the support tools may offer in helping developers achieve these goals.

3 SURVEY METHODOLOGY

To create this article, we considered principles from well-known literature review methodologies [74]. This section provides an overview of the methodology used for conducting this survey.

3.1 Research goals and questions

The goals of this survey are to investigate the types of tools available for developers to support their work with design decisions, understand the goals that developers have when working with design decisions, and identify how accomplishing these goals can be supported by various types of tools. Additionally, this survey aims to identify limitations in existing tools and explore opportunities for improvement or to introduce new tools that can better support developers. With these objectives in mind, we aim to address the following research questions:

RQ1. What types of tools are available to support developers working with design decisions?
RQ2. What are developers' goals when they work with design decisions?
RQ3. In what ways do tools support developers in accomplishing their goals working with design decisions?

We address RQ1 in Section 5, RQ2 in Section 6, and RQ3 in Section 7.

3.2 Identification and selection strategy

Search strategy. To identify relevant studies for RQ1 and RQ2, we used several methods; 1) We collected a set of research papers that we were already familiar with that covered developers' tasks, their interaction with tools, and tools designed for working with design decisions in code. 2) We also collected another set of related papers through searching literature databases, such as Google Scholar, DBLP, IEEEXplore, and the ACM digital library, using relevant keywords such as "developer tasks", "developer tools", "following rules", "changing code", "finding patterns", and "detecting defects". 3) We then used snowballing to identify additional studies by examining the references of the papers we found and the studies that cited them. To identify prior surveys related to the tools we identified in RQ2, we searched literature databases using keywords such as "survey" and "overview", as well as using the tool types in Section 5 as keywords. We then used further snowballing to identify recent surveys by examining surveys which cite older surveys.

Inclusion and exclusion criteria. The scope of the papers examined in this survey includes technical reports and papers presented at workshops, conferences, and journals in software engineering and human-computer interaction. To address RQ1, we included studies that focused on developers and their needs, goals, and challenges. For RQ2, we considered tools designed to support developers in completing tasks that involve working with various representations of design decisions, such as design patterns. We excluded studies and tools that were considered too preliminary, such as position papers, as well as tools that did not pertain to design decisions, such as debugging tools.

3.3 Data extraction

For each article about a tool, we made notes on the paper's self-positioning, including the tool's purpose and the tasks it supports, as well as its features, including its user interface. We also noted example use cases and potential future features of the tool mentioned in the paper.

4 DEFINITIONS

In this section, we define several key concepts related to design decisions: software design, design decisions, design rationale, design rules, and design patterns.

Software design. Software design is the process of creating a blueprint for a software system by defining, visualizing, creating, and testing the architecture, components, modules, and interfaces to fulfill specified requirements [67, 68].

Design decision. In this paper, we define a design decision as a choice between alternative technical approaches made in order to achieve a goal [113]. For example, a design decision might be to choose the Proxy pattern [56] instead of Facade or Factory [56] to represent elements in a document editor. Design decisions may be characterized by the *scope* of their impact, ranging from low-level choices which impact a few lines of code to cross-cutting architectural decisions which impact an entire codebase [132]. Design decisions may be simple, compound, or cross-cutting. "Simple decisions have a singular rationale and consequence. Compound decisions include several closely related rationales, but their consequences are generally contained in one component. Finally, cross-cutting decisions affect a wider range of components, and their rationale follows a higher-level concern

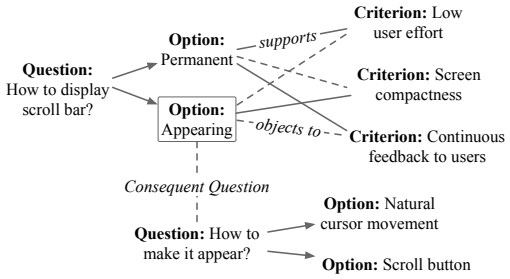

Fig. 1. An example argument in the QOC model of argumentation [99]. Design choices are specified as questions (design choices), options (design alternatives), and criteria (properties and requirements).

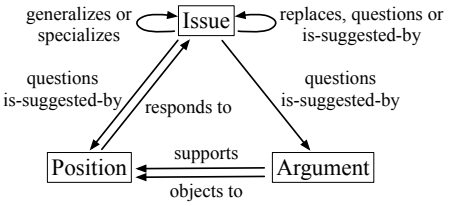

Fig. 2. In the IBIS model of argumentation [37], design choices (issues) can be generalized, specialized, or create other design choices. Each design choice is accompanied by several design alternatives (positions) which are supported or opposed by arguments.

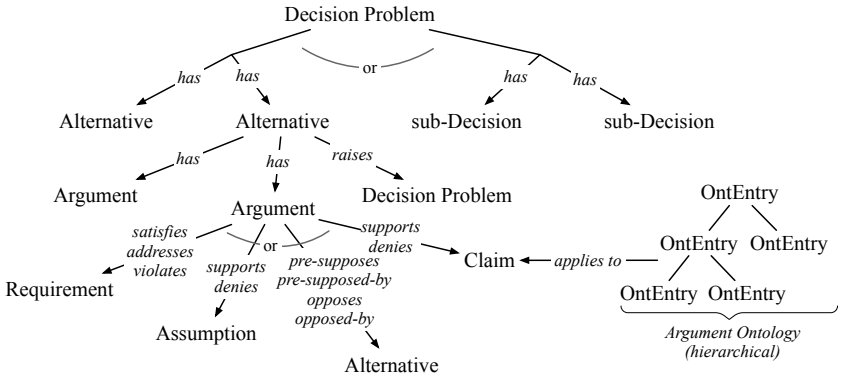

Fig. 3. In the DRL model in RATSpeak [24], design decisions are specified as design choices (decision problems), design alternatives, and arguments, which encompass requirements that are supported or violate, assumptions about alternatives, and the relationship with other alternatives and claims (design decisions).

such as the architectural quality of the system." [132] A variety of attributes may be associated with a design decision, such as its authors, history, area of impact (e.g., usability, security), current state (e.g., idea or implemented), risk, and cost [78].

Design rationale. Central to a design decision is its rationale [117]. Design rationale describes the alternatives which were considered and the justification for why the selected alternative was chosen [85, 107]. For example, a developer may select the Proxy pattern [55] in a document editor to reduce the cost of expensive object creations and only create them on-demand. Design rationale offers traceability which explicitly links requirements to the descriptions of artifact features that satisfy these criteria [44]. Design rationale may be captured and represented informally or through more structured formal approaches [36]. *Informal* approaches represent design decisions and rationale in raw documents [20] or other communication artifacts such as email, chat, audio, or video which record the design process [86]. *Formal* approaches impose a structure by which design decisions are represented through an *argumentation model* describing how the design decision was made and the rationale behind it [44], and as such they are widely known as design rationale models. For example, in Toulmin's model of argumentation, several elements are specified, such as a claim (a design decision), datum (data providing evidence for this claim), and a warrant which

connects the claim and the datum [143]. This model does not capture design alternatives or their arguments. Other prominent design rationale models are the Questions, Options, and Criteria (QOC) model [99], the Issue-Based Information System (IBIS) model [36, 123], and model used in the Decision Representation Language (DRL) [84]. In the QOC model, questions capture design choices, options describe design alternatives, and criteria enumerate properties and requirements that must be satisfied (Figure 1). Each question is associated with several options, and each option may satisfy or violate criteria. Design rationale is illustrate through a diagram, where lines connecting questions and options represent arguments. Similar to QOC, IBIS diagrammatically represents design issues (Figure 2). Specifically, in IBIS, issues (design choices), positions (design alternatives), and arguments are each diagram elements, and directed edges between them describe their relationships. This representation describes how a *resolution* is reached and a decision is made. IBIS influenced other design rationale models, including the Procedural Hierarchy of Issues (PHI) [51] and the model proposed by Potts and Bruns [120], which itself was extended in the model of DRL [84] by adding design rationale elements to the model (Figure 3). The DRL model then inspired later argumentation models, such as the Design Document Model (DDM) [64, 75] (also inspired by QOC) and RATSpeak [24].

Design rule. After a selected alternative has been chosen, this selection imposes a set of constraints that code must follow to be consistent with the design decision. These constraints are design rules [8]. Design rules impose "partial specifications to which the realization of one or more architectural [and non-architectural] entities have to conform to" [70]. For example, applying the Proxy pattern [55] to a document editor requires creating a proxy class and an entity class for each element in the editor, along with an editor class which may only access the proxy. Constraints may require or prohibit the existence of an entity or artifact, require specific properties for cross-cutting concerns (which may be hard to map to a specific location in source code), or constrain the conditions under which code is executed [78].

4.0.1 Design pattern. While a design decision is most often considered to be a choice made in a specific situation, a design decision can also be considered as a more general, and reusable, choice which balances competing considerations. This perspective was best popularized in the form of a design pattern, offering a solution to a problem in a context [55]. Many design patterns have been introduced, such as the GoF (Gang of Four) patterns on object oriented design [56], patterns for Agile programming [103], patterns for enterprise software [53], and patterns for specific programming languages such as Java [18]. Design patterns have often been closely associated with structural constraints on code elements, describing how a specific problem is solved by enumerating a few classes with specific roles and specifying constraints on their relationships (e.g., the Observer pattern [56] decouples a subject from an observer by maintaining a list of observers which are notified of state changes). Similar to design patterns, architectural styles also offer reusable solutions to a problem. However, instead of constraining how a few classes are connected, instead constrain how all elements in a system are connected. For example, in the Model-View-Controller [133], three types of elements have constraints on their interactions throughout the system.

5 TOOLS FOR INTERACTING WITH DESIGN DECISIONS IN CODE

A wide variety of tools have been proposed to support developers in working with design decisions in code. We identified 85 papers which introduce a tool that a developer may use to work with design decisions (Figure 4). These tools can be categorized along several dimensions, including what a developer is asked to do before or after making a design decision and how a design decision is represented. We used this categorization to created a taxonomy for tools (Figure 4). The types of tools we survey and the primary tasks they are intended to support are listed in Table 1.

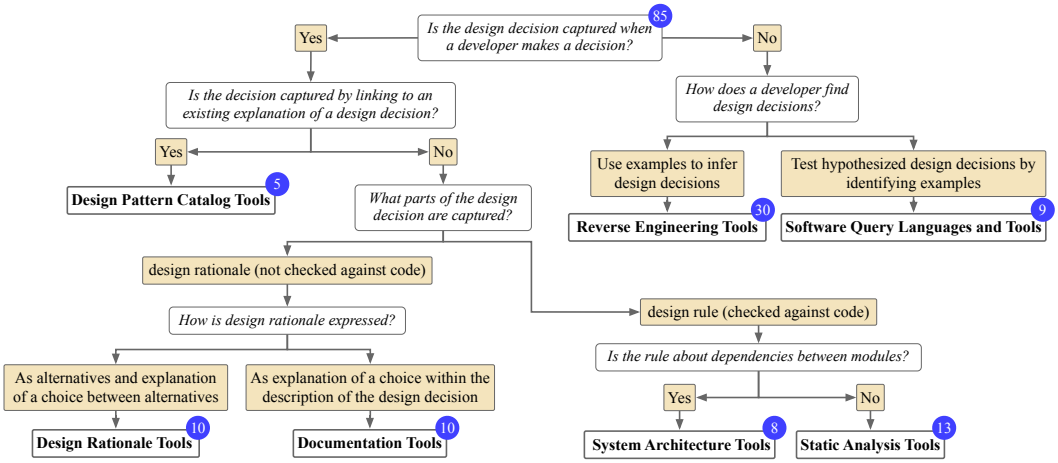

Fig. 4. A taxonomy of tools for supporting developers working with design decisions in code. Tools embody many choices about how decisions are captured and represented (italic text), resulting in a variety of approaches (bold text). The blue labels indicate the number of papers in each category.

Table 1. Tools focus on supporting developer work with design decisions in a primary task, but may also impact how developers achieve other goals (Section 6).

Rule Status	Tool Types	primary task	
Captured	Documentation Tools	document design decisions	Section 5.1
	Static Analysis Tools	document and check design rules	Section 5.2
	Design Rationale Tools	document alternatives and design rationale	Section 5.3
	Design Pattern Catalog Tools	document design patterns	Section 5.4
	System Architecture Tools	document and check module dependency rules	Section 5.5
Not Captured	Reverse Engineering Tools	infer potential design decisions	Section 5.6
	Software Query Languages and Tools	test hypothesized decisions by identifying examples	Section 5.7

Most broadly, tools may support developer work with design decisions that are either explicitly *captured* and written down at the point when a developer makes a decision or which remain *uncaptured*. Tools have envisioned several ways to offer support for explicitly captured decisions. Design decisions may be represented through a link to a more generic form of the design decision, as found in Design Pattern Catalog Tools (Section 5.4). Or design decisions may be directly documented, either as the design rationale explaining the choice or the design rule associated with the design decision. Design rationale may be documented either explicitly, as alternatives and the criteria used to choose between them (Design Rationale Tools, Section 5.3), or less formally, through unstructured text (Documentation Tools, Section 5.1). Other tools focus on checking that code remains consistent with a design rule. These may focus either on higher-level architectural rules (System Architecture Tools, Section 5.5) or lower-level code rules (Static Analysis Tools, Section 5.2).

As design decisions are often not written down and explicitly captured when the design decisions were originally made, a wide variety of tools have been designed to help support developer work with uncaptured design decisions. Tools may support developers as they attempt to infer design decisions from examples (Reverse Engineering Tools, Section 5.6) or to hypothesize design decisions and test these hypotheses against code (Software Query Languages and Tools, Section 5.7).

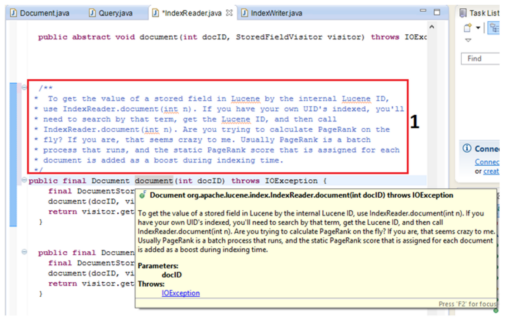

Fig. 5. CODES [150] generates JavaDoc from related discussions on Stack Overflow.

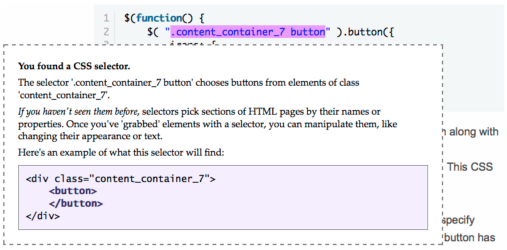

Fig. 6. Tutorons [63] generates context-relevant documentation for explaining complex options and syntax, such as for the `wget` command, REGEX queries, and CSS selectors.

Table 2. Novak et al. [111] identify several dimensions characterizing the capabilities of static analysis tools.

Dimension	Description	Approaches
Rule	Types of rules which may be checked	Style, Naming, Concurrency, Exceptions, Performance, Security, SQL, Maintainability, Correctness
Configurability	Ability to customize tool	Text document, GUI, XML, Rulesets
Technology	Analysis techniques used to identify rule violations	Dataflow, Syntax, Theorem proving, Model checking
Extensibility	If the tool can be extended with customized rules	Possible, Not possible
Developer Experience	How developers interact with the tool	Environment integration, Automatic Locating errors in code, Extensive help on faults, User interface, Command Line, GUI
Input	Types of files that can be loaded into tool	Source Code, Byte Code
Output	Presentation of the results from tool	HTML, XML, List, Text

5.1 Documentation Tools

Design decision documentation tools support systematic documentation of design decisions. While their primary purpose is to capture design decisions, documentation tools may also provide features to facilitate editing and updating design decisions over time, such as by linking design decisions to code [65, 104] (Figures 9 and 14), visualizing the relationships among decisions [89] , or generating documentation using external resources [150] (Figure 5).

Documentation tools vary in how a decision is represented, which may be **code-based** or **on-demand** [124]. In code-based documentation tools, design decisions are formulated as code patterns (e.g., ActiveDocumentation [104]), with a focus on lower-level decisions. In contrast, on-demand documentation tools are responsive, generating documentation from external resources based on a user query. For example, CODES [150] searches for related discussions in Stack Overflow and inserts found descriptions as JavaDoc documentation in the code (Figure 5). Tutorons [63] responds to user queries to generate code-specific documentation explaining complex options and syntax using prepared templates (Figure 6).

Related Survey. While we focus on tools for documentation in this survey, prior work has surveyed techniques for documentation. Alexeeva et al. [3] conducted a systematic literature review of approaches for documenting design decisions. They classified documentation techniques across five dimensions: goal, formalism for classifying decisions (e.g., properties or free-text), which decisions are documented, artifacts included with the decision, and tool-support.

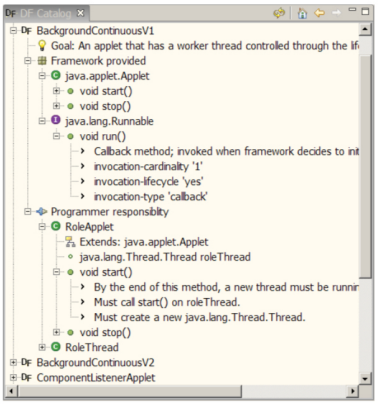

Fig. 7. In Design Fragments [47], developers write down design rules about framework interactions, which are then checked against code.

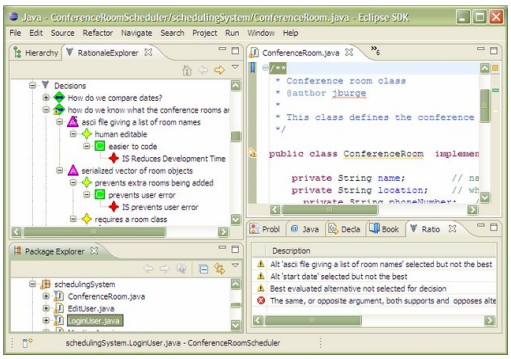

Fig. 8. SEURAT [25] enables documenting design rationale through a tree representation (left) and linking it to code (right).

5.2 Static Analysis Tools

Static analysis tools check code for conformance against pre-defined rules [151]. Static analysis tools may be used to check consistency between a design decision and code by checking that it follows the design rule reflecting the chosen alternative. Static analysis tools vary widely in their capabilities, including the types of rules they may check, their extensibility to new rules, the languages they support, and the developer experience of interacting with the tools (Table 2).

Static analysis tools are generally pre-configured to check for specific **universal** design rules which reflect constraints that should hold across all projects (e.g., PMD [38], FindBugs [66], and CheckStyle [26]). An *extensible* static analysis supplements this with support for developers to themselves author custom rules. These rules may document and check design rules imposed by design decisions. Tools support a variety of techniques and notations by which developers may author project-specific design rules. Some offer libraries for writing rules as code while others use specialized notations. Rules may be expressed as constraints on information collected while traversing the Abstract Syntax Tree (e.g., CheckStyle, PMD), XPath queries (e.g., PMD), or XML (e.g., Design Fragments [47], (Figure 7). Some tools (e.g., Tricorder [129] and RulePad [105]) specifically emphasize their ability to empower all developers to write their own rules.

Related Survey. In this survey, we categorize static analysis tools based on the scope of rules they check. Prior work [90] categorized 25 program analysis tools based on their popularity in research and search engines, their purpose, and the programming languages they support.

5.3 Design Rationale Tools

Design rationale tools enable developers to make the reasoning behind design decisions explicit, by documenting, organizing, maintaining, and reusing rationale. Design rationale tools may be **model-based**, using an explicit argumentation model which describes alternatives and criteria used to choose between them, or **artifact-based**, linking to existing artifacts which include unstructured textual descriptions of rationale.

Model-based design rationale tools enable developers to explore documented rationale through visual diagrams or textual representations. Tools often represent the design rationale explicitly, identifying alternatives that were considered and the criteria with which a developer selected an alternative through an argumentation model (Section 4). Many tools support nesting design

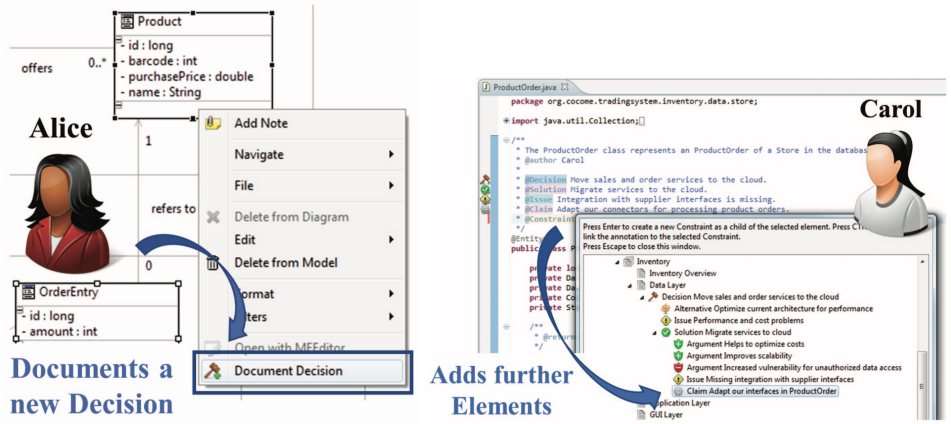

Fig. 9. DecDoc [65] documents design decisions through UML diagrams (left) and code annotations (right).

decisions, where an alternative may have design choices within it. Tools may display the argumentation model of alternatives and criteria through a visualization. Early tools such as InfoRat [23] display argumentation models using a textual representation through lists of items. gIBIS [37] uses the IBIS model to visualize design rationale as a graph, with nodes for issues (design choices), positions (alternatives), and arguments (Figure 2). SEURAT [25] lists design rationale information (including decisions, alternatives, and arguments) as levels in a tree (Figure 8). DecDoc [65] uses the Documentation Decision Model (DDM) [64] to capture design decisions through two separate approaches: architectural design decisions UML diagrams and design decisions in code through annotations (Figure 9). As graphs visualizing rationale may become large and cluttered, tools may make the diagrams zoomable (e.g., gIBIS) or collapsible (e.g., SEURAT), displaying details on demand (Figure 8).

Tools may also support the process of evaluating design rationale. For example, SEURAT [25] enables developers to annotate code fragments relevant to specific design issues, creating two-way links between rationale and code. Completeness and consistency is checked for violations of syntactic rules (e.g., multiple alternatives are selected) and semantic rules (e.g., the same argument is used for and against an alternative), with notifications of potential violations of requirements or when chosen alternatives may be non-optimal.

In contrast to model-based tools which explicitly represent alternatives, artifact-based tools offer links to existing artifacts containing text describing design rationale. Design rationale may be discussed on communication platforms such as chat and email. However, these artifacts are often disconnected from code and may be lost or forgotten. To support the use of these artifacts to explain rationale, artifact-based design rationale tools such as CodeLink [157] and REACT [4] organize design rationale information, automatically extracting discussion of rationale or enabling developers to annotate these discussions in non-code artifacts. CodeLink organizes emails containing rationale, making them easier to find by processing them to extract development contexts (i.e., time, project information, the author's task, and the author's development environment) and identify relevant source code. REACT enables developers to annotate design rationale elements (design choices, alternatives, pro-arguments, con-arguments, and decisions) in Slack.

Related Survey. In this survey, we classify design rationale tools into model-based and artifact-based tools. Prior work categorized early model-based design rationale tools, introduced between 1970 to 2000, as process or feature-oriented, identified an argumentation model (e.g., IBIS, QOC),

examined whether rationale was automatically extracted or captured from developers, and identified an information retrieval process (navigation, automatically triggered, query-based, or hybrid) [122].

5.4 Design Pattern Catalog Tools

Design pattern catalog tools systematically collect known design patterns introduced for common problems, enabling a new decision to be documented and explained by referencing a prior generalized decision (e.g., adopting the Strategy pattern). Catalogs represent design patterns using structured templates which enumerate attributes, facilitating search and exploration. Most famously, the "Gang of Four" introduced a catalog of design patterns for object-oriented design. They described each pattern through twelve properties, including jurisdiction (elements where the pattern applies), characterization (the pattern's functionality), intent (the rationale of the pattern), and motivation (a scenario where it applies) [55]. Later templates introduced a variety of additional properties, such as solution principle and run-time behavior [27, 43]. Catalog tools may also capture architectural patterns. Catalog tools may support extension to other types of patterns by modifying the underlying templates (e.g., SEURAT_Architecture [154]).

By representing design patterns through structured templates, catalog tools enable browsing and search along the enumerated dimensions. For example, DRIMER supports identifying and understanding patterns by integrating design patterns, examples, and their rationale in a unified model called Design Recommendation and Intent Model (DRIM) [116]. SEURAT_Architecture [154] integrates rationale management with design patterns by recording attributes including positive and negative consequences.

5.5 System Architecture Tools

System architecture tools enable developers to document architectural design decisions and check code against the constraints they impose. Two influential approaches are Reflexion Models (RM) [76] and Dependency-Structure Matrices (DSM) [130]. In RM tools, components are extracted from the code, visualized, and checked for conformance against a specified, intended architecture. For example, SAVE [76] visualizes the system as a graph, where nodes represent the system components and edges represent dependencies (Figure 10). Violations of constraints are illustrated with icons on edges. Like Reflexion Models, DSM tools capture dependency structure, but display dependency structure in a matrix form. The rows and columns correspond to system components, and each cell in the matrix denotes dependencies between components. For example, in Lattix's Dependency Manager tool [83], rows and columns correspond to classes, and cells count directed dependencies between classes. Constraints on dependencies between classes may be defined by marking cells as allowed or forbidden, with violations reported through small triangles shown on cells (Figure 11).

5.6 Reverse Engineering Tools

Reverse engineering tools extract design artifacts from a program to support developers in specified tasks [34]. In this survey, we focus on reverse engineering tools which might be applicable to detecting potential design decisions in code. Reverse engineering tools may help developers infer design decisions by identifying frequently co-occurring code fragments, suggesting the presence of a design rule associated with a design decision. These tools assume that the more a pattern is repeated, the more likely that pattern is intentional and reflects an underlying design rule. Reverse engineering tools may also be used to detect design patterns. Working from a predefined set of attributes defining design patterns, tools may identify instances of these in code.

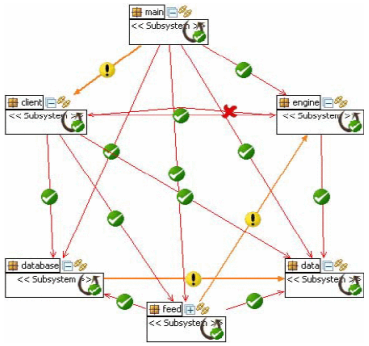

Fig. 10. SAVE [76] visualizes dependencies between system components, indicating potential violations with an edge icon.

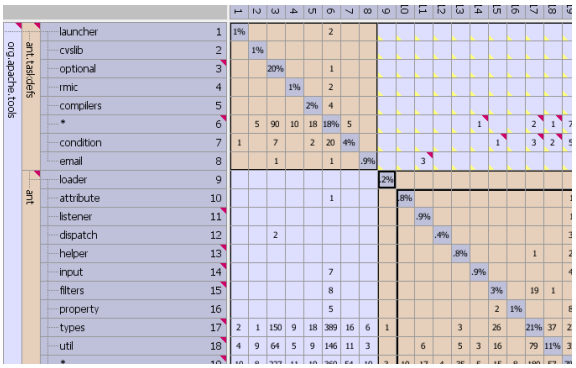

Fig. 11. Dependency-Structure Matrix tools visualize component dependencies in a matrix and indicate violations of dependency constraints (cells with triangles) [83].

5.7 Software Query Languages and Tools

Software query languages and tools enable developers to test a hypothesized design rule. Developers formulate a design rule as a query and then examine identified example code fragments that follow or violate the rule. Techniques vary in the query language used to express design rules, including general-purpose programming languages, domain-specific languages, and semi-natural languages.

General Purpose Programming Languages. Using general purpose programming languages to write queries reduces the required learning curve for the developer to learn the query language but adds complexity and overhead when writing complex queries. EG [10] supports extracting common and idiomatic examples for a given API method, but with queries limited to API methods. ArchJava [2] offers an architectural description language for checking architectural decisions against code, extending Java by adding constructs to define architectural components, connectors between components, and ports (sets of method calls).

Domain-Specific Languages. Domain-specific languages offer languages specifically designed for expressing queries, with widely varying complexity and expressiveness. Some query languages like Semmle.QL [39] are similar to well-known languages such as SQL, reducing learning barriers. For example, a Semmle.QL query to find classes that only implement the `compareTo` method without implementing the `equals` method is written as:

```
from Class c
where c.declaresMethod("compareTo") and not(c.declaresMethod(”equals”))
select c.getPackage(), c
```

Other query languages use novel notation, such as XPath in PMD [38], SOUL [40] (a functional language), and Acme [60] (an architecture description language). While offering considerable expressiveness, these languages require users to learn a new and often complex language to write queries. For example, a PMD XPath query to find classes without `private` fields is written as:

```
//ClassOrInterfaceDeclaration[count( ClassOrInterfaceBody/ClassOrInterface BodyDeclaration/
FieldDeclaration[@Private="true"])=0]
```

A SOUL query to find `bar` methods in classes is written as:

```
if jtMethodDeclaration(?m) {
public ?type bar(?paramList) { ?statements };
}
```

And an Acme query checking that a connection C1 between components A and B implies a connection C2 is written as:

```
(forall b :! B in sys.components | (forall cnp :! P1T in b.ports | (forall a :! A in
sys.components | (forall cnr :! P2T in a.ports | (connected (cnp, cnr) -> (exists cqr :!
P3T in b.ports | (exists cqp :! P4T in a.ports | connected (cqr, cqp)...)
```

Semi-Natural Languages. Natural language is an expressive and natural approach for writing queries. However, natural language is often ambiguous and difficult to interpret. Therefore, many tools use specifically designed semi-natural languages that seek to remain close enough to natural language to benefit from their naturalness and expressiveness while introducing structure to reduce ambiguity. Browse-By-Query (BBQ) [15] and srcQL [11] use semi-structured grammars for expressing queries. A BBQ query to find `bar` methods in classes is written as:

```
matching "bar" methods in all classes
```

A query in srcQL to find a function which contains an `fopen()` followed by an `fclose()` on the same variable is written as:

```
FIND src:function CONTAINS $X = fopen() FOLLOWED BY fclose($X)
```

Architectural design rules may also be specified through semi-natural language. For example, a design rule in Dictō [33] for the model-view-controller architectural style may be specified as:

```
Test=Package with name:"com.app.Test"
View=Package with name:"com.app.View"
Model=Package with name:"com.app.Model"
Controller=Package with name:"com.app.Controller"
Test, View can only depend on Model, Controller
```

6 DEVELOPER GOALS WHEN WORKING WITH DESIGN DECISIONS

To examine how developers can be better supported when working with design decisions, it is important to consider the **goals** they developers hope to accomplish. A goal specifies an objective that an individual seeks to achieve through a series of mental or physical activities [61, 97]. In programming tasks, goals encompass the tasks developers do (e.g., fix a defect, migrate to a new library version) as well as the questions and subgoals necessary to complete these tasks (e.g., determine the cause of a defect, determine the assignment statement which last wrote to a field). Goals often describe an objective by the developer to implement a change to a program or to gather information [79, 109]. In this survey, we consider goals developers may have when working with a design decision, examining the situations where developers may work with a design decision and how tools might effectively support these situations.

Broadly, a design decision can be thought of as transitioning through a lifecycle. It is first created, impacts subsequent work, and, eventually, may be revisited and changed. To make a design decision, a developer first formulates a problem. For many programming problems, there may be may be more than one solution, each with positive and negative consequences [95]. Developers may collect and evaluate each alternative, considering their advantages and disadvantages as well as existing design constraints [16]. After a design decision is made, it may be *captured* by the developer in, for example, a design document so that future developers can understand it [114] and prevent knowledge vaporization [30]. However, in practice, many design decisions remain *uncaptured*, undocumented, and tacit in the heads of decision makers [70, 147]. Over time, to understand, maintain, and reuse code, developers need to understand related design decisions [127] to follow and to be consistent [77, 137]. To detect uncaptured design decisions, developers may apply reverse engineering techniques to find implicit design decisions in code [13], or may conjecture design decisions by generating hypotheses and testing them by examining the code [138]. When

Table 3. When working with design decisions, developers seek to accomplish a number of distinct goals.

	Goal	Example
Goal 1 (Section 7.1)	**Identify potential alternatives**	*How should functionality be decomposed into classes to achieve extensibility and maintainability?*
Goal 2 (Section 7.2)	**Select an alternative as a design decision**	*Is the best alternative for this situation the Command Pattern or Publish/Subscribe?*
Goal 3 (Section 7.3)	**Document the chosen alternative**	*Communicate the design decision of selecting the Command pattern to future developers through documentation.*
Goal 4 (Section 7.4)	**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 (Section 7.5)	**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 (Section 7.6)	**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.*

considering changing design decisions, developers often seek to understand their rationale [70, 107, 137], which is the most frequently reported category of hard-to-answer questions by developers [81].

From this, we identified six key goals developers may have in working with design decisions: 1) identify potential alternatives, 2) select an alternative as a design decision, 3) document the chosen alternative, 4) check a hypothesized design decision against code, 5) find and follow relevant design decisions, and 6) determine why an alternative was selected (Table 3).

7 SUPPORTING DEVELOPER GOALS

While many tools are most closely associated with a single goal (the primary task identified in Table 1), many offer at least partial support for a wider variety of developer goals. In this section, we consider how a wide variety of approaches may offer support for each of the goals we pinpointed in the previous section and the challenges that may exist in achieving this support. Table 4 summarizes the support tools may offer developers in achieving these goals and the challenges developers may have when using these tools.

7.1 Goal 1: Identify potential alternatives

When developers face a problem writing code, they must find a solution [135]. To this end, developers may identify a range of potential solutions for further examination [16, 95] from multiple sources, including novel solutions [28, 110] as well as reusing solutions and ideas from other projects [52, 102, 148, 149] and developers' own past experiences [100]. Developers may also brainstorm and discuss ideas with their team [100]. Solutions may also be adapted from solutions to similar problems found within a developers' own project [32, 100]. Developers may also draw on solutions to more general problems, such as those written in tutorials, blog posts, books, or other mediums.

Tools may support developers in finding potential alternatives to a design problem, either by helping developers identify and understand similar decisions made within their own project or by helping identify general solutions to problems to adapt and reuse. For example, design rationale tools might help developers by recording a number of alternative solutions that they have considered in the past [127]. The following subsections describe the ways in which different tools may offer support in achieving this goal.

7.1.1 Documentation Tools. Documentation tools document design decisions with information which may inform developers about their potential applicability to future design problems. Documentation tools may help developers look for related design decisions [156] and understand the rationale behind these choices. For example, in ActiveDocumentation [104], design decisions are associated with labels to help developers locate decisions relevant to a specific topic. Developers

Table 4. Existing developer tools help developers achieving different goals involving design decisions.

	Goal 1: Identify potential alternatives	Goal 2: Select an alternative as a design decision	Goal 3: Document the chosen alternative	Goal 4: Check hypothesized design decision against code	Goal 5: Find and follow relevant design decisions	Goal 6: Determine why an alternative was selected
Documentation Tools	+ Developers may search and reuse documented decisions if applicable. - Usefulness of tools depends on the existence, clarity, and correctness of the documentation. - Developers need to map from the language used in the tool to that in their code.	+ Developers may reuse documented alternatives if applicable. - Usefulness of tools depends on the existence, clarity, and correctness of the documentation. - Developers need to map from the language used in the tool to that in their code.	- No Support.	+ Developers can look up the related design decisions if documented. - Usefulness depends on the existence, clarity, and correctness of the documentation. - Developers need to map from the language used in the tool to that in their code.	+ Developers can look up the related design decisions if documented. - Usefulness of tools depends on the existence, clarity, and correctness of the documented decisions.	+ Developers can look at rationale of documented design rules in the tool. - Usefulness depends on the existence, clarity, and correctness of the documented decisions.
Static Analysis Tools	- No Support.	- No Support.	+ Developers may document the chosen alternatives if the tools support custom rule authoring. - Developers need time, effort, budget, and motivation to use the tools. - Developers need to have specialized knowledge and expertise to write checkable rules.	+ Developers can author design rules in the tool and use the tool to check the code for conformance. - Developers need specialized knowledge and expertise to write checkable rules and interpret the results.	+ The tool checks the code against the rule for conformance. - Usefulness depends on the existence, clarity, and correctness of the documented decisions.	+ If recorded, developers can examine the rationale of design rules using the tool. - Usefulness depends on the existence, clarity, and correctness of the documented decisions.
Design Rationale Tools	+ Developers may search for and reuse documented decisions, if they exist. - Usefulness depends on the existence, clarity, and correctness of the documented design rationale. - Developers need to map between the language used to describe rationale and that in their code.	+ Developers may reuse documented alternatives, if applicable. - Usefulness depends on the existence, clarity, correctness, and language of the documented design rationale. - Developers need to map from the language used in describing rationale to that in their code.	+ Developers can document the decision rationale using the tool. - Developers need time, effort, budget, and motivation to use the tools.	+ Developers can look up decisions documented in the tool. - Usefulness depends on the existence, clarity, and correctness of the documented design rationale. - Developers need to map from the language used in describing the rationale to that in their code.	+ Developers can look up decisions documented in the tool. - Usefulness depends on the existence, clarity, and correctness of the documented design rationale. - Developers need to map from the language used for in describing rationale to that in their code.	+ Developers can use the tool to examine the rationale of design rules. - Usefulness depends on the existence, clarity, and correctness of the documented design rationale. - Developers need to map the language used for the rationale and their code.
Design Pattern Catalog Tools	+ Developers may search and look at catalog of design patterns, and find design pattern that addresses challenges. - Usefulness of tools depends on the existence, clarity, and correctness of the documented design patterns. - Developers need to map the from the identifiers using in the design patterns to those in their code.	+ Developers can see consequences defined for design patterns in the catalog. - Usefulness depends on the existence, clarity, and correctness of the documented design patterns. - Developers need to map from the identifiers used in the design patterns to those in their code.	+ Developers who are familiar with the design pattern catalog may use the identifiers to find the patterns in code. - Developers need time, effort, budget, and motivation to use the tools.	+ Developers who are familiar with the design pattern catalog may use the identifiers to find the design patterns in code and make changes consistently. - Usefulness depends on the existence, clarity, and correctness of the documented design patterns. - Developers need to map design pattern identifiers to their code.	+ Developers can use identifiers to match the design patterns in the catalog and to make changes consistently. - Usefulness depends on the existence, clarity, the correctness of the documented design patterns. - Developers need to map design pattern identifiers to those in their code.	+ Developers may go to the catalog to see why an alternative might have been chosen. - Usefulness depends on the existence, clarity, and correctness of the documented design patterns. - Developers need to map from the language used in the documented rationale to their code.
System Architecture Tools	- No Support.	- No Support.	+ Developers can configure the intended architecture in the tool. - Developers need time, effort, budget, and motivation to use the tools.	+ Developers can test hypothesized decisions. - Usefulness depends on the existence, clarity, and correctness of the documented decisions. - Developers need to have specialized knowledge to test decisions and interpret the results.	+ The tool visualizes the system architecture and shows inconsistencies with the intended architecture. - Usefulness depends on the existence, clarity, and correctness of the documented decisions. - Developers need to interpret the models.	+ Developers may look up the rationale if documented. - Usefulness depends on the existence, clarity, and correctness of the documented rationale.
Reverse Engineering Tools	+ Developers can identify and compare existing alternative implementations of a behavior. - Developers need to interpret the results. - The results depend on the tool configurations.	- No Support.	- No Support.	- No Support.	+ Tools may help find relevant design rules and identify any inconsistencies. - Developers need to interpret the results. - Tools may produce false positives.	- No Support.
Software Query Languages and Tools	+ Developers can write queries to look for and compare existing implementations of a behavior. - Developers may need specialized knowledge to write queries and interpret the results.	- No Support.	- No Support.	+ Tools may help check for hypothesized decision in code. - Developers may need specialized knowledge to write queries and interpret the results.	+ Developers may formulate queries and compare results to find inconsistencies. - Developers may require specialized knowledge to write queries and interpret the results. - Tools may produce false positives.	- No Support.

may then read descriptions of design decision to understand their rationale and find code examples which illustrate decisions.

7.1.2 Design Rationale Tools. Documented design rationale may help developers in reusing design decisions [44]. Model-based design rationale tools document design rationale by capturing design choices, alternatives, arguments for or against the alternatives, and rationale (Section 5.3). If design decisions share a similar context, their alternatives may be applicable [87]. Documented alternatives may potentially be reused when a developer makes new design choices [19, 102, 116]. However, arguments used in making a past design choice may no longer apply, as these may depend on the specific context of past design choices [87]. Even when alternatives do not apply directly, they may inspire new alternatives [28, 100].

7.1.3 Design Pattern Catalog Tools. Design patterns offer a solution to a problem in a context [55], offering developers a menu of readily available alternatives with which to solve problems. For example, to implement an element which may take one of multiple forms (e.g., graphical elements in an editor), the developer can look at the Gang of Four patterns for *object jurisdiction* and *structural purpose* [56]. Design pattern catalog tools support working with patterns, describing each with properties including their type, scope, intent, and motivation [52]. Many design pattern catalog tools offer features to facilitate the search and exploration process. A developer may browse patterns supporting the *intent* of creating objects in DRIMER [116]. Tools such as SEURAT_Architecture [154] visualize design patterns and their properties in tree structures and maintain links to code fragments where the patterns are applied.

7.1.4 Reverse Engineering Tools. Reverse engineering tools transform code into abstract forms and extract information to detect unknown design rules and design patterns, which may be used as alternatives for new design decisions. In many tools, extracted potential design rules are presented as code fragments without description or rationale (e.g., [139]), and it is left to developers to evaluate the fragments and infer a design rule and rationale. To do this, developers may examine the mined code fragments in the context of the codebase [138, 139]. Developers may then consider if the discovered design rules are solutions to their design problem. For example, a tool might extract code fragments which suggest to the developer the presence of a factory or prototype pattern, which the developer might then consider as alternatives when deciding how to support extensibility.

7.1.5 Software Query Languages and Tools. Software query languages and tools enable developers to query code to answer questions. Using these to identify alternatives may require more knowledge of the codebase than other tools, as the developer must first formulate an idea of a design or implementation to create a query. For example, a developer considering how to support extensibility might use a software query language to look for implementations of the Factory or Prototype pattern in their codebase.

7.1.6 Challenges. A central challenge with documentation and design rationale tools is the **quality** (e.g., clarity and correctness) of the content developers write down. If the documentation is vague, incomplete, or incorrect, it may mislead developers and create more harm than good [91]. Creating documentation is usually considered a burden imposed on developers [69, 106], and developers often skip documenting or maintaining the documentation of design decisions [91]. As a result, documentation is often missing, outdated, or incomplete [91], and many assumptions made when making decisions remain undocumented [59] (see Section 7.3.5). It is crucial for tools to consider ways in which the burden for developers in creating documentation might be reduced, such as by reducing the effort to create the documentation or providing immediate benefits that the authors of documentation receive in completing their task.

Another challenge developers face is bridging the gap between the concepts and **language** they use in framing their new design problem and the language and the vocabulary used in describing existing problems in documentation, catalogs of design patterns, and other artifacts. Concept assignment (matching human-oriented concepts to code or context) [17], the potential for vocabulary mismatch (differences in word choices) [54], and domain-specific vocabulary used in existing artifacts are key barriers to reusing information within existing artifacts [59]. In documentation tools and design rationale tools, documented decisions and alternatives are often described with system-specific terms and context [102, 131]. This may make it hard for developers to understand how to generalize and relate these decisions to the design problem they face. In contrast, design pattern catalogs take a step towards instead employing a generalized and standardized vocabulary. But there may still be challenges in adapting this terminology back to the context of the problem the developer is considering.

Reverse engineering alternatives from existing code brings additional challenges. Rather than produce alternatives which can be readily considered, these tools instead produce code snippets. Developers must then examine the code snippets to infer alternatives, which may require additional **knowledge and effort**. To write queries using software query languages and tools, developers need knowledge about the design and implementation of the code (Section 7.1.5) and may need to learn new specialized domain-specific languages (Section 5.7), knowledge which developers may lack [105]. In reverse engineering tools, **technical limitations** can cause challenges as the quality of the results depend heavily on the tool configuration, including the criteria used to find matches and specific threshold values chosen (e.g., [12, 92]). For example, reverse engineering tools based on code clone detection techniques may ignore code fragments with few occurrences [12]. In software query languages and tools, developers find code snippets by writing queries, and as such, incorrect queries lead to incorrect results.

Takeaways. When searching for potential alternatives, developers can leverage previously documented design decisions and the alternatives they encode from various sources. If decisions have been documented, they might leverage those in design rationale, documentation, or design pattern catalog tools. However, this may bring challenges with mapping the language used in documentation to that in the problem the developer is currently considering. Developers can also identify potential alternatives by examining existing code, including theirs, and they may receive some assistance from tools like reverse engineering and software query languages and tools. However, these tools only produce code snippets, not alternatives, and developers need to themselves have the knowledge necessary to infer the design alternative from the code snippet, a process which may also require considerable effort.

7.2 Goal 2: Select an alternative as a design decision

After identifying potential alternatives, developers must select one of the alternatives as the solution to their design problem. To choose a solution, developers need information about each alternative to understand its consequences. Developers consider and compare alternatives, weighing their consequences, and may externalize their thinking through whiteboards or notes [101], In choosing an alternative, developers may consider the concerns addressed, impacted decisions, assumptions and decisions that limit viable alternatives, and pros and cons of choosing each [16, 145].

A variety of developer tools and techniques may help support choosing between potential alternative solutions to a problem. Tools such as documentation tools enable developers to evaluate and reuse existing design decisions. These tools collect design decisions with information such as a description or rationale which help developers maintain consistency in decision-making and support

principled decision making [136]. Design rationale helps developers make important architectural decisions more rigorously and methodically [49]. Developer tools such as design rationale tools support developers in selecting an alternative by gathering information on alternatives, such as a description and consequences, in structured formats. By including information about rationale and consequences, design pattern catalog tools may assist developers in identifying advantages and disadvantages of applying a design pattern for a specific problem.

7.2.1 Documentation Tools. Code-based documentation tools record design decisions with information such as descriptions and rationale supporting developers in evaluating and selecting an alternative. For example, in ActiveDocumentation [104] each design decision is documented with a title, a description that may contain the rationale, labels specifying the scope of design decisions, and violated and example code snippets extracted from a codebase. When considering documented decisions as alternatives for another design choice, ActiveDocumentation helps developers in evaluating the alternatives by providing information about design rationale in descriptions, and related scopes and concepts through labels.

On-demand documentation tools which provide rich explanations for selected code snippets may also support developers in evaluating alternatives. For example, CODES [150] generates JavaDoc for a method using Stack Overflow discussions, which might contain useful information for evaluating the method when considered as an alternative.

7.2.2 Design Rationale Tools. Model-based design rationale tools record information about design rationale, enabling developers to assess and choose alternatives [22, 44]. In design rationale tools, each alternative is accompanied by opposing and supporting arguments, and if an alternative is reused, the associated arguments can be considered in evaluating the alternative. For example, in tools which apply the QOC model [99], each design choice ("question") is associated with alternatives ("options"), and each option satisfies some "criteria" and violates other criteria. Similarly, in tools applying IBIS [123], each design choice ("issue") is associated with several alternatives ("positions") with advantages and drawbacks discussed in "arguments." Using these, developers may assess each alternative by examining its associated arguments to choose an option.

To help choose alternatives, design rationale tools visually present design rationale information [23, 25, 140], support discussing decisions with teammates [37], and notify developers about missing or incorrectly chosen alternatives [25]. Design rationale tools present documented rationale models through diagrams [25, 140] or textual representations [23]. Design rationale diagrams often are intended to clearly illustrate the relationships among elements in the design rationale model [25, 37, 140]. Textual representations of design rationale information are often used in Model-free tools, offering search features to help developers locate relevant information [4, 157]. In textual representations, the relationship between different elements can be less clear, and it can be hard to obtain a broad understanding of all design rational elements. Some design rationale tools support developers' collaboratively discussing design rationale [37]. This is particularly useful for collaboratively making choices about specific issues, such as high-level design choices which impact several parts of the system and which are maintained by different developer teams [44]. These choices require communication between developers with varying skills and objectives [44]. Finally, tools may also support choosing alternatives by detecting inconsistencies among alternatives. For example, SEURAT examines the arguments of selected alternatives to identify incorrect decisions or arguments early in the design process and evaluate design decisions and alternatives [25]

7.2.3 Design Pattern Catalog Tools. Design pattern catalog tools record design pattern information in a structured format, making it easier for developers to directly compare alternatives across specific dimensions. Pattern catalogs record information such as descriptions, intent, characterizations,

motivation, and implementation details. These attributes may assist developers in evaluating design patterns from a particular perspective. For example, in SEURAT_Architecture [154] developers can compare design patterns by analyzing the attributes affected by each design pattern.

7.2.4 Challenges. Using documentation tools, design rationale tools, or design pattern catalogs to select an alternative may impose challenges on developers. To successfully select an alternative, information about each alternative should be **clear and coherent**. Incorrect documentation may misguide developers. **Language** may also impose a barrier. In reusing and adapting design choices, [102]. developers must understand the details of the design decisions and map the language used in the recorded alternative to the new project context.

Takeaways. Developers evaluate and compare alternatives to make a design decision. To accomplish this, they may examine information previously captured for each alternative. For instance, developers can review the rationale behind an alternative if it is documented in design pattern catalog tools, design rationale tools, or documentation tools. The usefulness of these tools depends on the quality of the documented information, as clear and coherent documentation is necessary to effectively evaluate and compare alternatives.

7.3 Goal 3: Document the chosen alternative

After making a design decision, or discovering an undocumented design decision, developers may document a design decision to avoid knowledge vaporization [62]. Documentation may also help developers to identify potential design alternatives when facing similar problems in the future [142] (Section 7.1.1). Developers may document a variety of information about a design decision, including a description of the decision, often written with project-specific vocabulary, and a rationale describing its reasoning and justification [78, 115, 145].

Tools change the process by which developers document a design decision by prescribing a format and structure in which it is documented. Model-based design rationale tools use models to record design decisions and provide a history of the decision making process. Design pattern catalog tools document the information of design patterns in a structured format, prescribing a template to capture specific attributes. Static analysis tools differ in focusing on documenting the design rule, ensuring the conformance of code to the rule, and focus less on offering rationale motivating the choice. Similarly, system architecture tools document architectural design rules about the allowed dependency structure of code and check code for conformity.

7.3.1 Static Analysis Tools. Extensible static analysis tools support developers in documenting specific types of design rules in a checkable format. Tools such as PMD [38] and FindBugs [66] find inconsistencies in code by documenting design rules, checking code, and reporting violations. Many design rules can be expressed as AST patterns [105], enabling the rule to be documented in a checkable representation. To use these tools to capture project-specific design rules, rather than the general defect patterns for which many of these tools were originally designed, developers must use the extensibility features of these tools to author their own rules. Tools require developers to author rules in special notations or write program analysis code in general purpose programming languages. For example, in PMD developers can author custom rules by writing XPath queries describing a prohibited code pattern [118]. XPath is a query language for XML data, in this case, the XML representation of the AST of the code [1]. To enable developers to write XPath queries, PMD provides PMD Designer [119] that, given input code, produces the XML representation, executes an XPath query on it, and presents the result as code fragments (Figure 12).

[1] https://developer.mozilla.org/en-US/docs/Web/XPath

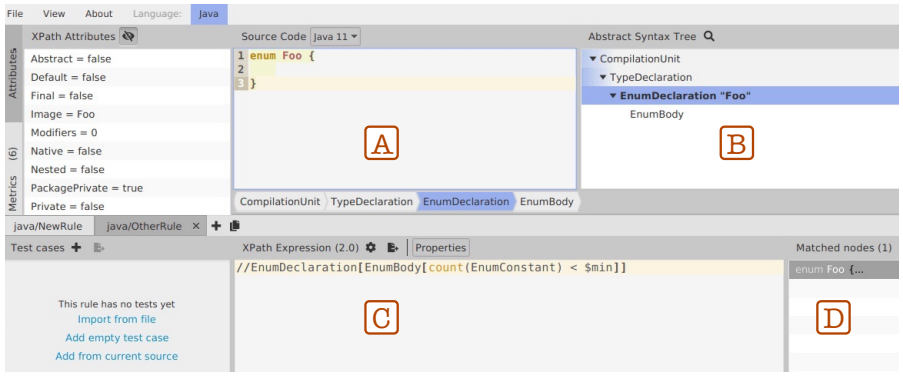

Fig. 12. PMD Designer [119] enables developers to author design rule AST patterns as XPath queries. Developers can view the visualization of the AST (B) of the input code fragment (A). They can use this information to write an XPath query (C) and checks the results of executing the query on the input code (D).

7.3.2 Design Rationale Tools. Design rationale tools enable developers to document information about design decisions, such as considered alternatives and explanations of their decision rationale with supporting and opposing arguments (Section 5.3). Design rationale tools may be model-based or artifact-based, which vary in flexibility and precision. Model-based tools store information in predefined models (Section 4), limiting flexibility in what can be documented but supporting extensive detail [25, 37]. Artifact-based tools document design rationale in unstructured text, offering flexibility but lacking the ability to check that specific information is written down [157].

In model-based design rationale tools, developers document elements of design rationale, which is then visualized through diagrams. Depending on the model, developers may document information such as design choices, alternatives, and relationships between issues or alternatives (Sections 4 and 5.3). For instance, SEURAT [25] uses the DRL model to document decisions, alternatives, arguments for and against each alternative, and links to related code fragments in the codebase.

In artifact-based design rationale tools, tools automatically or through manual steps extract and annotate relevant information from informal artifacts generated by developers. For example, CodeLink [157] collects and processes emails sent by developers to extract design rationale, finding and documenting information such as context and related code snippets from email text.

7.3.3 Design Pattern Catalog Tools. Design pattern catalog tools document design patterns by documenting properties for each design pattern. They are often preconfigured with standard design patterns such as the 26 Gang of Four patterns [55]. Tools may be extensible, enabling developers to document new design patterns by filling out templates. For example, SEUART_Architecture [154] offers a template recording name, type (e.g., architectural pattern or code idiom), description, and consequences.

7.3.4 System Architecture Tools. System architecture tools document architectural design decisions and visualize these through diagrams or matrices. For example, SAVE [76], a reflexion model tool [108], lets developers author design rules constraining dependencies between components and visualizes these in a graph (Figure 10). Lattix [83] depicts dependencies between components using a dependency-structure matrix. Architectural design decisions are documented by setting constraints on matrix cells which allow or forbid specific dependencies (depicted by the yellow corner triangles on cells in Figure 11).

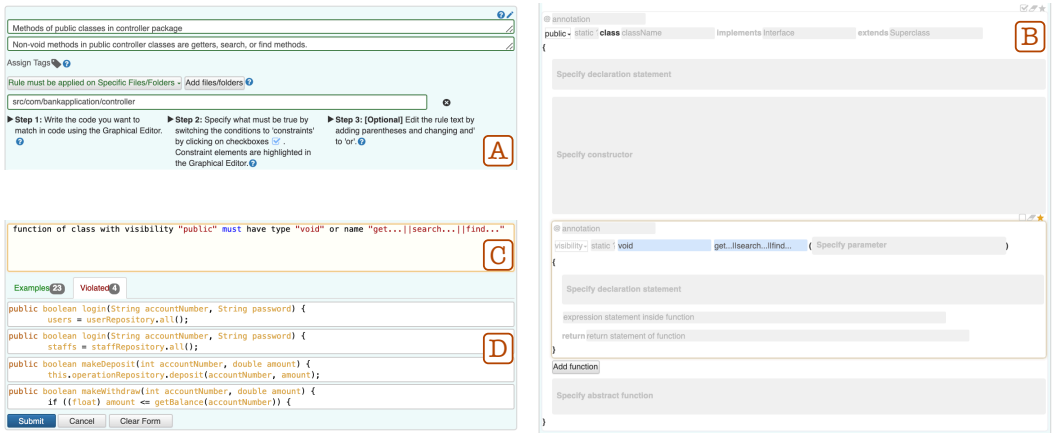

Fig. 13. RulePad [105] enables developers to document design rules in a checkable format using two bidirectionally connected interfaces. (A) Design rules properties are specified and (B) design rules are written in code-like template (C) or in a semi-natural language. (D) The tool offers immediate feedback of matches.

7.3.5 Challenges. In offering developers tool support with which they may document a chosen alternative, the key challenge is motivating developers to choose to document the decision in the first place. Despite admitting the importance and value of documentation, many developers report avoiding documenting decisions [50, 142] or documenting decisions without rationale [44, 48, 106, 141] due to reported barriers with the **time, effort, and budget** required [22, 31, 85, 142]. Developers report that they often have limited time and resources, which creating documentation requires. Many report that they postpone updating documentation, leading to knowledge vaporization [29, 62, 82]. Developers may be poorly **motivated** due to the inadequate rewards developers receive for creating documentation [62, 88]. Documenting decisions may be particularly challenging when the act of documenting is disconnected from the process of making the decision [51]. Developers also report personal reasons to skip documentation, such as being unaware they made a decision at all [30, 62, 142], concealing decisions and their rationale to avoid being challenged [30, 142], or believing that they will not reuse the decisions in the future [30].

Documenting a design rule in a checkable format using static analysis tools brings important advantages in making it possible to check new code against design rules and notify developers when their code is inconsistent. But to do so, developers need special **knowledge**) to use specialized notations with which to write design rules. For example, to write a custom rule in PMD [38], developers need to translate the intended rule into an XPath query or write program analysis code in Java (Section 5.2) [105]. Another challenge is formulating the rule itself (**specialization**) [105]. While developers may have an intuition about the rule and its implications, they may struggle to transform the rule into an abstract format and formulate it as an AST pattern. To address this challenge, RulePad offers other representations of design rules, snippet-based templates and semi-natural language, to help novice and experienced developers author design rules (Figure 13). The code-based template allows non-expert developers author design rules in templates that resemble code, and the semi-natural-language authoring allows experienced developers author design rules as unambiguous and structured textual representations [105].

Takeaways. Once an alternative has been chosen, developers may choose to document information about their decision. This can be achieved using various tools, such as static analysis

tools for documenting checkable design rules, design pattern catalog tools for recording design patterns, or system architecture tools for documenting architectural design decisions. The primary obstacle in documenting decisions is motivating developers to devote the necessary time, effort, and resources. To overcome this challenge, tools may provide incentives by offering benefits not just for understanding the decision later but for the process of making the decision, such as informing developers of code which may violate a decision a developer has just made.

7.4 Goal 4: Check hypothesized design decisions against code

When formulating new design decisions, developers may check these design decisions against the decisions and code which already exist. Tools may offer support to developers by letting them compare a hypothesized design decision against those which have already been documented or to test a design rule associated with a design decision against the code. Despite the existence of tools for documenting design decisions, design decisions often remain implicit and hidden in the code [82]. In these cases, developers may work to reverse engineer them from code or other artifacts. To uncover hidden design decisions, developers investigate artifacts such as the code, hypothesize potential design decisions, and check if these are consistent with the code [138].

7.4.1 Documentation Tools. Using code-based documentation tools, developers may compare a hypothesized decision to design decisions which have been documented. To the extent that documentation is correct, contradictions suggest hypotheses may be incorrect. For instance, if a developer hypothesizes that a set of elements interact through the mediator pattern (one of the GoF patterns [56]), but documentation indicates events are used instead, this might indicate the hypothesis is incorrect. Some code-based documentation tools support developers in finding relevant design decisions. For example, ActiveDocumentation [104] maintains links between code and design decisions and marks related decisions with tags (corresponding to concepts used in code), enabling developers to search for decisions related to specific code or concepts in question.

7.4.2 Static Analysis Tools. In documenting and checking design rules, static analysis tools such as PMD [119] and FindBugs [66] also offer a repository of design rules. As discussed in Section 7.3.1, extensible static analysis tools enable developers to write new project-specific design rules. In this way, static analysis tools may be used to test hypothesized design rules against code.

7.4.3 Design Rationale Tools. Design rationale tools may support developers in comparing new hypothesized decisions against existing documented decisions. Developers may manually check if a hypothesized decision can be inferred from or is consistent with documented design rationale.

7.4.4 Design Pattern Catalog Tools. By capturing and presenting well-known and project-specific design patterns in structured formats, developers may use design pattern catalogs to compare potential new decisions against those which have already been made. Like documentation and design rationale tools, documented information may help developers identify inconsistencies or conflicts between decisions.

7.4.5 System Architecture Tools. For decisions which are architectural and reflect constraints on relationships between elements in a system, system architectural tools may help developers to understand how existing decisions relate to a new hypothesized decision or even to directly test decisions for their consistency with existing code. A diagram or matrix representation of dependency structures in a project may help developers see quickly, at a glance, how new hypothesized decisions relate to those that already exist. For example, a developer hypothesizing that a portion of the system is implemented as a layered architecture, might quickly look for this structure in a DSM

diagram. For tools which implement conformance checks against code, developers may even go a step further and test a new hypothesized decision against code. A developer might formulate a new constraint for the layer and then test code for conformance against it.

7.4.6 Software Query Languages and Tools. Software query languages and tools enable developers to test rules associated with hypothesized decisions by formulating these rules as queries. Tools support developers by executing them against code and inspecting matching code snippets. For this purpose, developers need to formulate their proposed design rules in a format executable by software query languages and tools and examine the snippets that follow or violate the rules.

7.4.7 Challenges. In cases where **documentation** is maintained and updated, tools which help structure and organize this may make it easier for developers to gather relevant information to check if their newly formulated decisions are consistent with code. However, as discussed in Section 7.3.5, decisions are often not documented. When documentation is not available, developers use source code as the primary source of information and reverse engineer design decisions [82].

Tools can offer important support in testing rules associated with decisions against code. But they impose several important barriers on developers. As discussed in Section 7.3.5, developers must translate their newly formulated decisions into checkable formats executable by the tools, which may require **specialized knowledge** to write. Another challenge is **interpreting** the results. Tools often present the results without sufficient explanation (e.g., CloneDR [14]). Developers may have difficulty distinguishing accidental patterns from intentional design decisions.

Takeaways. When exploring code, developers often hypothesize design decisions that may have been made. To verify these hypotheses, developers can compare them against design decisions that have been previously documented in various tools, such as documentation tools, design rationale tools, design pattern catalog tools, static analysis tools, and system architecture tools. However, the usefulness of these tools is contingent on the existence, accuracy, and clarity of the information documented within them. To validate their hypotheses, developers can also inspect the code. They may use tools, such as static analysis tools, system architecture tools, and software query languages, to search for instances of their hypotheses within the code. However, to achieve success, developers must possess specialized knowledge to use these tools to express their hypotheses, as well as to interpret the results.

7.5 Goal 5: Find and follow relevant design decisions

When writing new code, developers need to make sure that it is consistent with existing design decisions. To do so, developers must find and follow the design decisions which have already been made. Failing to find or follow design decisions may lead to developer confusion, decreased comprehensibility of code, and code decay [45, 94] as well as defects. New design decisions may diverge from the original design (architectural drift) or violate the intended architecture (software erosion) [57, 58, 146, 155].

Tools may offer support to developers in finding and following design decisions. If documented (Section 5), tools may support developers in identifying and following relevant documented design decisions. If uncaptured, tools may help developers find and follow decisions by helping to extract decisions from code.

7.5.1 Documentation Tools. Developers can find documented design decisions using search features in IDEs or documentation tools. Documentation tools store design rules in specially designed data stores [38] or JSON files [104]. Design rules can be looked up based on properties such as their description, scope, or rationale. Documented decisions in comments may be looked up through

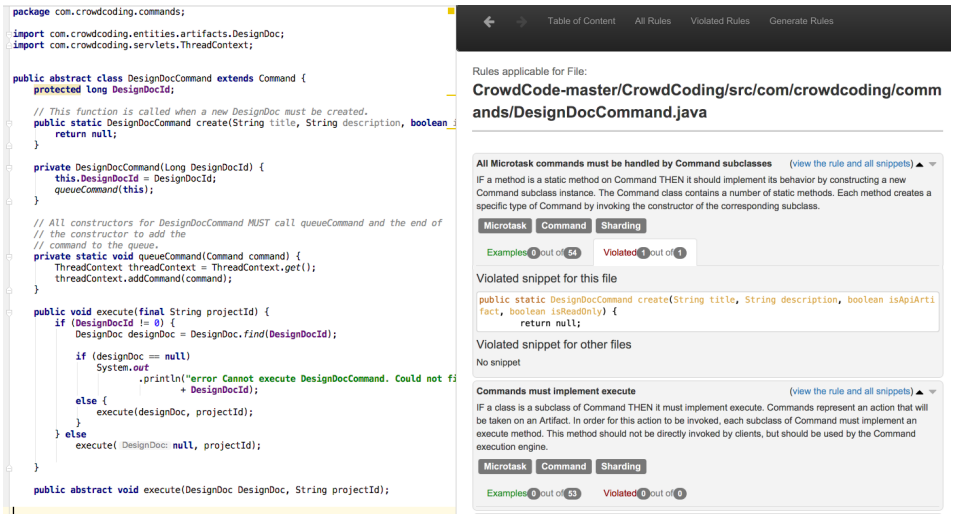

Fig. 14. In ActiveDocumentation [104] developers can find relevant design rules using the labels assigned to each design rules and the filtration feature that displays related design rules for each class.

IDE search features or other suitable IDE plugins. For instance, CODES [150] generates JavaDoc for specifications and descriptions of methods inserted in code, which can be looked up by the same tools used for searching the code.

Documentation tools use a variety of approaches to help developers retrieve relevant design rules, such as through search or by organizing rules. Developers may search for target design decisions through keyword queries. Other tools organize decisions to expedite search. For example, in ActiveDocumentation [104] developers can define labels based on topics applied in code and mark design rules with relevant labels. The labels and related design rules are accessible in the tool by clicking on the tags (Figure 14). Many documentation tools connect design decisions to code to enrich documentation. Maintaining a bidirectional connection between design rules and code helps developers find relevant design rules more easily (Figure 14).

7.5.2 Static Analysis Tools. By connecting design rules to code, static analysis tools enables developers to identify design rules simply by writing code and then running the static analysis tool to check if any design rules have been violated (e.g., PMD [38], FindBugs [66], CheckStyle [26]). After flagging a design rule violation, some tools also provide additional information about how to follow the design rule. For example, ActiveDocumentation [104], lists code snippets that follow design rules, which developers may then use as an example of how to follow the design rule.

7.5.3 Design Rationale Tools. Design rationale tools capture decisions in structured formats (model-based tools) or mark relevant information in non-code artifacts (artifact-based tools). Like documentation tools, model-based design rationale tools provide features for easier retrieval of design decisions. The structure may assist developers in finding design decisions. In addition, most tools include search or browse features to find design decisions through keywords or by intent. Some also connect rationale to code [25], which may support understanding how and where rules should be followed in code. For instance, using SEURAT [25] developers can find design decisions relevant to a code snippet by following links developers created when documenting design rationale.

7.5.4 Design Pattern Catalog Tools. Design pattern catalog tools use a structured representation to collect and categorize design patterns based on properties such as characteristics, jurisdictions, intents, or the problems they solve (Section 5.4). Many design pattern catalog tools allow developers to search for design patterns based on their properties and apply them to their code by following their constraints. Conversely, developers can look up design patterns applied to their code in design pattern catalog tools using the identifiers used in the code. However, mapping applied patterns to catalogs may be challenging, due to a potential for vocabulary mismatch between the terminology in catalogs and terminology used in a project. To address this, some catalog tools may be customized per-project or explicitly link between catalogs and pattern instances in code. For example, in SEURAT_Architecture [154] developers can document information of patterns applied in code and link the patterns to the documentation.

7.5.5 System Architecture Tools. As discussed in Section 7.3.4, design rules on system dependencies may be formulated as constraints over system elements. Developers may study these to understand architectural design decisions. After editing the code, these tools may also help developers uncover design rules by flagging new violations of architectural design rules that have been introduced.

7.5.6 Reverse Engineering Tools. Reverse engineering tools may help developer uncover hidden design rules by helping them extract them from code. Most reverse engineering tools detect repeated code fragments in code according to some predefined frequency threshold, and present them to a developer (e.g., [153]). The Developer then evaluates the results and may *find* implicit design rules in code. For example, code clone detectors may identify a particular method call chain for persisting data using API methods in a controller package, which is a design rule imposed by the API constraining the method call chains for persisting data. By examining these results, developers may learn about the design rule and the appropriate location in the codebase for invoking persistence functionality (i.e., the controller package).

7.5.7 Software Query Languages and Tools. Software Query Languages and Tools may be used to find a design rule and, to some extent, understand how to follow it. Software query languages and tools enable developers to query the codebase to *find* design rules. For example, developers can query for calls to a specific API method to find design rules related to it. This requires developer to interpret and make sense of these results to uncover hidden implicit design rules.

7.5.8 Challenges. To find and follow relevant design decisions, developers can look to identify relevant decisions that have been documented through a tool. However, this imposes several challenges on developers. One key barrier is **maintaining** documented design decisions. As discussed in Section 7.3.5, outdated and incomplete documentation misinforms developers, leading to software drift and erosion. Some tools like ActiveDocumentation [104] alleviates this issue by constantly checking rules against code and providing instant feedback. Another related challenge is the **quality** and comprehensibility of documented design decisions. To follow design decisions, developers first need to understand the decisions, their rationale, their constraints, and their implementation. Unclear descriptions may confuse developers, who may then fail to follow the decisions. Another obstacle in using tools is **motivating** developers to adopt and use the tools to find and follow design decisions. Despite being useful [7], many developers refrain from using tools for non-development tasks such as finding and following design decisions. For instance, studies of the usability of static analysis tools suggests that developers have several concerns, including the presentation of the results, the accuracy of the results (numerous warnings and high rates of false positives), inadequate information and examples for fixing violations, and separate workflows for software development and using the tools [35, 71, 128]. Some approaches mitigate these barriers,

such as by supporting tool customization, more structured and detailed presentations of information, and automating usage of a tool at required development steps, such as Code Review [71, 128].

In addition to these challenges, using design pattern catalog tools have to find and follow design decisions imposes additional difficulties. To find relevant design patterns, developers need to map identifiers in code consistent with the ones used in the catalogs. However, this does not always occur (Section 7.3.5), forcing developers to manually **map identifiers**, which is challenging.

Like design pattern catalog tools, system architecture tools require developers to **interpret** the models and map them to the source code. Model-based system architecture tools –reflexion models and dependency-structure matrices– represent the high-level structure of the codebase through diagrams and matrices. Therefore, if the tools lack enough support in mapping the elements, corresponding high-level elements in the diagrams to lower-level code elements can be difficult for developers. Some tools like Lattix [83] address this issue by visualizing the structure in a hierarchical presentation that can be expanded to show lower-level elements.

Using reverse engineering tools and software query languages and tools to find and follow design decisions depends on the efficiency and accuracy of the algorithms used to extract design decisions and the ability of developers to interpret and analyze the results. Reverse engineering tools and software query languages and tools potentially suffer from **false positives** (incorrectly detected patterns and rules). Some tools support manual or automatic techniques to address this issue. For example, Rasool et al. [121] used annotations for better retrieval of design patterns in code. However, it still relies on developers to maintain the annotations. Dong et al. [41] used behavioral and semantic analysis to eliminate falsely detected candidate design patterns. Another challenge is the reliance on developers to **interpret** and analyze the results. Tools require developers to examine potential design decisions to differentiate accidental patterns from intended design decisions. Therefore, the form of the presentation of extracted information is key.

Takeaways. When modifying code, developers must find and follow previously made design decisions, which may be supported through tools. Documentation tools, design rationale tools, design pattern catalog tools, and system architecture tools provide access to previously captured design decisions. These tools may also link documentation to code or check the code against the decisions. By utilizing these tools, developers may be offered a process to both find a decision as well as information about how to follow it. Additionally, after a decision has already been identified, reverse engineering tools and software query languages may assist in the process of writing new code consistent with a design decision by identifying previously written code that is consistent with a design decision.

7.6 Goal 6: Determine why an alternative was selected

Developers look to explain why an alternative was selected (its rationale) when they wish to re-evaluate or change a decision and understand the impact of these changes [21, 25, 80]. Developers report that answering questions about rationale is one of their most frequent hard-to-answer questions about code [81] and one of the most serious problems they face [82]. Tools have been designed to support developers in answering these questions by capturing rationale information in free form text or structured models, sometimes in association with design rules.

7.6.1 Documentation Tools. Documentation tools capture information about design decisions, including design rationale, in model-free representations as text. In code-based documentation tools like ActiveDocumentation [104], documented design rules may include a field for a description, which may be used to capture design rationale. In on-demand documentation tools like CODES [150], design rationale information is extracted from non-code artifacts such as Stack Overflow discussions.

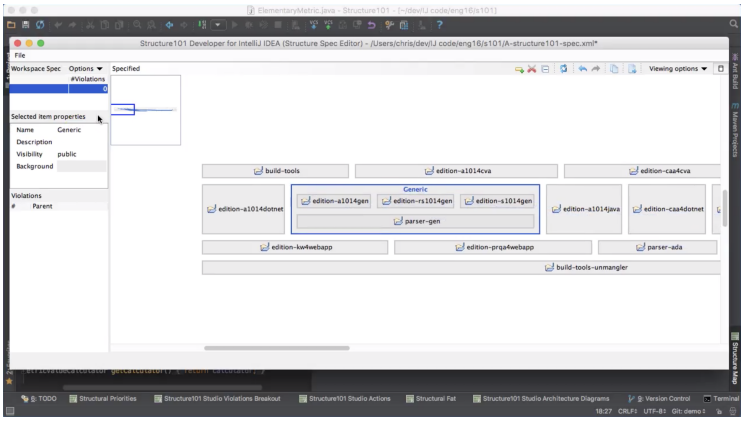

Fig. 15. Structure101 [98] supports describing architectural constraints through diagrams.

7.6.2 Static Analysis Tools. Associated with the design rule checked by static analysis tool may be design rationale that can be used to reason about design decisions. Design rationale may be captured in tools such as PMD [38] and FindBugs [66] and accessed in data stores as well as the IDE views of the tools when the design decision is violated .

7.6.3 Design Rationale Tools. Design rationale tools directly focus on documenting design rationale by recording alternatives, arguments, and other information which may enable reasoning about rationale (Section 4). Using a diagramatic or tree view of rationale, developers may reason about what alternatives were considered and why specific ones were chosen. The information recorded in artifact-based tools may not be as detailed as model-based tools as these tools do not enforce or check that specific information has been recorded. To find captured rationale for a code snippet, developers can query or browse the documented rationale. Tools may also connect design rationale directly to code (e.g., SEURAT [25]).

7.6.4 Design Pattern Catalog Tools. Design pattern catalog tools often use design rationale information to describe the *intent* of a design pattern, describing the motivation of a design pattern, its consequences, and its rationale [55]. For example, developers may explore the reasoning behind or the consequences of selecting a design pattern in SEURAT_Architecture [154], or look for the intent of a design pattern in DRIMER [116].

7.6.5 System Architecture Tools. System architecture tools may also support adding rationale to explain rules. For example, Structure101 [98] enables developers to add descriptions or rationale for patterns, constraints, or architectural elements in a free-form textual field (Figure 15).

7.6.6 Challenges. Tools that rely on developers to document design rationale ultimately rely on developers to first do this. **Documenting** rationale is considered overhead by many developers [48], and it is often easy for developers to ignore [142]. Moreover, in most documentation tools, static analysis tools, and system architecture tools, the information created takes the form of free-form **text**. This flexibility may enable developers to neglect documenting rationale, may make it harder to understand, and may leave it incomplete. The **language** used in describing rationale may also be hard to understand. As software evolves, vocabulary may change and the initial vocabulary used to author the rationale may become dated. Design rationale may also be out of date. Most design rationale tools are stand-alone tools, separated from the code, or provide only one-way links to

code snippets without checking the code or maintaining the links. Developers are responsible for **updating** documented rationale when the code changes, but this may be challenging.

Takeaways. Tools can assist developers in understanding the rationale behind choosing an alternative design. Documentation tools, design rationale tools, design pattern catalog tools, static analysis tools, and system architecture tools can capture information on design decisions, including the reasoning behind them. When it exists and is up to date, developers may leverage this information to better comprehend the reasoning behind design decisions.

8 FUTURE RESEARCH DIRECTIONS

In this article, we surveyed seven types of tools that help developers work with design decisions and six goals developers try to achieve during software development. This survey evaluated how different types of tools can be employed to achieve each goal and examined the challenges developers face when using the tools to achieve these goals. While addressing the challenges discussed in Section 6 paves the way for many future improvements, a number of broader questions and areas of research remain for exploration, which we also survey in the following.

8.1 Better support for documentation

One of the main mechanisms by which tools have been envisioned to assist developers is by facilitating the process of working with captured design decisions. Documentation, static analysis, and design rationale tools all rely on information to be first captured about a design decision at the point in time when a developer makes a decision. However, this approach brings several challenges, as discussed in this survey.

First, developers may not be motivated to document alternatives and design decisions explicitly through tools, as this requires the developer to spend time doing more work. To give developers more payoff for investing the time needed to create this content in documents, tools should be carefully designed to incentivize developers to do this work. One key challenge is that tools today ask developers to do work—write documentation—but only give benefits later—when developers revisit this documentation in the future in order to make future changes that are consistent. One way to address this is to create tools that, while creating documentation, also support the developer at the point in time when they are making the design decision and doing the work to write the documentation. For example, when a developer uses ActiveDocumentation [104] to document a design decision that they are currently making, the tool will immediately give them feedback about all of the points in the code that either follow or violate this design decision. This may be useful for the developer as they make the design decision in helping to identify code that they had not realized needs to be updated and in reducing the investigation work they may need to do to find all of the implications of the design decision they had just made.

Second, documenting checkable design rules in tools such as static analysis tools often requires specialized program analysis knowledge or complex query notations. In practice, developers are unlikely to have this knowledge, and may not be motivated to invest the time necessary to learn it. As a result, these tools today are rarely used in practice to document project-specific design rules. Even for developers that have the knowledge necessary to use these tools, existing tools and notations can make writing design decisions down in the necessary format time consuming, further reducing developers' motivation to document decisions. Tools can play an important role in addressing this barrier. Instead of requiring specialized knowledge, tools can instead use concepts with which developers are already familiar and provide dedicated tool support not just for checking design rules but also the process of creating and authoring design rules. For example, RulePad [105]

addresses these issues by using a semi-natural language in which to author design rules and offering a dedicated editor for crafting design rules.

Third, documentation must be regularly maintained and updated to remain useful and not become outdated. Offering support for updating documentation throughout the life of a software project is at least as important as supporting initial creation of documentation. One way tools may assist developers is by identifying cases where the documentation and code have diverged over time, and notifying the developer that these divergences have been created. Particularly if this can be identified at the point in time in which design decisions are first violated (e.g., when a developer writes new code), it may be faster and easier for the developer to rapidly revise the design decision to reflect their new intent as they are creating a new design decision.

Fourth, tools should be able to capture various types of information, including design rationale. While many existing tools provide free-form text to document information about design decisions, this approach can lead to important information being inadvertently left out. To overcome this issue, tools can use templates that require specific types of information to be recorded for every decision. Additionally, tools can support information management by linking together related design decisions. For instance, in design rationale tools like SEURAT [25], developers can link related alternatives, arguments, and chosen alternatives to help keep track of the decision-making process. This may aid in faster and more effective searches.

Fifth, it is essential for tools to support linking design decisions to code to ensure that developers can follow the intended design. By enabling easy access to the relevant documentation to understand the reasoning behind the decisions made during the design phase, developers may make more informed decisions when modifying code. Moreover, linked documentation is easier to update than disconnected documentation. Changes made to the code can be immediately reflected in the documentation, ensuring that the documentation is always up-to-date [104].

8.2 Better support for reverse engineering

Tools can also assist developers with design decisions by extracting information from code. Reverse engineering tools and software query languages and tools can help find patterns and query codebases, which can aid in finding alternatives in code, checking hypothesized decisions, and discovering design decisions in code. However, as we discussed in this survey, the results produced by these tools often contain false positives and require interpretation, which can be time-consuming and challenging for developers. To overcome these challenges, tools should offer more effective techniques and present results in a way that is tailored to developers' work context, making them easier to understand and apply. For example, tools can prune results only related to the recent activities of developers [72]. Another challenge is that using software query languages and tools requires specialized knowledge to write queries. To address this, tools can provide an intermediate interface or language for writing queries that does not require additional training. For example, using semi-natural languages to write queries can be helpful for novice developers in greatly reducing the necessary training [105].

8.3 Connecting software tools

One of the challenges in using developer tools is that they are often disconnected from other tools. To achieve their goals, developers must often use multiple tools, which reduces their productivity. Existing tools offer only full support to developers in achieving one or two goals and only partial support in achieving other goals. For example, ActiveDocumentation [104] directly supports developers in documenting and following design decisions, but offers only partial support in reasoning about design decisions. As a result, developers may need to rely on separate tools, such as specialized tools such as SEURAT [25] for reasoning about rationale. As these two tools are

disconnected, using them together today would require the developer to document each design decision twice, once in the format for each tool. Offering tools that support a broader range of goals, or at least more integration information sharing between tools, might help increase the benefits of using tools while dramatically reducing the costs.

8.4 Return on documentation investment

To better tailor tools to meeting developers' goals, further research is needed to better understand the information needed in achieving developer goals. Developers constantly make choices about what to document and how much detail to include in documenting it. To the extent that developers document information that is obvious, or fail to document what is not obvious, these choices may cause issues later. Documented information may help developers achieve later goals, but it comes as the cost of the time which must be invested to document it, which developers often find demotivating. Developers may question to what extent all of this documentation is truly worth the effort required, as modern processes such as agile development often eschew it.

8.5 Documenting vs. reverse engineering

There are two main approaches to working with design decisions: explicitly documenting decisions and reverse engineering tacit decisions from software artifacts. Traditionally, developers have been encouraged to write and maintain documentation. But a more recent emphasis has been on tools which support developers in extracting tacit decisions from artifacts. Reverse engineering tools extract information from software artifacts through mining [6, 93, 96, 125] and querying [73] software artifacts. The advantages of using reverse engineering tools to retrieve design decisions are that developers need only work with the tools at the point of need, when they are trying to understand decisions, rather than invest upfront in documentation effort and the maintenance required to maintain this documentation.

But, as discussed in this survey, one key challenge in using mining tools is their heavy reliance on developers to infer decisions from extracted patterns. One may argue that this process is inherently unreliable, and developers should not solely rely on these tools to discover decisions. Better understanding how this tradeoff works in practice is an important area for future research.

8.6 Usability issues

Despite their potential usefulness, many tools remain underused due to usability issues. As described in this survey, issues such as poor reliability, hard to understand results, and disconnected tools demotivate developers from using tools. Several studies have examined the usability issues involved with using static analysis tools (e.g., [35, 71]) and proposed suggestions to mitigate common issues, such as configuring the priority of errors based on development contexts (e.g., local programming, continuous integration, or code review) [152], more structured error messages [9], supporting collaborative environments [112], and offering immediate feedback [144]. But important research remains to more broadly understand the usability issues across the full range of tools for working with design decisions in code.

9 CONCLUSION

Design decisions are central to software, directly determining its correctness, comprehensibility, and maintainability. Design decisions progress through a lifecylce during software development; they may captured by developers when decisions are made or remain uncaptured. There are many open areas for future work, including better motivating developers to document decisions, supporting the process of documenting decisions as well as updating documentation, offering more information when reverse engineering decisions, improving the usability of tools, and improving the integration

between tools which support different developer goals. In addition, further research is needed to better understand exactly what information developers require when achieving their goals, to help refine the information documentation tools ask developers to record as well as the presentation of information which might be reverse engineered. More broadly, important questions remain about if and when it is necessary for developers to take the time to document decisions or the extent to which reverse engineering tools can provide many of the same benefits.

REFERENCES

[1] Mohammed Ghazi Al-Obeidallah, Miltos Petridis, and Stelios Kapetanakis. 2016. A survey on design pattern detection approaches. *International Journal of Software Engineering (IJSE)* 7, 3 (2016), 41–59.

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: connecting software architecture to implementation. In *International Conference on Software Engineering (ICSE)*. 187–197. https://doi.org/10.1145/581339.581365

[3] Zoya Alexeeva, Diego Perez-Palacin, and Raffaela Mirandola. 2016. Design decision documentation: A literature overview. In *European Conference on Software Architecture*. 84–101. https://doi.org/10.1007/978-3-319-48992-6_6

[4] Rana Alkadhi, Jan Ole Johanssen, Emitza Guzman, and Bernd Bruegge. 2017. REACT: An Approach for Capturing Rationale in Chat Messages. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 175–180. https://doi.org/10.1109/ESEM.2017.26

[5] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Computing Surveys (CSUR)* 51, 4 (2018), 81:1–81:37. https://doi.org/10.1145/3212695

[6] Mohsen Anvaari and Olaf Zimmermann. 2014. Semi-automated Design Guidance Enhancer (SADGE): A Framework for Architectural Guidance Development. In *European Conference on Software Architecture*. 41–49. https://doi.org/10.1007/978-3-319-09970-5_4

[7] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs Fixit. In *International Symposium on Software Testing and Analysis (ISSTA)*. 241–252. https://doi.org/10.1145/1831708.1831738

[8] Carliss Young Baldwin and Kim B Clark. 2000. *Design rules: The power of modularity*. Vol. 1. MIT Press.

[9] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How Should Compilers Explain Problems to Developers?. In *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 633–643. https://doi.org/10.1145/3236024.3236040

[10] Celeste Barnaby, Koushik Sen, Tianyi Zhang, Elena Glassman, and Satish Chandra. 2020. Exempla Gratis (EG): Code Examples for Free. In *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1353–1364. https://doi.org/10.1145/3368089.3417052

[11] Brian Bartman, Christian D. Newman, Michael L. Collard, and Jonathan I. Maletic. 2017. srcQL: A syntax-aware query language for source code. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 467–471. https://doi.org/10.1109/saner.2017.7884655

[12] Hamid Abdul Basit and Stan Jarzabek. 2009. A Data Mining Approach for Detecting Higher-Level Clones in Software. *Transactions on Software Engineering* 35, 4 (2009), 497–514. https://doi.org/10.1109/TSE.2009.16

[13] Ira D. Baxter and Michael Mehlich. 1997. Reverse Engineering is Reverse Forward Engineering. In *Working Conference on Reverse Engineering (WCRE)*. 104–113. https://doi.org/10.1109/WCRE.1997.624581

[14] Ira D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance (ICSM)*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

[15] BBQ. 2015. Browse-By-Query. http://browsebyquery.sourceforge.net/

[16] Reza Beheshti. 1993. Design decisions and uncertainty. *Design Studies* 14, 1 (1993), 85–95. https://doi.org/10.1016/S0142-694X(05)80007-9

[17] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1993. The Concept Assignment Problem in Program Understanding. In *International Conference on Software Engineering (ICSE)*. 482–498. https://doi.org/10.1109/ICSE.1993.346017

[18] Joshua Bloch. 2016. *Effective java*. Pearson Education India.

[19] Frances M. T. Brazier, Pieter H. G. van Langen, and Jan Treur. 1997. A compositional approach to modelling design rationale. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 11, 2 (1997), 125–139. https://doi.org/10.1017/S0890060400001918

[20] David C. Brown and Rahul Bansal. 1989. Using Design History Systems for Technology Transfer. In *Computer-Aided Cooperative Product Development, MIT-JSME Workshop*. 544–559. https://doi.org/10.1007/BFb0014295

[21] Bernd Bruegge and Allen H. Dutoit. 2009. *Object-Oriented Software Engineering: Using UML, Patterns and Java* (3rd ed.). Prentice Hall Press.

[22] Janet E Burge. 2005. *Software engineering using design RATionale*. Ph.D. Dissertation. Worcester Polytechnic Institute.

[23] Janet E. Burge and David C. Brown. 2000. Reasoning with Design Rationale. In *International Conference on Artificial Intelligence in Design (AID)*. 611–629. https://doi.org/10.1007/978-94-011-4154-3_30
[24] Janet E Burge and David C Brown. 2004. An integrated approach for software design checking using design rationale. In *Design Computing and Cognition*. Springer, 557–575. https://doi.org/10.1007/978-1-4020-2393-4_29
[25] Janet E. Burge and David C. Brown. 2008. Software Engineering Using RATionale. *Journal of Systems and Software* 81, 3 (2008), 395–413. https://doi.org/10.1016/j.jss.2007.05.004
[26] Oliver Burn. 2004. CheckStyle. http://checkstyle.sourceforge.net
[27] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley Publishing.
[28] Bill Buxton. 2010. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann.
[29] Rafael Capilla, Juan C Dueñas, and Francisco Nava. 2010. Viability for codifying and documenting architectural design decisions with tool support. *Journal of Software Maintenance and Evolution: Research and Practice* 22, 2 (2010), 81–119. https://doi.org/10.1002/smr.419
[30] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 2016. 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software* 116 (2016), 191–205. https://doi.org/10.1016/j.jss.2015.08.054
[31] Rafael Capilla, Francisco Nava, and Carlos Carrillo. 2008. Effort Estimation in Capturing Architectural Knowledge. In *International Conference on Automated Software Engineering (ASE)*. 208–217. https://doi.org/10.1109/ASE.2008.31
[32] Rafael Capilla, Francisco Nava, and Juan C. Dueñas. 2007. Modeling and Documenting the Evolution of Architectural Design Decisions. In *Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK/ADI)*. 9. https://doi.org/10.1109/SHARK-ADI.2007.9
[33] Andrea Caracciolo, Mircea Filip Lungu, and Oscar Nierstrasz. 2015. A unified approach to architecture conformance checking. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 41–50. https://doi.org/10.1109/wicsa.2015.11
[34] Elliot J. Chikofsky and James H. Cross. 1990. Reverse engineering and design recovery: a taxonomy. *IEEE Software* 7 (1990), 13–17. https://doi.org/10.1109/52.43044
[35] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *International Conference on Automated Software Engineering (ASE)*. 332–343. https://doi.org/10.1145/2970276.2970347
[36] E Jeffrey Conklin and KC Burgess Yakemovic. 1991. A process-oriented approach to design rationale. *Human–Computer Interaction* 6, 3-4 (1991), 357–391. https://doi.org/10.1207/s15327051hci0603&4_6
[37] Jeff Conklin and Michael L Begeman. 1988. gIBIS: A hypertext tool for exploratory policy discussion. *Transactions on Information Systems (TOIS)* 6, 4 (1988), 303–331. https://doi.org/10.1145/62266.62278
[38] Tom Copeland. 2005. *PMD Applied*. Centennial Books.
[39] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. 2008. *.QL: Object-Oriented Queries Made Easy*. Springer Berlin Heidelberg, 78–133. https://doi.org/10.1007/978-3-540-88643-3_3
[40] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. 2011. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *International Conference on Principles and Practice of Programming in Java (PPPJ)*. 71–80. https://doi.org/10.1145/2093157.2093168
[41] Jing Dong, Dushyant S. Lad, and Yajing Zhao. 2007. DP-Miner: Design Pattern Discovery Using Matrix. In *International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS)*. 371–380. https://doi.org/10.1109/ECBS.2007.33
[42] Jing Dong, Yajing Zhao, and Tu Peng. 2009. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering* 19, 06 (2009), 823–855. https://doi.org/10.1142/s021819400900443x
[43] Zoya Durdik. 2016. *Architectural Design Decision Documentation through Reuse of Design Patterns*. Vol. 14. KIT Scientific Publishing.
[44] Allen H. Dutoit, Raymond McCall, Ivan Mistrík, and Barbara Paech. 2006. *Rationale Management in Software Engineering*. Springer Berlin Heidelberg, Chapter Rationale Management in Software Engineering: Concepts and Techniques, 1–48. https://doi.org/10.1007/978-3-540-30998-7_1
[45] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. *Transactions on Software Engineering* 27, 1 (2001), 1–12. https://doi.org/10.1109/32.895984
[46] George Fairbanks. 2010. *Just enough software architecture: a risk-driven approach*. Marshall & Brainerd.
[47] George Fairbanks, David Garlan, and William Scherlis. 2006. Design Fragments Make Using Frameworks Easier. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 762–763. https://doi.org/10.1145/1167515.1167480

[48] Davide Falessi, Lionel C Briand, Giovanni Cantone, Rafael Capilla, and Philippe Kruchten. 2013. The value of design rationale information. *Transactions on Software Engineering and Methodology (TOSEM)* 22, 3, Article 21 (2013), 32 pages. https://doi.org/10.1145/2491509.2491515

[49] Davide Falessi, Giovanni Cantone, Rick Kazman, and Philippe Kruchten. 2011. Decision-making techniques for software architecture design: A comparative survey. *Computing Surveys (CSUR)* 43, 4 (2011), 33. https://doi.org/10.1145/1978802.1978812

[50] Davide Falessi, Giovanni Cantone, and Philippe Kruchten. 2008. Value-Based Design Decision Rationale Documentation: Principles and Empirical Feasibility Study. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 189–198. https://doi.org/10.1109/WICSA.2008.8

[51] Gerhard Fischer, Andreas C. Lemke, Raymond McCall, and Anders I. Mørch. 1991. Making Argumentation Serve Design. *Human-Computer Interaction* 6, 3-4 (1991), 393–419. https://doi.org/10.1080/07370024.1991.9667173

[52] Scott D. Fleming, Christopher Scaffidi, David Piorkowski, Margaret M. Burnett, Rachel K. E. Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *Transactions on Software Engineering and Methodology (TOSEM)* 22, 2 (2013), 14:1–14:41. https://doi.org/10.1145/2430545.2430551

[53] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc.

[54] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. 1987. The Vocabulary Problem in Human-System Communication. *Commun. ACM* 30, 11 (1987), 964–971. https://doi.org/10.1145/32206.32212

[55] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *European Conference Object-Oriented Programming (ECOOP)*. 369–378. https://doi.org/10.1007/3-540-47910-4_21

[56] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[57] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. 2013. A comparative analysis of software architecture recovery techniques. In *International Conference on Automated Software Engineering (ASE)*. 486–496. https://doi.org/10.1109/ASE.2013.6693106

[58] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. 2011. Enhancing architectural recovery using concerns. In *International Conference on Automated Software Engineering (ASE)*. 552–555. https://doi.org/10.1109/ASE.2011.6100123

[59] David Garlan, Robert Allen, and John Ockerbloom. 2009. Architectural Mismatch: Why Reuse Is Still So Hard. *IEEE Software* 26, 4 (2009), 66–69. https://doi.org/10.1109/MS.2009.86

[60] David Garlan, Robert T Monroe, and David Wile. 2000. *Acme: Architectural description of component-based systems.* Vol. 68. Cambridge University Press, 47–68.

[61] Howard Garland. 1985. A cognitive mediation theory of task goals and human performance. *Motivation and Emotion* 9, 4 (1985), 345–367. https://doi.org/10.1007/BF00992205

[62] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. 2007. Using Patterns to Capture Architectural Decisions. *IEEE Software* 24 (2007), 38–45. Issue 4. https://doi.org/10.1109/MS.2007.124

[63] Andrew Head, Codanda Appachu, Marti A Hearst, and Björn Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 3–12. https://doi.org/10.1109/vlhcc.2015.7356972

[64] Tom-Michael Hesse and Barbara Paech. 2013. Supporting the collaborative development of requirements and architecture documentation. In *International Workshop on the Twin Peaks of Requirements and Architecture*. 22–26. https://doi.org/10.1109/TwinPeaks-2.2013.6617355

[65] Tom-Michael Hesse, Arthur Kuehlwein, and Tobias Roehm. 2016. DecDoc: A tool for documenting design decisions collaboratively and incrementally. In *International Workshop on Decision Making in Software ARCHitecture*. 30–37. https://doi.org/10.1109/march.2016.9

[66] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 132–136. https://doi.org/10.1145/1028664.1028717

[67] IEEE. 1990. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990* (1990), 1–84. https://doi.org/10.1109/IEEESTD.1990.101064

[68] Daniel Jackson. 2021. *The Essence of Software: Why Concepts Matter for Great Design.* Princeton University Press. https://doi.org/10.1515/9780691230542

[69] Anton Jansen and Jan Bosch. 2005. Software architecture as a set of architectural design decisions. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 109–120. https://doi.org/10.1109/WICSA.2005.61

[70] Anton Jansen, Jan Bosch, and Paris Avgeriou. 2008. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software* 81, 4 (2008), 536–557. https://doi.org/10.1016/j.jss.2007.08.025

[71] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *International Conference on Software Engineering (ICSE)*. 672–681. https://doi.org/10.1109/icse.2013.6606613

[72] Mik Kersten and Gail C. Murphy. 2006. Using task context to improve programmer productivity. In *International Symposium on Foundations of Software Engineering (FSE)*. 1–11. https://doi.org/10.1145/1181775.1181777

[73] Markus Kimmig, Martin Monperrus, and Mira Mezini. 2011. Querying source code with natural language. In *International Conference on Automated Software Engineering (ASE)*. 376–379. https://doi.org/10.1109/ase.2011.6100076

[74] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Keele University and University of Durham.

[75] Anja Kleebaum, Marco Konersmann, Michael Langhammer, Barbara Paech, Michael Goedicke, and Ralf H. Reussner. 2019. Continuous Design Decision Support. In *Managed Software Evolution*. Springer, 107–139. https://doi.org/10.1007/978-3-030-13499-0_6

[76] Jens Knodel and Daniel Popescu. 2007. A comparison of static architecture compliance checking approaches. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 12–12. https://doi.org/10.1109/wicsa.2007.1

[77] Amy J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *International Conference on Software Engineering (ICSE)*. 344–353. https://doi.org/10.1109/ICSE.2007.45

[78] Philippe Kruchten. 2004. An Ontology of Architectural Design Decisions in Software-Intensive Systems. In *Groningen Workshop on Software Variability Management*.

[79] Thomas D. LaToza. 2020. Information Needs: Lessons for Programming Tools. *IEEE Software* 37, 6 (2020), 52–57. https://doi.org/10.1109/MS.2020.3014343

[80] Thomas D LaToza, David Garlan, James D Herbsleb, and Brad A Myers. 2007. Program comprehension as fact finding. In *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 361–370. https://doi.org/10.1145/1287624.1287675

[81] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer Questions About Code. In *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*. Article 8, 6 pages. https://doi.org/10.1145/1937117.1937125

[82] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *International Conference on Software Engineering (ICSE)*. 492–501. https://doi.org/10.1145/1134285.1134355

[83] Lattix. 2020. Lattix Architect. https://www.lattix.com

[84] Jintae Lee. 1991. Extending the Potts and Bruns model for recording design rationale. In *International Conference on Software Engineering (ICSE)*. 114–125. https://doi.org/10.1109/ICSE.1991.130629

[85] Jintae Lee. 1997. Design rationale systems: understanding the issues. *IEEE Expert* 12 (1997), 78–85. Issue 3. https://doi.org/10.1109/64.592267

[86] Jintae Lee and Kum-Yew Lai. 1991. What's in Design Rationale? *Human Computer Interaction* 6, 3-4 (1991), 251–280. https://doi.org/10.1080/07370024.1991.9667169

[87] Jintae Lee and Kum-Yew Lai. 1992. *A comparative analysis of design rationale representations*. Technical Report. Massachusetts Institute of Technology.

[88] Larix Lee and Philippe Kruchten. 2007. Capturing Software Architectural Design Decisions. In *Canadian Conference on Electrical and Computer Engineering*. 686–689. https://doi.org/10.1109/CCECE.2007.176

[89] Larix Lee and Philippe Kruchten. 2008. A Tool to Visualize Architectural Design Decisions. In *International Conference on the Quality of Software-Architectures (QoSA 2008)*, Vol. 5281. 43–54. https://doi.org/10.1007/978-3-540-87879-7_3

[90] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. 2018. A survey on code analysis tools for software maintenance prediction. In *International Conference in Software Engineering for Defence Applications*. 165–175. https://doi.org/10.1007/978-3-030-14687-0_15

[91] Timothy C Lethbridge, Janice Singer, and Andrew Forward. 2003. How software engineers use documentation: The state of the practice. *IEEE Software* 6 (2003), 35–39. https://doi.org/10.1109/ms.2003.1241364

[92] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. *Transactions on Software Engineering* 32 (2006), 176–192. Issue 3. https://doi.org/10.1109/TSE.2006.28

[93] 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. https://doi.org/10.1016/j.cad.2011.08.002

[94] Mikael Lindvall, Roseanne Tesoriero Tvedt, and Patricia Costa. 2002. Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture. In *International Software Metrics Symposium (METRICS)*. 77–86. https://doi.org/10.1109/METRIC.2002.1011327

[95] Michael Xieyang Liu, Jane Hsieh, Nathan Hahn, Angelina Zhou, Emily Deng, Shaun Burley, Cynthia Bagier Taylor, Aniket Kittur, and Brad A. Myers. 2019. Unakite: Scaffolding Developers' Decision-Making Using the Web. In *Symposium on User Interface Software and Technology (UIST)*. 67–80. https://doi.org/10.1145/3332165.3347908

[96] 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). https://doi.org/10.1115/1.3470018

[97] Edwin A. Locke, Karyll N. Shaw, Lise M. Saari, and Gary P. Latham. 1981. Goal setting and task performance: 1969-1980. *Psychological Bulletin* 90, 1 (1981), 125–152. https://doi.org/10.1037/0033-2909.90.1.125

[98] Headway Software Technologies Ltd. 2019. Structure101. https://structure101.com

[99] Allan MacLean, Richard M Young, Victoria ME Bellotti, and Thomas P Moran. 1991. Questions, Options, and Criteria: Elements of Design Space Analysis. *Human–Computer Interaction* 6, 3-4 (1991), 201–250. https://doi.org/10.1080/07370024.1991.9667168

[100] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. 2014. Supporting informal design with interactive whiteboards. In *Conference on Human Factors in Computing Systems (CHI)*. 331–340. https://doi.org/10.1145/2556288.2557411

[101] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. 2015. How Software Designers Interact with Sketches at the Whiteboard. *Transactions on Software Engineering* 41, 2 (2015), 135–156. https://doi.org/10.1109/TSE.2014.2362924

[102] Christian Manteuffel, Paris Avgeriou, and Roelof Hamberg. 2018. An exploratory case study on reusing architecture decisions in software-intensive system projects. *Journal of Systems and Software* 144 (2018), 60–83. https://doi.org/10.1016/j.jss.2018.05.064

[103] Robert C. Martin. 2002. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall. https://doi.org/10.1002/pfi.21408

[104] Sahar Mehrpour, Thomas D. LaToza, and Rahul K. Kindi. 2019. Active Documentation: Helping Developers Follow Design Decisions. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 87–96. https://doi.org/10.1109/vlhcc.2019.8818816

[105] Sahar Mehrpour, Thomas D. LaToza, and Hamed Sarvari. 2020. RulePad: interactive authoring of checkable design rules. In *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 386–397. https://doi.org/10.1145/3368089.3409751

[106] Cornelia Miesbauer and Rainer Weinreich. 2013. Classification of design decisions–an expert survey in practice. In *European Conference on Software Architecture*. 130–145. https://doi.org/10.1007/978-3-642-39031-9_12

[107] Thomas P Moran and John M Carroll. 1996. *Design rationale: Concepts, techniques, and use*. CRC Press.

[108] Gail C. Murphy, David Notkin, and Kevin Sullivan. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-level Models. In *Symposium on Foundations of Software Engineering (FSE)*. 18–28. https://doi.org/10.1145/222124.222136

[109] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52. https://doi.org/10.1109/mc.2016.200

[110] Brad A. Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Amy J. Ko. 2008. How designers design and program interactive behaviors. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 177–184. https://doi.org/10.1109/VLHCC.2008.4639081

[111] Jernej Novak, Andrej Krajnc, and Rok Žontar. 2010. Taxonomy of static code analysis tools. In *International Convention MIPRO*. 418–422.

[112] Steve Oney, Christopher Brooks, and Paul Resnick. 2018. Creating Guided Code Explanations with chat.codes. *Human-Computer Interaction* 2, CSCW (2018), 131:1–131:20. https://doi.org/10.1145/3274400

[113] David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058. https://doi.org/10.1007/978-3-642-48354-7_20

[114] David Lorge Parnas and Paul C. Clements. 1986. A rational design process: How and why to fake it. *Transactions on Software Engineering* SE-12, 2 (1986), 251–257. https://doi.org/10.1109/tse.1986.6312940

[115] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *International Conference on Mining Software Repositories (MSR)*. 227–237. https://doi.org/10.1109/MSR.2017.63

[116] Feniosky Peña-Mora and Sanjeev Vadhavkar. 1997. Augmenting design patterns with design rationale. *AI EDAM* 11, 2 (1997), 93–108. https://doi.org/10.1017/S089006040000189X

[117] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *Software Engineering Notes* 17, 4 (1992), 40–52. https://doi.org/10.1145/141874.141884

[118] PMD. 2020. Writing a custom rule PMD Source Code Analyzer. https://pmd.github.io/latest/pmd_userdocs_extending_writing_pmd_rules.html

[119] PMD. 2021. PMD Designer. https://pmd.github.io/latest/pmd_userdocs_extending_designer_reference.html

[120] Colin Potts and Glenn Bruns. 1988. Recording the Reasons for Design Decisions. In *International Conference on Software Engineering (ICSE)*. 418–427.

[121] Ghulam Rasool, Ilka Philippow, and Patrick Mäder. 2010. Design pattern recovery based on annotations. *Advances in Engineering Software* 41, 4 (2010), 519–526. https://doi.org/10.1016/j.advengsoft.2009.10.014

[122] William C. Regli, Xiaochun Hu, Michael Atwood, and Wei Sun. 2000. A Survey of Design Rationale Systems: Approaches, Representation, Capture and Retrieval. *Engineering with Computers* 16, 3-4 (2000), 209–235. https://doi.org/10.1007/pl00013715

[123] Horst W. J. Rittel. 1972. On the planning crisis: Systems analysis of the first and second generations. *Bedriftsøkonomen* 8 (1972).

[124] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. 2017. On-demand developer documentation. In *International Conference on Software Maintenance and Evolution (ICSME)*. 479–483. https://doi.org/10.1109/icsme.2017.17

[125] Benjamin Rogers, James Gung, Yechen Qiao, and Janet E. Burge. 2012. Exploring techniques for rationale extraction from existing documents. In *International Conference on Software Engineering (ICSE)*. 1313–1316. https://doi.org/10.1109/ICSE.2012.6227091

[126] Chanchal Kumar Roy and James R Cordy. 2007. *A survey on software clone detection research.* Technical Report 115. Queen's School of Computing TR. 64–68 pages.

[127] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc. 1990. Recognizing Design Decisions in Programs. *IEEE Software* 7, 1 (1990), 46–54. https://doi.org/10.1109/52.43049

[128] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

[129] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *International Conference on Software Engineering (ICSE)*. 598–608. https://doi.org/10.1109/icse.2015.76

[130] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. 2005. Using Dependency Models to Manage Complex Software Architecture. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 167–176. https://doi.org/10.1145/1103845.1094824

[131] Sandra Schröder and Georg Buchgeher. 2019. Formalizing Architectural Rules with Ontologies - An Industrial Evaluation. In *Asia-Pacific Software Engineering Conference (APSEC)*. 55–62. https://doi.org/10.1109/APSEC48747.2019.00017

[132] Arman Shahbazian, Youn Kyu Lee, Duc Le, Yuriy Brun, and Nenad Medvidovic. 2018. Recovering Architectural Design Decisions. In *International Conference on Software Architecture (ICSA)*. 95–9509. https://doi.org/10.1109/ICSA.2018.00019

[133] Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall.

[134] Abdullah Sheneamer and Jugal Kalita. 2016. A survey of software clone detection techniques. *International Journal of Computer Applications* 137, 10 (2016), 1–21. https://doi.org/10.5120/ijca2016908896

[135] Patrick C. Shih, Gina Venolia, and Gary M. Olson. 2011. Brainstorming under constraints: why software developers brainstorm in groups. In *BCS Conference on Human-Computer Interaction*. 74–83. https://doi.org/10.14236/ewic/hci2011.30

[136] Simon Buckingham Shum and Nick Hammond. 1994. Argumentation-based design rationale: what use at what cost? *International Journal of human-computer studies* 40, 4 (1994), 603–652. https://doi.org/10.1006/ijhc.1994.1029

[137] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *Transactions on Software Engineering* 34, 4 (2008), 434–451. https://doi.org/10.1109/TSE.2008.26

[138] Elliot Soloway, Jeannine Pinto, Stanley Letovsky, David C. Littman, and Robin Lampert. 1988. Designing Documentation to Compensate for Delocalized Plans. *Commun. ACM* 31, 11 (1988), 1259–1267. https://doi.org/10.1145/50087.50088

[139] Boya Sun, Gang Shu, Andy Podgurski, and Brian Robinson. 2012. Extending static analysis by mining project-specific rules. In *International Conference on Software Engineering (ICSE)*. 1054–1063. https://doi.org/10.1109/ICSE.2012.6227114

[140] Alistair G. Sutcliffe and Michele Ryan. 1998. Experience with SCRAM, a SCenario Requirements Analysis Method. In *International Symposium on Requirements Engineering*. 164–171. https://doi.org/10.1109/ICRE.1998.667822

[141] Antony Tang, Paris Avgeriou, Anton Jansen, Rafael Capilla, and Muhammad Ali Babar. 2010. A comparative study of architecture knowledge management tools. *Journal of Systems and Software* 83, 3 (2010), 352–370. https://doi.org/10.1016/j.jss.2009.08.032

[142] Antony Tang, Muhammad Ali Babar, Ian Gorton, and Jun Han. 2006. A survey of architecture design rationale. *Journal of Systems and Software* 79, 12 (2006), 1792–1804. https://doi.org/10.1016/j.jss.2006.04.029

[143] Stephen E Toulmin. 1958. *The uses of argument.* Cambridge University Press.

[144] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2018. JIT Feedback: What Experienced Developers Like About Static Analysis. In *International Conference on Program Comprehension (ICPC)*. 64–73. https://doi.org/10.1145/3196321.3196327

[145] Jeff Tyree and Art Akerman. 2005. Architecture decisions: Demystifying architecture. *IEEE Software* 22, 2 (2005), 19–27. https://doi.org/10.1109/ms.2005.27

[146] Vassilios Tzerpos and Richard C. Holt. 2000. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Working Conference on Reverse Engineering (WCRE)*. 258–267. https://doi.org/10.1109/WCRE.2000.891477

[147] Jan Salvador van der Ven, Anton G. J. Jansen, Jos A. G. Nijhuis, and Jan Bosch. 2006. *Design Decisions: The Bridge between Rationale and Architecture*. Springer Berlin Heidelberg, Chapter 16, 329–348. https://doi.org/10.1007/978-3-540-30998-7_16

[148] Uwe van Heesch and Paris Avgeriou. 2011. Mature Architecting - A Survey about the Reasoning Process of Professional Architects. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 260–269. https://doi.org/10.1109/WICSA.2011.42

[149] Hans van Vliet and Antony Tang. 2016. Decision making in software architecture. *Journal of Systems and Software* 117 (2016), 638–644. https://doi.org/10.1016/j.jss.2016.01.017

[150] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. 2014. CODES: Mining Source Code Descriptions from Developers Discussions. In *International Conference on Program Comprehension (ICPC)*. 106–109. https://doi.org/10.1145/2597008.2597799

[151] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25, 2 (2020), 1419–1457. https://doi.org/10.1007/s10664-019-09750-5

[152] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 38–49. https://doi.org/10.1109/saner.2018.8330195

[153] Vera Wahler, Dietmar Seipel, J. Wolff, and Gregor Fischer. 2004. Clone detection in source code by frequent itemset techniques. In *International Workshop on Source Code Analysis and Manipulation*. 128–135. https://doi.org/10.1109/SCAM.2004.6

[154] Wei Wang and Janet E Burge. 2010. Using rationale to support pattern-based architectural design. In *ICSE Workshop on Sharing and Reusing Architectural Knowledge*. 1–8. https://doi.org/10.1145/1833335.1833336

[155] Rainer Weinreich and Iris Groher. 2016. Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review. *Information and Software Technology* 80 (2016), 265–286. https://doi.org/10.1016/j.infsof.2016.09.007

[156] Yunwen Ye, Gerhard Fischer, and Brent Reeves. 2000. Integrating active information delivery and reuse repository systems. In *International Symposium on Foundations of Software Engineering (FSE)*. 60–68. https://doi.org/10.1145/357474.355053

[157] Vera Zaychik and William C Regli. 2003. Capturing communication and context in the software project lifecycle. *Research in Engineering Design* 14, 2 (2003), 75–88. https://doi.org/10.1007/s00163-002-0027-8