2018 IEEE International Conference on Software Architecture

Recovering Architectural Design Decisions

Arman Shahbazian, Youn Kyu Lee, Duc Le, Yuriy Brun, and Nenad Medvidovic
University of Southern California
Los Angeles, CA, USA 90089-0781
{armansha, younkyul, ducmle, neno}@usc.edu

University of Massachusetts Amherst
Amherst, MA, USA, 01003-9264
brun@cs.umass.edu

***Abstract*—Designing and maintaining a software system's architecture typically involve making numerous design decisions, each potentially affecting the system's functional and nonfunctional properties. Understanding these design decisions can help inform future decisions and implementation choices and can avoid introducing regressions and architectural inefficiencies later. Unfortunately, design decisions are rarely well documented and are typically a lost artifact of the architecture creation and maintenance process. The loss of this information can thus hurt development. To address this shortcoming, we develop RecovAr, a technique for automatically recovering design decisions from the project's readily available history artifacts, such as an issue tracker and version control repository. RecovAr uses state-of-the-art architectural recovery techniques on a series of version control commits and maps those commits to issues to identify decisions that affect system architecture. While some decisions can still be lost through this process, our evaluation on Hadoop and Struts, two large open-source systems with over 8 years of development each and, on average, more than 1 million lines of code, shows that RecovAr has the recall of 75% and a precision of 77%. Our work formally defines architectural design decisions and develops an approach for tracing such decisions in project histories. Additionally, the work introduces methods to classify whether decisions are architectural and to map decisions to code elements. Finally, our work contributes a methodology engineers can follow to preserve design-decision knowledge in their projects.**

I. Introduction

Software architecture has become the centerpiece of modern software development [37]. Developers are increasingly relying on software architecture to lead them through the process of creating and implementing large and complex systems. Understanding of a software system's architecture and the set of decisions that led to its creation is crucial for making new decisions about the system both at the design and implementation levels. Further, engineers who are aware of the architectural impacts of their changes deliver higher-quality code [27]. Unfortunately, the design decisions made during the software lifecycle are typically not well documented and so the rationale for these choices is often lost [14]. It is thus desirable for architects and developers to be able to automatically recover past design decisions. Unfortunately, modern architectural recovery techniques, e.g., [40], [11], focus on recovering "what" the architecture of a system looks like, and not "why" the architecture looks the way it does, a symptom of a phenomenon known as knowledge vaporization in software systems [13]. Recovering the design decisions that lead to an architecture is one way to capture this "why."

In this paper, we make the observation that modern software development offers access not only to the system itself but also to the history of its development (e.g., a version control repository) and a database of issues, change requests, and tasks, often partially augmented with reasons and justifications mapped to specific code changes that address them (e.g., an issue tracking system). Guided by these observations and the fact that access to architects who designed the system is limited and expensive, we develop RecovAr, a technique for automatically recovering design decisions made during the development process.

While RecovAr is independent of a system's architectural paradigm (e.g., component-based, microservices, SOA, system-of-systems), it does assume the existence of suitable means of obtaining static architectural structure from implementation artifacts. Specifically, our work discussed in this paper relies on two existing techniques that recover such architectural structure from code, *ACDC* [40] and *ARC* [11].[1] By recovering this architectural information at multiple points during the system's development and mapping the changes in the structure to the rich information available in the system's issue tracking systems, RecovAr can recover many (though not all) of the system's design decisions and traceability links between those decisions and code changes, issues, and other documentation.

We applied RecovAr to over 100 versions of Hadoop and Struts, two large, widely adopted open-source systems, with over 8 years of development each and, on average, more than 1 million lines of code. We found that RecovAr can accurately uncover the architectural design decisions embodied in the systems, recovering 75% of the decisions with a precision of 77%.

This paper makes the following contributions:

- We formally define the notion of an architectural design decision and develop an approach for tracing such decisions in existing software project histories.
- We introduce methods to classify whether decisions are architectural and to map decisions to code elements.
- We empirically examine how design decisions manifest in software systems, evaluating our approach on two large, widely-used systems.

[1]Existing literature refers to these and similar techniques as "architecture recovery techniques," and the produced artifacts as "architectures." For legacy and simplicity reasons, we will also use this terminology in the remainder of the paper, with the understanding that what is recovered is only a partial, structural view of a system's static architecture.

978-1-5090-5729-0/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSA.2018.00019

- We develop a methodology for preserving design-decision knowledge in software projects.

The remainder of this paper is organized as follows. Section II summarizes the background necessary for our approach. Section III describes RecovAr and Section IV evaluates it. Section V places our work in the context of related research and Section VI summarizes our contributions.

II. Background

Research has demonstrated that software architecture plays a critical role in the evolution and maintenance of software systems [10], and that awareness of the architectural implications of code changes results in higher-quality code [27]. This has led to the development of several architectural recovery techniques that help counteract the challenges brought about by architectural drift and erosion [41], [11], [10], [40].

These recovery techniques are typically based on some slant of the originally proposed view of software architecture as four Cs building blocks: components, connectors, configurations, and constraints. Despite being simple and appealing, this view has proven to be incomplete and has required further elaboration. To that end, recent work has approached architecture from the perspective of design decisions. Our work builds on these two bodies of research, describing a software architecture as a set of design decisions (Section II-A) and automatic architectural recovery (Section II-B).

A. Architectural Design Decisions

For many years, research community and industry alike had been focused on the result, the consequences of the design decisions made, trying to capture them in the *architecture* of the system under consideration, often using a graphical representation. Such representations were, and to a great extent still are, centered on views [17], as captured by the ISO/IEC/IEEE 42010 standard [12], or the use of an architecture description language (ADL) [23]. However, this approach to documenting software architectures can cause problems such as expensive system evolution, lack of stakeholders communication, and limited reusability of architectures [33].

Architecture as a set of design decisions was proposed to address these shortcomings. This new paradigm focuses on capturing and documenting rationale, constraints, and alternatives of design decisions [41]. More specifically Jansen et al. defined architectural design decisions as a description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, the design rules, and additional requirements that (partially) realize one or more requirements on a given architecture [13], [5]. The key element in their definition is rationale, i.e., the reasons behind an architectural design decision. Kruchten et al. proposed an ontology that classified architectural decisions into three categories: (1) existence decisions (ontocrises), (2) property decisions (diacrises), and (3) executive decisions (pericrises) [16]. Among the three categories, existence decisions — decisions that state some element or artifact will positively show up or disappear in a system — are the most prevalent and capture the most volatile aspects of a system [16], [13]. Property and executive decisions are enduring guidelines that are mostly driven by the business environment and affect the methodologies, and to a large extent the choices of technologies and tools.

Inspired by the described existing work, the notion of design decisions used in this paper values the *rationales* and *consequences* as two equally important constituent parts of design decisions. However, not all design decisions are created equal. Some design decisions are straightforward, with clear singular rationale and consequence, while some are cross-cutting and intertwined [5], i.e., affect multiple components and/or connectors and often become intimately intertwined with other design decisions. To distinguish between different kinds of design decisions we classify them into three categories: (1) simple, (2) compound, and (3) cross-cutting. Simple decisions have a singular rationale and consequence. Compound decisions include several closely related rationales, but their consequences are generally contained to one component. Finally, cross-cutting decisions affect a wider range of components and their rationale follows a higher level concern such as architectural quality of the system.

B. Architecture Recovery, Change, and Decay Evaluator

To capture the consequence aspect of design decisions, we build on top of the existing work in architecture modeling and recovering. To obtain the static architectures of a system from its source code, we use the state-of-the-art workbench, called *ARCADE* [4], [20]. *ARCADE* employs a suite of architecture-recovery techniques and a set of metrics for measuring different aspects of architectural change. It constructs an expansive view showcasing the actual (as opposed to idealized) evolution of a software system's architecture.

ARCADE allows an architect (1) to extract multiple architectural views from a system's codebase and (2) to study the architectural changes during the system's evolution as reflected in those views. *ARCADE* provides access to multiple recovery techniques. We use two of the techniques in this paper: *Algorithm for Comprehension-Driven Clustering* (*ACDC*) [40] and *Architecture Recovery using Concerns* (*ARC*) [11]. *ACDC* and *ARC* approach a system's architecture from different perspectives, and have been shown to exhibit the best accuracy and scalability among the recovery techniques provided in *ARCADE*. *ACDC*'s view is oriented toward components that are based on structural patterns (e.g., components consisting of entities that together form a particular subgraph). On the other hand, *ARC*'s view produces components that are semantically related due to sharing similar system-level concerns (e.g., a component whose main concern is handling distributed jobs).

To measure architectural changes across the development history of a software system, *ARCADE* provides several architecture similarity metrics: cvg [20] and a2a [4], MoJo [39], and MoJoFM [43]. These are system-level similarity metrics calculated based on the cost of transforming one architecture into another. Using similar principles, RecovAr conducts the change analysis and extracts a system's architectural changes (see Section III-A).

III. The RecovAr Approach

Knowledge vaporization in software systems plays a major role in increasing maintenance costs, and exacerbates architectural drift and erosion [13]. The goal of RecovAr is to uncover architectural design decisions in software systems, and thereby help reverse the course of knowledge vaporization by providing a crisper understanding of such decisions and their effects.

In this section, we further elaborate on the definition and classification of design decisions. We describe how architectural changes can be recovered from the source code of real software systems. We also describe a process whereby architectural design decisions are identified in real systems. Finally, our approach enables engineers to continuously capture the architectural decisions in software systems during their evolution.

Section II-A identified two constituent parts of an architectural design decision, *rationale* and *consequence*. The static architecture of a system explicitly captures the system's components and possibly other architectural entities, but rationale is usually missing or, at best, implicit in the structural elements. For this reason, our approach focuses on the consequences of design decisions. We have developed a technique that leverages the combination of source code and issue repositories to obtain the design decision consequences. Issue repositories are used to keep of track of bugs, development tasks, and feature requests in a software development project. Code repositories contain historical data about the inception, evolution, and incremental code changes in a software system. Together, these repositories provide the most reliable and accurate pieces of information about a system.

RecovAr automatically extracts the required information from a system's repositories and outputs the set of design decisions made during the system's lifecycle. In order to achieve this RecovAr first recovers the static architecture of the target system. RecovAr then cross-links the issues to their corresponding code-level changes. These links are in turn analyzed to identify candidate architectural changes and, subsequently, their rationales.

A high level overview of RecovAr's workflow is displayed in Figure 1. RecovAr begins by recovering the static architecture of a system. This step is only required if an up-to-date, reliable, documented architecture is not available.

After recovering or obtaining the architectures of different versions of its target system, RecovAr follows through three distinct phases. In the first phase (**Change Analysis**) RecovAr identifies how the architecture of the system has changed along its evolution path. The second phase (**Mapping**) mines the system's issue repository and creates a mapping (called *architectural impact list*) from issues to the architectural entities they have affected. Finally, the third phase (**Decision Extraction**) creates an overarching decision graph by putting together the architectural changes and the architectural impact list. This graph is in turn traversed to uncover the individual design decisions. In the remainder of this section we detail each of the three phases.

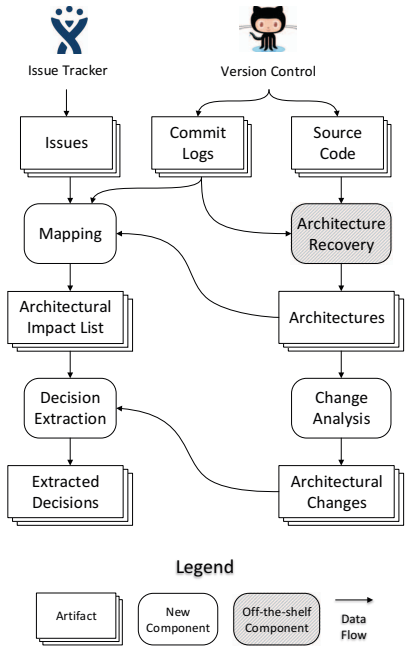


Fig. 1. Overview of RecovAr. Using the existing source code, commit logs, and extracted issues obtained from code and issue repositories, our approach automatically extracts the underlying design decisions. Implementation of the new components spans over 4,000 Source Lines of Code (SLoC).

A. Change Analysis

Architectural change has been recognized as a critical phenomenon from the very beginnings of the study of software architecture [28]. However, only recently have researchers tried to empirically measure and analyze architectural change in software systems [20], [4]. These efforts rely on architectural change metrics that quantify the extent to which the architecture of a software system changes as the system evolves. This work has served as a motivation and useful foundation in obtaining a concrete view of architectural changes.

Specifically, we have designed *Change Analyzer* (*CA*), which is inspired by the manner in which existing metrics (e.g., a2a [4], MoJo [39], and MoJoFM [43]) measure architectural change. These metrics consider five operations used to transform architecture A into architecture B: addition, removal, and relocation of system entities (e.g., methods, classes, libraries) from one component to another, as well as addition and removal of components themselves [1], [22], [26]. We use a similar notion and define architectural change as a set of *architectural deltas*. An architectural delta is: (1) any modification to a component's internal entities including additions and removals (relocation is treated as a combined addition to the destination component and removal from the source component), or (2) additions and removals of entire components. We then aggregate these deltas into architectural change instances. Algorithm 1 describes the details of the approach used to extract the architectural deltas and changes.

CA works in two passes. In the first pass, *CA* matches the most similar components in the given pair of architectures.

Algorithm 1: Change Analysis

Input: *ArchitectureA*,*ArchitectureB*
Output: *Changes* ⇐ a set of architectural changes
1 Let *ComponentsA* = *ArchitectureA*'s components
2 Let *ComponentsB* = *ArchitectureB*'s components
3 Let E_{all}, E_{chosen} = ∅
4 **if** $|ComponentsA| \neq |ComponentsB|$ **then**
5 $\quad$ *Balance*(*ComponentsA*,*ComponentsB*)
6 **foreach** $c_a \in$ ComponentsA **do**
7 $\quad$ **foreach** $c_b \in$ ComponentsB **do**
8 $\quad\quad$ $cost = CalculateChangeCost(c_a, c_b)$
9 $\quad\quad$ $e = \{c_a, c_b, cost\}$
10 $\quad\quad$ *add e to* E_{all}
11 $E_{chosen} = MinCostMatcher(ComponentsA, ComponentsB, E_{all})$
12 **foreach** $e \in E_{chosen}$ **do**
13 $\quad$ $Changes = GetChangeInstances(e.c_a, e.c_b) \cup Changes$
14 *return Changes*

Algorithm 2: *GetChangeInstances* method

Input: *ComponentA*,*ComponentB*
Output: *Changes*
1 Let *EntitiesA* = *ComponentA*'s entities
2 Let *EntitiesB* = *ComponentB*'s entities
3 **if** $EntitiesA \cap EntitiesB = \emptyset$ **then**
4 $\quad$ *Change* ch_1, ch_2
5 $\quad$ $ch_1.deltas = EntitiesA$
6 $\quad$ $ch_2.deltas = EntitiesB$
7 $\quad$ *return* $\{ch_1, ch_2\}$
8 **else**
9 $\quad$ *Change ch*
10 $\quad$ $ch.deltas = (EntitiesA \cup EntitiesB) - (EntitiesA \cap EntitiesB)$
11 $\quad$ *return* $\{ch\}$

In the second pass, *CA* compares the matched components, extracts the architectural delta(s), and clusters them into architectural change instances.

The objective of the matching pass is to find the most similar components in a way that minimizes the overall difference between the matched components. Since two architectures can have different numbers of components, *CA* first balances (Algorithm 1, line 5) the two architectures. To do so, *CA* adds "dummy" (i.e., empty) components to the architecture with fewer components until both architectures have the same number of components. After balancing the architectures, *CA* creates a weighted bipartite graph from architecture A to architecture B and calculates the cost of each edge. Existence of an edge denotes that component C_A has been transformed into component C_B. The cost of an edge is the total number of architectural deltas required to effect the transformation.

Figure 2 displays a simple example of two architectures and the corresponding bipartite graph with all possible edges. *MinCostMatcher* (Algorithm 1, line 11) takes the two architectures and the set of edges between them, and selects the edges in a way that ensures a bijective matching between the components of the two architectures with the minimum overall cost (sum of the costs of the selected edges). *MinCostMatcher* is based on linear programming; its details are omitted for brevity.

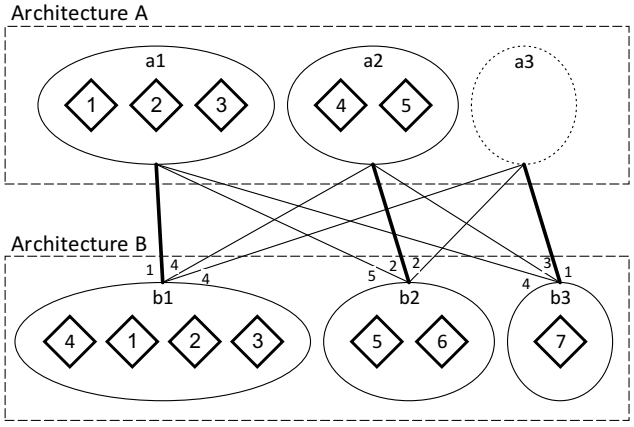


Fig. 2. Calculating the costs of the edges and finding the perfect matching. The bold connectors are the selected edges that lead to minimum overall cost.

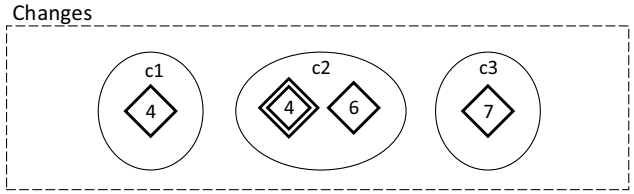


Fig. 3. Extracted changes between the architectures depicted in Fig. 2. Double-lined diamonds indicate removals while regular diamonds denote additions.

In the second pass, *CA* extracts the architectural deltas between the matched components. If there are no common architectural entities between two matched components, we create two change instances, one for the component that has been removed and one for the newly added component. The reason is to distinguish between transformations of components and their additions and removals. Figure 3 depicts the extracted changes of our example architectures.

B. Mapping

The output of *CA* is a set of architectural changes that is a superset of the consequences of design decisions. The goal of *Mapping* is to find all the issues that point to the rationale of the design decisions that yielded those consequences. To that end, *Mapping* first identifies the issues that satisfy two conditions: (1) they belong to the version of the system being analyzed and (2) they have been marked as resolved and their consequent code changes have been merged with the main code base of the system. *Mapping* then extracts the code-level entities affected by each issue. These code-level entities are identified by mining the issues' commit logs and pull requests. Using one or more architecture recovery methods available in *ARCADE*, the code-level entities are translated into corresponding architectural entities. The list of all issues, as well as the mapping between the issues and the architectural entities affected by them is called the *Architectural Impact List*.

Figure 4 displays a graph-based view of this list. It is possible for issues to have overlapping entities (e.g., i2 and i3 are both connected to entity 5). It is also important to note that

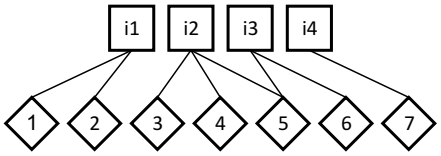


Fig. 4. Architectural impact list. Squares represents issues and diamonds represent entities. An edge from an issue to an entity means that resolving that issue resulted in modifying that entity.

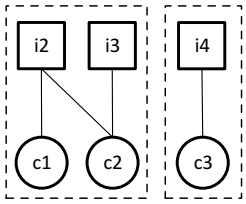


Fig. 5. The overarching decisions graph contains two decisions D1 and D2. Squares denote issues, and circles denote changes.

the presence of an edge from an issue to an entity does not necessarily indicate architectural change (e.g., entities 1 and 5 are not part of any of the architectural changes in Figure 3). This is intuitively expected, since a great many of issues do not incur substantial enough change in the source code and thereby the architecture of the system.

C. Decision Extraction

In its final phase, RecovAr creates the overarching decision graph by putting together the architectural changes and their pertaining issues. This graph is traversed and individual design decisions are identified. Algorithm 3 details this phase.

Algorithm 3 traverses the architectural impact list generated in the *Mapping* phase and the list of changes. If there is an intersection between the entities matched to issues and the entities involved in changes, then it adds an edge connecting the issue with the change. The intuition behind this is that an issue contains the rationale for a decision if it affects the change(s), which are the consequences, of that decision. We note that, hypothetically, there can be situations in which an issue is the cause of a change without directly affecting any architectural deltas in that change. For example, if an issue leads to removing all the dependencies to an entity, that entity might get relocated out of its containing component by the architecture recovery technique. However, detecting these situations in a system's architecture is not possible with existing recovery techniques, because they abstract away the dependencies among internal entities of a component. Although such information could easily be incorporated, RecovAr would be unable to deal with such scenarios as currently implemented.

The decisions graph for our running example is depicted in Figure 5. The *FindDecisions* method in Algorithm 3 removes all orphaned changes and issues, and in the remaining graph locates the largest disconnected subgraphs. Each disconnected subgraph represents a decision. The reason is that these disconnected subgraphs are the largest sets of interrelated

Decision Type	**Issue(s)**	**Change(s)**
Simple	[1] Job tracking module only kept track of the jobs executed in the past 24 hours. If an admin checked the history after a day of inactivity, e.g., on Monday, the list would be empty.	① `hadoop.mapred` component was modified.
Compound	[1] UTF8 compressor does not handle end of line correctly. [2] Sequenced files should support custom compressors.	① `CompressionInputStream` was added and `CompressionCodec` was modified.
Cross-cutting	[1] Random seeks corrupt the `InputStream` data. [2] Streaming must send status signals every 10 seconds. [3] Task status should include timestamp for job transitions.	① `hadoop.streaming` was modified. ② `hadoop.metrics` component was modified. ③ `hadoop.fs` was modified.

Fig. 6. Examples of recovered Hadoop decisions.

Algorithm 3: *Decision Extraction*

Input: *ArchitecturalImpactList*, *Changes*
Output: *Decisions*

```
Let DecisionsGraph = bipartite graph of decisions
foreach (issue, entities) ∈ ArchitecturalImpactList do
    foreach c ∈ Changes do
        if c.deltas ∩ entities ≠ ∅ then
            connect(issue, c) in DecisionsGraph
Decisions = FindDecisions(DecisionsGraph)
return Decisions
```

rationales and consequences that do not depend on other issues or changes. Intuitively, we expect that, in a real-world system, only a subset of issues will impose changes whose impact on the system can be considered architectural. Furthermore, each of those issues will reflect a specific, targeted objective. Therefore, in a typical system, the graph of changes and issues should contain disconnected subgraphs of reasonable sizes. This is discussed further in our evaluation in Section IV.

In Section II, we identified three different types of decisions: (1) Simple decisions are the decisions that consist of a single change and a single issue. These decisions have a clear rationale and consequence. (2) Compound decisions are the decisions that include multiple issues and a single change. These decisions are similar to simple decisions and the issues involved are closely related to an overarching rationale. Finally, (3) cross-cutting decisions are the decisions that include multiple changes and one or more issues. These decisions have a higher-level, compound rationale — e.g., improving system reliability or performance — that requires multiple changes to be achieved.

For illustration, Figure 6 lists three real examples of decisions, one of each type, uncovered from Hadoop. Information in the *Issue(s)* column contains the summaries of the issues

System	Domain	Versions	Issues	MSLoC
Hadoop	Distributed Processing	68	2969	30.0
Struts	Web Apps Framework	36	1351	6.7

Fig. 7. Subject systems analyzed in our study.

pertaining to that decision. Each boxed number indicates a separate issue or change. The data in the *Change(s)* column are short descriptions of the changes involved in a given decision. The simple decision in the top row is an update to satisfy a requirement by changing the *job tracking module*. The compound decision in the middle row describes the two sides of a problem that is resolved by changing the *compression module* of Hadoop. Finally the uncovered cross-cutting decision in the bottom row is about a series of changes applied to increase the reliability of Hadoop's task execution.

Applying RecovAr continuously throughout a project's lifecycle (e.g., as can be done with testing [31], [25], [24]), helps preserve architectural knowledge and could encourage engineers to write architecturally-conscious issue descriptions, increasing system quality [32].

IV. Evaluation

We have empirically evaluated RecovAr's applicability and accuracy. IV-A discusses the real-world systems on which RecovAr was applied, demonstrating its applicability. Sections IV-B and IV-C discuss RecovAr's precision and recall.

A. Applicability

Figure 7 describes the two subject systems we have used in our evaluation. These systems were selected from the catalogue of Apache open-source software systems [2]. We selected Hadoop [3] and Struts [35] because they are widely adopted and fit the target profile of candidate systems for our approach: they are open-source, have accessible issue and code repositories, and log the fixing commits (i.e., the changes applied to the system to resolve the corresponding issues). Furthermore, these systems are at the higher end of the Apache software systems' spectrum in terms of size and lifespan. Both of these projects use GitHub as their version control and source repository, and Jira [15] as their issue repository. We analyzed more than 100 versions of Hadoop and Struts in total. Our analyses spanned over 8 years of development, and over 35 million SLoC, and over 4,000 resolved issues.

An overview of the results of applying RecovAr to the two subject systems is depicted in Figure 8. These results are grouped by (1) system (Hadoop vs. Struts) and (2) employed recovery technique (*ARC* vs. *ACDC*). In this table, *No. of Iss. in Decisions* represents the total number of issues that were identified to be part of an architectural design decision. On average, only about 18% of the issues for Hadoop and 6% of the issues for Struts have had architecturally significant effects, and hence have been considered parts of a design decision. This is in line with the intuition that only a subset of issues will impose changes whose impact on the system can be considered architectural. Moreover, this observation bolsters the importance of RecovAr for understanding the current state of

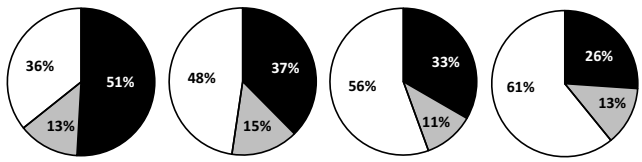
Systems	Hadoop		Struts	
	ACDC	ARC	ACDC	ARC
No. of Iss. in Decisions	427	674	70	94
No. of Changes	950	3935	220	1359
No. of Decisions	112	149	27	23
Avg. Issues/Decision	3.81	4.52	2.59	4.94
Avg. Changes/Decision	1.77	2.36	1.77	2.21

Fig. 8. Number of changes, recovered decisions, and frequencies of issues and changes per decision.

a system and the decisions that have led to it. Without having access to RecovAr, architects would have to analyze 5-to-15 times more issues and commits to uncover the rationales and root causes behind the architectural changes of their system. The remainder of Figure 8 displays the total number of detected architectural changes (*No. of Changes*), the total number of uncovered architectural design decisions (*No. of Decisions*), and the average numbers of issues and changes per decision (*Avg. Issues/Decision* and *Avg. Changes/Decision*, respectively). It is worth mentioning that not all the detected changes were matched to design decisions, which we will elaborate on further when evaluating RecovAr's recall (Section IV-C).

As displayed in Figure 8, depending on the technique used to recover the architecture, the number of uncovered design decisions varies. The reason is that *ACDC* and *ARC* approach architecture recovery from different perspectives: *ACDC* leverages a system's module dependencies; *ARC* derives a more semantic of a system's architecture, detecting concerns via information retrieval techniques. Therefore, the nature of the recovered architectures and changes, and consequently the uncovered design decisions, are different. Recent work has shown that these recovery techniques provide complementary views of a system's architecture [20]. The propagation of these complementary views to our approach has yielded some tangible effects. For instance, RecovAr running *ARC* was able to uncover a decision about refactoring the names of a set of classes and methods in Hadoop, while RecovAr running *ACDC* could not uncover that decision. The reason is that *ARC* is sensitive to lexical changes by design.

RecovAr aims to uncover three kinds of architectural design decisions (recall Section II). Our results confirmed the presence of all three kinds in our subject systems. Figure 9 depicts the distribution of different kinds of decisions detected for each pair of systems and recovery techniques. While the relative proportion of simple and cross-cutting decisions varies across systems and employed recovery techniques, the number of



(a) Hadoop-ACDC (b) Hadoop-ARC (c) Struts-ACDC (d) Struts-ARC

Fig. 9. Distribution of types of decision in the subject systems: solid black denotes simple decisions; grey denotes compound decisions; white denotes cross-cutting decisions.

compound decisions is consistently the smallest.

B. Precision

To assess RecovAr's precision, we need to determine whether the uncovered architectural design decisions are valid. As captured in the premise of RecovAr, architectural design decisions are not generally documented, hence a ground-truth for our analyses was not readily available.

To overcome this hurdle, we devised a systematic plan to objectively assess the uncovered design decisions. We defined a set of criteria targeting the two aspects of an architectural design decision — rationale and consequence — and used them as the basis of our assessment. Two PhD students carried out the analysis and the results of their independent examinations were later aggregated. In the remainder of this section, we will elaborate on the details of the conducted analyses.

We use four criteria targeting different parts of an architectural design decision (two targeting rationales and two targeting consequences). Each criterion is rated using a three-level-scale, with the numeric values of 0, 0.5, and 1. In this scale, 0 means that the criterion is not satisfied; 0.5 means that the satisfaction of the criterion is confirmed after further investigation by examining the source-code, details of the issues, or commit logs; finally, 1 means that the criterion is evidently satisfied. The reason we use a three-level scale in our analysis is to measure the precision of RecovAr's results from the viewpoint of non-experts, and to distinguish the decisions according to the effort required for understanding them. To that end, any criterion whose evaluation requires (1) in-depth system expertise, (2) inspection of information other than that captured in design decisions, and/or (3) having access to the original architects of the system, is given a rating of 0.

The criteria for assessing rationales are two-fold:

1) *Rationale Clarity* indicates whether the rationale and its constituent parts are easily understandable. This is accomplished by looking at issue summaries and pinpointing the problems or requirements driving the decision.
2) *Rationale Cohesion* indicates the degree to which there is a coherent relationship among the issues that make up a given rationale. *Rationale Cohesion* is only analyzed if the decision is shown to possess *Rationale Clarity*.

The criteria for assessing consequences are also two-fold:

1) *Consequence-Rationale Association* assesses whether the changes and their constituent architectural deltas are related to the listed rationale.
2) *Consequence Tractability* assesses whether the size of the changes is tractable. In other words, is the number of changes and their constituent deltas small enough to be understandable in a short amount of time?[2]

The two PhD students independently scored every decision based on the above criteria. The three-level scale allowed us to develop a finer-grained understanding of the decisions' quality.

As illustrative examples, we explain the scoring procedures for two decisions in Hadoop. Listing 1 displays a simple design decision as uncovered by RecovAr in Hadoop version 0.9.0. The rationale consists of a single issue that explains the intent is to separate the user logs from system logs. However, the rationale summary does not explain why this needs to happen. Looking at the issue in Jira, the reason is that system logs are cluttering the user logs, and system logs need to be cleared out more frequently than user logs. Since we had to look at the issue to understand "why" this decision was made, the *Rationale Clarity* in this case was scored 0.5. Since we only have one issue, the *Rationale Cohesion* is not applicable. The consequence involves one change with a single architectural delta, i.e., adding the `TaskLog`. The relationship of this change to the issue is clear and the change size is tractable. Therefore, *Consequence-Rationale Association* and *Consequence Tractability* each received 1.

[2]Our evaluation considered decisions that included more than five changes not to satisfy this criterion, but this heuristic can be relaxed.

Decision Types	Hadoop		Struts	
	ACDC	ARC	ACDC	ARC
Simple	0.89	0.95	0.90	0.99
Compound	0.50	0.52	0.76	0.56
Cross-Cutting	0.61	0.76	0.78	0.77
Overall	0.72	0.72	0.81	0.71

Fig. 10. Average scores of recovered decisions per recovery-technique for Hadoop and Struts.

```
Rationales:
 Issue 1:
   Desc: Seperating user logs from system
         logs in map reduce
   ID  : HADOOP-489
Consequences:
 Change 1:
   Added: org.apache.hadoop.mapred.TaskLog
```

Listing 1. A simple decision from Hadoop v. 0.9.0

Listing 2 is a cross-cutting example from Hadoop 0.10.1. Although the rationales seem unrelated, after inspecting the code and issue logs, we realized that *LzoCodec* will be available only if the *Native Library* is loaded. Therefore, this decision received 0.5 for *Rationale Cohesion*.

```
Rationales:
 Issue 1:
   Desc: Implement the LzoCodec to support
         the lzo compression algorithms
   ID  : HADOOP-851
 Issue 2:
   Desc: Native libraries are not loaded
   ID  : HADOOP-873
Consequences:
 ...
```

Listing 2. Part of a cross-cutting decision from Hadoop v. 0.10.1

Figure 10 displays the average scores of the analyzed decisions, grouped by the decision type and the recovery technique used for uncovering the decisions. Figures 11 and 12 display the cumulative distributions of the decision scores for Hadoop and Struts, respectively. The right-leaning feature of these distributions indicates that higher-quality decisions are more prevalent than lower-quality ones. The threshold of acceptability for measuring precision is adjustable, but in our evaluation we required that a decision scores at least 0.5

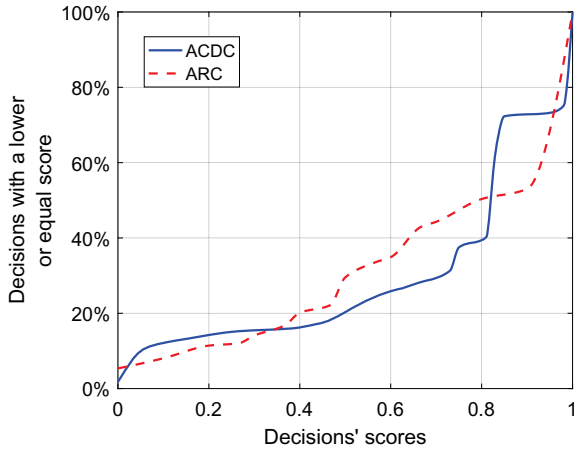


Fig. 11. Smoothed cumulative distribution of the decision scores for Hadoop.

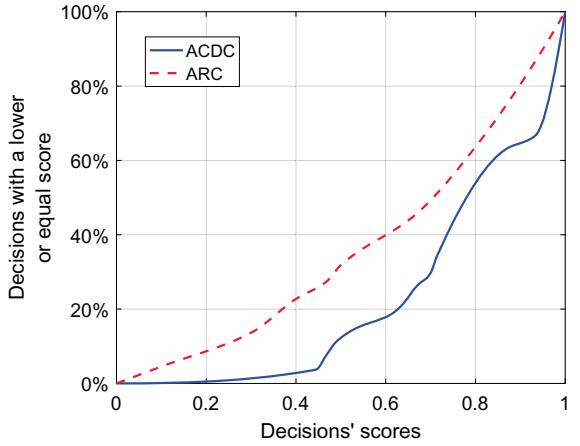


Fig. 12. Smoothed cumulative distribution of the decision scores for Struts.

in the majority (i.e., at least three) of the criteria. In our analyses, on average (considering both *ARC* and *ACDC*) 76% of the decisions for Hadoop and 78% of the decision for Struts met this condition. Figure 13 depicts a descriptive view of the results of our evaluation, classifying the decisions by the required criteria. The values denote the proportion of decisions that have at least partially satisfied the criteria corresponding to a given intersection.

Most of the unacceptable decisions were made in the newly introduced major versions of the two systems. This is consistent with prior findings: The number of architectural changes between a minor version (e.g., 0.20.2) and the immediately following major version (e.g., 1.0.0) tends to be significantly higher than the architectural change between two consecutive minor versions [4]. In these cases, the decision sizes (number of rationales and consequences) tend to be higher than our conservative thresholds, and these decisions tend to be rated as unacceptable. However, these decisions still provide valuable insight into why the architecture has changed.

The reason that the *ARC*-based decisions generally score lower (i.e., they are less right-leaning) than the *ACDC*-based

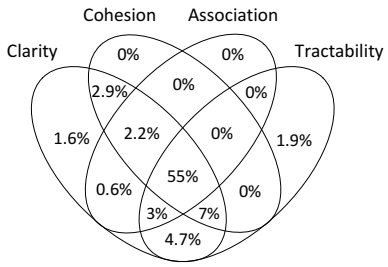


Fig. 13. Classification of the recovered decisions based on the satisfied criteria.

ones is due to the nature of changes extracted by *ARC*. As discussed previously, *ACDC* adopts primarily a structural approach to architecture, while *ARC* follows a semantic approach, which requires a higher level of system understanding. Therefore, attaining a conclusive rating for these decisions was not possible by only looking at the decision elements defined earlier. Our findings suggest that the uncovered decisions based on *ARC* are more suitable for experienced users.

C. Recall

Another target of our evaluation was the extent to which RecovAr manages to successfully capture the design decisions in our subject systems. Based on the definition of the architectural design decisions (recall Section II), every architectural change is a consequence of a design decision. We thus use the coverage of architectural changes by the identified design decisions as a proxy indicator for measuring RecovAr's recall.

Our initial analysis reported low recall values, indicating that a relatively small fraction of the extracted changes formed design decisions. The first row of Figure 14 displays the results of this analysis. The recall of the extracted architectural changes was consistently around 20% in our subject systems regardless of the used recovery technique. To understand the cause of this, we manually examined the detected architectural changes for which RecovAr could not locate the rationale. We were able to identify two major reasons why an architectural change was not marked as part of a design decision by RecovAr. The first was when architectural change was happening in off-the-shelf components that are integrated with the system and evolve separately. These can be third-party libraries, integrations with the other Apache software projects, or even changes in the core Java libraries that are detected by the recovery techniques. Examples of this phenomenon for Struts include changes to the Spring Framework's architecture [34], and for Hadoop changes to Jetty [8] and several non-core Apache Common projects. The second reason is what we call the "orphaned commit" phenomenon. Orphaned commits are those commits that conceptually belong to an issue, but (1) were not added to an issue, (2) have been merged with the code-base before their containing issues have been marked as resolved, or (3) a human error in the issue data rendered them useless for our approach (e.g., incorrectly specified affected version).

We consider orphaned commits a shortcoming of our approach that can affect its recall. Orphaned commits might also limit RecovAr's ability to recover the initial architectural

	Hadoop		Struts	
	ACDC	**ARC**	**ACDC**	**ARC**
Before Cleanup	20%	19%	21%	24%
After Cleanup	85%	67%	80%	63%

Fig. 14. RecovAr's recall before (top row) and after (bottom row) the clean-up of the raw-data.

design decisions that are not documented as issues. This is less concerning when issue trackers are used in tandem with project management tools for task assignments in the early stages of development. However, the imposed changes on a system's architecture do not capture the original intentions of the developers and architects. Therefore, we carefully inspected the architectural changes to eliminate the ones caused by external factors. In our inspection, we created a list of namespaces whose elements should not be considered architectural changes caused by the developer decisions. Partial lists of these namespaces for Hadoop and Struts are displayed for illustration in Listings 3 and 4, respectively. We verified each entry by searching the system's code repository and confirming that the instances where imported and not developed internally by the developer teams.

```
com.facebook.*
java.lang.*
org.apache.commons.cli.*
javax.ws.rs.*
...
```

Listing 3. Imported namespaces for Hadoop

```
com.opensymphony.xwork2.util.*
java.io.*
org.apache.commons.*
org.springframework.*
...
```

Listing 4. Imported namespaces for Struts

We subsequently reevaluated RecovAr's recall. The results are displayed in the second row of Figure 14. The recall was 73% on average after eliminating externally caused changes. This also reveals an interesting byproduct of RecovAr: by using RecovAr or a specially modified version of it, we can detect the parts of a system that are not developed or maintained by the system's core team. This information can be used for automatic detection of external libraries and dependencies in software systems, and can help the recovery techniques in extracting a more accurate view of a system's core architecture.

D. Threats to validity

We identify several potential threats to the validity of our study with their corresponding mitigating factors. The key threats to external validity involve our subject systems. We chose the two systems in our evaluations from the higher end of the Apache spectrum in terms of size and lifespan; each have a vibrant community, and are widely adopted. Another threat stems from the fact that both of our systems use GitHub and Jira. However, RecovAr only relies on the basic issue and commit information that can be found in any generic issue tracker or version control system. The different numbers of versions analyzed per system pose another potential threat to validity. This is unavoidable, however, since some systems simply undergo more evolution than others.

The construct validity of our study is mainly threatened by the accuracy of the recovered architectural views and of our detection of architectural decisions. To mitigate the first threat, we selected the two architecture recovery techniques, ACDC and ARC, that have demonstrated the greatest accuracy in a comparative analysis of available techniques [10]. These techniques are developed independently of one another and use very different strategies for recovering an architecture. This, coupled with the fact that their results exhibit similar trends, helps to strengthen the confidence in our conclusions. The manual inspection of the accuracy of the design decisions uncovered by our approach is another threat. Human error in this process could affect the reported results. To alleviate this problem, two PhD students independently analyzed the results to limit the potential biases and mistakes. Moreover, the inspection procedure was designed to be very conservative.

V. Related Work

Tyree et al. [38] described the importance of design decisions in demystifying the software architecture and filling in the shortcomings of traditional approaches, such as RM-ODP (Reference Model for Open Distributed Processing) [29], or 4+1 [17]. They devised a methodology for architects to document architectural design decisions, requirements, and pertinent assumptions. Other decision centric approaches (e.g., [7], [44]) have been proposed to direct the derivation of target architectures from requirements. These techniques aim to make design rationale reusable. RecovAr can augment these techniques and reduce the architects' burden by pointing to the existing decisions where such documents do not exist.

Jansen and Bosch et al. [13], [5] defined architectural design decisions and argued for the benefits of the invaluable information getting lost when architecture is modeled using purely structural elements. Several researchers focused on studying the concrete benefits of using design decisions in improving software system's quality [36], [21], and decision making under uncertainty [6]. Falessi et al. extensively studied design rationale and argued for the value of capturing and explicitly documenting this information [9]. A recent survey by Weinreich et al. [42] showed that knowledge vaporization is a problem in practice, even at the individual level. However, unlike RecovAr, none of these research studies have focused on automatic recovery of undocumented design decisions.

Roeller et al. [30] proposed RAAM to support reconstruction of the assumptions picture of a system, i.e., early architectural design decisions. A serious shortcoming of this approach is that the researchers need to acquire a deep understanding of the software system to reconstruct the assumptions. ADDRA [14] was designed to recover architectural design decisions in an after the fact documentation effort. It was built on the premise that in practice, software architectures are often documented after the fact, i.e. when a system is realized and architectural design decisions have been taken. Similar to RAAM, and unlike

our approach, ADDRA also relies on architects to articulate their "tacit" knowledge.

VI. Contributions and Future Work

In this paper, we took a step toward addressing the problems arising from knowledge vaporization and architectural erosion [19]. We formally defined the notion of an architectural design decision. We introduced RecovAr, a technique that uses a project's readily available history artifacts (e.g., an issue tracker or code repository), to automatically recover the architectural design decisions embodied in that system. We empirically examined how design decisions manifest in software systems, using two large, widely-adopted open-source software systems. While our approach may not recover all the design decisions in a software system, in our evaluation RecovAr exhibited high accuracy and recall. Finally, our developed methodology helps preserve design-decision knowledge in software projects.

There are a number of remaining research challenges that will guide our future work. There is a slew of information in software repositories that can help increase the accuracy of our approach [18]. These include comments, commit messages, documentations, pull requests, tests, etc. RecovAr can be extended with a summarization technique to provide succinct summaries of the recovered rationales and consequences. Furthermore, we will investigate models that employ RecovAr to predict the architectural consequences of issues based on their description thus helping engineers make better-informed decisions during design and code review time [32].

Acknowledgments

This work is supported by the National Science Foundation under grants no. CCF-1453474, CCF-1564162, CCF-1618231, and CCF-1717963, by the U.S. Office of Naval Research under grant no. N00014-17-1-2896, and by Huawei Technologies.

References

[1] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for change: A reconfiguration language for distributed systems. *Distributed Systems Engineering*, 1(5):313, 1994.
[2] Apache software foundation. http://apache.org/, 2018.
[3] Apache Hadoop. http://hadoop.apache.org/, 2018.
[4] P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering*, 2016.
[5] J. Bosch. Software architecture: The next step. In *European Workshop on Software Architecture*, pages 194–199, 2004.
[6] J. E. Burge. Design rationale: Researching under uncertainty. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 2008.
[7] X. Cui, Y. Sun, and H. Mei. Towards automated solution synthesis and rationale capture in decision-centric architecture design. In *WICSA*, pages 221–230, 2008.
[8] Eclipse Jetty. https://eclipse.org/jetty/, 2018.
[9] D. Falessi, L. C. Briand, G. Cantone, R. Capilla, and P. Kruchten. The value of design rationale information. *TOSEM*, 22(3):21, 2013.
[10] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *ASE*, 2013.
[11] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai. Enhancing architectural recovery using concerns. In *ASE*, pages 552–555, 2011.
[12] ISO/IEC 42010: 2011 systems and software engineering—recommended practice for architectural description of software-intensive systems. Technical report, ISO, 2011.
[13] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *WICSA*, pages 109–120, 2005.
[14] A. Jansen, J. Bosch, and P. Avgeriou. Documenting after the fact: Recovering architectural design decisions. *JSS*, 81(4):536–557, 2008.
[15] Jira. https://www.atlassian.com/software/jira, 2018.
[16] P. Kruchten. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen Workshop on Software Variability*, pages 54–61, 2004.
[17] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.
[18] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. Automated extraction of rich software models from limited system information. In *WICSA*, pages 99–108, 2016.
[19] D. Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *ICSA*, 2018.
[20] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *MSR*, pages 235–245, 2015.
[21] I. Malavolta, H. Muccini, and V. Smrithi Rekha. Supporting architectural design decisions evolution through model driven engineering. *Software Engineering for Resilient Systems*, pages 63–77, 2011.
[22] N. Medvidovic. ADLs and dynamic architecture changes. In *Second International Software Architecture Workshop*, 1996.
[23] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *TSE*, 26(1):70–93, 2000.
[24] K. Muşlu, Y. Brun, and A. Meliou. Data debugging with continuous testing. In *ESEC/FSE NIER*, pages 631–634, 2013.
[25] K. Muşlu, Y. Brun, and A. Meliou. Preventing data errors with continuous testing. In *ISSTA*, pages 373–384, 2015.
[26] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE*, pages 177–186, 1998.
[27] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman. Are developers aware of the architectural impact of their changes? In *ASE*, pages 95–105, 2017.
[28] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), 1992.
[29] RM-ODP. http://www.rm-odp.net/, 2018.
[30] R. Roeller, P. Lago, and H. van Vliet. Recovering architectural assumptions. *JSS*, 79(4):552–573, 2006.
[31] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, 2003.
[32] A. Shahbazian, D. Nam, and N. Medvidovic. Toward predicting architectural significance of implementation issues. In *MSR*, 2018.
[33] M. Shahin, P. Liang, and M. R. Khayyambashi. Architectural design decision: Existing models and tools. In *WICSA/ECSA*, pages 293–296, 2009.
[34] Spring application framework. https://spring.io/, 2018.
[35] Struts. http://struts.apache.org/, 2018.
[36] A. Tang, M. H. Tran, J. Han, and H. Van Vliet. Design reasoning improves software design quality. In *QoSA*, pages 28–42, 2008.
[37] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.
[38] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, 2005.
[39] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. In *RE*, pages 187–193, 1999.
[40] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Working Conference on Reverse Engineering*, pages 258–267, 2000.
[41] R. Weinreich and I. Groher. Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review. *Information and Software Technology*, 80:265–286, 2016.
[42] R. Weinreich, I. Groher, and C. Miesbauer. An expert survey on kinds, influence factors and documentation of design decisions in practice. *Future Generation Computer Systems*, 47:145–160, 2015.
[43] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *International Workshop on Program Comprehension*, pages 194–203, 2004.
[44] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster. Reusable architectural decision models for enterprise application development. In *QoSA*, pages 15–32, 2007.