Roles and Impacts of Hands-on Software Architects in Five Industrial Case Studies

Inayat Rehman
Mehdi Mirakhorli
Rochester Institute of Technology
{ixr8582, mxnvse}@rit.edu

Meiyappan Nagappan
University of Waterloo
mei.nagappan@uwaterloo.ca

Azat Aralbay Uulu
Matthew Thornton
Rochester Institute of Technology
{axa8095, mjthornton}@rit.edu

ABSTRACT

Whether software architects should also code is an enduring question. In order to satisfy performance, security, reliability and other quality concerns, architects need to compare and carefully choose a combination of architectural patterns, styles or tactics. Then later in the development cycle, these architectural choices must be implemented completely and correctly so there will not be any drift from envisioned design. In this paper, we use data analytics-based techniques to study five large-scale software systems, examining the impact and the role of software architects who write code on software quality. Our quantitative study is augmented with a follow up interview of architects. This paper provides empirical evidence for supporting the pragmatic opinions that architects should write code. Our analysis shows that implementing architectural tactics is more complex than delivering functionality, tactics are more error prone than software functionalities, and the architects tend to introduce fewer bugs into the implementation of architectural tactics compared to the developers.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; *Software implementation planning*; *Programming teams*;

KEYWORDS

Hands-on Architect, Coding, Design Profile, Human Modeling

ACM Reference Format:
Inayat Rehman Mehdi Mirakhorli, Meiyappan Nagappan, and Azat Aralbay Uulu Matthew Thornton. 2018. Roles and Impacts of Hands-on Software Architects in Five Industrial Case Studies. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.
https://doi.org/10.1145/3180155.3180234

1 INTRODUCTION

The architectural design of any complex system is driven by quality concerns such as reliability, availability, security, and performance [3, 9, 21, 25]. Architects are often responsible for addressing such concerns through design decision-making, which often involves implementing and deploying standard architectural tactics (e.g., audit trails for security, load balancing for performance, active redundancy for availability) [3].

In the software engineering community, there are divided and diverse pragmatic opinions regarding *who needs architects* [17], *what responsibilities architects should have* [7], and *whether architects should code* [8]. Some practitioners argue that since architects are responsible for the integrity of the entire system, and for satisfying business goals while mitigating risks, they should not write code. In this mindset, architects should delegate other lower-level decisions, including those regarding coding, to developers who typically have a narrower set of responsibilities [1, 3, 19].

On the other side of this spectrum is a group of practitioners who oppose the "no coding" mentality and who argue that software projects need "Hands-on Architects", architects that also write code [6, 7, 11]. Some critics even take this criticism further and argue that so called "PowerPoint architects" [37] are both ineffective and expensive. "PowerPoint" architects are architects who join project kickoff meetings, draw a variety of diagrams, and leave before implementation starts. They are not effective because of their lack of ongoing feedback during the development cycle. [37].

Some other practitioners choose a moderated perspective. Martin Fowler, in his recent keynote talk [13], explores different ways to stimulate collaboration and communication between programmers and architects, advocating for architects to pair-program with developers. Ward Cunningham [11], in his anti-pattern called "Architects Don't Code", addresses the same issue, presenting a development problem in which "The Architect responsible for designing your system hasn't written a line of code in two years." Ward encourages communication between architects and developers during implementation.

In his new book, "Software Architecture for Developers" [6], Simon Brown advocates a transition from "ivory tower" architects to architects who take on roles of coaching and collaboration. All of these practitioners agree on the necessity for a form of architectural design. Arguing that most software developers are not architects and lack extensive design skills [6], Simon Brown advocates for increasing developers' design knowledge.

These discussions are just a few samples of pragmatic perspectives; following any of these opinions depends largely on trust in the person advocating it. While there have been numerous studies [12, 14, 15, 22, 24, 35, 36] on the impact of architecture on a software system, no empirical study has examined the influence that software architects can have on coding activities. The novel contribution of this paper lies in using both qualitative (interviews)

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180234

and quantitative (mining software repository) techniques to answer the following question:

> **Problem Statement:** Is there any empirical evidence to support that software projects will benefit from Hands-on software architects?

Our study finds that architects write code and architectural tactics implemented by these hands-on architects will have less bugs than those implemented by (non-architect) developers. In four out of five case studies, the code committed by architects, had significantly fewer bugs than code committed by developers. The intermediary analytical data and the findings of this study are confirmed by the architects in each case study.

The **contribution** of this paper is twofold: First, it adapts a **human behavioral modeling** approach from psychology [28] to model individual developers' decision-making behaviors, architecture-level problem-solving, and soft and technical skills in the area of software design. The outcome of this approach is a model called the **developer's design profile** that captures an architect's portfolio of architectural experience, activities, and applied skill sets in a project. This approach uses the developers' information, such as their email address, to identify the their technical page on social media (e.g., LinkedIn); based on that, the model creates a design profile for each developer. For the developers without online profiles, we used other resources (e.g., discussion forums), or directly contacted them to create their design profiles. Similar approaches have been previously used to connect developers to their social profiles, but no researcher has mined this information to create developers' design profiles and architectural experience portfolio. The design profiles represent developers' technical skills, work experience, and whether they have previously served in an architect role. Our interviews with developers showed that, this approach can accurately identify the original architects. These profiles are used to examine how architects can help with low-level programming activities.

This paper's second contribution is in providing empirical evidence, driven from mining software repositories to examine pragmatic opinions on: *whether architects write code*, *why software projects should have hands-on architects*, and, more broadly, *what are the impact of developers' design background on software qualities*.

2 CASE STUDY DESIGN

To test our research questions, we designed a *two-stage study* in which first bottom-up data analytics-based approaches were used to investigate each research questions, then in the second stage, the findings were confirmed and substantiated using interviews with the original architects.

2.1 Research Questions

In this study, we investigate the following research questions,and offer our rationale for each:

Do architects write code? In a recent blog post [29] on if architects should write code, that was intensely discussed by developers on reddit/programming [30], we observed two distinct groups of comments - one set of developers stated that they have never seen an architect who writes code and another group stated they have always assumed architects cannot write code. We wanted to empirically investigate this evidence to determine whether architects wrote code or not.

What type of code do architects write? Assuming the answer to the previous question was yes, we wanted to investigate what form of coding activities these hands-on architects contributed to.

Is there any empirical evidence to support that software projects will benefit from Hands-on software architects? Lastly, we wanted to examine the potential impact of architects contributing to coding activities.

2.2 Studied Systems

Five open-source projects were selected for in-depth analysis and studied in our empirical experiments. For each of these projects, we extracted the change logs from the version control system, including a list of Java files modified as part of a release, and a separate list of defect fixes. The projects were chosen according to the following criteria:

- **(i)** Representation of different application domains, including areas such as performance-centric and business-centric applications.
- **(ii)** Availability of change logs with commit messages formatted in a way that differentiates between a defect fix and other types of code modifications.
- **(iii)** Accessibility of design documents.
- **(iv)** Utilization of a wide range of architectural tactics.
- **(v)** Representing large-scale industrial systems.

These projects were carefully solicited from a list of 47 open source projects that we previously used in an in-depth architecture study [33]. From these 47 projects, we selected those matching the above criteria and that had many architectural tactics in common. We did not include utility projects (e.g. Apache Camel) that primarily implemented architectural patterns and did not have any user centered functionality. Table 1 summarizes the description of each project and the tactics implemented in them.

We designed a ***two-stage*** study set-up to conduct our investigation, wherein **data analytics-based** approaches were used to answer research questions, and then followed by **semi-structured interviews** to confirm the findings and obtain a deeper understanding. **Interview goals** were very specific and are described in the beginning of each of the following sections. We use interviews as a means to confirm our the intermediary analytical-data that we base our conclusion on. This differs from traditional interview or survey studies that aim to obtain consensus from large numbers of participants. We used interviews as an unorthodox approach to work closely with architects and to confirm the data and results obtained through our data analytics approach, as well as to substantiate the findings through architects' first-hand experiences.

2.3 Concepts and Definitions

In order to investigate the problem statements and research questions framing our study, we first define the following key concepts and terms:

- *Architectural Tactic*: Formally defined as "a means of satisfying a quality-attribute-response measure by manipulating

Table 1: Projects Studied

Project Name	Project Description	Tactics Used
Camel	A versatile open-source integration framework based on known Enterprise Integration Patterns.	Audit, Authenticate, Checkpoint, Credential, Heartbeat, Load Balancing, Ping-Echo, Pooling, RBAC, Scheduler, Secure Session
Hadoop	A map-reduce framework that allows for the distributed processing of large data sets across clusters of computers.	Audit, Authenticate, BRAC, Checkpoint, Heartbeat, Load Balancing, Ping-Echo, Pooling, Scheduler, Secure Session, Validation Interceptor
Hive	A data warehouse infrastructure for providing data summarization, query, and analysis.	Redundancy, Audit, Authenticate, Checkpoint, Credential, Heartbeat, Kerbrose, Load Balancing, Ping-Echo, Pooling, RBAC, Scheduler, Secure Session, Validation Interceptor
OfBiz	An open-source ERP product for the automation of enterprise processes.	Audit, Authenticate, Checkpoint, Credential, Load Balancing, Ping-Echo, Pooling, RBAC, Scheduler, Secure Session, Validation Interceptor
OpenJPA	An open-source implementation of the Java Persistence API specification.	Audit, Authenticate, Checkpoint, Credential, Load Balancing, Ping-Echo, Pooling, RBAC, Scheduler, Secure Session
Pig	A convenient tool created at Yahoo to analyze large datasets easily and efficiently.	Authenticate, Checkpoint, Credential, Heartbeat, Kerbrose, Load Balancing, Ping-Echo, Pooling, RBAC, Scheduler, Secure Session

some aspects of a quality attribute model through architectural design decisions," [1] architectural tactics describe design solutions that satisfy a wide range of quality requirements.

- *Tactical Commits*: Source code commits that partially or fully implement an architectural tactic.
- *Functional Commits*: Source code commits that pertain not to the implementation of an architectural tactic, but rather to the functionality of the software system or the support utilities.
- *Tactical File*: A source file that contains tactic implementation (at least one tactical commit).
- *Functional File*: A source file that only contains functional commits and does not contribute to the implementation of an architectural tactic.

3 DO ARCHITECTS WRITE CODE?

In this section, we describe a **human modeling approach** to identify architects in each project and answer RQ①. We followed up the approach with interviews to: (i) confirm whether our human modeling approach correctly identifies the original architects of the system; and (ii) ask the architects to elaborate on their programming roles and responsibilities (if any).

3.1 Approach

We adapt the framework presented by Pew and Mavor's *Modeling Human and Organizational Behavior* [28] and adapt it for documenting and modeling human behavior in software development environments. This original approach argues that human modeling should focus on capturing human behavior in certain areas of expertise relevant to an organization. Example areas identified by the authors focused on capturing human behavior with regard to *multitasking*, *memory and learning*, *decision making*, *situation awareness*, *planning*, and *behavioral moderators*. Then they used various data collection mechanisms to document an individual's performance in those areas.

Figure 1 illustrates an overview of the approach followed to answer our first research question. The first step was to determine the areas of expertise to be used in the human modeling approach and to distinguish architects from developers. The second step was to model each developer based on evidence of their competency in those areas. The outcome of this step was the developers' *design profiles*. Example of such profiles are shown in Figure 2 (right side). In the third step of the approach, we peer-reviewed all the profiles and grouped them into architects and developers (non-architect) in a group meeting. In the fourth step of our approach, we interviewed architects to confirm the results of our modeling approach and confirm/update list of architects. In last step, we reviewed the code to analyze the architects' contributions to the project repository. Below, we discuss each of these steps in more detail.

3.1.1 Identifying the Key Areas of Architecture Expertise Modeling. We used Philippe Kruchten's article on "What do software architects really do?" [26] to elicit the skills and behavioral areas relevant to a software architect or architects' team. This article discusses a set of commonly agreed-upon architect roles, responsibilities, and technical and behavioral skills. From these, we compiled the list of key areas for our human modeling approach, which distinguishes architects from regular developers. These areas included assessing individuals *decision making*, *project planning*, *risk/trade-off analysis*, *working out risk mitigation strategizing*, *problem-solving and moderation*, and *architecture mentoring*.

decision making is an expected behavior modeling area that entails the individual's ability to understand the requirements, qualities, making choices, synthesizing a solution and exploring alternatives. *Project Planning* reflects and individual's ability to understand a software's composition, and work-breakdown-structure, which enables him/her in planning and effort estimation aspects or partition of work across multiples teams. *Risk/trade-off analysis*, is a key expertise of an experienced architect, comprises the capacity to identify risks and trade-offs. *Risk mitigation strategizing*, a key design skill, captures an individual's ability to devise mitigation techniques to minimize system risks. minimize the risks in the system, a key design skill. *problem-solving and moderation* define a key behavioral aspect of architects; because of their technical expertise, architects are drawn into problem-solving and fire-fighting activities that go beyond solving strictly architectural issues. *Architecture mentoring*, is the last expected behavior modeling area that can help distinguish architects from developers by identifying individuals who provide regular design reviews, write guidelines, and mentor developers in order to maintain the architectural integrity of the system.

We created a template *design profile* to be used to model each developer. This template contained sections for the following information (if available):

- **Personalized background details:** An engaging description of the developer's broad knowledge, including his/her name and social and technical background.
- **Developer's Major Roles:** A list of the developer's highlighted roles in his/her LinkedIn profile, or identified from other resources.
- **Key Behavioral and performance areas:** A list of areas in which evidence of the developer's proficiency were present.

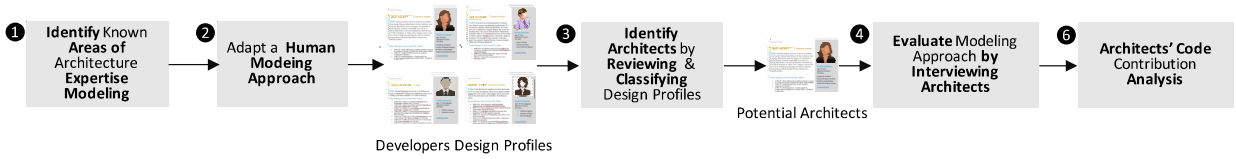

Figure 1: Approach followed to examine RQ①

- **Sample Commit Messages:** A sample of the developer's commit messages that provide evidence of his/her expertise in key architectural performance and behavioral areas.
- **Length of expertise and job titles:** This includes information about the developer's age (extracted or estimated), previous experience in software development and IT, as well as their length of engagement in each position.

3.1.2 Model Developers Design Profiles. We created and ran a script to parse the commits of all the GitHub repositories and identify each developer who contributed code. From this list, we first removed the developers who were not active: had less than 5 commits to the entire project. This threshold was chosen heuristically and eliminated a small fraction of developers who did not significantly contribute to the project. We used the name and email address of these developers to search for their social profile on the web. The search was manually conducted, and included: (i) an analysis of the developer's LinkedIn profile; (ii) a search for software publications; (iii) inspection of GitHub commit messages; and lastly (iv) a generic Google search. The developers with multiple email accounts were identified and their profile was merged. After scraping these sources, the data was compiled and analyzed to provide a binary (Yes or No) answer to the 5 key-behavioral areas defined above. We decided to treat these key areas as binaries, because evaluating such aspects in a more complex scale was difficult. We primarily used the content from developers' commit messages and issue-tracking posts to identify evidence of their activities in *decision-making*, *project planning*, *risk/trade-off analysis*, *risk mitigation*, and *problem-solving*. This information, along with the relevant commit messages, was compiled onto a single developer-profile (Figure 2). These profiles were reviewed by three members of our team to assess the quality of the data analysis and revise if necessary.

3.1.3 Identify Architects. The design profile created for each developer was used to evaluate their design background and architecture knowledge. Each profile was reviewed by three of our team members, which allowed us to confidently conclude whether the individual should be categorized as an *architect* or *developer (non-architect)*. For many of the developers, this was an easy task, as they have already indicated the architect role in their LinkedIn profile. We had a negligible number of profiles that required us to look at other sources to classify them as *architect*. This approach resulted in an accurate representation of the developers' architecture experience.

3.1.4 Evaluating Correctness of the Models in Identifying the Architects. In this step of the experiment, we contacted all the architects to verify the information in their design profile and confirm whether they were the actual project architects. We conducted an initial interview with them, asking them the following questions:

- *IQ1:What is your role in the project?*
- *IQ2: We found that the following developers have previously served as architect in other projects, did they participate in the architecture design of your project, too?*
- *IQ3: Who else is/are the architect(s) of the system?*

Through this interview process, we identified the original architects of each project.

3.1.5 Architects Code Contribution Analysis. In this step of the experiment, we developed a *code contribution extractor* script that mined the commits histories, extracted each developer's (architects in this experiment) added/modified code snippets. This resulted

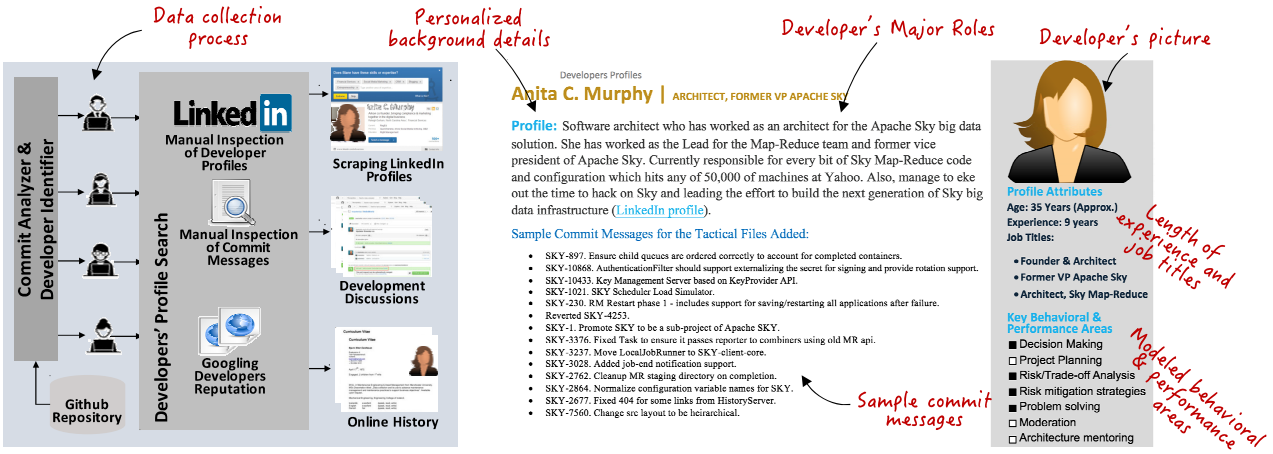

Figure 2: Developer "Design Profile" Creation Process and Sample Profile

in a code contribution matrix for architects. We also conducted a follow up semi-structured interview with software architects to confirm the results of our findings, and also obtain qualitative data explaining these results.

- *IQ4: Did your project's architect write code?*
- *IQ5: If so, then to what extent did they participate?*
- *IQ6: What was theirs/your role?*

3.2 Results

Table 2 demonstrates the accuracy of our design profile-based approach to identifying the project architects. The first rows show the total number of active developers in each project at the time of this study [1], while the second row indicates the total number of architects identified using our human modeling approach. In the third row are true positive cases, confirmed by interviewing the developers, whereas the fourth row shows false positive cases, in which we incorrectly identified a developer as an architect. Such cases occurred mostly for developers who had served as an architect in a previous job (as claimed in their LinkedIn profiles), but were not part of the architect team in the project we studied. The fifth row shows the true negative cases, wherein we failed to identify individuals who were the original architects of the projects. Last row presents the accuracy of our approach in identifying architects. After interview confirmation, we updated the list of architects based on the collected feedback and used that list for the remaining of our study.

Table 2: Results and Accuracy of Human Modeling Approach in Identifying Architects

	Hadoop	Hive	OfBiz	OpenJPA	Pig
#Developers	77	61	24	20	18
#Architects	7	7	4	4	2
#Confirmed	6	6	4	4	2
#Incorrect	1	1	0	0	0
#Missed	0	0	0	0	0
%Accuracy	85%	85%	100%	100%	100%

Table 3 shows the percentage of code commits contributed by the architects in each project. Architects code contribution varies from 27% in Hadoop and Pig, to 47% in OpenJPA. These results indicate that the architects of the system in our study have extensively participated in the coding activities. While this result may seem obvious to some, there has been no empirical evidence so far that shows that architects do or do not write code. These results indicate that, at least in the studied projects, architects do indeed write code.

Table 3: Percentage of Code Commits by Architects

	Hadoop	Hive	OfBiz	OpenJPA	Pig
% of Commits	27%	32%	41%	47%	27%

[1] September 2017

In reviewing our findings with the architects of these systems, all architects confirmed that they were actively engaged in design as well as coding, testing, and other development activities.

One of the architects in Apache Hadoop project, offered the following in response to the question of whether Hadoop architects write code:

> *"We do all get our hands dirty on a regular basis. I also think architects need to write tests, to understand how to design for testability, and get involved in the low-level issues of metrics and monitoring, of deployment and fielding support calls: otherwise you don't get systems designed for management and supportability. I'm pleased to say my colleagues and I do get to do all this."*

This architect added that, as a side effect of growth, after a few years, several of the architects in the core team moved into project management and stopped writing code.

An architect from Apache Hive states that

> *"There aren't traditional architects in Hive. What I have done is substitute -driver- (including design and implementation) of a major feature or subsystem in for -architect-, we made both macro and micro level decisions and played roles as designers, implementers, and testers."*

Therefore, quantitative data as well as confirmation from the architects of these systems provide a positive answer to our research question RQ① that:

> There are architects with several years of experience who actively write code.

4 WHAT TYPE OF CODE DO ARCHITECTS WRITE?

In this section we describe the **code classification and analysis approach** used to investigate what type of code architects write and to answer RQ②. We followed up by interviewing the architects to: (i) confirm all the intermediary data; and (ii) substantiate the final results. The architects code contribution was confirmed through interviews. This resulted in an accountability matrix for each architect, indicating who had modified a tactical files and who had modified a functional files. The code contribution matrix was discussed with the architects to examine their responsibilities during development and the nature of their coding activities.

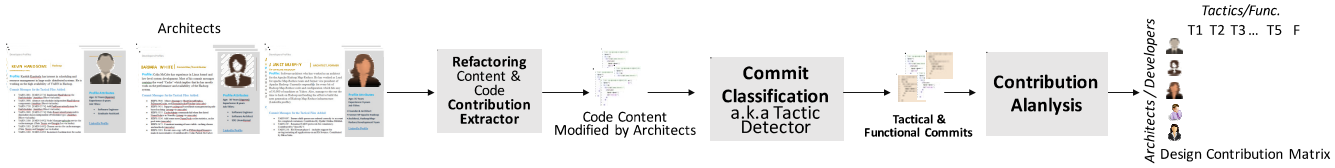

Figure 3: Approach followed to examine RQ②

4.1 Approach

Figure 3 provides an overview of our approach to investigating RQ②. The first step was to extract code commits from each developer and identify those committed by architects. We used our *code contribution extractor* script (described in previous section) to extract this data. In the second step, we adapted an existing technique [34] to classify code commits as functional or tactical. To ensure accuracy, we also confirmed the results of code classification using the manual code review of tactical commits. Our last step comprised an analytical module that created statistics about the type of code written by architects as well as developers. This technique enabled us to build a *design contribution matrix* that accurately identified the type of code commits made by an architect.

4.1.1 Code Contribution Extractor. We used our *code contribution extractor* script to identify the commit hash and extract the commit content (code snippets added/deleted/modified) for commits made by architects as well as developers.

4.1.2 Code Classification Approach. All the projects' code commits were analyzed to identify whether they implemented (or contributed to) tactical or functional system aspects. As it was infeasible to manually evaluate each code commit, we adapted a classification algorithm [34] to detect architectural tactics and differentiate them from functional code commits. The tactic classifier included three phases: preparation, training, and classifying [34]. In the **preparation phase**, standard information retrieval techniques are used to prepare the data for training and classification purposes. For example, all source code is preprocessed by splitting source code variable names into primitive parts, removing stop words, and stemming words to their root forms. In the **training phase**, a set of textual descriptions of the tactics were processed to produce a set of *weighted indicator terms* that are considered representative of each tactic type. For example, a term such as *priority* is found more commonly in code related to the *scheduling* tactic than in other parts of the code, and therefore receives a higher weight for that tactic. Finally during the **classification phase**, the indicator terms learned in the training phase are used to compute the likelihood that a given source code is associated with a given tactic. Each source code is assigned a score for each tactic, and source code snippets scoring over a pre-defined threshold (t_{cl}) are labeled as being associated with the relevant tactic. In this way, it is possible for a source code to be associated with more than one tactic. Tactic classifiers have been trained to find the following architectural tactics: *asynchronous method invocation*, *audit trail*, *authentication*, *checkpoint and rollback*, *hash-based method authentication*, *heartbeat*, *kerberos*, *ping-echo*, *resource pooling*, *role-based access control (RBAC)*, *scheduling*, and *secure session management*, *Validation Interceptor*.

As the tactic classifier had previously been trained to detect the tactics used in this study, training was not repeated for this paper. Instead, the code commits of the five projects in our study were classified using the indicator terms learned from previous training sessions [34]. Based on previous experiments reported elsewhere [31], the classification threshold t_{cl} of 0.5 obtained a set of tactic-related code snippets with a high degree of confidence. Several previous experiments [31] have shown high accuracy of such classification techniques, which is confirmed with our extensive manual inspection of the results we obtained.

Using this technique, we classified the commits by each developer as tactical or non-tactical. This was done to determine which developers contributed to tactical or functional system fragments. Although we could have conducted this classification at a file level, for more accuracy we performed it at commit level. Therefore, for each refactoring content committed by a developer, we knew whether it changed a tactical code or only impacted a functional-code (non-tactical areas of the code).

4.1.3 Analysis of Architects' Code Contributions. We constructed the *design contribution matrix*. displays this matrix: each row represents a developer/architectâĂŹs contributions. Each column referrers to a tactic, and the values of the cells represent the total number of a developerâĂŹs commits for a specific tactic. Last column represents each developerâĂŹs number of non-tactical (or functional) commits. From the design contribution matrix, we calculated the *commit-type metrics*, defined as::

- *Architect-tactical*: Number of tactical commits made by an architect to a file implementing the tactic.
- *Developer-Tactical*: Number of tactical commits made by a non-architect developers to a file implementing the tactic.
- *Architect-Functional*: Number of functional (non-tactical) commits made by an architect to a non-tactical file.
- *Developer-Functional*: Number of functional commits made by a non-architect developer to a non-tactical file.

We conducted another round of interviews with architects to confirm their design contribution matrix established using our data analytic technique, and to obtain detailed information on their design and coding activities. The initiating questions were:

- *IQ7: Did you make macro-level or micro-level decisions?*
- *IQ8: Which parts of the system did you work on?*
- *IQ9: We observed that you actively contributed to the implementation of the following tactics, do you confirm that?*

4.2 Results

Table 4 reports the total number of tactical-code and functional-code commits made by developers, and architects. These numbers show that architects write a non-trivial amount of tactical and non-tactical code. Furthermore, developers also contribute to tactics, as well as more heavily to the functional fragments of the system.

Table 4: Commit-Type Contribution Metrics

Commit-Type	Hadoop	Hive	OfBiz	OpenJPA	Pig
Architect-Tactical	4122	4104	1766	1087	109
Architect-Functional	24940	15234	13450	19443	4214
Developer-Tactical	15318	5222	1458	460	227
Developer-Functional	62589	34390	19666	22130	10983

Architects in the five case studies confirmed that they contributed to both tactical and functional system fragments. They also confirmed that they have made both *macro-level decisions* impacting several components of the system as well as *micro-level decisions* impacting specific behavior, properties or components.

About macro-level architectural decisions, an anonymous architect stated that:

> *"In this project, two founders were the original architects and defined the overall architecture/structure of project. Natalia (anonymized) also has laid down the most fundamental aspects of the project: the entity engine, the service engine, the data model, the OFBiz-specific DSL named Minilang."*

Regarding the micro-level contribution this architect adds that:

> *"Jack (Anonymized) joined when the product was mature, he worked on implementing patterns/tactics and refactoring various aspects of the framework like the implementation of the Minilang DSL and did some work on concurrency related enhancements. I helped with enhancing the performance architecture using pooling and multi-threaded design"*

An architect discussing developers' contributions to tactics added that:

> *"There are certain tactics, that functional developers are rarely, if ever, touching these parts of the project."*

On the same topic, a third architect from the Hive project mentioned that:

> *"In an open source project, you don't have the ability to dictate architecture decisions. It's all persuasion. So certainly in availability and reliability solutions that I've developed, I spent time building and communicating a vision for how things should work. But things didn't always end up the way I envisioned them. Had I had dictatorial powers some of the details in HCatalog would have ended up different than they did."*

Architects write code to implement architectural tactics to address quality concerns. They also write code to shape the initial structure of the projects based on their mental model. Some architects limit their coding to specific tactics or patterns. Some architects write tactical, functional code, tests and engage in other development activities.

5 WILL SOFTWARE PROJECTS BENEFIT FROM HANDS-ON ARCHITECTS?

Results from the previous section demonstrate that architects write both tactical and functional code. The interviews further indicated that architects' develop architectural tactics, shape the structure of the system, and regularly communicate these design decisions to the other developers. Our discussion motivated a follow-up study to investigate whether architects do a better job in implementing high tactics. In this section we describe a **code quality and accountability analysis** conducted at the file-level in order to answer RQ③. Using this approach:

(i) We analyzed the defect density and volatility(change frequency) of tactical and functional files independent of the contributors.
(ii) We performed such analysis on tactical files with respect to whether architects or developers committed the code.

Statistical tests were performed to examine the impact of architects writing code and to compare that with non-architect developers in each project. We followed up the quantitative analysis through interviews of contributors in the studied projects to confirm the results of our findings, obtain qualitative data explaining our results, and gather first-hand experience from architects on tactical-code quality issues.

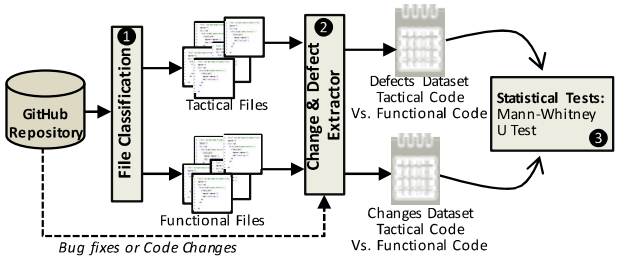

Figure 4: Approach for Tactical vs. Functional Code Quality Analysis

5.1 Approach for Tactical vs. Functional Code Quality Analysis

In this experiment, we compared the quality of tactical and functional files independent of contributors or ownership.

We designed a data analytic-based approach to examine and compare the *change frequency* and *bug-density* in each projectsâĂŹ tactical and functional files. Figure 4 illustrates an overview of our approach. To perform this analysis, we used tactic classifier at the source file-level to classify each file as tactical or functional. Then we extended the data analytic approach used in the previous section and developed a defect extractor to calculate: (i) modification frequency; and (ii) defect density for both tactical and functional files. Finally, the third step entailed performing a statistical test on both datasets to validate our motivating hypothesis.

5.1.1 File Classification. In this step, we used the tactic classifier (described in section 3) and used it against the source files (instead of commits), to classify each file as tactical or non-tactical. A

5.1.2 Change and Defect Extraction. *Change & Defect Extractor* parses the commit logs of projects and identifies each tactical or functional fileâĂŹs number of changes and defects/bugs. This component then generates two distinct datasets. The first dataset is used to compare the defect density of tactical files versus functional files. The second dataset compared the change frequency in both types of files.

5.1.3 Statistical Analysis. The final step of the experiment was to validate the motivating hypothesis using the two datasets collected from all five projects. The obtained datasets do not follow a normal distribution: the two distributions are skewed and have

differing shapes; and the independent variables consist of two categorical, independent groups that represent tactical file and functional file. There is no relationship between the observations in each group of the independent variable, or between the groups themselves.

Based on all these characteristics, the Mann-Whitney U test (MWU) is a recommended statistical test [10]; therefore, we used it to compare the change frequency and and bug density between tactical and functional files.

Table 5: Results for comparison of change frequency

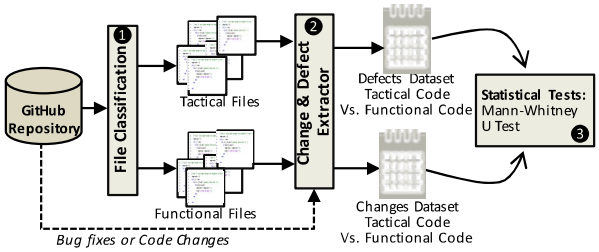

Table 6: Defect Density Results

	Hadoop		Hive		OfBiz		OpenJPA		Pig	
p-value	*p=0.000*		*p=0.000*		*p=0.000*		*p=0.000*		*p=0.002*	
	Mean	**M.Rank**	**Mean**	**M.Rank**	**Mean**	**M.Rank**	**Mean**	**M.Rank**	**Mean**	**M.Rank**
Functional	2.63	2657.76	2.29	1631.31	1.18	557.53	2.53	2016.85	3.45	782.15
Tactical	11.83	3789.13	12.07	2399.24	5.44	2399.24	9.58	2502.81	12.64	1157.89

5.1.4 Results. The MWU test results are reported here. Since the two distributions have different shapes, the MWU test can only be used to compare "mean ranks," and not median [10]. Tables 5 and 6 report the mean rank (M.Rank) calculated in MWU test and P-values in each project. To provide additional data points about the differences between the two groups, we also report means for bug density and change frequency. However, we should emphasize that M.Rank is the actual output of MWU test and the differences between two groups is judged accordingly. Mean or average is reported because it might be more tangible for non-academic readers.

Change Frequency: The test found that there is a statistically significant difference between mean and mean-rank of change frequency for tactical and functional files (p-value< 0.001), providing evidence that tactic related files changed more frequently than non-tactic related ones. For instance, the mean change frequency in the ofBiz system for tactic-related classes was 53.32, compared with an average change of 19.17 for functional-files. Similarly, for other projects, the average change frequency for tactic-related files was higher than the functional files. A similar pattern was observed in the mean-rank of the changes across the studied projects.

Defect Density: We performed a similar analysis of bug density for tactical files versus functional files. The non-parametric test indicated that tactic related files had significantly more bugs than functional ones (p-value< 0.001). For example, the average number of bugs in tactical files for Apache Hadoop was 11.83, compared to an average of 2.63 bugs for functional files. For Apache Hive, the average number of bugs was 12.07 for tactical files and 2.29 for functional files.

> Tactical-code undergoes a greater number of changes and contains more defects than functional code.

These results generated a number of hypotheses, as well as questions to follow up on with the architects. Our first hypothesis was that *tactical files undergo more rigorous tests and therefore have more bugs.* The second hypothesis was that, *tactics are more complex and harder to implement for a developer with less of a design background.*

We reached to architects to discuss the results but also designed experiments to investigate the relationship between developers-versus-architects contribution and defects in tactical files.

- *IQ10: How was your testing effort distributed in the project?*
- *IQ11: Do you think tactical files had more bugs because they were more rigorously tested*
- *IQ12: Are tactics inherently more complex and harder to implement?*

Architects responded similarly to these questions. An architect from the OfBiz project mentioned that:

> *" IQ10: The community focused more on functional tests since start. IQ11: I think it's the contrary, it's because we have less tests for the tactics and it's now harder to add them than it would have been from start. "*

All the architects uniformly indicated that tactics were more complex to implement and maintain compared to the functional aspects of the systems.

An Architect from Hadoop added that some of the tactical bugs were more "*firefighting*":

> *"IQ10: What may be surfacing is 'stuff done with time for thinking, review and test-first development' and 'stuff we rush out the week before a release', to stop Facebook clusters crashing, or simply to diagnose why a large US bank is getting error messages without stack traces in the kerberos code".*

An architect from Hive project added that:

> *"High availability, performance, and security share several traits that make them bug prone. They are complex. They are areas many people don't spend a lot of time working in. They are very hard to unit test. To me the key here is having people with experience in these areas.".*

5.2 Approach for Architects vs. Developers Tactical-Code Quality Analysis

Our interviews indicated that architects must often communicate their design decisions with other developers to ensure the maintainability and integrity of architectural tactics. We conducted an experiment to examine whether tactical code written by architects are less error prone. In previous analysis, we classified each commit as a tactical commit or non-tactical commit. Then we created a dataset of the the following *design Ownership metric.* The relationships between these metrics and the bugs in tactical files will help examine our third research question.

- *Design Ownership*: the proportion of architectsâĂŹ tactical code ownership. It is calculated based on the ratio of number of tactical commits that the architects have made relative to the total number of commits for that file.

In order to evaluate the question of whether architects introduce more or less bugs in tactical files compared to non-architect developers, we carry out two comparisons:

C1: We compared the number of bugs in tactical files that architects had a design ownership of 50% or more.

C2: We compared the number of bugs in tactical files where $Architect - Tactical > Developer - Tactical$ to the number of bugs in tactical files files where $Architect - Tactical < Developer - Tactical$. In other words, we compared the number of bugs in tactical files with tactical commits predominantly from architects to files that had tactical commits predominantly from non-architects.

To carry out the comparisons, we used the one-tailed Mann Whitney U (MWU) test; we thereby determined whether architects indeed cause fewer bugs. We use the MWU test since it is non-parametric and does not require a normal distribution of bugs (as opposed to the T-Test). In order to collect the data for the above the comparisons, we used the *commit-type metrics* described in sub-section 4.1.3.

We compared the two sets of files for C1 (and then C2) using the MWU test (we used an alpha value of $p < 0.05$). We also calculated the CohenâĂŹs d effect size for the comparison. The results for the MWU test for C1 and C2 for our five case study systems appear in Table 7 and 8 respectively. We also report the six number summary (min, 1st quartile, median, mean, 3rd quartile, and max) for each of the sets in tables 7, and 8 respectively.

From Table 7 we can see that four of the five case study systems (Hive, OfBiz, OpenJPA, and Pig) yielded statistically significant results for the MWU test: the tactical files predominantly owned by architects are less likely to have bugs. We can draw the same conclusions from Table 8, in which we only consider tactical *commit-type metrics* and not the design ownership metric. From Table 7, and 8 we can also see that in Hadoop does not have statistically significant results. Although the results in 8 shows that, similar to the other projects, tactical files predominantly owned by architects have less buts, although the results are not statistically significant.

The statistical results indicate that:

> Tactical-code with more commits from architects have fewer defects. This provides empirical evidence on why architects should be engaged in the development of tactical fragments in a system.

Overall, in four out of five projects, we found strong statistical results, providing a positive answer to the research question posed in this paper. We believe that this paper's findings should motivate researchers to expand this work and further investigate the controversial and pragmatic opinions on "architects and coding." Moreover, this paper draws attention to this issue and encourages researchers to approach it from different perspectives, or to replicate our study with other qualitative and quantitative experiments.

6 THREAT TO VALIDITY

This section summarizes our findingsâĂŹ threats to validity and the mechanisms used to mitigate these threats.

Construct validity The construct validity of our study is mainly threatened by the accuracy of the recovered architectural tactics. To mitigate this threat, we tuned the thresholds of the underlying classifier in the tactic detection method, which increased the precision of detected tactical commits. Increasing the classification thresholds has been demonstrated as an effective way to accurately detect tactical code snippets [31], furthermore, we manually reviewed all the tactic related items to ensure the reliability of the data.

External validity One threat to the external validity of the results relates to the systems studied in this paper. To mitigate this threat, we used five open source projects with diverse characteristics in: application domain, industry adoption , size, and richness of architectural tactics. Another threat is the fact that we only used Apache systems that use Jira as their issue repository and are implemented in Java. However, the diversity of the selected case studies helps to reduce this threat, as does the wide adoption of Apache software products, Java, and Jira. Generalizing the conclusions from an empirical study in software engineering is difficult, because this is largely dependent on several context variables [2]. For this reason, we cannot assume that the results of our study generalize beyond the specific system we examined. However, our results provide sufficient insight into the research questions investigated in this paper. Furthermore, since the studied systems are representative of large industrial applications, these findings may be relevant to other large-scale systems.

7 RELATED WORK

Several practitioners have previously provided pragmatic recommendations on the roles of software architects and whether they should be engaged in low-level programming tasks or not [6–8, 11]. None of these recommendations have been examined through an in-depth empirical study. This paper presents a first-of-its-kind empirical study that utilizes mining software repository techniques to provide more insight into this debate.

Prior studies have explored the impact of code ownership on the quality of a software system [4, 5, 16, 20]. These studies found that proportion of ownership, number of major contributors, and number of minor contributors to a software component has a statistically significant impact on defect introduced in a component. However, to the best of our knowledge, no study examined the impact of developers design background on software quality.

There have been several research papers on studying the relationship between architectural smells and bugs in a software system [23, 24]. In this line of work, a bad architecture, manifested as architectural smells [18], can increase the complexity, possibly leading to poor software quality [23]. Furthermore, researchers [27] in MSR and architecture communities have studied the impact of other architectural factors such as architectural changes and design decay on software defects. They suggest that developers need to be aware of such changes. However, these studies do not look at these problems from developers perspective. They also do not inspect whether architects can help prevent the design decay during software implementation. In a small precursor code-review

Table 7: Bug comparison for tactical files owned by architects vs. non-architect developers

	Design Owned by Architects						Design Owned by non-Architect Developers						P value	Eff Size
Project	Min	1st.Qu	Median	Mean	3rd.Qu	Max	Min	1st.Qu	Median	Mean	3rd.Qu	Max		
Hadoop	0	3	5	14.46	14	592	0	2	5	9.628	8	1283	p>0.5	-
Hive	0	0	0	0.2031	0	14	0	0	1	4.237	4	327	0.0	0.391 (small)
OfBiz	0	0	0	0.4821	0	23	0	0	0	2.639	2	103	0.0	0.329 (small)
OpenJPA	0	0	0	1.062	1	21	0	0	2	5.772	5	172	0.0	0.374 (small)
Pig	0	0	1	1.394	2	7	0	2	4	9.851	9	92	0.0	0.57 (medium)

Table 8: Comparison of bugs where one groups makes more tactical contribution

	Architect-Tactical > Developer-Tactical						Developer-Tactical > Architect-Tactical						P for S>N	Eff Size
Project	Min	1st.Qu	Median	Mean	3rd.Qu	Max	Min	1st.Qu	Median	Mean	3rd.Qu	Max		
Hadoop	0	3	5	10.44	11	213	0	2	5	11.76	9	1283	p>0.5	-
Hive	0	0	0	0.4509	0	27	0	0	2	5.367	4	327	0.0	0.48 (small)
OfBiz	0	0	0	1.909	1	38	0	0	0	3.206	3	103	0.0003	0.197(negligible)
OpenJPA	0	0	1	4.306	4	172	0	2	3	7.785	6	74	0.0	0.275 (small)
Pig	0	1	2.5	3.877	3.75	31	0	2	4	10.66	10	92	0.00047	0.456 (small)

[32], which motivated this work, we investigated the importance of implementing architectural tactics rigorously and robustly. Two open-source projects, Hadoop, and OfBiz, were analyzed for tactic-related changes. The results of this small study hinted that that tactic-related classes may go through more refactoring and expose more bugs. However, the work was not extensive enough to make a generalizable conclusion. The work conducted in this paper, investigated this issue further. First we conducted a detailed study of 4 additional systems from various of software domains. This was followed by persona creation and investigation of the relationship between developers design background and defects.

8 CONCLUSION

The results of this study show that architects write code and tactics implemented by architects will have less bugs than those implemented by developers. The results provide a concrete reason why software architects should be engaged in coding activities.

8.1 Message to the industrial audience

The following take home messages are for practitioners:

Architects should write code: the ultimate value of a good design is delivered when architects thoughts are transfered to code, implemented and evolved and maintained based on low level coding and technological constraints. Hands-on architects write better tactical code than developers (section 5). Tactical code snippets owned and maintained by architects is less error prone.

Developers need a deep understanding of architecture: if you do not have a hands-on architect, educate the developers on the bigger picture of the system, but also details of tactic implementation, their properties and best ways to implement them.

Hands-on architects should write the tactical code, but also developers need to be educated on how to implement and maintain tactics: The empirical data shows, hands-on architects introduce less bugs in tactical code snippets (section 5), they are better prepared to implement these critical solutions. But it also shows that developers inevitably will modify these code snippets (section 4). While architects can write and maintain the tactical code, it is important to share the knowledge with the developers and keep them informed.

Tactics are inherently more complex than software functionality: architects uniformly indicated that (section 4) implementing tactics is more complex than delivering functionality. Every change by developers in the tactics should require a follow up code review to prevent bugs that result in architectural failures.

Firefighting situations that impact tactics require your deeper attention: hands-on architects consider the pushing deadlines a factor that can compromise code quality. Tactical code are more complex (interviews in section 4), and typically fewer number of developers are familiar with those concepts. Rushed code fixes on tactical code snippets require refactoring and testing to maintain tactics and ensure their code quality (interviews section 5).

8.2 Message for Academic Audience

The following take home messages are for practitioners:

Architects do/should write code: The results of this study first shows that architects write code and secondly provide empirical evidence for a pragmatic recommendation on why architect should write code. We encourage other researchers and practitioners to perform and report their findings of similar analyses so that we can build a body of knowledge regarding design background and quality in various domains and contexts; also, obtain a better understanding of pragmatic approaches to architecture design.

Techniques to help architects and developers effectively communicate and collaborate during coding activities: architects have the experience but also original understanding of how the solution should be. Based on the collected data analytics and interviews, a gap in the current development workflow is the lack of techniques, tools or practices to help developers and architects communicate, stimulate design thinking in developers, engage architects in review of tactical code snippets written by the developers.

Testing architectural tactics Architects discussed that it is difficult to write and run unit tests for tactical code snippets to ensure their correctness and integrity. While for functional fragments of the systems, typically there are concrete tests defining accepted outcome from the system for each use case. Future research is required to automated testing of tactics behavior,

Bug triaging: the results (section 5) indicate that the tactical code snippets are more error prone and those code snippets committed by non-architects are more likely to have bugs. Although tactics impact a relatively smaller fragments of a system, they play a significant role in satisfaction of key quality attributes, business goals and success of the projects. The findings of this paper can be used to build new bug triaging to prioritize code inspection and limit the developers efforts on important bugs.

REFERENCES

[1] Felix Bachmann and Len Bass. 2001. Introduction to the Attribute Driven Design Method. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. IEEE Computer Society, Washington, DC, USA, 745–746. http://dl.acm.org/citation.cfm?id=381473.381623

[2] Victor R. Basili, Forrest Shull, and Filippo Lanubile. 1999. Building Knowledge Through Families of Experiments. *IEEE Trans. Softw. Eng.* 25, 4 (July 1999), 456–473. https://doi.org/10.1109/32.799939

[3] Len Bass, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice*. Adison Wesley.

[4] Christian Bird, Nachiappan Nagappan, Harald Gall, Premkumar Devanbu, and Brendan Murphy. 2010. *An Analysis of the Effect of Code Ownership on Software Quality across Windows, Eclipse, and Firefox*. Technical Report MSR-TR-2010-140. http://research.microsoft.com/apps/pubs/default.aspx?id=140711

[5] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don'T Touch My Code!: Examining the Effects of Ownership on Software Quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 4–14. https://doi.org/10.1145/2025113.2025119

[6] Simon Brown. [n. d.]. Software Architecture for Developers. https://leanpub.com/software-architecture-for-developers. ([n. d.]).

[7] Frank Buschmann. 2009. Introducing the Pragmatic Architect. *IEEE Softw.* 26, 5 (Sept. 2009), 10–11. https://doi.org/10.1109/MS.2009.130

[8] Frank Buschmann and Joerg Bartholdt. 2012. Code Matters! *IEEE Software* 29, 2 (2012), 81–83. https://doi.org/10.1109/MS.2012.27

[9] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. 2009. On Non-Functional Requirements in Software Engineering. In *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*. 363–379. https://doi.org/10.1007/978-3-642-02463-4_19

[10] G. W. Corder and D. I. Foreman. 2009. *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. New Jersey: Wiley.

[11] Ward Cunningham. [n. d.]. Architects Don't Code. Cunningham & Cunningham, Inc., http://c2.com/cgi/wiki?ArchitectsDontCode. ([n. d.]).

[12] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the Impact of Design Flaws on Software Defects. In *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*. 23–31. https://doi.org/10.1109/QSIC.2010.58

[13] Molly Dishman and Martin Fowler. 2015. Agile Architecture. O'Reilly Software Architecture conference keynote. (2015).

[14] Schahram Dustdar and Harald C. Gall. 2003. Architectural concerns in distributed and mobile collaborative systems. *Journal of Systems Architecture* 49, 10-11 (2003), 457–473. https://doi.org/10.1016/S1383-7621(03)00092-4

[15] Pascal Fenkam, Harald C. Gall, Mehdi Jazayeri, and Christopher Krügel. 2002. DPS : An Architectural Style for Development of Secure Software. In *Infrastructure Security, International Conference, InfraSec 2002 Bristol, UK, October 1-3, 2002, Proceedings*. 180–198. https://doi.org/10.1007/3-540-45831-X_13

[16] Matthieu Foucault, Jean-Rémy Falleri, and Xavier Blanc. 2014. Code Ownership in Open-source Software. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14)*. ACM, New York, NY, USA, Article 39, 9 pages. https://doi.org/10.1145/2601248.2601283

[17] Martin Fowler. 2003. Who Needs an Architect? *IEEE Softw.* 20, 5 (Sept. 2003), 11–13. https://doi.org/10.1109/MS.2003.1231144

[18] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Identifying Architectural Bad Smells. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (CSMR '09)*. IEEE Computer Society, Washington, DC, USA, 255–258. https://doi.org/10.1109/CSMR.2009.59

[19] B. Vande Ghinste. [n. d.]. Do Architects Need To Code? http://blogs.msdn.com/b/. ([n. d.]).

[20] Michaela Greiler, Kim Herzig, and Jacek Czerwonka. 2015. Code Ownership and Software Quality: A Replication Study. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 2–12. http://dl.acm.org/citation.cfm?id=2820518.2820522

[21] Daniel Gross and Eric S. K. Yu. 2001. From Non Functional Requirements to Design through Patterns. *Requir. Eng.*, 18–36.

[22] Alan Grosskurth and Michael W. Godfrey. 2005. A Reference Architecture for Web Browsers. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. 661–664. https://doi.org/10.1109/ICSM.2005.13

[23] Lorin Hochstein and Mikael Lindvall. 2005. Combating Architectural Degeneration: A Survey. *Inf. Softw. Technol.* 47, 10 (July 2005), 643–656. https://doi.org/10.1016/j.infsof.2004.11.005

[24] Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, Lu Xiao, Sam Malek, and Yuanfang Cai. 2015. A Study on the Role of Software Architecture in the Evolution and Quality of Software. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 246–257. http://dl.acm.org/citation.cfm?id=2820518.2820548

[25] Philippe Kruchten. 2004. An Ontology of Architectural Design Decisions. (2004), 55-62 pages.

[26] Philippe Kruchten. 2008. What do software architects really do? *Journal of Systems and Software* 81, 12 (2008), 2413 – 2416. https://doi.org/10.1016/j.jss.2008.08.025 Best papers from the 2007 Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, April 10-13, 2007.

[27] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2015. An Empirical Study of Architectural Change in Open-source Software Systems. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 235–245. http://dl.acm.org/citation.cfm?id=2820518.2820547

[28] Anne S. Mavor, Richard W. Pew, and National Research Council (U.S.). 1998. Modeling human and organizational behavior : application to military simulations / Richard W. Pew and Anne S. Mavor, editors. (1998), xi, 418 p. : pages.

[29] Mehdi Mirakhorli. [n. d.]. Why Should Software Architects Write Code? http://blog.ieeesoftware.org/2016/02/why-should-software-architects-write.html. ([n. d.]).

[30] Mehdi Mirakhorli. [n. d.]. Why Should Software Architects Write Code? http://tinyurl.com/ArchitectureSavvy. ([n. d.]).

[31] Mehdi Mirakhorli and Jane Cleland-Huang. [n. d.]. Detecting, Tracing, and Monitoring Architectural Tactics in Code. In *IEEE Transactions on Software Engineering*.

[32] Mehdi Mirakhorli and Jane Cleland-Huang. 2015. Modifications, Tweaks, and Bug Fixes in Architectural Tactics. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. 377–380. https://doi.org/10.1109/MSR.2015.44

[33] Mehdi Mirakhorli, Patrick Mäder, and Jane Cleland-Huang. 2012. Variability Points and Design Pattern Usage in Architectural Tactics. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, Article 52, 52:1–52:11 pages.

[34] Mehdi Mirakhorli, Yonghee Shin, Jane Cleland-Huang, and Murat Cinar. 2012. A Tactic Centric Approach for Automating Traceability of Quality Concerns. In *International Conference on Software Engineering, ICSE (1)*.

[35] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. 2006. How Design Predicts Failures. *Softwaretechnik-Trends* 26, 2 (2006). http://pi.informatik.uni-siegen.de/stt/26_2/01_Fachgruppenberichte/04-WSR2006/20-SchroeterZimmermannZeller.pdf

[36] Mary Shaw and Paul Clements. 2006. The Golden Age of Software Architecture. *IEEE Softw.* 23, 2 (March 2006), 31–39. https://doi.org/10.1109/MS.2006.58

[37] Venkat Subramaniam. 2005. *Practices Of An Agile Developer*. Pragmatic Bookshelf.