

Using Architecture to Change Code: Studying Information Needs

Thomas D. LaToza

Institute for Software Research International, Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
+1 412-268-1964
tlatoya@cs.cmu.edu

Abstract

While architecture is widely viewed as important, the mechanisms by which understanding architecture helps developers better change code are less clear. We are conducting a laboratory study to gather requirements for the design of future tools and techniques for making architecture more explicit while interacting with code.

Categories and Subject Descriptors D.2.11 [Software Engineering]: Software Architectures – *languages, patterns.*

General Terms Design, Experimentation, Human Factors

Keywords Changes tasks, empirical study, components and connectors, design decisions, breakdowns

1. Introduction

The difficulty or success of writing new features or fixing bugs can be conceived as determined by the ease of obtaining the information necessary to complete the task. Making a change to code might require knowing or finding information formalized in contracts or protocols, information recoverable by tools such as if class A has methods which might call class B, or information about why a layered architecture was adapted at best captured only in comments or design documents. While some of this information may be local information of little importance for the system as a whole, other information may be architectural in nature. Yet, to the best of our knowledge, no empirical study of how developers interact with architectural information has ever been conducted. We would like to design better diagram or visualization systems, specification languages, or coding conventions that make architecture more explicit during coding tasks. But little is known about the characteristics of hard to obtain architectural information – strategies developers use to find it, questions developers ask while seeking it, tasks for which it is used, information it consists of, and breakdowns which occur when using existing strategies for uncovering it. While we expect that a central problem lies in architecture's implicit presence during code interactions, we understand few of the requirements for making architecture explicit.

One approach would be to conjecture specific problems, design tools and techniques to solve these problems, and then empirically evaluate their effects. Our approach is to instead study how developers currently interact with architecture to elicit specific problems and understand their nature and context. This has the

advantage of allowing us to select genuine and important problems and ensures that our solutions respect the context in which the problems occur.

We are conducting a lab study investigating how developers interact with architecture. Developers will perform changes to a system that require architectural information. We will then analyze their interactions with architectural information and how these interactions influence their task performance.

2. Architectural Information

To guide our analysis of developer interactions with architectural information, we propose several dimensions to characterize these interactions. We do not intend this as a final model of but as an initial hypothesized model to be updated as our analysis proceeds. We hope that these and other dimensions arising from the analysis of our data will help distinguish and elicit specific requirements for making architecture explicit.

2.1 Information Characteristics

Software architecture spans the properties of important elements in a system, their relationships, and the rationale behind important decisions made about them [2]. Architectural information can be characterized by the form it takes in code – information at particular program points, constraints on allowed programs, rationale about decisions made, or properties of the execution of the program. Developers seek answers to questions to determine this information. Questions involving information at program points ultimately results in assertions about whether some statements exist in code – do there exist any statements where element A calls element B? These questions could be answered by inspecting the code or a tool that automatically inspects the code. In contrast, the other forms require some additional information to answer. Constraints prescribe what changes to code conform to architectural design intent. For instance, a layered architectural style restricts relationships a lower layer can have with a higher layer. Simply inspecting code for the presence or absence of these dependencies does not ensure that this constraint was intended – these dependencies might simply not occur in the code. Thus, constraints may be more difficult to reverse engineer. Design rationale concerns why a particular design decision was made. A layered architecture might be adopted to make it easier to allow third party companies to be able to write the top layer. Execution property questions require either executing the program or making a guess about how the program will execute. For instance, understanding whether a GUI element will execute quickly enough to prevent flickers following update events requires the program to be executed or the developer to understand how it will execute.

2.2 Better design decisions

We hypothesize that developers who do not possess relevant architectural information make suboptimal changes. Developers might make poor decisions about where to make a change – the methods or classes that should be modified to accomplish a change. Developers might miss existing functionality that could be reused to accomplish their task. This might result in code duplication or excessively long implementations. Developers might misunderstand the anticipated changes an architecture is trying to make easier and make changes destroying these benefits. Developers might not understand how a particular configuration of elements allows a quality attribute to be achieved.

2.3 Strategies and breakdowns

Strategies describe the usage of tools and techniques for obtaining architectural information. Developers might choose to use a debugger, read code, traverse relationships in code using an IDE, formulate hypotheses using their domain knowledge, or read comments or documentation. We hypothesize that the choice of combinations of tools and techniques strongly determines the architectural information uncovered. Strategies may also be hierarchic. Developers may begin with specific high level goals of information to obtain and employ strategies that result in lower level information being sought. Uncovering some information may suggest other information to seek. Thus, developers might be particularly sensitive to uncovering some information simply because it suggests other useful information to seek. Strategies result in questions that a developer would like answered.

Breakdowns describe how strategies go wrong when developers' understanding of the code or architectural information becomes inaccurate by differing from the actual information. Developers might misread or misinterpret code. Developers might never realize that they should be asking a question to obtain some information when nothing suggests it is relevant. Developers may be aware that some information is needed, but simply be unable to find it. Developers may uncover some information but forget it due to the limitations of human working memory. Or developers may realize they need some information but forget they need it later. Developers might discover some information but not realize it is important enough to read carefully. Thus, in addition to studying what questions a developer seeks to answer, we can also study what questions a developer should have been asking to have more quickly focused on the necessary information to obtain.

3. Study Design

To study interactions with architectural information, we will observe developers performing code change tasks in a controlled lab setting. A controlled setting allows us to carefully design the tasks to ensure that architectural information is important to the task and to directly compare several developers' work solving the same problem. Developers will make changes to jEdit – a small (more than 50,000 lines of code) open source code editor written in Java. It has been previously used for a study of code navigation [3]. Participants will be recruited from computer science graduate students at Carnegie Mellon and will be screened for experience with Java.

Participants will be randomly assigned to one of two conditions. Participants in the control condition will receive with each task description a paragraph description of the functionality of each of

several classes important to the task and a class diagram depicting how they are related. Participants in the C&C condition will also receive a component and connector diagram depicting the runtime relationships between each of the important classes. We expect that C&C participants will more quickly understand what architectural elements exist and how they are related. We also expect that they will employ different strategies and will have different problems once they have discovered this information. Two additional conditions providing design rationale documentation for relevant decisions and annotations in the code specifying important architectural information may also be run if time permits.

Participants will first complete a survey of background knowledge. They will then be given the description of the first of two tasks. After reading the task, they will be asked to spend 1.5 hours using Eclipse 3.2 to complete the task. After the first task, developers will be given the description of the second task and a clean copy of the jEdit codebase. Finally, developers will engage in an approximately one hour semi-structured interview exploring decisions they made, the architectural information they uncovered, the architecture they perceived the system to have, strategies they employed, difficulties they experienced, and information they would have liked to have available. Raters will rate participants' changes for conformance with the architecture. Participants' behavior will be observed using a recording of their screen. Participants will also be instructed to think aloud to verbalize their thoughts and goals as they work. This audio will be recorded. To ensure both tasks require using architectural information and are neither too trivial nor too difficult for the time provided, we are iterating the task requirements and information provided from problems experienced by pilot participants.

Data analysis will focus on bucketing interactions with architectural information using our hypothesized dimensions. From this, we can understand the frequency of particular interactions with architecture. We can also find differences between participants and understand how strategies shaped these differences. Based on architectural information it is clear participants did not uncover or inaccurate information they believe, we can then ask what went wrong. Working backward, we will reconstruct the breakdown chains experienced [1].

4. Conclusions

We hope that our study will produce a corpus of requirements for future tools and techniques for making architecture more explicit during coding tasks.

References

- [1] Ko, A. J. and Myers, B. A. A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages and Computing*, 16, 1-2 (Feb – April 2005), 41-84.
- [2] Perry, D. E. and Wolf, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes* 17, 4 (Oct. 1992), 40-52.
- [3] Robillard, M. P., Coelho, W., and Murphy, G. C. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Transactions on Software Engineering*, 30, 12 (Dec. 2004), 889-903.