Dimensions of Software Architecture for Program Understanding[†]

Hausi A. Müller[∓] Kenny Wong[∓] Scott R. Tilley[±]
hausi@csr.uvic.ca kenw@csr.uvic.ca stilley@sei.cmu.edu

[∓]Department of Computer Science
University of Victoria
Victoria, BC V8W 3P6

[±]Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Software architecture is usually considered in terms of software construction rather than software understanding. Architectures for construction typically embody design patterns based on software engineering principles. In contrast, architectures for understanding represent change patterns and business rules based on conceptual models. This paper presents three dimensions of software architecture for program understanding. In each dimension, the user of the architecture plays a central role.

Keywords: program understanding, software architecture, software evolution.

1 Introduction

Software architecture has traditionally been oriented toward new software construction, not toward software understanding for maintenance and evolution. This orientation has lead to a view of software architecture as consisting of design patterns, reusable components, and structural dependencies. In this view, architecture is a framework for tracking requirements, a technical basis for design, a managerial basis for cost estimation and process management, an effective basis for reuse, and a basis for dependency and consistency analysis [1].

While helpful, this view ignores the importance of a multitude of architectures based on change patterns, business rules, or conceptual models used during program understanding. The users and uses are different. Architectures for software design and construction cater to designers, integrators, testers, and inspectors; architectures for software understanding cater to software engineers who need to understand software systems during long-term maintenance. For software understanding especially, we argue for an alternative, yet compatible, interpretation of architecture that places the individual user in a central role.

The need to consider individual users is partly evidenced by the following fact. Software exists as a hybrid of two imperatives: intangibility and tangibility [2]. The intangibility imperative views software as an abstract idea similar to mathematics. The tangibility imperative concerns more immediate, accessible, and physical concepts. The main observation is that intangibles are best addressed and understood at the level of individuals, not committees. Thus, a comprehensive treatment of understanding software requires support not only for groups, but for individual users.

[†]This work was supported in part by the British Columbia Advanced Systems Institute, the IBM Software Solutions Toronto Laboratory Centre for Advanced Studies, the IRIS Federal Centres of Excellence, the Natural Sciences and Engineering Research Council of Canada, and the University of Victoria.

2 A user-oriented interpretation

For software understanding, the notion that there is an architecture existing in an objective state, independent of the engineers probing it, is flawed. We believe there is no single, definitive architecture of a software system. Instead, there are many architectures corresponding to many diverse purposes and users. The single-architecture view is limited and ignores levels of user involvement, thereby lacking context. In effect, architectures are user-created and depend in part on how we observe it and what we choose to see.

In addition, there is no right architecture for a software system that is valid for all users. Each user has a different interpretation of what the software is about. These are not mere subjective and virtual "perceptions" of a single, true architecture; an architecture to a user is very real and meaningful. Individual maintainers, managers, and customers have their own architectures.

An architecture for understanding may span various levels ranging from the concrete to the abstract: implementation, structural, functional, and behavioral [3]. For one user, an architecture may contain a combination of detailed code, structural design patterns, object interaction behavior, and functional purpose. For another user, an architecture may contain a combination of entirely different information: code complexity measures, maintenance effort, risk analyses, and personnel assignments. There is no conflict in having multiple architectures for understanding. Moreover, these architectures are continually changing and in flux, not only because the software is evolving, but also because the users and their needs are changing. Program understanding necessitates a subtle appreciation of these rich and diverse architectures.

The idea of multiple architectures has a major impact on the design of tools and methodologies for program understanding. Many program understanding tools deal with the software as if it has a single architecture, such as module interconnections, that is right for everyone. This ignores the fact that users each have their own conceptual models, methods of reasoning, and needs for understanding. Diverse needs such as business rules and change patterns are lost in a module interconnection architecture. Annotating a module diagram with this information is inadequate, cluttering, and relegates the information as second class to some users. It is irrelevant and to a user whether this information is collected into a grand, unified architecture. What is needed is tool support for expressing and analyzing a tapestry of multiple, user-oriented, architectures for understanding.

Unlike architectures for construction, which typically deal with exact information for purposes such as code generation, debugging, and integration, architectures for understanding must allow and deal with inexact or partial information. The art of effective understanding is to know what to leave out and what to ignore [4]. Incompleteness is the norm, not only for practical reasons of the scalability and immaturity of the analysis methods, but because there cannot be a definitive architecture in an evolving system. Even inconsistency and conflicts from trying to consolidate two architectures are acceptable, since humans can easily deal with incomplete and inconsistent information. This differs from the specifications and design architectures used by computers, which need to be exact, complete, and consistent.

3 The role of the user

Since architectures for understanding are fundamentally user-created entities, it is logical that program understanding tools and methodologies in-