Architectural Congruence: Toward Exploring the Software Development Process Through an Architectural Perspective

Anita Sarma
Institute for Software Research
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
asarma@cmu.edu

John Georgas
Department of Computer Science
Northern Arizona University
PO Box 15600
Flagstaff, AZ 86011 USA
John.Georgas@nau.edu

Abstract

While the architecture of a software system is a critical artifact, it is often overlooked during development. Instead, developers focus on the numerous relationships and dependencies between source code files, bug and feature reports, and communications between other developers. We propose to begin bringing architecture back to the forefront of development, by extending the TESSERACT environment to support the exploration of the relationships and dependencies between the architecture, the technical artifacts of software development, and the social interactions of developers. With this tool support in place, software engineers may now begin exploring and reasoning over the development process in architectural terms, promoting congruence between their understanding of a system's architecture and how this architecture effects and is affected by the social and technical activities of software development.

1. Introduction

The architecture of a software system is one of the most important artifacts of the development lifecycle: it is the architecture that captures the essential design decisions that frame a system's functionality. The architecture forms the structure around which code is fleshed out, and fundamentally affects the system's non-functional properties.

However, the importance of architecture also extends beyond just technical concerns, playing a distinct role in coordination. Task assignments are often created along the separations between architectural boundaries by, for example, assigning different modules to different teams. However, the architecture also frames, and is framed by, the organizational structure of these teams. Simply stated, the architecture of a software system is inextricably linked to the organization of those that develop it. This is commonly referred to as Conway's Law [4], which states that the organizational structure of developers is reflected in the architecture of the product they eventually produce.

Despite this importance of architecture from both a technical and coordination perspective, it is unfortunately not at the forefront of the software development process. Software developers tend to focus more on the numerous artifacts with which they interact on a day-to-day basis: source code files, unit tests, bug reports, and emails to and from other developers. This is not surprising, given the large number of these artifacts and the need to achieve a basic understanding of the numerous relationships and correlations between them. Developers, for example, need to be able to edit, compile, and debug their changes, while also ensuring that their work integrates with the rest of the changes taking place elsewhere in the project. These activities are not trivial, require considerable effort and consume most of the time and attention of software developers. Finally, the lack of a direct mapping between architecture and the rest of software artifacts makes it difficult for developers to treat architecture as a first-class artifact. Consequently, the architecture fades away into the background, becoming less important and often abandoned, quickly becoming obsolete. Architecture is simply still not a

part of the conventional, everyday development toolbox, despite efforts to make it so.

We aim to augment the software development process – rather than fundamentally alter it – in such a way as to bring architecture closer to the forefront of the development process. We propose an approach that leverages architecture to tie together information about source code files, code changes, defect and new feature reports, and developer communication records. This cross-linked architectural model will provide a cohesive and concise multi-level view of the overall development process, while still allowing developers to access lower-level information. Our approach rests on the integration of architectural information into the TESSERACT [19] tool, and building support for focusing on the exploration of different technical and social data about a system's development process through an architectural lens. By providing capabilities such as these, we aim to focus on supporting *architectural congruence*: bringing our understanding of the prescriptive architecture of a software system closer to the artifacts that implement it, and the social and organizational processes that created it.

We posit that fulfilling this vision will bring benefits organized along two dimensions: *development process awareness*, and *software quality*. By understanding the relationship between architecture and the communication patterns of its developers, we can better understand how to organize teams and assign work [15]]. By correlating code changes that span multiple components, for example, we can identify component dependencies and realign team assignments to match these dependencies: Therefore, we can ensure that the development of these components is assigned to a single team, so that the shared mental models and work practices of this team promote easier integration of these components.

We can also better understand the qualities of the architectural design itself. On a simple level, we can identify code changes that affect parts of the architecture that they should not, by tracking the mappings between architectural elements and code units that change. Using these mappings, we can also reason about modularity and brittleness. Displaying and understanding source code changes in architectural terms will, for example, clearly illustrate the elements of the architecture which are under constant evolution and modification. This frequency of change may indicate brittleness in those architectural components, and motivate more careful testing in future maintenance. Explicitly identifying elements of the architecture that are affected by a change may also indicate the degree of modularity exhibited by the architecture: continual changes to one component for an extended period of time, for example, may motivate a re-design that better modularizes and separates the concerns captured by that particular component. Furthermore, this work enables exploration of the relationships between other architectural concepts, such as architectural style, in the context of an organization's structure.

2. Background and Related Work

Our works relies on the integration of insights from software architecture and research in providing support for exploration and awareness of the software development process.

2.1. Software Architecture

Software architecture revolves around capturing the decisions that frame the design of a software system [22]. Most commonly, these decisions relate to the logical components that encapsulate functionality and the connections that coherently compose them into systems. Architectural styles [21] provide guidance for the design of software architectures that are particularly well-suited for specialized tasks or specific application domains. Architectures, however, are not just design-time artifacts: By explicitly modeling architectures using architecture description languages (ADLs) [13], the utility of architecture extends to all phases of software development: architectures, for example, are used for guiding runtime evolution [17] and designing self-adaptive systems [11].

Architectural models can also be used as the central artifact in the development process. The ARCHEVOL system [16], for example, focuses on supporting an architecture-centric development process by explicitly mapping the architecture to changing source code artifacts. The LIGHTHOUSE approach [7] focuses on providing a view of a system's emerging design, as discovered through code analysis during development, so that developers may trace the effects of their work on others. Other tools [2] support traceability between multiple software development processes, such as quality assurance and testing, through a central architectural model.

The work we discuss differs from this previous work by focusing on post-hoc analysis of development

information, which is less disruptive to established processes than tools that are intended to be used during development. Rather than directly aimed at providing capabilities to be used during development, we aim at supporting the exploration and analysis of what actually took place, with an eye toward better understanding the development process and gleaning insights to be used in future efforts.

2.1. Development Exploration and Awareness

A typical software project is composed of thousands of software artifacts, including code, design documents, bug reports, communication records, and task assignments. Further, these project elements are intrinsically inter related. Software Development environments are increasingly trying to capture and convey these relationships to the user. Consider the example where editing a file of source code may affect many other relevant artifacts. Considerable research effort has been focused on using a variety of techniques such as static code analyses [1], task definition [12], text analysis [5], and records of prior developer activity to identify these related artifacts [10] and make them easily accessible when they are likely to be useful. Similarly, there exists basic support for tracking when changes to source code may require certain test cases to be regenerated [18].

There is also an increasing interest in understanding and using relationships among individuals in a team to improve software development. Research has focused on increasing awareness among developers about each other's relevant activities [20], and on using the social relations among developers to identify implicit teams [3] or to predict software defects [14]. Such efforts often draw on social network analysis (SNA).

While these development environments serve as first steps towards investigating and understanding the myriad relationships among different project elements, they provide limited support. Most of these environments typically focus on the code archive and at the maximum only link two kinds of data (e.g., test suites or communication records). Doing so has three drawbacks. First, the information is at a low granularity making it difficult to gain an overall picture of changes taking place at the systems level. Second, information at such granularity can quickly lead to information overload for the user. Finally, these environments only present a partial picture of the project relationships, since other elements of the project are ignored. While this partial picture is better than nothing, it still leaves significant amounts of information unavailable for the developer.

Developers are intuitively aware of this need, and actively engage in informal communications while developing or restructuring code. There is a need for environments that allow investigation of socio-technical information in a project in useful, actionable ways, such that they don't overwhelm the user. The need for such tools is reflected by findings from field studies, which have shown that developers find it difficult to decipher how their work binds them with that of others. Consequently, they spend a significant portion of their time in managing their changes or in finding the right person with whom to communicate [9] .

3. Research Plan

In order to reify our approach, we will need to provide support for: modeling the relationships between explicit architectural models and the social and technical artifacts of development, and visualize and explore these relationships.

3.1. Data Model

The current incarnation of TESSERACT relies on collecting data from tools commonly used in large software projects: a configuration management system, a bug-tracking database, and project mailing lists. By extracting and cross-linking information from these tools, the tool builds a number of visualizations showing: the relationships between source code, developers, and bug reports; the communication patterns between developers; and how well the technical aspects of development match the project's social interactions.

Integrating architectural information with TESSERACT's current design demands that artifacts such as source code, bug reports, and email communication are linked to an explicit architectural model, which shows the relationships between each of these and the elements of the architecture. Beyond the modeling problem involved in this integration, an additional challenge is imposed by the fact that – due to the post-hoc exploratory nature of proposed tool – the architectural model must be generated after parts or even the entire development process is complete.

In order to minimize additional development, we plan on addressing the integration of architectural information into TESSERACT by focusing on the links

between architectural elements, such as components and connectors, and the source code files that implement these elements. TESSERACT already includes support for modeling and correlating relationships between source code files, bug reports, and developer communication emails. Based on this support, the mapping between source code files and architecture becomes the focal point of this initial integration. This provides a direct relationship between architecture and source code, and an indirect relationship between architecture and bug reports or social communications. For this initial integration, we will rely on modeling architectures through an extension of the core xADL 2.0 ADL [8]. We will extend the current definition of the language to support one-to-many associations between architectural elements and units of source code. As our work progresses, we plan on exploring whether this linkage is sufficient, or whether direct connections between architecture and all other artifacts are needed.

Given that our initial plans for this effort are targeted to post-hoc analysis of development process information, another challenge involves the recovery of architectural models. The initial body of information from the GNOME project, for example, used for the initial deployment of TESSERACT does not contain architectural information: The project is simply not guided by an explicit architectural model throughout development. As a result, in order to create these links between architectural elements and source code files, we must first recover the architecture. Based on the architectural recovery techniques of the FOCUS approach [13], we will model the descriptive architecture of the GNOME project, which – integrated with the information on source code changes, bug reports, and communication information – will serve as the basis for the development and initial usage of our architectural extension to TESSERACT.

3.2. Initial Design

The primary goal of TESSERACT is to provide an interactive exploratory environment with which developers can investigate relationships across architecture, code, bugs, and communication records in mailing lists and bug databases. It also allows developers to visually investigate how these relationships evolve over time and find interesting development patterns in their project. Specifically, TESSERACT consists of five cross-linked panes each showing a different facet of the software project:

- The *Project activity* pane (Figure 1(a)) displays the overall activities in a project (code commits at the top and communication at the bottom) as a time series display. This view helps users visually identify when there are spurts in activity; these can be either code contributions or communication. Users can select a particular time period for their investigation, which is reflected in all other panes.
- The *Files/Bugs network* pane (Figure 1(b)) displays a bi-partite network of artifact associations and bugs associated with particular artifacts. Here, file-to-file associations are created by linking files that are frequently changed together (logical coupling). The edges of the network represent the number of times pairs of files have been committed together and can be 'thresholded' by the user. Textual listing of the file names allows quick identification of specific files via a search function. Bugs that are associated with a particular file are also represented in the same graph, enabling users to quickly identify which artifacts have bugs associated with them and whether sets of files that are changed together tend to have common bug associations.
- The *Architecture* Pane (Figure 1(c)) presents the architecture of the project, where components that are being edited in the selected time frame are highlighted in gray. Further, components that have bugs associated with it are bordered in red, drawing the users' attention. Similar to the previous pane, a textual listing of components is provided for easier search and navigation.
- The *Developers network* pane (Figure 1(d)) displays developers and links among them. Two developers are linked if they either edited the same artifact or interdependent artifacts. The edges in this network are colored (in green or red) to show when developers have communicated via either email or the bug repository (e.g., comments or activities in Bugzilla). The thickness of the edges is based on the number of times developers communicated. Similar to the file network, a textual listing of the developer names is provided.
- The *Issues* pane (Figure 1(e)) displays defect or feature related information as a stacked area chart, as well as, in a detailed listing.

TESSERACT enables exploration of project data and its underlying relationships in multiple ways. First, clicking on an entity (graph element or line in the textual lists) will highlight that entity and will also show all related entities in the other panes. For example, selecting a node (bug) in the file/bug network highlights all the files and architectural components

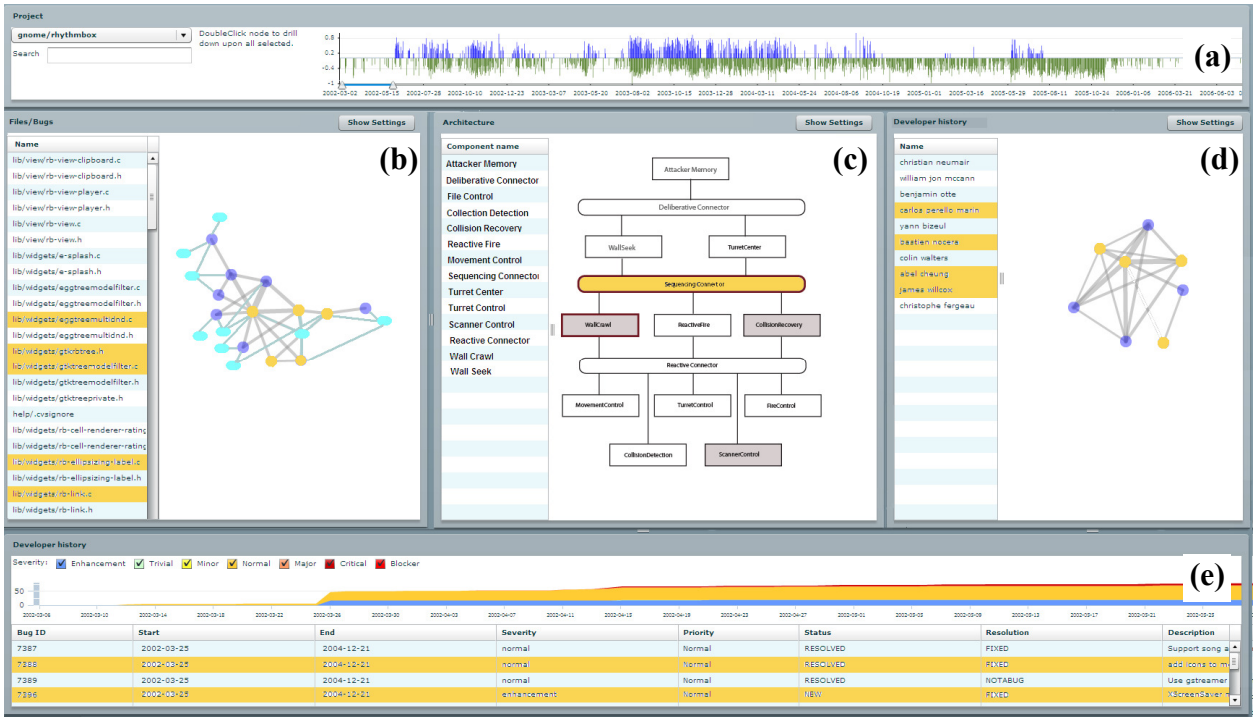

Figure 1. TESSERACT architecture extension UI mockup showing five display panes: (a) project activity pane with code commits (top) and communication (bottom), (b) file and bug bi-partite network, (c) architecture, (d) developer communication network, and (e) textual listing of bugs.

that are included in that bug fix and also highlights the developer who had fixed the bug. Similarly, selecting a developer in the developer network highlights all the files, bugs, and architectural components that the developer has worked on. Second, hovering over a node in any of the nodes, displays additional information about the node and highlights its neighbors. Third, a user can pan (background drag), zoom (wheel), and move individual nodes in the graph. Fourth, search functionality allows users to quickly find an entity when they know its full or partial details. Finally, TESSERACT allows users to change the perspective of their investigation by drilling down on specific artifact(s) or developer(s). For instance, a user might drill-down to view a particular component in the architecture to find which files and developers are associated with that component.

3.3. Sample Usage Scenario

Here we present an example usage scenario where TESSERACT can enable the investigation of the technical and social relationships in a project through an architectural perspective. Assume the hypothetical case where a manager is interested in investigating the development history of a particular component (Sequencing Connector) from the last release till the current date. To do so, she first selects the appropriate time period from the *Activity* panel, following which she selects the particular component in the *Architecture* pane. On performing this step the following information is highlighted in TESSERACT (see Figure 1): (1) files that are associated with that component and have changed in the desired time frame, (2) developers who have edited those files, and (3) any bugs that are associated with those files.

Drawing on her knowledge of the project, the manager uncovers the following facts. First, the chosen component involves artifacts that occupy a central position in the artifact network implying intricate dependencies with two other components, she realizes that the code structure is not appropriately reflecting the architecture and decides she would need to talk to the team about it. Second, some of the files that belong to the component had had defects, most of which had low priority (normal) and were all fixed. Finally, she confirms that the component was largely developed by the team that was assigned the task. However, she is surprised to find that an outlying developer who is a member of a team responsible for another component (Scanner Control) has been involved. Further investigation of the code contributions of the outlier reveals that that developer made unplanned modification to the Sequencing Connector and that an undocumented interface method was added so that the

component Scanner Control could call the Sequencing Connector component.

The architecture clearly indicates that all calls from Scanner Control should connect through another component (Reactive Connector) and the new interface method violates the architectural assumptions. Further investigation of the developer communication network shows that the said developer has not communicated with the rest of the team indicating a possibility that the team may not be aware of this architecture violation. She decides to immediately contact the main development team and the outlier developer to resolve this issue immediately.

4. Discussion and Future Work

TESSERACT is the first exploratory environment that not only allows the investigation of the different socio-technical relationships that exist among project elements, but also helps a user understand their evolution over time. More specifically, TESSERACT creates networks among code, bugs, and developers. It creates file-to-file associations based on which files are frequently changed together, and then connects which artifacts are associated with which bugs. TESSERACT also creates a developer communication network by creating links among developers who have communicated either via mailing list or discussion via the bug repository. A critical contribution of TESSERACT is the cross-linking of different project elements in such a way that users can investigate a pattern or project occurrence across multiple project elements and over time.

Here, we have discussed extending the capabilities of TESSERACT to provide an architectural perspective to project investigations. The architecture of a project provides the right level of abstraction, using which individuals can correlate lower granularity changes (at the level of files/bugs). Doing so, not only presents changes taking place at the design level, but also helps in making the information presentation and investigation more scalable.

Although, the architecture of a project is a critical element that captures the design decisions of a project and to a large extent determines the team interactions, it frequently becomes an obsolete artifact. Developers tend to focus more on the numerous artifacts with which they interact on a day-to-day basis: source code files, unit tests, bug reports, and emails to and from other developers. This is not surprising, given the fact that developers have to contend with an extremely large set of such artifacts and that there exists no tool support which treats architecture as a central element. In our work, we aim to overcome this problem and enable the investigation of project level changes through an architectural perspective.

4.1. Planned Validation

Evaluation of any software tool involves two criteria: usefulness and usability. There exists a large body of literature that argues for the need for architecture to be considered as a central element [22] and field studies that show that doing so is rarely possible in the real world [6]. These studies help ascertain the usefulness of a tool such as TESSERACT, that allows the investigation of project changes through an architectural perspective. We plan to augment this body of knowledge by demonstrating our tool to software architects, managers, and developers and collecting their feedback on how they would use the tool in their daily activities. Our central goal would be to elicit how these stakeholders currently manage their tasks and how TESSERACT can help them.

We plan to validate the usability of the tool by performing user studies both in controlled lab settings and deployment in the field. Towards this direction, we will first instrument TESSERACT with real projects to ensure that it scales well to real projects and so that users can investigate their own projects during our study. Doing so will also help us validate the efficacy of our architectural recovery techniques from available source code.

We plan to conduct a series of user studies through which we will determine how users investigate a given set of changes when using our tool as compared to using other available tools. Based on our understanding of these studies and after implementing feedback from our studies, we will identify sites that treat architecture as a central component (e.g., the aerospace industry). We will then deploy TESSERACT to these sites and investigate the usefulness and usability of the tool in real-life settings.

4.2. Enabled Future Research

The approach we discussed in the previous section is focused on the exploration of the relationships between an instance of an architecture and the technical and social interactions that result in its implementation. Our work, however, enables the investigation of a broader

set of questions and research directions in the intersection of architectural concepts and the practical aspects of software development.

One very exciting question we plan on exploring next is the relationships of architectural styles with the social structures of software developers. The selection of a specific architectural style for a software system is primarily based on the technical characteristics of the system in question: Does the style foster needed qualities, for example, or has the style been shown effective in the problem domain in the past?

What the tool support we propose to build does, however, is to enable asking questions about the effects or demands (if any) of specific architectural styles on the social structures of the developers that will use it. By exploring and understanding these relationships, the richness of considerations during the selection of an architectural style is greatly enhanced: Does the style fit our established social practices, or how must we arrange our development teams to better align their organization with the style's social demands? By providing the fundamental tool support for even asking such questions, we look forward to exploring if such correlations exist and how to leverage them in better matching communication and teaming practices with architectural styles.

5. Acknowledgements

We would like to thank Larry Maccherone and Patrick Wagstrom for their help in the development of TESSERACT. This effort is partially funded by the NSF grant number IIS-0414698 and IIS 0534656, and the Software Industry Center and its sponsors, particularly the Alfred P. Sloan Foundation. Effort also supported by a 2007 Jazz Faculty Grant.

6. References

[1] Arnold, R.S. 1996. *The Year 2000 problem: Impact, Strategies and Tools.* Software Evolution Technology, Inc. Tech. Report.Tech Report.

[2] Asuncion, H.N., et al. 2007. *An End-to-End Industrial Software Traceability Tool.* European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. ACM. p. 115-124.

[3] Bird, C., et al. 2008. *Chapels in the Bazaar? Latent Social Structure in OSS.* 16th ACM SigSoft International Symposium on the Foundations of Software Engineering. p. 24-35.

[4] Conway, M.E.,1968. *How do committees invent?* Datamation, 14(4): p. 28-31.

[5] Cubranic, D., et al.,2005. *Hipikat: A Project Memory for Software Development* IEEE Transactions on Software Engineering, 31(6): p. 446-465.

[6] Curtis, B.,1992. *Insights from Empirical Studies of the Software Design Process.* Future Generation Computer Systems, 7(2-3): p. 139-149

[7] da Silva, I., et al. 2006. *Lighthouse: Coordination through Emerging Design.* OOPSLA workshop on Eclipse Technology eXchange. p. 11-15.

[8] Dashofy, E., et al.,2005. *A Comprehensive Approach for the Development of Modular Software Architecture Description Languages.* In ACM Transactions on Software Engineering and Methodology, to appear. , 14(2): p. 199-245.

[9] de Souza, C.R.B. and D. Redmiles. 2008. *An Empirical Study of Software Developers' Management of Dependencies and Changes.* Thirteeth International Conference on Software Engineering. p. 241-250.

[10] DeLine, R., et al. 2005. *Towards Understanding Programs through Wear-Based Filtering.* ACM Symposium on Software Visualization. p. 183-192.

[11] Georgas, J.C. 2008. *Supporting Architecture- and Policy-Based Self-Adaptive Software Systems.* Thesis. in Informatics. University of California, Irvine.

[12] Kersten, M. and G.C. Murphy. 2006. *Using Task Context to Improve Programmer Productivity.* Fourteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 1-11.

[13] Medvidovic, N. and V. Jakobac,2006. *Using Software Evolution to Focus Architectural Recovery.* Automated Software Engineering, 13(2): p. 225-256.

[14] Meneely, A., et al. 2008. *Predicting Failures with Developer Networks and Social Network Analysis.* ACM SIGSOFT International Symposium on the Foundations of Software Engineering. p. (to appear).

[15] Mockus, A. and G. Weiss,2001. *Globalization by Chunking: A Quantitative Approach.* IEEE Software, 18(2): p. 30-37.

[16] Nistor, E., et al. 2005. *ArchEvol: Versioning Architectural-Implementation Relationships.* Twelfth International Workshop on Software Configuration Management. ACM. p. 99-111.

[17] Peyman, O., et al. 1998. *Architecture-based Runtime Software Evolution.* Twentieth International Conference on Software Engineering. IEEE Computer Society. p. 177-186.

[18] Ren, X., et al. 2004. *Chianti: A Tool for Change Impact Analysis of Java Programs.* Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 432-448.

[19] Sarma, A., et al. 2009. *Tesseract: Interactive Visual Exploration of Socio-Technical Relationships in Software Development,.* Thirty-first International Conference on Software Engineering. p. (to appear).

[20] Sarma, A., et al. 2008. *Empirical evidence of the benefits of workspace awareness in software configuration management.* Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM. p. 113-123.

[21] Shaw, M. and P. Clements. 1997. *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*. Computer Software and Applications Conference. p. 6-13.

[22] Taylor, R., et al., 2009. *Software Architecture: Foundations, Theory, and Practice* John Wiley & Sons. pages. 750.