Towards Engineered Architecture Evolution

Sagar Chaki CMU/SEI
Andres Diaz-Pace CMU/SEI
David Garlan CMU/CS
Arie Gurfinkel CMU/SEI
Ipek Ozkaya CMU/SEI

Abstract

Architecture evolution, a key aspect of software evolution, is typically done in an ad hoc manner, guided only by the competence of the architect performing it. This process lacks the rigor of an engineering discipline. In this paper, we argue that architecture evolution must be engineered — based on rational decisions that are supported by formal models *and* objective analyses. *We believe that evolutions of a restricted form —* close-ended evolution*, where the starting and ending design points are known a priori — are amenable to being engineered. We discuss some of the key challenges in engineering close-ended evolution. We present a conceptual framework in which an architecture evolutionary trajectory is modeled as a sequence of steps, each captured by an operator. The goal of our framework is to support exploration and objective evaluation of different evolutionary trajectories. We conclude with open research questions in developing this framework.*

1. Introduction

The generally accepted definition for software evolution is the process by which systems change, adapting to the marketplace and inheriting characteristics from preexisting programs [19]. A key aspect of software evolution is architecture evolution. As a system evolves, its architecture is impacted. Conversely, planning for architecture evolution is a powerful tool to guide and plan for software evolution.

Most work on software evolution focuses on practices towards managing code. However, the software architecture is the linchpin for ensuring that a software intensive system achieves its business and quality goals over the lifetime of the system [17, 7]. Thus, the importance of software architecture practices for evolving systems is becoming increasingly evident in both commercial and government sectors. For example, a recent report from Gartner Research [9] highlights that many organizations are moving from traditional client-server architectures to other architectures, prompted by emerging demands for agile business processes. A similar opinion is echoed by a report [2] from a U.S. Army workshop, which recommends that the architecture be designed with evolution in mind.

In this paper, we outline our position on planning architecture evolution (or evolution, for short) as an engineered process. Currently, planning for evolution is an *ad hoc* activity. This does not mean that the evolution is chaotic or unplanned, but rather that it is non-generalizable (i.e., specific to a particular instance). Sometimes, an ad hoc evolution begins with an architect producing an evolution plan. More often, the plan is constructed as evolution is taking place. This plan is captured in a variety of forms: a list identifying key milestones, timelines to achieve them, work to be done at each milestone, intermediate release cycles, etc. Such ad hoc planning is, at best, the state-of-the-practice in current organizations undergoing evolution. Since such planning is not supported by sound engineering methods and tools, the quality of the result depends, almost completely, on the competence of the architect.

Our aim is to improve this state of affairs through "engineered evolution." Specifically, we envision that evolution is planned in a generalizable and repeatable manner. Moreover, the planning process is backed up by rational decisions based on models and analyses derived from architecture-centric practices. Our vision is rooted in the argument made by Jazayeri — "Software is more than source code; models and meta-models are important to software evolution" [18].

In this paper, we focus on a class of evolutions – *close-ended* (or closed) evolutions – which we believe are amenable to being engineered. An evolution is closed if the properties of the current and target architectures are known a priori. The challenges observed in planning for such evolution center around lack of tools and practices that can assist an architect in better scheduling critical activities, analysis of alternate evolution trajectories, and systematically rationalizing the decisions made. We sketch a conceptual framework — based on a set of generic, reusable operators — aimed at enabling the architect to model and construct a feasible plan for evolving an architecture from its current to a desired state, with appropriate intermediate release points.

The rest of this paper is organized as follows. We survey the related work in Section 2. We define closed evolution in Section 3 and identify key challenges in planning for closed

evolution in Section 4. In Section 5, we outline our approach for modeling an evolution trajectory with operators. Section 6 concludes the paper.

2. Related Work

The theoretical foundations for software evolution can be traced to Lehman's laws of software evolution [19] and Parnas' ideas about "software aging" [26]. There is a large body of work on maintaining and aligning software to changing requirements, business goals and practices. These efforts, although mainly focused on the code structures of the system, have led to important evolution concepts such as: code modularization criteria [25], maintainability indicators [19], and code refactoring [22]. Lehman's laws were based on observations of the evolution of several industrial software systems in the late 1970's, and influenced much work in software evolution [21]. Nevertheless, architecture evolution, by itself, has received scant research attention. We survey related work in four areas that influence the emergence of engineered evolution: maintenance, planning, cost-benefit analysis, and formal modeling.

Maintenance. Maintenance is a fine-grained, short-term activity of the product life-cycle that focuses on localized changes [31]. Bennett and Rajlich [8] have proposed a software model for evolution, in which, after the initial development of a system, it sequentially goes through the stages of evolution, servicing, phase-out, and close-down. However, if we interpret maintenance as evolution, then only modifications that preserve the conceptual integrity of the system are possible. Otherwise evolvability is lost. The ISO/IEC 14764:1999 standard categorizes maintenance into perfective, corrective, adaptive and preventive maintenance [16]. This categorization only implies, but does not identify, possible engineering aspects of architecture evolution.

Planning. This area is concerned with delivery of functionality in increments, and has been mainly addressed by release planning techniques [28]. Release planning is about the allocation of features to ordered releases within certain constraints (e.g., priorities, resources, cost). A feature is a selling unit provided to the stakeholders. Features operate at the system level or at the infrastructure level, but are not directly related to architectural structures. In the context of IT applications, Erder and Pureur [12] have provided advice on how to realize a system architecture on a time continuum, based on successive states of architecture infrastructure (called plateaus), each of which provides a stable substrate for delivering a series of functionality (called waves). Although this approach is useful in theory, the guidelines are oriented to information technology and business architectures rather than to architectural design.

Cost-benefit analysis. This area encompasses the estimation of cost, resources, benefit, and uncertainty factors, so as to evaluate which courses of action are economically feasible. Techniques for cost estimation of software systems have been around for many years, e.g., COCOMO [10] and function point analysis [1]. Such techniques do not provide guidance for estimating benefit and uncertainty at the architecture design level. Planning for architecture evolution is a combination of the need to deal with uncertainty, along with costs and expected benefits. Real options analysis in the context of design has been proposed as a technique to tackle this problem [5]. Real options provide the right, but not the obligation, to take some action in the future. For instance, it is possible to estimate the value of a software structure as a function of the flexibility that modular designs provide by using real options [30]. That is, the use of flexibility mechanisms creates designs that are more (or less) evolvable. Modularity is not the only way to utilize options in software design. Real options have been used for valuing the stability of middleware [4], and for analyzing the economic value of applying certain architectural patterns [23]. However, all these techniques are not yet integrated into an architecture evolution process.

Formal modeling. We review two approaches: architecture description languages (ADLs) and architectural styles. An ADL is a formal language to describe a software architecture. While, there is no agreement on what exactly an ADL must provide (see a survey in [20]), it is generally accepted that the main elements of an ADL are components, connectors, and behavioral interactions. By providing formal syntax and semantics, ADLs facilitate analysis, which provide input for engineered evolution. However, existing ADLs do not provide any techniques to specify architectural change in the context of evolution [6].

An architecture style "defines a vocabulary of components and connector types, and a set of constraints on how they can be combined" [29]. Architecture styles provide a way to capture knowledge about common classes of systems, and a leverage for analyzing their properties. Recently, several researchers have proposed to extend styles to architecture evolution and defined a concept of *evolution styles*. Garlan [13] defines an evolution style as a set of evolution paths among classes of systems, e.g., evolutions from a web-based architecture to J2EE. Le Goaer et al. [15] define an evolution style at a much lower level of abstraction in terms of the structural changes involved.

3. Common Architecture Evolutions

We classify the common types of architecture evolution as follows: maintenance focused, open-ended (or open), and close-ended (or closed).

Maintenance focused evolution. Maintenance focused evolution aims to ensure an architecture that is fit enough to weather different classes of changes, fixes, and new require-

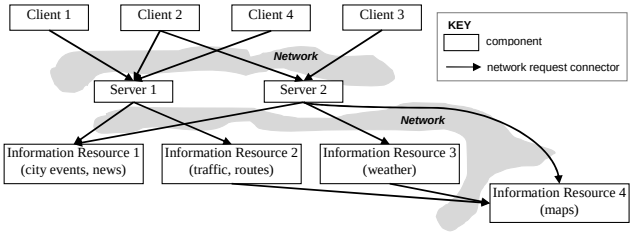

Figure 1. The initial CIS architecture.

ments. For example, the architecture of the Eclipse Rich Client Platform is focused on system evolution without key architectural changes, via the plug-in framework. Thus, in general, maintainability and modifiability are concerns that architectures need to be designed for. Design decisions that address these concerns need to be worked into the architecture [3]. Once such an architecture is specified, most of the maintenance practices are at a low-level of abstraction, centering on code evolution, component evolution, requirement evolution, and refactoring. When the architecture can no longer handle necessary changes, high-level structural modifications are needed. This leads to architecture transformations, hence evolutions, that go beyond maintenance.

Open evolution. Open evolution is characterized by high uncertainty. While aspects of the new architecture direction are known, business, technical, and market conditions prevent architects to shape a clear architecture a priori. For example, 3D computer graphics is an important feature in video game software. Many games that are successful in particular consoles cannot be easily evolved to different consoles, or environments such as handheld devices. An added challenge is anticipating the constraints that new hardware might impose and plan for evolving to those new platforms. The architecture focus of open evolution is often flexibility. Management of uncertainty becomes a critical aspect of open evolution. This is exemplified by the work on architecture real options [30, 4, 23].

Closed evolution. Closed evolution is where the characteristics of the current and envisioned system's architectures are known. For example, evolution from thick client to thin client architecture is an example of a closed evolution. Such an evolution is motivated by new technical solutions (e.g. incorporating CORBA for message exchanging), domain changes (e.g. need to support distributed call centers as opposed to central ones), new business models (e.g. businesses selling services as opposed to products), etc. There are many historical examples of closed evolutions, e.g., from mainframe to client-server, from stand-alone application to web-based, and from text-based to graphical user interface-based systems. From a modeling perspective, the two ends of closed evolution can be captured as instances of particular architectural styles [13].

Today, a common scenario of closed evolution is the migration of client-server systems (CS) to a service-oriented

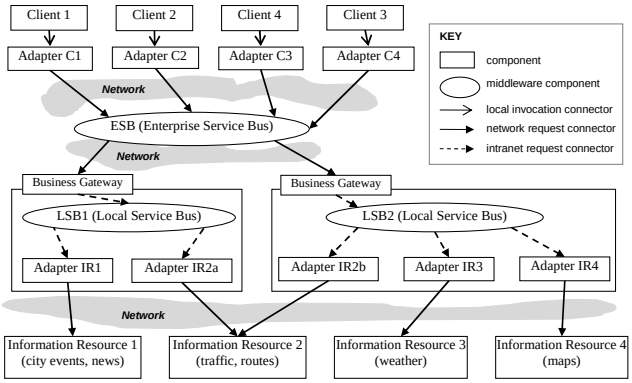

Figure 2. The CIS system evolved to an SOA.

architecture (SOA). As a running example, consider the city information system (CIS) website [11, 23]. In CIS, users retrieve information using a web browser and information resources host data about city events, places, weather, etc. The initial design of the system, based on the CS style, is shown in Fig. 1. In this design, accessing an information resource requires that its location be hard-coded in the server. When the server (information provider) receives a request from a client, it runs the appropriate service and returns the information. The owner of CIS makes a profit based on the number of hits the website gets, and believes that more information resources carried in the system will attract more users. The CIS is expected to evolve over time through the addition of new information resources.

Evolving the CIS system to an SOA paradigm changes the resulting architecture as shown in Fig. 2. In order to achieve this end state the architect needs to describe the steps for the evolution. The goal is to find an optimal way of keeping the old architecture elements while introducing the new ones. Thus, intermediate release points make the transition easier. In short, the architect must design the trajectory for the evolution, rationalized by optimal resource allocations, release points, and ordering of key tasks.

We believe that closed evolutions are interesting, yet sufficiently restricted to be engineered. Nevertheless, there are many parameters that need to be aligned in reference to each other. This makes closed evolution challenging.

4. Challenges in Closed Architecture Evolution

The key engineering challenges in closed evolution fall under the following categories: resource estimation, evaluation of alternative evolution trajectories, and the lack of tool support for estimation and evaluation.

Resource estimation. A common issue in resource estimation is the need to differentiate between local changes (e.g., to an interface or component) and architectural changes (e.g., to the system topology) [7]. Architectural

changes affect several elements, hence, can have ripple effects that need to be accounted for. To account for ripple effects and estimate the effort involved, every step in an evolution plan must be traceable to its source. For example, in CIS, a precondition to adding a Local Service Bus (LSB) is the existence of a gateway for a business domain (see Fig. 2). The gateway, in turn, depends on service identification, and the Enterprise Service Bus (ESB). Finally, the ESB requires adapters for the clients. Thus, resource estimation for adding one LSB element triggers decisions that affect the entire architecture.

Another issue in resource estimation is domain knowledge. This is especially true when adopting a new technical infrastructure, such as when moving from CS to SOA. In such cases, the architect needs technical information on the new domain to capture the key architectural properties of the new infrastructure required for resource estimation. For example, for the CIS evolution from Fig. 1 to Fig. 2, an architect needs to understand the preconditions for removing the connections from weather information resources to maps, without affecting the system functionality.

Evaluation of alternative evolution trajectories. There are many criteria to evaluate trajectories, e.g., length (shortest vs. longest), risk aversion, managing uncertainty, immediacy of benefits, resource utilization (i.e., budget, time, manpower), etc. The choice of which criteria to use is driven by business and mission goals. During evolution planning, the architect needs to decide on the order of the key steps with respect to such criteria. This often leads to an emergence of several alternative evolutionary directions.

For example, in CIS, the final architecture requires an ESB. There are many strategies to get there. One is to let clients temporarily talk directly to the business gateway, while the company evaluates different ESB options. Another, is to first deploy a subset of services, do a a short-term feasibility demonstration of the infrastructure, and only then identify all the services of the SOA design and migrate the components.

This leads to, at least, the following three evolutionary trajectories. Note that we only list the steps but not the intermediate architectures.

1. introduce adapters, ESB, business gateways;
2. introduce adapters, business gateways, connect adapters to business gateways, introduce ESB, remove connections between adapters and gateways, reconnect adapters and gateways to ESB;
3. put basic ESB in place, identify few key services, introduce adapters to connect these services to information resources and clients, test system, migrate rest of the components, improve ESB capabilities as needed.

In comparison, the second trajectory is redundant with respect to the first: it first connects adapters and then disconnects them. However, its benefits – a continuously working system – may exceed the costs. The third trajectory is more complex, but provides a risk mitigation by a partial deployment in the early stages of the migration.

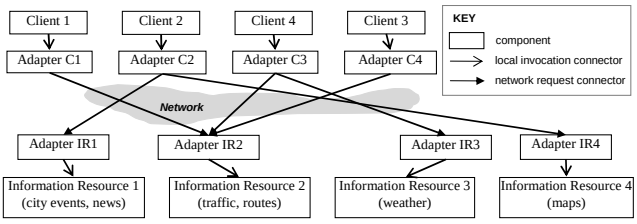

Figure 3. An intermediate CIS system.

Lack of tool support. Rational decision making for evolution requires a combination of quality-attribute based reasoning, resource estimation, and effective alternative evaluation. Practical tools that leverage formal analyses to aid in these aspects of planning and managing evolution are lacking. While there are commercial and prototypical tools for modeling with formal (e.g., ADL) and semi-formal (i.e., UML) languages, these tools provide only limited assistance for architecture representation. Architects need tool-supported analysis to help them make rational decisions. For example, in the trajectories introduced above, the decision to pick one over the other requires the architect to address issues such as dependencies between applying various adapters, added cost due to ripple effects, added development effort due to redundant connectors, etc. Without tool support, tracking these categories of information and reasoning about them collectively is a daunting task.

5. Modeling Architecture Evolution

We argue that in order to deal with the challenges outlined in Section 4, planning for evolution should be captured in an analyzable model with a certain degree of precision. In this section, we present our vision for such a model for closed evolution. We assume a closed evolution from an initial architecture A_0 to some given final architecture A_n. Reasoning about such an evolution requires the architect to also reason about the intermediate architectures (states). These intermediate states need to be planned and architected as well. Thus, we propose to model an evolutionary trajectory as a sequence of architectures: $\langle A_0, A_1, \ldots, A_n \rangle$. This is a simple way to encode a trajectory. It can be used to represent releases, keep track of necessary changes, and argue for advantages of different trajectories. We do not propose that the architect designs each state within a trajectory. The model is intended to provide a means to facilitate analysis and to formally capture the evolution process.

For instance, if A_0 captures the client-server architecture of Fig. 1 and A_n captures the SOA of Fig. 2, Fig. 3 shows an intermediary architecture A_1. This architecture repre-

sents a point-to-point service integration using adapters in between clients (service users) and information resources (service providers), rather than a true ESB infrastructure, as discussed by Robinson [27]. A_1 can be part of trajectory 1, as discussed in Section 4.

A trajectory can be constructed with a very high level of abstraction and coarse granularity, where the architectures within the trajectory are only A_0 and A_n. Alternatively, a trajectory can be captured in a very low level of granularity by representing every key structural change as a state within the trajectory. While the first approach does not provide the architect with the planning path, the second approach is not feasible and leads to "analysis paralysis."

To find a suitable level of abstraction between these two extremes, we need a mechanism to capture key structural changes (and steps) between intermediate architectures. To do so, we extend the trajectory with operators. Formally, a trajectory is a sequence $\langle A_0, O_0, A_1, O_1, \ldots, O_{n-1}, A_n \rangle$, where each A_i is an architecture, as before, and each O_i is an operator. Intuitively, an operator O_i represents the steps taken between architectures A_i and A_{i+1}.

What exactly is an operator? On one hand, it is a semantic action, such as "identify a service", or "adapt a client to use a service". On the other hand, it is a structural transformation of an architecture — some components and connectors are introduced and removed. To separate between these two views of an operator, we classify operators into abstract and concrete. An abstract operator represents only the semantic meaning of an architectural transformation. Each concrete operator is an instance of some abstract operator O. It inherits the semantic meaning of O and provides the additional detail required to implement O on the target architecture. In other words, an abstract operator provides an interface, and the concrete operator provides an implementation of that interface. In our running example, some abstract operators are:

- *IS*: Identify service(s),
- *ACS*: Adapt client(s) to use service(s),
- *CSSP*: Convert server(s) into service provider(s), and
- *AESB*: Add an Enterprise Service Bus (ESB).

Several concretizations of the above operators are shown in Fig. 4. Operator *IS* is instantiated into four services that capture the main business functionality of CIS. Operator *ACS* is instantiated into corresponding adapters for each of the client components of Fig. 1. Operator *CSSP* leads to five adapters that bridge communications with the information resources of Fig. 1. We assume that "Information Resource 2" will have to be accessed from two functional domains, each with its own communication protocol (e.g., XML and SOAP, respectively). This may justify the use of two separate adapters. Operator *AESB* is instantiated into three operators, which refer to two types of ESB: an ESB per se and an LSB. An LSB is an instance of ESB that provides connectivity support for a single domain [24]. All these operator instances ultimately boil down to basic changes in the architectural specification, such as: delete component, add component, connect two components through a connector, remove a connector, etc.

To estimate the cost of transforming an architecture using an operator, each abstract operator needs to be analyzed based on its *properties*. For example some properties of the *AESB* are: components affected, new dependencies introduced, new components introduced (e.g., security manager, and service registry), interfaces changed, new interfaces introduced, etc. These properties lead to a cost function. Each concrete operator provides the arguments to the cost function of its corresponding abstract operator. The cost of the trajectory is a combination of the costs of each constituent transformation, also taking into account combined affects. For example, in CIS, applying *AESB* requires all of the clients and new information providers to have a connection to the ESB. This affects the dependencies between the clients and the information providers. Furthermore, the ripple effects from removing the interfaces on the information sources must be taken into account

We envision that the architect will first select a bag of operators $\mathcal{O}$, and then construct a set $\{T_i\}$ of trajectories that lead from A_0 to A_n via the application of operators from $\mathcal{O}$. For instance, the three trajectories described in Section 4 are obtained by applying some of the operators shown in Fig. 4. Next, the architect uses cost-benefit analysis techniques to compare between the T_i's. The costs are computed using the properties estimated during operator instantiation. The benefits are estimated from the results of quality attribute analysis [3] on the architectures in the trajectories. For example, Ozkaya et al. [23] outline an approach to compute the cost-benefit of particular types of operators (specifically, architectural patterns) such as: insert a proxy (adapter), use a broker, or use a client-server-dispatcher; which can be integrated into possible trajectories.

Developing a meaningful and reusable set of operators is a fundamental activity proposed by our approach. We believe that, over time, a repertoire of abstract, as well as domain-specific, sets of operators will emerge. In the case of evolution from client-server to SOA, there are guidelines [24, 27] prescribing how an SOA infrastructure can be progressively designed and deployed. We expect that abstract operators can be mined from such guidelines.

6. Conclusion and Future Work

In this paper, we argue for planning a closed, architecture evolution as an engineered activity. We outline a framework for modeling evolution as a set of trajectories obtained via the application of operators. Operators is a formal way to

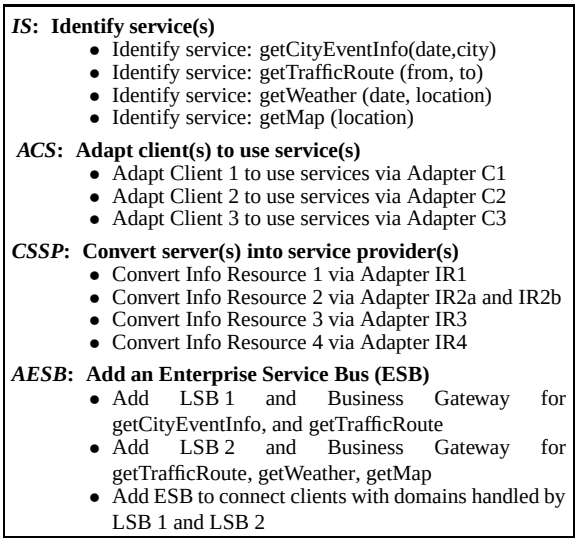

Figure 4. Operators: abstract to concrete.

identify and represent key evolutionary steps at a level of abstraction that is suitable for objective evaluation of alternative trajectories. Moreover, they help identify the architectural properties needed for cost-benefit analysis. Our current work focuses on defining and modeling useful operators, and investigating tool support [14]. It is motivated by the following research questions:

- Is it possible to define reusable operators for certain common classes of evolution?
- How can quality attribute theories, such as performance, aid cost-benefit analysis of evolution?
- Can operators be grouped under major quality attributes, such as operators that enhance performance, modifiability, etc. How does this grouping relate to existing work in architectural styles and tactics?

We hope to address these questions in future research.

Acknowledgment: This work has benefited from numerous discussions with F. Bachmann, J. Batman, R. Kazman, M. Klein, R. Nord, and B. Schmerl.

References

[1] A. Albrecht and J. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *TSE*, 9(6), 1983.

[2] W. Anderson, J. Bergey, M. Fisher, C. Graettinger, W. Hansen, and R. Obenza. Army Workshop on Lessons Learned from Software Upgrade Programs. Tech. Rep. CMU/SEI-2001-SR-021, CMU/SEI, 2001.

[3] F. Bachmann, L. Bass, and R. Nord. Modifiability Tactics. Tech. Rep. CMU/SEI-2007-TR-002, CMU/SEI, 2007.

[4] R. Bahsoon, W. Emmerich, and J. Macke. Using Real Options to Select Stable Middleware-Induced Software Architectures. *IEE Proc. Software*, 152(4), 2005.

[5] C. Baldwin and K. Clark. *Design Rules: The Power of Modularity*. MIT, 2000.

[6] O. Barais, A. Le Meur, L. Duchien, and J. Lawall. Software Architecture Evolution. In *Software Evolution*. 2008.

[7] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd ed.)*. AW, 2003.

[8] K. Bennett and V. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Proc. of the Conf. on The Future of Software Engineering*, 2000.

[9] M. Blechar and D. Sholler. Key Issues for Information and Application Architectures Management. Gartner Res., 2007.

[10] B. Boehm. *Software Engineering Economics*. 1981.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[12] M. Erder and P. Pureur. Transitional Architectures for Enterprise Evolution. *IT Professional*, 8(3), 2006.

[13] D. Garlan. Evolution Styles: Formal Foundations and Tool Support for Software Architecture Evolution. Tech. Rep. CMU-CS-08-142, CMU, 2008.

[14] D. Garlan and B. Schmerl. Ævol: A Tool for Defining and Planning Architecture Evolution. In *Proc. of ICSE'09*, 2009. Accepted for publication.

[15] O. L. Goaer, D. Tamzalit, M. Oussalah, and A. Seriai. Evolution Styles to the Rescue of Architectural Evolution Knowledge. In *Proc. of SHARK'08*, 2008.

[16] ISO/IEC. ISO/IEC 14764:1999 — Information Technology – Software Maintenance, 1999.

[17] M. Jazayeri. On Architectural Stability and Evolution. In *Proc. of Ada-Europe'02*, 2002.

[18] M. Jazayeri. Species Evolve, Individuals Age. In *Proc. of IWPSE'05*, 2005.

[19] M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proc. of IEEE*, 68(9), 1980.

[20] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *TSE*, 26(1), 2000.

[21] T. Mens and S. Demeyer, editors. *Software Evolution*. 2008.

[22] W. Opdyke and R. Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *Proc. of SOOPPA'90*, 1990.

[23] I. Ozkaya, R. Kazman, and M. Klein. Quality-Attribute Based Economic Valuation of Architectural Patterns. Tech. Rep. CMU/SEI-2007-TR-003, CMU/SEI, 2007.

[24] A. Papkov. Develop a Migration Strategy from a Legacy Enterprise IT Infrastructure to an SOA-based Enterprise Architecture. IBM, 2005.

[25] D. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. of the ACM*, 15(12), 1972.

[26] D. Parnas. Software Aging. In *Proc. of ICSE'94*, 1994.

[27] R. Robinson. Understand Enterprise Service Bus Scenarios and Solutions in Service-Oriented Architecture. IBM, 2004.

[28] G. Ruhe. *Handbook of Software Engineering and Knowledge Engineering*, volume 3, chapter Release Planning. World Scientific, 2005.

[29] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice, 1996.

[30] K. Sullivan, W. Griswold, Y. Cai, and B. Hallen. The Structure and Value of Modularity in Software Design. In *Proc. of FSE'01*, 2001.

[31] N. Weiderman, J. Bergey, D. Smith, and S. Tilley. Approaches to Legacy System Evolution. Tech. Rep. CMU/SEI-97-TR-014, CMU/SEI, 1997.