Crossroads
The ACM Student Magazine

Architecture-Centric Development: A Different Approach to Software Engineering

By ***John C. Georgas***, ***Eric M. Dashofy***, and ***Richard N. Taylor***

Introduction

Every student of computer science has practiced software development in introductory courses generally focusing on programming and algorithms. These courses typically introduce programming language constructs such as branches and loops in addition to basic design principles such as abstraction, modularity, and separation of concerns. For the majority of small-scale problems, these constructs and principles suffice. However, as system complexity increases, these techniques are insufficient for ensuring a successful outcome that fulfills the requirements and quality goals set out. The largest software projects have staggering failure rates: the Standish Report [18] states that nearly a third of projects are cancelled before completion and more than half suffer from serious cost overruns.

Large-scale software engineering is an inherently complex activity which involves the multi-person creation and manipulation of a large number of mostly intangible and highly dynamic artifacts. These can include requirement specifications, high-level software designs, source code, testing information, and post-deployment maintenance and evolution needs. Because of the complexity of the activity, much effort has gone into the improvement of the software engineering process: the steps taken during the construction of software. Widely-touted standards for achieving quality such as the Capability Maturity Model [6] focus almost exclusively on process. Yet, despite all this process improvement, we do not see a significant corresponding improvement in the software we build. A good process does not guarantee a good product.

We believe that there is another road to improving software quality and project success rates. Software architecture is a discipline that is able to connect and integrate the various stakeholders, activities, and products involved in software engineering. Software architecture also allows engineers much greater control and insight into their systems earlier in the development process and can foster early identification and avoidance of problems. As a result, architecture can help steer the project toward success rather than stumbling into failure due to a lack of understanding.

Every software system, from the smallest toy example to the largest multi-organizational system has an architecture which can vary in quality. Architecture captures the set of principal design decisions that are made about a system. Design decisions are choices made about how a system will be developed and how it will work, and these choices can include structure, organization, functionality, behavior, or more non-functional properties such as usability and aesthetics. The importance of these design decisions varies for each software system and is a function of the system's stakeholders, their concerns, and their specific needs. A key insight is that, although architecture is fundamentally a design-centric activity, architecture pervades the entire lifecycle. Architectural concerns focus on what is essential about a system and influence requirements and govern activities such as collaborative design, system development and implementation, evolution enactment, and adaptation.

Deriving full benefits of software architecture requires a proper application of the discipline. Systems with good architectures are likely to succeed, while systems with poor architectures are almost certainly doomed to fail. This paper introduces a "toolbox" of concepts, tools, notations, and methods that, when combined with traditional good software engineering techniques and processes, comprise a new, architecture-centric method of software development. The entire discussion is set in the context of promising research in the field of software architecture and provides advice and an overview of useful research approaches, methods, and tools that can be adopted by the practicing software engineer.

An Introduction to (Good) Software Architecture

This section introduces the most basic concepts in software architecture: the ingredients of good architecture and required activities. These topics will serve as a foundation for our later discussions.

Basic Architectural Elements

Good architectures, at the most basic level, capture a software system's structure in terms of interconnected high-level architectural elements. These elements are *components* and *connectors*, linked to one another in specific *configurations* [15]. Components are the architectural elements that are primarily responsible for computation. As components embody the functional aspects of the system, traditional design strategies such as functional decomposition can contribute significantly to the identification and bounding of components. Connectors are elements that are solely responsible for facilitating and managing the communication between components. Connectors allow independent components to interact without direct knowledge of each other. Since independence of components is a key driver of reuse, connectors can lead to significant cost savings. Finally, configuration combines components and connectors in a single, coherent whole. Configuration drives the behavior of a software system.

These architectural components and connectors are not abstract logical representations only useful during design, but should be explicitly modeled, retained, and implemented so that they accompany a system and are used throughout its development and deployment.

Architecture Modeling

Architecture modeling is the process of capturing and documenting architectural design decisions and can take place in a variety of ways. Some architects and developers capture architectural design decisions using natural language documents and abstract "box-and-arrow" diagrams. While these informal forms of documentation are useful in many respects, they generally lack the rigor and precision needed to comprehensively describe architectures.

Achieving an appropriate level of rigor and precision in architecture modeling is a key enabler of architecture's value in activities other than design. Good models provide a basis for architectural analysis, understanding and communicating about a system's design, and guiding a system's evolution. Architecture description languages (ADLs) are notations designed specifically to allow for the rigorous and precise specification of software architectures.

Various ADLs have been developed to document concepts important to a particular domain or area of interest to stakeholders. The Wright language [2], for example, focuses on the formal analysis of connector-based component interactions and therefore includes language support for capturing this aspect of software architecture. A new branch of ADL research has focused on extensibility, that is, creating ADLs that stakeholders can easily adapt to support new concerns or combinations of concerns. Examples in this paper use the XML-based xADL 2.0 language [7], which is unique because it focuses on the modular definition of ADLs through the use of the extensibility mechanism provided by XML schemas. xADL 2.0 is accompanied by a tool-set that spans the software engineering lifecycle from design, through implementation mappings, to maintenance and evolution.

Return on Investment: The Value of Architecture

In this section, we discuss techniques and tools applicable at various points in the software engineering lifecycle. These techniques have been demonstrated to add significant value to software engineering either through quality improvements, reduced time-to-market, or reduced costs.

Design and Analysis

Architecture development and design starts with the system's stakeholders. Developing an architecture is fundamentally a *concern-driven* activity. Maier and Rechtin, in their book *The Art of Systems Architecting* [13], use a diagram similar to Figure 1 to describe the relationship between concerns and architecture.

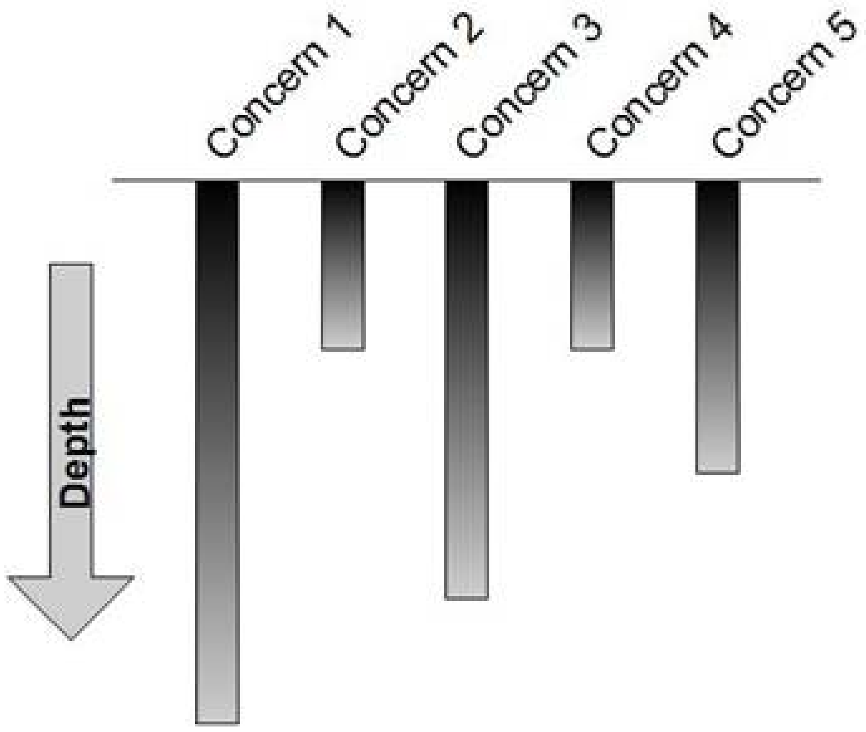


Figure 1: An illustration of the relationship between concerns and architecture.

Each bar on this figure represents a concern of one or more stakeholders. Some concerns, represented by the deeper bars, are more important than others: for example, in a medical system, reliability may be much more important than reconfigurability. Architecturally, the level of detail used to capture a concern should be proportional to the depth of that concern. Less important concerns will be addressed at a more abstract level of detail, while more important concerns will be addressed more precisely.

Although this diagram is deceptively simple, it conveys a profound message: the architecture of a system is driven and scoped based on what stakeholders perceive as important. Of course, which concerns are important will vary from system to system. Maintaining the focus on concerns ensures that attention is paid to the right aspects of the system and prevents designs from becoming focused on non-critical elements.

Architectural Styles

Once concerns are identified and prioritized, they can be used to identify and select appropriate architectural styles, which are key drivers of cost savings and help to prevent systems from diverging from their original designs. Reusing successful solutions is one of the fundamental principles of successful software engineering practice, and the same holds true in the context of software architecture. The creation of reusable components and connectors is one common form of architectural reuse. A more comprehensive kind of architecture-level reuse is accomplished through the adoption of *architectural styles* [**17**]. An architectural style is a named "package" of architectural design decisions expressed so that it can be applied to many different products with the goal of improving or helping to ensure certain target qualities.

Probably the best-known example of an architectural style is *pipe-and-filter*. In this architectural style, components (called filters) interact with the outside world solely through an input and an output interface. Data transferred through these interfaces is always in the form of character streams, although the specific format of these streams depends on the application. Streams are connected to components by connectors (called pipes). This style is linear and allows no branches. The pipe-and-filter rules have been specifically chosen to improve certain software qualities. For example, by standardizing component interfaces, interoperability is improved. By keeping components independent and allowing them to work on data as soon as it becomes available, pipe-and-filter applications can also better take advantage of multiprocessing and work more efficiently. When one types

```
ls -l | grep "foo" | more
```

at a UNIX command line, one is actually defining and executing a pipe-and-filter architecture composed of three components (`ls`, `grep`, and `more`) and two connectors (pipes). It should be noted that while pipe-and-filter rules may improve certain software qualities, the style itself cannot guarantee that any given composition will do anything useful.

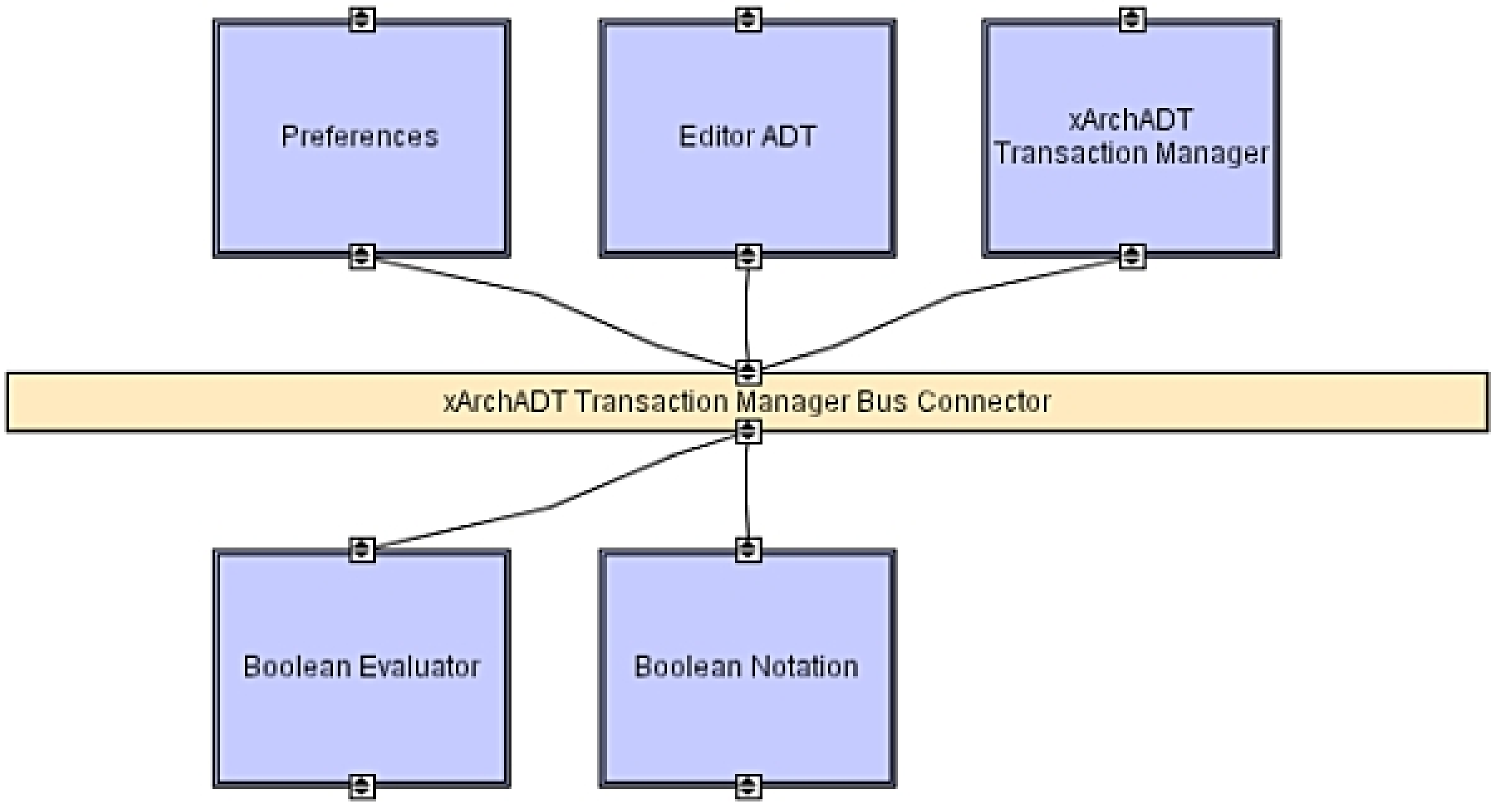
While pipe-and-filter is a trivially simple architectural style, more complex styles provide more value for particular domains. The Representational State Transfer architectural style [**10**] is the foundation of the modern Web and the HTTP/1.1 protocol and encompasses specific compositional principles that are aimed at optimizing network interactions. The requirements of stateless client-server interactions and client caching, for example, promote the qualities of system scalability and increased user-perceived performance. The C2 architectural style [**19**] combines aspects of many simpler styles -- such as layering and implicit invocation -- to create an amalgam which is particularly well-suited to developing highly dynamic, heterogeneous applications.

By leveraging styles, engineers can effectively get "free" engineering knowledge, patterns, and tools that can significantly reduce costs. If the architecture is developed based on stakeholder concerns, the desirable application qualities will emerge by selecting a style that induces qualities that are a close match to the most important concerns identified by the system stakeholders.

Modeling and Visualization

The selection of modeling notations and visualization techniques should also be concern-driven. From natural language to UML to ADLs, architects have a wide variety of notations and visualizations to choose from. Choices here should also be driven by previously-identified concerns of importance. Good architects must be well-versed in the panoply of modeling notations available, and make use of them liberally -- a common mistake in architectural modeling is to limit modeling to a single notation due to convenience, breadth of adoption, or to minimize effort.

Modeling, or capturing architectural design decisions, is closely intertwined with architectural visualization -- the manner in which design decisions are depicted to and manipulated by stakeholders. In fact, all modeling notations have at least one canonical visualization that is native to the notation. For xADL 2.0, the canonical visualization is textual because the architecture is represented in XML. However, many visualizations can be applied to the same model in order to display the same information in substantially different ways. Each visualization has different advantages and disadvantages that an architect must consider.


Preferences
Editor ADT
xArchADT Transaction Manager
xArchADT Transaction Manager Bus Connector
Boolean Evaluator
Boolean Notation

```
<component type="Component" id="xArchTrans">


<description>xArchADT Transaction Manager</description>

<interface id="xArchTrans.IFACE_TOP">


<description>xArchADT Transaction Manager Top
Interface</description>


<direction>inout</direction>

<type type="simple" href="#C2TopType"/>

<signature type="simple" href="#xArchTrans_type_topSig"/>


</interface>


<interface id="xArchTrans.IFACE_BOTTOM">
```

```
<description>xArchADT Transaction Manager Bottom
Interface</description>

<direction>inout</direction>

<type type="simple" href="#C2BottomType"/>

<signature type="simple"
href="#xArchTrans_type_bottomSig"/>

</interface>

<type type="simple" href="#xArchTrans_type"/>

</component>

<link id="xarchtrans_to_xarchadtbus">

<description>xArchTrans to xArchADT Bus
Connector</description>

<point>

<anchorOnInterface href="#xArchTrans.IFACE_BOTTOM"/>

</point>

<point>
```

```
<anchorOnInterface href="#xArchADTBus.IFACE_TOP"/>

</point>

</link>
```

Figure 2: Different architectural visualizations: the top shows a small part of the ArchStudio architecture graphically, while the bottom shows the textual definition of a component and link specified in xADL 2.0.

Figure 2 shows two visualizations that refer to the same xADL 2.0 model: a model of a part of the architecture of the ArchStudio environment [12]. One visualization is graphical and it depicts the structure of the system in a conventional "boxes and arrows" way. It is immediately obvious to a human what the structure of the system is and how the components and connectors are organized. It does leave out some details, however: the descriptions of the various interfaces and links, for example. The other visualization consists of text in a trimmed, less verbose subset of the xADL 2.0 ADL. This visualization hides nothing, but it is much more difficult for a human to understand the overall architectural configuration through a textual specification. Just as architects should consider using multiple notations to capture architectural decisions, they should also use multiple visualizations to present those decisions.

Architectural Analysis

Architectural analysis is concerned with answering questions that cannot be answered simply by knowing the structure of a system. For example, "Will the system be reliable?" There are many analysis methods that can be applied to architecture ranging from manual ones to those that are fully automated. For example, one method of architectural analysis is an inspection in which a team of stakeholders gathers and, often assisted by a checklist or other agenda, reviews the architecture using different visualizations to look for problems. In this sense, analysis can be as much about process as it is about notations and tools.

The ultimate objective of architectural analysis is, of course, fully automated analysis: the architecture is specified, and a tool determines whether the architecture achieves a certain quality or exhibits a given property. The extent to which this is possible is directly dependent on how the architecture is specified. Many notations, especially those based on a formal mathematical model, such as Wright, were developed specifically to facilitate certain kinds of automated analyses. Usually, these notations are accompanied by tools supporting these analyses. Other efforts have focused on developing new analysis methods for widely used notations, particularly UML; many of these are surveyed in [**8**]. For example, Engels, et al. [**9**] have developed an approach that determines whether statechart behavioral specifications for base and derived classes are compatible.

The availability of particular analyses to check software architecture against different concerns should be taken into account when choosing modeling notations and depth of detail. For example, if reliability and concurrency are important concerns, a notation that can be automatically checked for deadlock freedom should be selected.

Implementation

A good, well-analyzed architectural design is useless unless there is a clear mapping between this architecture and its implementation. The less complete or automated this mapping, the more opportunity there is for *architectural drift*: the phenomenon in which the system's implementation gradually diverges from its intended design.

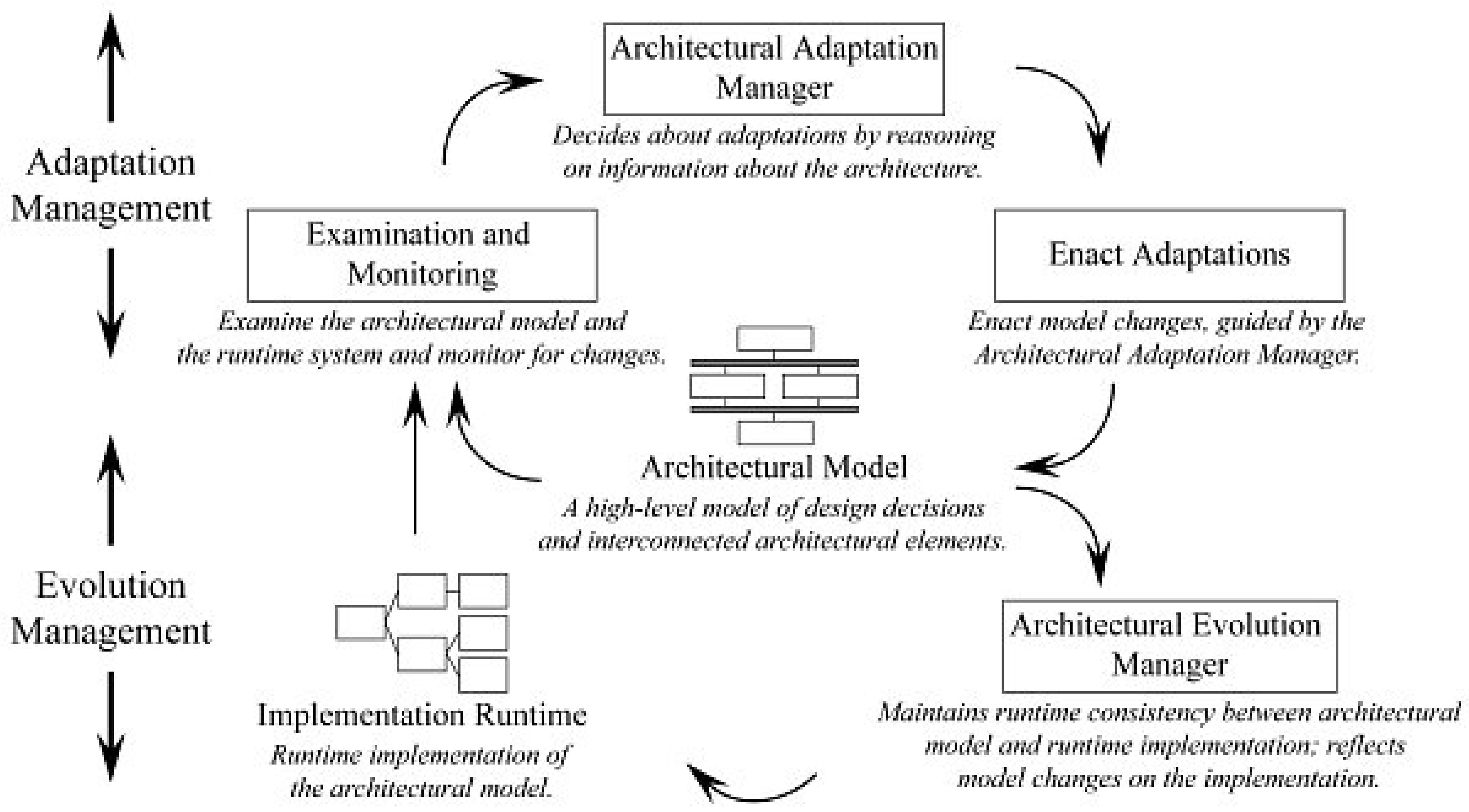
There are a variety of systems available which explicitly connect software architectures with implementations. The Archstudio environment and the xADL 2.0 language provide frameworks for linking architectural elements with their implementations through architectural frameworks. Frameworks are composed of software libraries that bridge the gap between architectural concepts -- such as components, connectors, and links -- and programming-language concepts like objects and procedure calls. Some frameworks, such as the one accompanying Archstudio, support the capability of automatically reflecting architectural changes to corresponding implementations during runtime. Other research efforts take an approach that is more focused on the source code artifacts. For example, ArchJava [**1**] embeds architectural information into Java source code to provide links between architectural and implementation artifacts.

As with analysis methods, the existence of implementation mapping techniques and tools can and should affect an architect’s choice of notations and modeling depth. While some mapping technologies are applicable to all kinds of implementation artifacts, others are bound to specific programming languages, operating systems, or middleware. If a project has specific needs in terms of the environment in which it is expected to run, then choosing notations with compatible implementation technologies is required. Techniques and tools that allow "round-trip" engineering, which automatically propagates architectural changes to the implementation, are the most preferable.

Evolution and Maintenance

Evolution and maintenance activities are primarily concerned with the operation of a software system after it is deployed. The goal of these activities is to ensure that a deployed software product continues to operate in a satisfactory way, adapting to new situations and needs. In current practice, evolution and maintenance activities account for the majority of costs incurred in software development [16].

Architecture provides the alternative of using architectural models as the basis for maintenance tasks. Updating a software component to use a newly available library, for example, can be achieved by changing its architectural connections so that it is linked to this new library. Focusing on this level of abstraction spares developers from having to delve into low-level component source code, which is harder to understand and modify.



Adaptation Management
Evolution Management
Architectural Adaptation Manager
Decides about adaptations by reasoning on information about the architecture.
Examination and Monitoring
Examine the architectural model and the runtime system and monitor for changes.
Enact Adaptations
Enact model changes, guided by the Architectural Adaptation Manager.
Architectural Model
A high-level model of design decisions and interconnected architectural elements.
Implementation Runtime
Runtime implementation of the architectural model.
Architectural Evolution Manager
Maintains runtime consistency between architectural model and runtime implementation; reflects model changes on the implementation.

Figure 3: An illustration of the evolution and adaptation management processes. An explicit architectural model is the key to the unification of these processes and to the leveraging of architecture for dynamic evolution and adaptation during system runtime.

The general process of using software architecture for evolution and maintenance, outlined in the lower half of Figure 3, begins by explicitly specifying the architecture of a software system using a machine-readable ADL. Once the connections between the architecture and its implementation are made, a system's architectural model becomes a reliable representation of its current state. This architectural model can then be used as the basis for system evolution. Although fine-grained changes to component behavior must still be made at the source code level, higher-level design changes can be expressed as changes to its architectural model. For example, adding distributed spell checking functionality to a word processing system can be effected by changing an architectural connector to use the remote spell checking facilities rather than an existing local one. Architectural changes are then detected and enacted on the running system through an *architectural evolution manager*, which ensures changes to the architecture are reflected on the running implementation.

This approach, adopted by a variety of researchers [**5**, **7**], has several benefits over traditional source-code-based evolution techniques: it prevents implemented systems from drifting away from their intended architectures as they evolve, it forces developers to maintain and update the architecture rather than leaving it to languish as a document used only during design, and it forms the basis for principled software evolution both off-line and while the system is running.

Adaptation

While software evolution is concerned with modifying systems at runtime, software adaptation involves deciding when and what modifications are appropriate. During software adaptation, systems are dynamically evolved to meet new functional or non-functional goals. The eventual goal is to construct systems that are self-adaptive, that is, systems that can autonomously change themselves. Software architecture can provide the foundational formalism and enactment platform for basing self-adaptive behavior on architectural models. This simplifies the kinds and the number of artifacts that must be considered during adaptation, and therefore has the potential to greatly ease the development and management of self-adaptive systems.

The vision we advocate for architecture-based self-adaptive systems -- initially outlined in [**14**] -- centers on an architectural model which can be used for dynamic software evolution. In addition, the logic for guiding adaptive behavior is also based on the architectural description of a system. The top half of Figure 3 presents an overview of this approach. Information about a software system is gathered through an examination of its architectural model and corresponding implementation. An *architectural adaptation manager* (AAM) is responsible for reasoning and deciding on what changes must be made to the architecture. Changes deemed necessary are then enacted as changes to the architectural model. These changes are then reflected on the running system through the evolution management process discussed in the previous section. The process stays the same whether the system is self-adaptive or not. In a human-driven system, the AAM acts as the coordination point where manual adaptation directives are injected into the system rather than encoding an automated decision-making process.

This approach necessitates the adoption of an architectural notation which supports (or can be extended to support) the specification of when and what kinds of adaptations are necessary. A variety of techniques have been developed to support the specification of adaptive properties at the architectural level: contracts for the interaction of object-oriented systems [3] and expert system rule-based policies [11] are two examples. Regardless of the specific representational approach adopted, the key insight is the use of dynamic architectural models which act as the foundation upon which to reason about self-adaptive behavior. Just as system evolution is better handled at the architectural level, the architectural level is also the most appropriate level of abstraction when it comes to the functional and non-functional concerns that adaptive systems must address.

Future Directions

Several general directions for near term software architecture research are evident: the first is concerned with how architecture is used during the software design activity, the second with the kinds of quality analyses that can be enabled through the strong adoption of software architecture, and the third with where architecture is applied in terms of application domains.

Architecture Supporting Design

One promising research direction is a broader investigation of the ways software architecture can improve to better support the design process. One aspect of this improvement is greater architectural support for the coordination of the everyday, collaborative practices of large-scale software engineering. Another aspect of this investigation will involve new ways of visualizing and manipulating software architecture models which go beyond drawing simple two-dimensional "box-and-arrow" diagrams. Two possibilities for improving architectural visualization are the inclusion of multiple layers into a single architectural description, and the manipulation of designs through alternative means such as three-dimensional or even virtual reality interfaces.

Finally, the research community will need to investigate different criteria for evaluating designs. Fundamental software engineering principles dictate a set of technically-oriented criteria that are currently used for evaluating designs, such as the avoidance of communication bottlenecks. There is still the potential, however, for determining a sense of *architectural aesthetics*: intuitive, human-centered criteria that establish a sense over whether an architectural solution is appropriate or progressing well. These aesthetics can facilitate discourse and communication between architects -- similar, after a fashion, to the criteria used to determine the artistic value of artwork through an evaluation of the use of color, form, symmetry, or specific compositions.

Architectural Quality Analysis

To more fully fulfill architecture's touted abilities to increase software success and quality, the research community must also provide better support for modeling and analyzing specific software qualities at the level of architectural design. For example, Banerjee, et al. [4] have performed a promising initial investigation in the area of applying architecture to the specific quality of trustworthiness. Their approach breaks down trustworthiness into various aspects, identifies general strategies for improving the trustworthiness quality, and finally identifies how each strategy can be implemented at the architectural level. Research such as this must be expanded to explore and model other software qualities at the architectural level, such as security, user-friendliness, and expected system behavior.

Architecture Applied

Finally, we believe it would be extremely worthwhile to revisit the thorough application of software architecture to various domains in strong collaboration with domain experts. Early research on domain-specific software architectures revealed that tremendous value can come from exploring deep architectural commonalities that occur in specific domains. As both the theory and practice of software architecture have substantially improved since these early research efforts, we believe it would be very worthwhile to once more direct efforts at extracting styles and patterns from domains such as autonomous space exploration and aircraft control.

This is not, however, simply an issue of technology transition: the research community cannot simply use these real-world domains as testbeds. Really examining the effectiveness of architectural practice will require a strong partnership and a two-way commitment from both researchers and domain experts. Current practitioners must be willing to expand their adoption of architectural techniques and wholeheartedly support their deployment, while researchers must be willing to study the results of their efforts in concrete settings and be prepared to discover the limits of the discipline itself.

Conclusion

Software engineering is a complex activity that involves a large variety of development artifacts with complex interdependencies at differing levels of abstraction. This paper supports the position that the discipline of software architecture should play a central role in providing a coherent view of a software system and its accompanying development process. Architectural models represent the set of design decisions involved in software engineering and must therefore serve as the primary point of integration between development activities and stakeholder concerns.

Applied throughout the lifecycle, good architectural practice has the potential to increase the understandability of a software system and the development process used to create it, ensure that qualities of particular relevance are met, and reduce the overall cost of software engineering. As a tool for improving design practices, architecture offers an effective way of modeling software systems while supporting reasoning and analyses about behavior before development is completed. While primarily a design-related discipline and artifact, architecture also has the potential to grant substantial benefits to the entire software development process including runtime evolution and adaptation.

In this paper, we have identified some specific approaches, tools, and techniques that show how architecture can be used to improve a broad spectrum of development activities. Of course, this does not mean that architecture can or should replace current practices shown to be effective; it does, however, mean that software architecture should be used as the core artifact that provides a coherent view and connects all of the people, activities, and artifacts involved in software engineering.

Acknowledgements

The authors would like to thank Anita Sarma and André van der Hoek for their help in formulating and clarifying the ideas presented in this paper. This work was supported in part by NSF Grant CCF-0430066 and by The Boeing Company.

References

1
Aldrich, J., Chambers, C., and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pp.187-197, May 19-25, 2002.

2
Allen, R., and Garlan, D. A Formal Basis for Architectural Connection. In *ACM Transactions on Software Engineering and Methodology*. 6(3), pp.213-249, July, 1997.

3
Andrade, L. and Fiadeiro, J.L. An architectural approach to auto-adaptive systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, pp.439-444, 2002.

4
Banerjee, S., Mattmann, C., Medvidovic, N., and Golubchik, L. Leveraging Architectural

Models to Inject Trust into Software Systems. In *Proceedings of the ICSE 2005 Workshop on Software Engineering for Secure Systems--Building Trustworthy Applications (SESS05)*, St. Louis, Missouri, May, 2005.

5

Cheng, S., Huang, A., Garlan, D., Schmerl, B., and Steenkiste, P. *Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure*. IEEE Computer, 37 (10), October, 2004.

6

CORPORATE Carnegie Mellon University, Paulk, M.C., Webber, C.V., Curtis, B., and Chrissis, M. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley Longman Publishing Co., Inc., 1995.

7

Dashofy, E.M., Van der Hoek, A., and Taylor, R.N. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 14(2), pp.199-245, April, 2005.

8

Elaasar, M. and Briand, L. An Overview of UML Consistency Management. *Carleton University, Technical Report*, pp.2-51, August 24, 2004.

9

Engels, G., Heckel, R., and Kuster, J. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML'2001)*. pp.272-286, Toronto, Ontario, Canada, 2001.

10

Fielding, R.T., and Taylor, R.N. Principled Design of the Modern Web Architecture. In *ACM Transactions on Internet Technology (TOIT)*. 2(2), pp. 115-150, May, 2002.

11

Georgas, J.C. and Taylor, R.N. Towards a Knowledge-Based Approach to Architectural Adaptation Management. In *Proceedings of ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*, October, 2004.

12

Institute for Software Research. *ArchStudio 3 Homepage*. <**http://www.isr.uci.edu/projects/archstudio/**>, 2005 (19 January 2006).

13

Maier, M. and Rechtin, E. *The Art of Systems Architecting*. 2nd ed. 344 pgs., CRC Press: Boca Raton, FL, 2000.

14

Oreizy, P., Gorlick, M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. An Architecture-Based Approach to Self-Adaptive Software. In *IEEE Intelligent Systems*, 14(3), pp.54-62, May/June, 1999.

15

Perry, D.E., and Wolf, A.L. Foundations for the Study of Software Architecture. In *ACM SIGSOFT Software Engineering Notes*. 17(4), pp.40-52, October, 1992.

16
Schach, S.R. *Classical and Object-Oriented Software Engineering*. McGraw-Hill Professional, 1995.

17
Shaw, M., and Clements, P. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *Proceedings of the Computer Software and Applications Conference*. pp.6-13, August, 1997.

18
Standish Group, The. *The CHAOS Report (1994)*. **http://www.standishgroup.com/sample_research/chaos_1994_1.php**, 1994 (19 January 2006).

19
Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*. 22(6), p. 390-406, June, 1996.

Biographies

John C. Georgas (**jgeorgas@ics.uci.edu**) is a PhD candidate in the Donald Bren School of Information and Computer Sciences Department of Informatics at the University of California, Irvine, and is advised by Professor Richard N. Taylor. His main research interests lie in the area of architecture-based software engineering, dynamic self-adaptive systems, system modeling, and design processes.

Eric M. Dashofy (**edashofy@ics.uci.edu**) is a PhD candidate in the Department of Informatics at the University of California, Irvine, and is advised by Professor Richard N. Taylor. His research interests are in the area of software architecture and modeling. He is a co-developer of the xADL 2.0 extensible architecture description language and the primary maintainer of the ArchStudio 3 architecture-based software development environment.

Richard N. Taylor (**taylor@ics.uci.edu**) is a Professor of Information and Computer Sciences at the University of California at Irvine and the Director of the Institute for Software Research. He received the PhD degree in Computer Science from the University of Colorado at Boulder in 1980. His research interests are centered on software architectures, especially event-based and peer-to-peer systems and the way they scale across organizational boundaries.