UNIVERSITY OF CALIFORNIA,
IRVINE

Open Architecture Software: A Flexible Approach to Decentralized Software Evolution

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Peyman Oreizy

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor David S. Rosenblum
Professor David Notkin

2000

The dissertation of Peyman Oreizy is approved
and is acceptable in quality and form
for publication on microfilm:

Committee Chair

University of California, Irvine
2000

To all my teachers throughout the years,
for their dedication and encouragement, and
for teaching me what everyone should know.

To my family,
for their love and support, and
for teaching me what I thought I knew but did not.

To Debbie,
for teaching me what I never imagined I would know.

Thank you all.

Table of Contents

List of Figures

List of Tables

Acknowledgments

There are so many people to thank!

First and foremost, I would like to thank Richard N. Taylor, my graduate advisor. He was instrumental in teaching me the skills I needed to *do* research, for providing opportunities to meet and work with other researchers in the field, for working relentlessly to create a productive and hassle-free research environment, and, most importantly, for giving me the intellectual freedom to pursue my own interests. Without his guidance, this dissertation would not have been written.

David S. Rosenblum and David Notkin, both of whom served on my thesis and candidacy committees, provided invaluable insights on this work. Their keen observations and questions have guided me and this work. I would also like to thank Michael Dillencourt whose unbridled enthusiasm inspired me to go to graduate school and who advised me during the two years I spent in the theory group before I recognized my true calling.

My research has also benefited from conversations with numerous colleagues and friends. Michael Gorlick and Dennis Heimbinger eagerly let me bounce ideas off of them. Debra J. Richardson critically reviewed my survey paper on the subject. Michael Franz, William H. Parker, and Nalini Venkatasubramanian served on my candidacy committee and provided new perspectives on this work. Alexander L. Wolf, Robert Balzer, Alfonso Fuggetta, Drew Bliss, Cliff Dibble, Mike Hatalski, David Hilbert, Peter Kammer, Nenad Medvidovic, Thomas Kistler, Rohit Khare, Roy Fielding, Jason E. Robbins, Mark Bergman, Erin Bradner, David McDonald, and E. James Whitehead, Jr. all provided critical feedback. I sincerely appreciate their suggestions, time, and guidance.

The past several years would have been much less rewarding without David Hilbert, Nenad Medvidovic, and Jason E. Robbins. I learned about in-the-trenches research with and through these friends.

In all of this, Essie Lev, my undergraduate counselor and friend, taught me the most valuable lesson of all—to have fun. Her passing was a great loss.

The Department of Information and Computer Science at UC Irvine, its staff, and its computing support group provided a top-notch research environment that was conducive to research. Also, the University of Washington generously provided me with office space during my dissertation writing phase.

Alas, doing research requires a work environment, nourishment, trips to conferences, computing equipment, etc., all of which requires money! My research was generously supported by the Defense Advanced Research Projects Agency, and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and F30602-99-C-0174. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory or the U.S. Government.

Curriculum Vitae

EDUCATION

2000 Ph.D. in-progress, Software Research Group, Computer Science, University of California, Irvine, Advisor: Dr. Richard N. Taylor.
1995 M.S. Computer Science, University of California, Irvine
1993 B.S. Computer Science, University of California, Irvine

PROFESSIONAL EXPERIENCE

Sept. 1995 - present	Graduate Student Researcher Software Research Group Computer Science University of California, Irvine
Sept. 1993 - Sept. 1995	Teaching Assistant Information and Computer Science University of California, Irvine
June 1995 - Sept. 1995	Apple Computer Researcher Advanced Technology Group
June 1993 - Sept. 1994	Microsoft Corporation Software Design Engineer Intern (6/93 - 9/93), Contractor (9/93 - 6/94), Intern (6/94 - 9/94)
Oct 1992 - June 1993	Institute of Transportation Studies University of California, Irvine Software Design Engineer

PUBLICATIONS

Refereed Journal Articles

J-1 Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, vol. 14, no. 3, pages 54-62. May/June 1999. Draft published as UCI Technical Report 98-27 (NR-1).

J-2 Peyman Oreizy, Richard N. Taylor. On the role of software architectures in runtime system reconfiguration. *IEE Proceedings—Software*, vol. 145, no. 5, October 1998.

J-3 Roy T. Fielding, E. James Whitehead Jr., Kenneth M. Anderson, Gregory A. Bolcer, Peyman Oreizy, Richard N. Taylor. Support for the virtual enterprise: Web-based development of complex information products. *Communications of the ACM*, vol. 41, no. 8, pages 84-92. August, 1998.

J-4 Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, Deborah L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pages 390-406. June 1996.

Refereed Conference Publications

RC-1 Peyman Oreizy, Richard N. Taylor. On the role of software architectures in runtime system reconfiguration. *Proceedings of the Fourth International Conference on Configurable Distributed Systems (ICCDS 4)*, pages 61-70, Annapolis, Maryland, May 4-6, 1998. 26 of 39 submissions accepted.

RC-2 Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor. Architecture-based runtime software evolution. *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, pages 177-186, Kyoto, Japan, April 19-25, 1998. 41 of 209 submissions accepted.

RC-3 Nenad Medvidovic, Peyman Oreizy, Richard N. Taylor. Reuse of off-the-shelf components in C2-style architectures. *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pages 190-198, Boston, MA, May 17-19, 1997. 21 out of 63 submissions accepted. Reprinted in Proceedings of the 1997 International Conference on Software Engineering (ICSE'97), pages 692-700, Boston, MA, May 17-23, 1997. Also reprinted in Software Engineering Notes, vol. 22, no. 3, pages 190-198.

RC-4 Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, pages 24-32, San Francisco, CA, October 16-18, 1996. 17 of 95 submissions accepted. Reprinted in Software Engineering Notes, vol. 21, no. 6, pages 24-32, November 1996. Draft published as UCI Technical Report 96-6 (NR-5).

Magazine Columns

MC-1 Peyman Oreizy, Gail Kaiser. The Web as enabling technology for software development and distribution. *IEEE Internet Computing, Column on collaborative work*, vol. 1, no. 6, pages 84-87, November/December 1997.

Refereed Workshop Publications

RW-1 Peyman Oreizy and Richard N. Taylor. Coping with application inconsistency in decentralized software evolution. *Proceedings of the Second International Workshop on the Principles of Software Evolution (IWPSE 2)*. Fukuoka, Japan. July 16-17, 1999.

RW-2 Peyman Oreizy. A flexible approach to decentralized software evolution. *Doctoral Workshop at the International Conference on Software Engineering (ICSE'99)*. Los Angeles, CA, May, 1999.

RW-3 Peyman Oreizy. Decentralized software evolution. *Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1)*. Held in conjunction with ICSE'98. Kyoto, Japan. April 20-21, 1998.

RW-4 Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor, David S. Rosenblum. Software architecture and component technologies: bridging the gap. *Proceedings of the OMG-DARPA Workshop on Compositional Software Architectures*, Monterey, CA, January 6-8, 1998.

RW-5 Peyman Oreizy. The WWW as an enabling technology for software engineering. *Workshop on Software Engineering and the World Wide Web*, held in conjunction with ICSE'97, Boston, MA, May 19, 1997. MC-1 is an updated version of this paper.

Invited Workshop Publication

IW-1 Peyman Oreizy. Issues in modeling and analyzing dynamic software architectures. *Proceedings of the First International Workshop on the Role of Software Architecture in Testing and Analysis*, Marsala, Sicily, Italy. June 30 - July 3, 1998.

Non-Refereed Publications

NR-1 Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, Alexander L. Wolf. Self-adaptive software. *UCI-ICS Technical Report 98-27*, Department of Information and Computer Science, University of California, Irvine, August 1998. J-1 is a revised version of this paper.

NR-2 Peyman Oreizy, David S. Rosenblum, Richard N. Taylor. On the role of connectors in modeling and implementing software architectures. *Technical Report UCI-ICS-98-04*, Department of Information and Computer Science, University of California, Irvine, February 1998.

NR-3 Roy T. Fielding, E. James Whitehead Jr., Kenneth M. Anderson, Gregory A. Bolcer, Peyman Oreizy, Richard N. Taylor. Software engineering and the WWW: the cobbler's barefoot children, revisited. *UCI Technical Report 96-53*. November 1, 1996. J-3 is an updated version of this paper.

NR-4 Peyman Oreizy. Issues in the runtime modification of software architectures. *Technical Report UCI-ICS-96-35*, Department of Information and Computer Science, University of California, Irvine, August 1996.

NR-5 Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. *Technical Report*

UCI-ICS 96-6, Department of Information and Computer Science, University of California, Irvine, January 1996.

PUBLICATIONS IN REVIEW

REV-1 Peyman Oreizy, Richard N. Taylor. Decentralized software evolution. Submitted to the International Conference on Software Engineering, 2000.

REV-2 Nenad Medvidovic, Peyman Oreizy, Richard N. Taylor, Rohit Khare, Michael Guntersdorfer. xADL: enabling architecture-centric tool integration with XML. Submitted to the International Conference on Software Engineering, 2000.

FORMAL PRESENTATIONS

Coping with application inconsistency in decentralized software evolution. Second International Workshop on the Principles of Software Evolution (IWPSE 2). Fukuoka, Japan. July 16-17, 1999.

A flexible approach to decentralized software evolution. Doctoral Workshop at the International Conference on Software Engineering 1999 (ICSE'99), Los Angeles, CA. May 18, 1998.

A flexible approach to decentralized software evolution. Topic defense. UC Irvine, April 14, 1999.

Decentralized software evolution. Candidacy exam. UC Irvine, December 14, 1998.

Runtime software evolution. Lecture in ICS 221 (a graduate class in software engineering). UC Irvine, November 3, 1998.

Issues in modeling and analyzing dynamic software architectures. First International Workshop on the Role of Software Architecture in Testing and Analysis, July 1, 1998, Marsala, Sicily, Italy.

On the role of software architectures in runtime system reconfiguration. Fourth International Conference on Configurable Distributed Systems (ICCDS 4), Annapolis, MD, May, 1998.

An architecture-based approach to runtime software evolution. International Conference on Software Engineering 1998 (ICSE'98), Kyoto, Japan, April 1998.

Decentralized software evolution. International Workshop on the Principles of Software Evolution, Kyoto, Japan, April 1998.

The WWW as an enabling technology for software engineering. Workshop on Software Engineering and the World Wide Web, Boston, MA, May 19, 1997.

Dynamic software architectures. Second EDCS Architecture/Generation Workshop, Santa Fe, NM, April 1997.

Web-based Software Development. W3C Distributed Authoring and Versioning (WebDAV) Symposium, December 5, 1996, Sunnyvale, CA.

SOFTWARE

ArchStudio: ArchStudio represented the first software development environment to bridge the gap between architecture-based design, analysis, system generation, execution, and runtime evolution. The most recent version of the environment is open and extensible, and includes a number of commercial- and research-off-the-shelf tools.

ArchShell: A key component of the ArchStudio environment, ArchShell pioneered the use of software architectures as the basis for runtime software evolution and dynamic adaptation in 1996. The tool allows a software architect to evolve a running implementation by changing its architectural model. Semantic annotations on the model permit ArchShell to verify and govern changes.

Extension Wizard: Another component of the ArchStudio environment, Extension Wizard permitted new architectural components and changes to be packaged, deployed, and installed in a running system using a off-the-shelf Web browsers.

PROFESSIONAL ACTIVITIES

Reviewer:
- IEEE Transactions on Software Engineering, 1998, 1999
- ACM Transactions on Software Engineering and Methodology, 1997, 1998
- Software - Practice & Experience, 1999

PROFESSIONAL ASSOCIATIONS

- Association for Computing Machinery (ACM)
- ACM Special Interest Group on Software Engineering (SIGSOFT)

Abstract of the Dissertation

Open Architecture Software:
A Flexible Approach to
Decentralized Software Evolution

by

Peyman Oreizy

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2000

Professor Richard N. Taylor, Chair

Customizing an existing off-the-shelf software package is an effective alternative to building one anew. Unfortunately, most software packages either lack provisions for customization by independent third-parties or support a predetermined class of customizations.

This dissertation advances the state-of-the-art in two areas. First, it presents a framework for understanding, evaluating, and comparing software customization techniques. Second, it proposes a new software customization technique, called *open architecture software*, that offers greater degrees of adaptability and consistency than previous techniques.

The comparison framework provides a practical and general-purpose technique for measuring the adaptability of decentralized software evolution techniques and is based on the concept of *software open points*, which, intuitively, represent the design decisions within a software package's implementation that third-parties can change. The framework is used to compare six popular techniques: application programming interfaces (APIs), software plug-ins, scripting languages, component architectures, event-architectures, and open-source software development.

Open-architecture software permits independent third-parties to change a software system by changing its architecture, i.e., the assemblage of its functional parts and their interrelationships. The essence of this technique lies in exposing this internal architecture, either in part or in full, as an explicit and malleable part of the package deployed to users. The architectural model doubles as a storehouse of semantic information that analysis tools can use to verify application invariants, and thereby offer certain assurances over third-party changes.

The claims of this dissertation are validated in two ways. First, a conceptual evaluation demonstrates that open architectures permit unprecedented changes in comparison to previous software customization techniques. Second, three case studies describe our experience using the open architecture approach. One case study uses the Mozilla Web browser, a large open-source software system comprising over 1.2 million source lines of code. Our experience demonstrates that a large, monolithic legacy system can be retrofitted with an open architecture, permitting third-parties to evolve the system in interesting and practical ways.

CHAPTER 1 Introduction

The best way to attack the essence of building software is not to build it at all.
– Fredrick Brooks, Mythical Man Month, Anniversary ed. 1995

Customizing an existing off-the-shelf software package is an effective alternative to building one anew. Unfortunately, most software packages either lack provisions for customization by independent third-parties or support a predetermined class of customizations.

This dissertation advances the state-of-the-art in two areas. First, it presents a framework for understanding, evaluating, and comparing software customization techniques. Second, it proposes a new software customization technique, called *open architecture software*, that offers greater degrees of adaptability and consistency than previous techniques.

The rest of this chapter serves as a broad overview of the dissertation. It begins by motivating the general problem of software customization, and then reviews and evaluates previous approaches to the problem. Next, it sketches the open-architecture approach and summarizes the case studies used to evaluate it. The chapter concludes by summarizing the contributions of the dissertation.

1.1 Perspective

After four decades of practical experience and research, software development remains an expensive, time-consuming, and error-prone activity. In a February 1999 report, the President's Information Technology Advisory Committee concluded that, “The demand for software has grown far faster than our ability to produce it. Furthermore, the Nation needs software that is far more usable, reliable, and powerful than what is being produced today” (PITAC Report 1999). To make matters more desperate, Fredrick Brooks convincingly argued and predicted in 1987 that no single advance in software development would result in “even one order of magnitude improvement in productivity, in reliability, in simplicity” (Brooks 1987). Twelve years later, Brooks’ prediction continues to hold true, for in the years since, no single advance has convincingly challenged it.

In sharp contrast to software technology, microprocessor technology has maintained the astonishing rate of improvement predicted by Gordon Moore over twenty-five years ago: a doubling of circuit density every eighteen months[1] (Moore 1965). This has lead to dramatically smaller, faster, and cheaper hardware systems. As a result, there exists an ever widening productivity gap between hardware and software development.

One way to close the software productivity gap, as many including Brooks have observed, is to avoid *building* software systems altogether and instead *(re)use* available off-the-shelf packages. Reuse magnifies the productivity of software developers by amortizing a software application’s production costs across every use of that application. Readily available off-the-shelf packages, such as word processors, spreadsheets, financial accounting packages and the like, routinely eliminate the need for custom-built applications. In many cases, these

1. Moore’s initial prediction was a doubling every twenty-four months. He later revised it to eighteen.

off-the-shelf packages provide capabilities that exceed the needs of users. The viability and beneficial consequences of application reuse are evidenced by today's flourishing shrink-wrapped commercial software marketplace and the high consumer demand for its products.

At times, a user's needs exceed that of available off-the-shelf packages, making as-is application reuse infeasible. In such cases, three remedies are often considered. One, the user can learn to cope with an existing package either by working around the deficiency or by revising their needs to correspond with the capabilities of an available package. Two, if certain capabilities are vital and non-negotiable, the user can procure a custom-solution, which often incurs considerable expense, delay, and risk. Three, an existing package can be *modified* to satisfy the user's crucial requirements.

This dissertation concerns the last strategy, in which an existing off-the-shelf package is modified, i.e., evolved or *customized,* to suit the needs of particular users. We refer to this form of software evolution, i.e., evolution by independent third-parties, as *decentralized software evolution*. Software customization combines some of the virtues of "reusing off-the-shelf packages" with "building custom packages." Under ideal conditions, customization benefits the three key parties involved, i.e., the original developers of the package, the users of the package, and the third-party developer who customize the package. The package's original developers sells more product since customization is a form of use. The users receive a tailored product in less time and at lower cost as compared to having a system fashioned anew. The third-party developers deliver a product in less time, at lower cost, of richer function, and that is better tested and documented as compared to a custom-built package.

Too often, an off-the-shelf package does not permit third-parties to make the necessary changes or—and just as important—the change undermines confidence in the software's integrity. Consider the problem from the user's perspective:

"An off-the-shelf package meets 90% of our requirements, but the missing 10% is vital."

Reality 1	Reality 2
"Unfortunately, the application cannot be modified to satisfy the missing 10%. We have no choice but to build a custom application from scratch."	"We can modify the application to provide the missing 10%. Unfortunately, the effort we must expend to regain confidence in the modified application is prohibitively high. We have no choice but to build a custom application from scratch."

Either way, the opportunity for reusing an off-the-shelf package is curtailed, causing a potentially small change in functionality to necessitate a disproportionately large amount of effort.

1.2 Previous Approaches

A large variety of software customization techniques have been developed to date. They can be roughly categorized into three classes: behaviorally closed, behaviorally open, and source-code based.

Behaviorally closed techniques permit third-parties to customize an application by selecting among and combining its *built-in* functionality, usually in a restricted fashion. Popular techniques include keyboard macros and application preferences. Keyboard macros, for example, record a sequence of user actions for later playback. This permits a new command to be defined in terms of a sequence of built-in commands. Application preferences and user-configurable options often enable, disable, or parameterize built-in application behaviors. For example, application preferences can allow users to change the application's user interface color scheme or the default file format used to store documents. Used effectively, these techniques can provide a rich source of customizations. They are advantageous in that they commonly do not require programming expertise to use, which makes them accessible to a broad class of users. As a result, a user can *self-tailor* their application to best meet their individual needs. The shortcoming of these techniques is that they can only express customizations in terms of the application's built-in behaviors—through selection, combination, and parameterization. Due to this limitation, we collectively refer to these customization techniques as *behaviorally closed*.

Behaviorally open techniques permit third-parties to customize an application by augmenting its build-in behaviors with novel behaviors. This permits a richer class of customizations than behaviorally closed techniques since new behaviors are not restricted to being compositions of built-in application behaviors. Popular techniques of this category include application programming interfaces (or APIs), software plug-ins, scripting languages, component architectures, and event architectures. These techniques differ with respect to the types of change supported. For example, APIs, plug-ins, scripting languages, and event architectures permit third-parties to augment or extend built-in application behaviors but prevent (or severely restrict) removal and replacement of built-in behaviors. This is referred to as the *restricted scope problem*. Component architectures expose an application's internal structure to third-parties and can permit replacement and removal of built-in application behaviors, though change is shackled by the tight coupling among functional parts. This is referred to as the *restricted change problem*.

The third class of techniques, i.e., source-code based techniques, provide third-parties with the application's source code and permit them to modify it directly, as they see fit. Open source software (Raymond 1999) is an example of this technique. Theoretically, these techniques permit third-parties to change all aspects of application behavior. The trade-off is that third-parties must thoroughly understanding (enough of) the application's source code to reason about the proper ways to change it. Often, this places a considerable burden on third-parties. This is referred to as the *change analysis problem*. Once a change has been made, the application must be thoroughly retested to verify that the change does not interfere with the proper functioning of built-in application behaviors. This is necessary to regain confidence in the modified application. This is referred to as the *change fragility problem* since errors are often difficult to detect. A related problem concerns merging changes from multiple, independent third-parties in the same application. Since third-parties work independently of one another, their mutual changes may interfere with one another when combined. This is referred to as the *change composition problem*.

Considered broadly, the behaviorally closed and behaviorally open techniques avoid the change analysis and fragility problems that plague source code techniques by restricting the scope and types of changes, i.e., by sacrificing the degree of adaptability available to third-parties. Conversely, source code techniques avoid the restricted scope and change problems that plague other techniques by divulging the application's source code, and consequently, cannot offer strict guarantees over consistency.

In general, the benefits and shortcomings of available techniques are poorly understood. This makes it difficult for practitioners to select techniques that are suitable for their needs. Chapter 2 places software customization into the broader context of software evolution, discusses these techniques in more detail, and considers the opportunities and risks they introduce. Chapter 3 surveys behaviorally open techniques and presents a comparison framework highlighting their capabilities and shortcomings.

1.3 A New Perspective and Approach

Can we devise software customization techniques that circumvent or reduce these problems? The thesis of this research is that there exists a middle ground between available techniques wherein application developers can make a more balanced trade-off between adaptability and consistency.

This dissertation describes a novel software customization technique that strikes this balance. The technique, called *open-architecture software*, permits independent third-parties to change a software system by changing its architecture, i.e., the assemblage of its functional parts and their interrelationships. The essence of this technique lies in exposing this internal architecture, either in part or in full, as an explicit and malleable part of the package deployed to users. This allows third-party developers to use the application's architectural model as the basis for customization, enabling them to introduce new modules, replace or remove existing modules, or alter module bindings. The architectural model doubles as a storehouse of semantic information that analysis tools can use to verify application invariants, and thereby offer certain assurances over third-party changes.

The novel aspects of this technique are:

- *system-model based change* — an abstract model of the software system is deployed as an integral part of the system's implementation and serves as the basis for third-party change, i.e., third-parties change the system's implementation by changing its system model.

- *governed change* — third-party changes to the system model are relayed to an agent called the *architecture evolution manager*, which validates each change before making a corresponding change to the system's implementation. The model guides these tasks.
- *multiple, reconfigurable connectors* — a system's functional parts interact with one another through discrete elements called *connectors*, which can be reconfigured as necessary to support third-party changes. Furthermore, each connector can implement a different composition policy, thereby allowing these policies to change to suit the specific context of reuse.

The benefits of this technique over previous techniques include intellectual leverage over change, diversity of change, and new opportunities for governing change.

Open-architecture represents a behaviorally open technique that reduces the problems of restricted scope and change by exposing the application's architecture to third-parties and allowing them to change it in a flexible manner. Although open-architectures cannot match the adaptability offered by source code techniques, they ameliorate the problems of change analysis, fragility, and composition. As a result, open architectures offer a better balance between adaptability (what changes are possible) and consistency (what assurances over change are provided).

1.4 Evaluation

The claims of this dissertation are validated in two ways. First, a conceptual evaluation demonstrates that open architectures permit unprecedented changes by (a) enumerating the types of change supported by previous techniques, (b) demonstrating that open architectures support these changes, and (c) identifying types of change that are supported by open architectures but not by the others.

Second, our experience using open architectures in three applications is summarized and evaluated. Two of the applications, each under 10,000 source lines of code and built in-house, were implemented entirely in a component- and event-based architectural style. The open-architecture approach was then used to expose their architectures and several third-party changes were made to each system. The changes demonstrate a wide variety of change types and the role constraints play in governing changes. The third application, a large commercial software system comprising over 1.2 million source lines of code, was modified to expose a small subset of its architecture using the open-architecture approach. Several new features were then added to the system by changing its architecture. The changes demonstrate that a large, monolithic legacy system can be retrofitted with an open architecture, permitting third-parties to evolve the system in interesting and practical ways.

1.5 Research Contributions

The contributions of this dissertation include:

1. A *measure* for evaluating and comparing the flexibility of decentralized software evolution techniques (Section 3.2).
2. A practical, system model-based *approach* to decentralized software evolution, called open-architecture software, that engenders novel types of adaptability and consistency. The approach complements existing techniques and can be used on a subset of a legacy software system, i.e., in an incremental manner (Chapter 5).
3. A *conceptual architecture* for implementing system model-based approaches to decentralized software evolution, such as open-architectures (Chapter 5).
4. A policy neutral *mechanism* for governing third-party changes to a software system using system-models (Chapter 6).

5. A more effective *characterization* of the relationships between system models, the types of third-party changes they support, and the assurances they provide (Chapter 4).

1.6 Organization of this Dissertation

Chapter 2 provides a broad introduction to software customization and defines many of terms used throughout the dissertation. Readers familiar with software customization can skip this material, but should review the definitions in Section 2.4. Chapter 3 surveys behaviorally open software customization techniques and presents a framework for characterizing and comparing them. Informed by this framework, Chapter 4 identifies the shortcomings of previous techniques. Chapter 5 presents the open-architecture approach. Chapter 6 and 7 then present case studies that report on the use of open-architectures in three applications: two small applications developed in-house and one large independently-built commercial software system. Chapter 8 evaluates the open architecture technique, including its risks, and Chapter 9 summarizes the contributions of this research and suggests areas for future study.

CHAPTER 2 Software Customization

Software customization broadly refers to the activity of modifying a software system to better suit the needs of a particular task. This chapter reviews the fundamental concepts and techniques of software customization. The material covered here spans a wide range of topics, not all of which is required to understand the remainder of this dissertation. Readers familiar with this topic should review the definitions in Section 2.4 before skipping ahead to Chapter 3.

The chapter begins by placing software customization in the context of software evolution and highlighting its unique characteristics. Next, the key stakeholders in the software customization process are identified and each of their concerns and needs identified. Next, essential terms and concepts are defined. Then, the breadth of software customization techniques is reviewed. The chapter concludes by examining relate work.

2.1 Software Evolution

Software evolution broadly concerns the activities of changes to a software system (Ghezzi et al. 1991) and has been the subject of much research (Lehman and Belady 1985, Parnas 1972, Lientz and Swanson 1980, Gamma et al. 1995, IWPSE 98, IWPSE 99). Table 2-1 places decentralized software evolution in the broader context of software evolution by partitioning commonly used software evolution techniques based on *when* they can be applied and by *whom*. Software systems can either be modified by a centralized authority, such as a single vendor (top row), or by a decentralized group, such as multiple independent vendors (bottom row). Software systems may also be modified during the design and implementation phase (left column) or after it has been deployed to users (right column). Given a context for software evolution—a *when* and *by whom*—the table helps determine which techniques are best suited for the task.

		When	
		Design-time (or pre-deployment) evolution	**Post-deployment evolution**
Who	**Central authority (e.g., single vendor)**	Design notations, methods, and tools; process systems; group communication and collaboration tools; configuration management	Release management systems; binary patch files; configurable distributed systems
	Decentralized group (e.g., multiple independent software vendors)	Same as above, with special support for loose coordination among geographically distributed team members (multiple sites or cross-organizational); open source	APIs, software plug-ins, scripting languages, open source, component architectures, and event-based systems

Table 2-1: This 2x2 matrix categorizes different techniques used to support software evolution based on *who* can evolve the system and *when* evolution can take place.

Centralized, design-time evolution: A team of closely collaborating team members, working on a single system, with common goals, and shared access to design and implementation artifacts characterizes this category of evolution. Most software evolution techniques and tools support this category. For example, design notations and methods, such as the Unified Modeling Language (UML) (Booch et al. 1998), provide guidelines for system design and graphical notations for design capture. Group communication and collaboration tools, such as e-mail, revision control tools, and configuration management systems, help team members coordinate and manage software changes. The great majority of software development projects fall into this category of evolution.

Decentralized, design-time evolution: A collection of loosely coordinated teams, working on different systems with a shared infrastructure differentiates this category from its centralized counterpart. Examples of this category include the various versions of Linux and the Apache Web, the different version of which have been tailored to a particular application domain by different organizations. Also falling into this category are large, possibly geographically distributed, teams of a single organization as coordination and synchronization of project artifacts, people, and development processes becomes a significant impediment to development (e.g., see Herbsleb and Grinter 1999, Perry et al. 1998). Perry and Kaiser (1991) have studied the impact of large development teams on software engineering environments. They observe that as a team grows above approximately 20 members, the number *and* complexity of interactions increases substantially. As a consequence, additional rules and mechanisms are necessary to aid cooperation among team members. Fielding and Kaiser (1997) report on the processes and tools used by the Apache Group, a globally distributed software development team that develops the popular open-source Apache Web server. They identify the importance of e-mail communication, archival of e-mail communication (as a means to support group memory), a shared information space accessible by all project members, coordination tools that support loose collaboration, and at least one shared goal among participants in fostering a successful project. Cutkosky et al. (1996) report similar experiences in the manufacturing domain.

Centralized, post-deployment evolution: Once a system is deployed to users, the original application developers use software updates or "patches" to evolve the system. These updates can incorporate bug fixes or additional features. Since a single authority—the original developer—creates updates, change conflicts do not arise. As a result, most of the technologies in this category are concerned with the efficient distribution and installation of updates. Since most user environments lack a proper software development environment, software updates are rarely deployed in source code form. Instead, developers create binary patch files, which encode byte-level changes between the original and updated versions of the application binary. Developers distribute these to users, who apply the patch to update their installed systems. More sophisticated tools, such as Tivoli's TME/10 (1998) and Software Dock (Hall et al. 1999), use software module dependency information and intelligent agents, respectively, to guide system updates.

Decentralized, post-deployment evolution: Multiple software developers independently evolving a system differentiates this category from its centralized counterpart. This category of software evolution is commonly referred to as *software customization*. Popular applications that use this type of evolution include Adobe Photoshop, GNU Emacs, and Netscape Communicator. Each offers a wide assortment of third-party add-ons that augment or "evolve" the application's capabilities. The original application developers implement and document various extension mechanisms to enable third-party developers to build add-ons. Commonly used techniques include application programming interfaces (APIs), scripting languages, and software plug-ins.

There are far reaching consequences inherent in supporting this last category of software evolution. For example, consider an application in which multiple add-ons may be used simultaneously. Since binary patch files from two independent parties cannot be merged, an alternative add-on packaging and installation mechanism becomes necessary. Furthermore, combining independently developed add-ons may lead to unforeseen interactions among the add-ons and the host application. Consequently, approaches for avoiding and/or detecting and resolving inconsistencies must be used.

Research to date has primarily focused on design-time techniques. Recent work has also begun to investigate centralized, post-deployment evolution, such as in the areas of software deployment and intelligent updates (e.g., Hall et al. 1999), runtime software evolution (e.g., see Oreizy et al. 1998, Oreizy and Taylor 1998, Kramer and Magee 1985), and reconfiguration of distributed systems (e.g., see ICCDS 1992-1998).

Meanwhile, the software industry's interest in providing third-party customizable systems has increased. This is evidenced by the growing trend toward customizable packages and the trend toward "applications-as-platform". The remainder of this chapter and this dissertation concern decentralized, post-deployment evolution.

2.2 Motivation

Software customization represents a middle ground between *buying an off-the-shelf application* and *building an application anew*—one *buys* an off-the-shelf application and *modifies* it to suit their needs. As such, customization can exhibit many of the virtues of each, while at the same time, minimizing each of their shortcomings.

Many have proclaimed the potential benefits of software customization as an alternative to today's mega-sized, one-size-fits-all applications. Wirth (1995) and others have expressed the disadvantages of building monolithic applications that attempt to suit everyone's needs: everyone gets stuck paying for features they do not need or use, which complicate the product unnecessarily and wastes computing resources. Wirth sees software customization as a means of remedying the situation, whereby "ideally, only a basic system with essential facilities would be offered, a system that would lend itself to various extensions. Every customer could then select the extensions genuinely required for a given task." Don Norman (Rheinfrank 1995) and others have long advanced the idea that applications customized for a particular task, i.e., *task-oriented software*, can improve usability and help overcome the complexity of today's general-purpose, feature-rich, difficult to use software systems. The essence of the argument is that, by focusing on a particular end-user task, software developers can (a) eliminate features that do not pertain to the given task and (b) tailor necessary features to better suit the given task, such as by selecting more appropriate application defaults or by using task-specific vocabulary and graphical elements in the application's user interface. Alvin Toffler (Ewell 1998) argues that mass customization, or "demassification", is a modern societal trend that has begun to influence business products and practices. He notes that although consumers enjoy the low prices engendered by mass produced, one-size-fits-all products, they prefer products that are customized and tailored to their particular needs. In the context of business information systems, Fayad and Cline (1996) argue that to stay competitive in today's rapidly changing business environment, businesses must leverage adaptable software systems that can be incrementally altered as business needs change.

The success of numerous customizable software applications, such as Adobe Photoshop, GNU Emacs, Visio, Microsoft Office, Linux, Netscape Communicator, and the Apache Web server and the third-party add-on marketplaces each has created, demonstrate the benefits of this approach. The use of software customization techniques can enable "decentralized innovation," wherein third-parties can provide innovative new capabilities independent of the original application vendor.

Although users generally prefer an application that has been customized to suit their particular needs, Catalano (1997) warns that they do not want to assemble and test products themselves. As evidence, Catalano points to failures of commercial third-party add-ons for popular applications, such as Microsoft Office. Although Catalano provides little evidence for the reasons behind the failures—i.e., does it stem from bad marketing or technical difficulties—he does reveal that software customizable is not without risk.

Too often, an off-the-shelf package does not permit third-parties to make the necessary changes or—and just as important—the change undermines confidence in the software's integrity. Often, these problems are not revealed until a significant investment has been made in attempting to customize a package.

Another important factor to consider is the *relative cost of reuse*, i.e., the ratio between the cost of actually reusing an asset and that of developing it from scratch. Selby (1998) examined data from nearly 3,000 NASA software modules, and observed that modules reused without modification had a relative cost of reuse as low as 0.03 to 0.04. In contrast, modules reused with less than 25% of its code modified had relative cost of reuse as high as 0.55—

more than a ten fold increase. Unfortunately, the study did not account for the relative complexity of source code—some programs are easier to understand and modify than others—simply the proportion of it that was examined and modified. Similar studies should be performed for customization techniques that do not rely on understanding and modifying the host application's source code, but on learning and using other customization mechanisms, such as an application's programming interface (API). We discuss many of the other risks of software customization in Section 2.5.

Finally, it should be noted that software customization is not a panacea. Evolvability is an intrinsic property of a software package's design and implementation. Software customization techniques can, at best, *reveal* this evolvability to third-party developers; they cannot engender novel kinds of evolvability. Thus, a package's design and implementation places a fundamental limitation on the potential benefits of customization techniques. Stated another way, customization techniques can at best make a particular change as easy for third-parties as it would be for the original software developers.

2.3 Customization Process and its Key Stakeholders

Developing software systems that support decentralized evolution introduces unique issues into the software development process. A holistic understanding of the issues must recognize its stakeholders and their individual concerns and needs. Figure 2-1 depicts an idealized high-level process identifying the four key stakeholders and their interrelationships. Each role is described in more detail below. Each spiral represents the process carried out by a participant using Boehm's Spiral process model (Boehm 1988). The quadrants of each spiral partition the stakeholder's key concerns and tasks as they proceed from initial product concept, to product

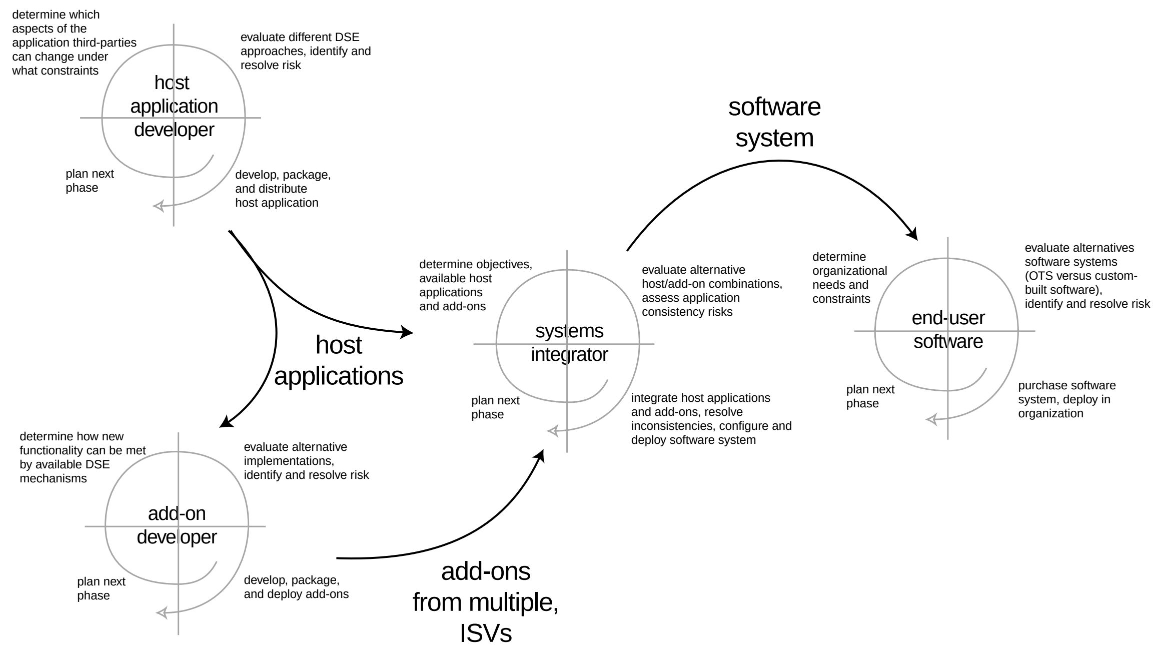

determine which aspects of the application third-parties can change under what constraints
evaluate different DSE approaches, identify and resolve risk
host application developer
plan next phase
develop, package, and distribute host application
software system
host applications
determine objectives, available host applications and add-ons
evaluate alternative host/add-on combinations, assess application consistency risks
systems integrator
plan next phase
integrate host applications and add-ons, resolve inconsistencies, configure and deploy software system
determine organizational needs and constraints
evaluate alternatives software systems (OTS versus custom-built software), identify and resolve risk
end-user software
plan next phase
purchase software system, deploy in organization
determine how new functionality can be met by available DSE mechanisms
evaluate alternative implementations, identify and resolve risk
add-on developer
plan next phase
develop, package, and deploy add-ons
add-ons from multiple, ISVs

Figure 2-1. A canonical high-level software customization process depicting the four key participants—the host application developer, the add-on developer, the systems integrator, and the end-user—and the artifacts that flow between them.

delivery, and then to product maintenance. Each process spirals outward through multiple releases. The quadrants represent: identifying objectives, alternatives, and constraints (upper-right); evaluating alternatives, identifying and resolving risks (lower-right); development and verification of the next iteration product (lower-left); and planning the next phase (upper-left). It should be noted that each participant is an independent entity and that it there is no explicit assumption of synchronization between their individual processes.

- The *host application developer* creates an application that others can customize. In addition to the traditional activities of software development, host application developers must evaluate the costs and benefits inherent to allowing third-parties to customize their application. To do this effectively, they must determine the allowable types of third-party customizations, the particular mechanisms that they would implement to enable third-parties to customize their application (upper-left), and the impact that third-party customization would have on other aspects of software development, including verification and testing, backward compatibility, documentation, and technical support (lower-right). Then, the application developers would implement the application plus the necessary mechanisms to support customization, verify, and deploy the application to others.

- The *third-party add-on developers* customize a particular host application to better suit the needs of an end-user community or application domain by creating software add-ons. The add-on developer identifies add-on requirements by either consulting with systems integrators and end-users or by trying to predict the needs of the market (upper-right). Add-on developers then evaluate alternative strategies for developing the add-on, which involves determining whether or not the add-on could be built using the customization mechanisms provided by the host application, and, if multiple customization mechanisms could be used, weighing the pluses and minuses of each in order to select the most appropriate one (lower-right). Once appropriate OTS applications and customization mechanisms are evaluated, the add-on developers would implement, test, and deploy the add-on (lower-left). Subsequent revisions and maintenance would then be considered and planned as necessary (upper-left). The particular techniques an add-on developer uses depends largely on the customization mechanisms provided by the host application, but it generally requires programming expertise.
- The *system integrators*, sometimes called "solution providers", build systems by assembling OTS applications and add-ons to suit the particular needs of an end-user community or application domain. The role of the system integrator is not to create applications or add-ons, but to assemble and integrate software systems for end-users

from existing parts. System integrators work with end-users to determine requirements and to identify suitable candidate OTS applications (upper-right). If there is a gap between the end-user's requirements and the capabilities of candidate OTS applications, system integrators collaborate with the host application developers and multiple, independent third-party add-on developers to determine if the gap can be reduced by customizing the OTS applications with add-ons (lower-right). Systems integrators then acquire candidate OTS applications and add-ons, install and configure them into candidate systems, and verify the degree to which they meet the end-users requirements. The assembled system is then deployed to the end-users (lower-left). The systems integrator then assumes the role of system maintenance (upper-left).

- The *end-users* acquire and use applications to support their objectives. The end-user process involves determining requirements, identifying alternative software systems (north-east), working with a system integrator to evaluate the alternative software systems and determine the degree to which they satisfy the requirements (south-east), acquiring, installing, and deploying the software system (south-west), and planning subsequent software needs (north-west).

Although we have presented these roles separately, they are often combined in practice. In the shrink-wrapped software industry, for example, consumers often take on the role of system integrator and end-user. Consumers purchase applications and add-ons from various vendors and assemble it themselves. In corporate markets, the MIS organization plays the role of third-party developer and system integrator.

Others have used a variety of names to describe similar stakeholder roles. Boehm and Scherlis (1992) use the term "megaprogramming" to describe the practice of building and evolving software by piecing together software components. Megaprogramming represents a combination of the task performed by third-party developers and system integrators, though we do not assume that software systems are evolved at the level of software components. Brooks (1995 p286) distinguishes between three types of megaprogrammers: (1) single-application/specialization megaprogrammer, who write code to customize a single application, (2) multiple-application/new domain megaprogrammers, who write code to integrate multiple applications, and (3) a function programmer, who writes new code available for megaprogrammers. In relation to our model, roles (1) and (2) are specializations of our system integrators, while role (3) combines our host application developer and third-party add-on developer. We distinguish between these two roles because while both can make changes to the application, the host application developer has complete access to the application's source, whereas third-parties can only change the application through the customization mechanisms made available to them by the host application developers.

2.4 Definitions

This section defines several key terms used throughout this dissertation and distinguishes them from related terms.

General-purpose or flexible. A software system is consider general-purpose and/or flexible if it can be used in a wide variety of situations without requiring any changes to its source code.

Evolvable. A software system is consider evolvable if it can be easily modified to satisfy new requirements. This conventional definition does not designate who can make changes, but the host application developer is assumed. Synonyms include *adaptable* and *modifiable* software.

Customizable. A software system is considered customizable if it allows third-parties to modify its behavior, usually without having access to its source code. Third-parties can add new behaviors, or replace and remove built-in application behaviors. More restricted forms of change are assumed by *extensible* and *configurable* software.

Extensible. A software system is considered extensible if it allows third-parties to *add* new behaviors to it, usually without having access to its source code (Pountain and Szyperski 1994). Extensibility is a limited form of customizability since it generally precludes the replacement and removal of built-in application behavior.

Behaviorally closed customization technique. A software customization technique is behaviorally closed if the range of customizations it offers to third-parties is exclusively through the selection and combination of built-in application behaviors.[2] Host application developers that use a behaviorally closed technique must, in essence, provide a rich collection of "atomic" behaviors that third-parties can select and combine to define new behaviors.

Behaviorally open customization technique. A software customization technique is behaviorally open if third-parties can add novel behaviors to the host application, i.e., define

2. For our purposes, a program is a specification written in a language with well defined, formal semantics that is executable on some machine (i.e., an executable specification language (Wing 1990) and is distinct from the inputs it operates upon. Altering this executable specification—by removing or replacing a portion of it, by augmenting it with another executable specification, or by changing its functional composition (i.e., from f(g(x)) to g(f(x)))—constitutes a "behavioral change." Although programs may treat inputs as data or program, such distinctions are immaterial for the purpose of defining behavioral change. Also, the particular representation of the executable specification that is modified is inconsequential—it may be programming language source code, module bindings, or compiled machine language instructions.

new "atomic" behaviors. This permits a richer class of customizations than behaviorally closed techniques since new behaviors are not restricted to being compositions of built-in application behaviors.

Decentralized change and composition. A software customization technique supports decentralized change and composition if it allows independently developed third-party add-ons to be combined in the same host application and if it attempts to either (a) avoid composition inconsistency or (b) detect and resolve composition inconsistencies.

2.5 Key Concerns and Issues

The process diagram in Figure 2-1 on page 19 reveals the interrelationships between software customization's four key stakeholders. The decisions of one stakeholder often impact another. By considering how one participant's changing decision impacts the other participants, we can identify several pertinent issues that impact software customization. These are:

Adaptability: The host application developer must decide which aspects of the host application third-parties can change and under what conditions. Such decisions circumscribe the types of changes third-parties can make and, ultimately, the types of systems that system integrators can assemble for end-users. For example, third-party plug-ins for Netscape Communicator can support new data types (such as an MPEG-2 video stream), but cannot replace the browser's bookmark facility (Netscape Corporation Inc., 1998). Therefore, it is imperative to understand the types of third-party changes enabled by different customization mechanisms.

Consistency: Third-party developers, as a consequence of working independently of one another, may change the host application in inconsistent or incompatible ways. For example, a poorly written module for the Apache Web server may compromise the security of a Web site, or less ominous, two Netscape Communicator plug-ins may try to register for the same media type. Therefore, host application developers should either devise strategies that prevent inconsistencies from occurring or use mechanisms that detect and resolve inconsistencies if they occur.

Backward compatibility and support effort: Once host application developers adopt customization techniques, they are pressured to maintain backward compatibility such that add-ons developed for the current release will continue to work with future releases. If backward compatibility is not maintained, third-party developers must modify their add-ons, and system integrators must reassess system assemblies with every host application release. The degree to which backward compatibility can be maintained largely depends on the resiliency of the abstractions exposed to third-party change. For example, although the Microsoft Windows operating system and its API have evolved significantly over the last decade, a large majority of the application programs that rely on its API have continued to operate without change. This is because the abstractions exposed through the API have been relentlessly maintained by its implementors. In contrast, the changing abstractions exposed to Windows device drivers has required that they be replaced with practically every operating system release.

Degree of integration: The customization mechanism used by the host application developer impacts the degree to which third-party add-on developers and system integrators can "seamlessly" integrate add-ons with the host application. Thomas and Nejmeh's (1992) four dimensions for evaluating tool integration in software development environments—presentation (i.e., user interface), data, control, and process—seem equally well suited for evaluating the degree of integration between host applications and their add-ons. We introduce a fifth dimension, documentation integration, which concerns the degree to which the help facilities of the host application and its add-ons are integrated. For example, in Emacs (Stallman 1984), end-users use the same mechanisms for retrieving documentation regarding built-in editing modes and commands as they do for ones provided by add-ons.

Adoption effort. Using a customization technique incurs cost and effort on the part of the host application developer, third-party developers, and system integrators. Adoption effort is an important discriminant of customization techniques, especially for third-party developers and system integrators where the effect is multiplicative. On one hand, exposing the host application's source code is easy for its developers—they can distribute the source code, build scripts, and on-hand documentation—but onerous for third-party developers, each of whom must understand and customize the host application at the source-code level, and the system integrators, each of whom must integrate, test, and resolve add-on inconsistencies at the source-code level. On the other hand, a component technique places more of the burden on the host application developers by requiring that they program according to strict, documented component interfaces, and reduces the burden on third-party developers since they can understand and change the host application at a higher level of abstraction than source code.

End-user effort. Third-party developers and system integrators usually insulate end-users from the complexities of implementing, integrating, and testing add-ons. In some cases, end-users assume these roles. In such situations, the level of end-user expertise and effort required to use a customization technique is an important consideration. For example, highly domain-specific scripting languages, like spreadsheet formula languages, accommodate end-user development better than general-purpose programming languages.

Implementation issues. Customization techniques can impose constraints on the implementation of add-ons. For example, a customization mechanism that uses a scripting language, such as Microsoft Office's Visual Basic for Applications, requires that all add-ons use a single implementation language. Other constraints may restrict the use of multiple threads of control, disallow the run-time addition and removal of add-ons, mandate a granularity for change, or sacrifice runtime performance.

Document portability. In some situations, end-users expect documents to be portable independent of the add-ons installed for the host application. For example, users of Adobe Photoshop can share images with one another irrespective of the plug-ins each uses since Photoshop uses a standardized, add-on-independent file format. Document portability is not a direct consequence of using any particular customization technique, but it is an important end-user issue that host application developers, third-party developers, and system integrators must consider.

Security and trust. In a traditional software marketplace, end-users trust a single entity—the software vendor—to guarantee certain measures of system reliability, stability, performance, etc. When systems are assembled from multiple, independently developed add-ons, end-users implicitly trust all the parties who contributed to the system: the host application developer, third-party developers, and the system integrator. If end-users assemble the system themselves, they become partially responsible as well. customization techniques are generally neutral with regard to such issues. Nevertheless, all the stakeholders must consider the security and trust implications of using software customization techniques.

Social and economic issues. In addition to the technical issues raised above, numerous social and economic issues impact software customization. For example, host application developers must provide incentives for third-party developers, foster collaboration and coordination among developers and system integrators, and determine licensing and software redistribution rights (Whitehead et al. 1995, Mackay 1990).

2.6 Related Areas of Study

Software research has, by and large, not directly investigated the topic of software customization. This is peculiar given the extensive research that has been done in tangential topics. For example, research has investigated numerous aspects of software reuse, from low-level code and component-based reuse to high-level design reuse and domain-specific software architectures (Krueger 1992). Yet, reuse at the granularity of software applications is largely ignored. Research on component-based software development has focused on components that range in size from hundreds to tens of thousands of lines of code, but has not generally regarded a software application as a single software component. Research on software evolution has primarily focused on techniques for designing and evolving software systems from an application developers point of view, not by independent third-parties. Configurable distributed systems and research in runtime software evolution (Magee and Kramer 1996, Purtilo 1994, Oreizy et al. 1998) have explored techniques for changing fielded software systems during runtime, i.e., centralized, post-deployment software evolution, but not its decentralized counterpart.

Design strategies, approaches, and methodologies, such as information hiding (Parnas 1972), object-oriented design (Booch 1994), mediator-based design (Sullivan and Notkin 1992), adaptive object-oriented design (Lieberherr 1996), design patterns (Gamma et al. 1995), aspect-oriented design (Kiczales et al. 1997a), etc., help *design* evolvable software systems, but lack any mechanisms for supporting software customization. Good design is a necessary but insufficient criteria for customization.

Application generators and 4th generation languages generate executable specifications from a high-level specification language that is suited for a particular application domain. Although useful, these tools do not necessarily lead to customizable software systems.

Tool integration technologies, such as Field (Reiss 1990) and Softbench (Gerety 1990), strive to address some of the same concerns as software customization, but with different assumptions. In tool integration, independently developed programs (i.e., tools) are integrated into a new whole (i.e., an environment). In decentralized software evolution, a program (i.e., a host application) is customized by independently developed programs specifically design for it (i.e., add-ons). Ultimately, tools with better software customization techniques give environment builders more leeway for integrating them into an environment.

End-user customization concerns software customization performed directly by its users, and has been studied extensively in the literature (see (Cypher et al. 1993, Mørch 1998)). In this domain, providing task-specific semantic primitives, simple control structures, and good visualizations of program state are critical in supporting users that lack programming expertise (Zarmer et al. 1992).

2.7 Conclusions

Software customization continues to have a significant impact on software development and the software marketplace. Numerous off-the-shelf packages provide software customization techniques. Yet, there are no clear characterizations of the unique issues that developers face when using these techniques or of their relative merits and shortcomings.

CHAPTER 3

Decentralized Software Evolution

This chapter evaluates and compares several techniques for decentralized software evolution (DSE) with regard to adaptability and consistency. In doing so, we establish a framework for understanding, comparing, and evaluating DSE techniques. Others can extend the framework with other DSE techniques. We use the evaluation criteria established here in subsequent chapters of the dissertation to identify the limitations of existing techniques (Chapter 4), to motivate our proposed technique (Chapter 5), and to evaluate its merits (Chapter 8).

Since countless systems, past and present, utilize some form of DSE, performing an exhaustive survey is infeasible. We instead examined a variety of techniques used by popular software systems of the day and those described in the research literature. Based on this examination, we identified six characteristic DSE techniques: application programming interfaces (or APIs), software plug-ins, scripting languages, component architectures, event architectures, and source code techniques. In general, other techniques can be derived by constraining or combining these six techniques.

Section 3.1 briefly describes the six characteristic techniques. Section 3.2 and 3.3 evaluate these techniques with regard to adaptability and consistency, respectively. Section 3.4 relates the comparison framework produced here with that of other work in the area. Section 3.5 discusses our findings.

3.1 Characteristic Techniques

Six popular behaviorally open DSE techniques are:

Application programming interfaces (APIs). With this technique, the host application exposes an interface—a collection of functions, types, and variables—to third-party developers. Third-party add-ons use and invoke interface elements to alter the host application's behavior. APIs are used by many operating systems and commercial applications.

Operating systems were perhaps the first "host applications" to offer APIs for DSE. In our context, the operating system represents the host application and third-party applications represent add-ons. The Microsoft Windows 95 API, for example, exports over one thousand functions and data types (Petzold 1996). Application programs invoke API functions that operate on files, manipulate graphical user interface elements, communicate with hardware devices, etc. McLellan et al. (1998) and Ran and Xu (1996) have described various techniques for organizing APIs to facilitate comprehension by third-party developers.

An API can also expose a meta-interface (Kiczales et al. 1991)—a collection of functions used to alter the behavior of the API itself. For example, Unix's ioctl() API function modifies parameters associated with a particular input/output device, affecting the behavior of other API functions such as read() and write(). Open implementation (Kiczales et al. 1997b) uses meta-interfaces as a means of allowing clients of the API to provide hints about how they will use the API. The implementation of the API uses these hints to "optimize" its behavior to the context of use.

Scripting languages. With this technique, the host application provides its own programming language and run-time environment to third-party developers. Third-parties use this language to implement add-ons. Add-ons, when executed by an interpreter, alter the behavior of the host application. Scripting languages commonly provide domain-specific language constructs and built-in functions that facilitate the implementation of add-ons, especially for end-users who lack programming expertise. Examples in AppleScript (Apple Computer, Inc. 1996), Visual Basic for Applications (Microsoft VBA 1997), elisp (Stallman 1984), and Tcl/Tk (Ousterhout 1994).

Spreadsheet formula languages, macro systems, and programming-by-demonstration systems are essentially scripting languages specialized for domain-specific needs. Spreadsheet formulas are expressed as mathematical formulae and stored in individual spreadsheet cells. The execution semantics are governed by the spreadsheet's recalculation engine, which determines the dependencies between spreadsheet cells and reevaluates expressions as needed. Macro systems confine the language to user-interface operations, and typically only allow them to be created indirectly, by observing and recording the end-user's actions. Execution occurs when the recorded macro is played back. End-user programming systems are similar to macro recorders, though they attempt to abstract and generalize the actions performed by the end-user so that they can be applied in broader contexts.

Plug-ins. With this technique, the host application pre-defines an interface that third-party add-ons or plug-ins implement. The host application uses and invokes the plug-in's interface, thereby altering its own behavior. A plug-in is akin to a behavioral place holder in the host application.

API callback mechanisms, such as those used by Open Implementation (Kiczales et al. 1997b) and function hooks (Nørmark 1996), are effectively software plug-ins in-the-small. For example, Emacs (Stallman 1984) and Interlisp (Teitelman and Masinter 1981) use hooks to inform third-party add-ons when certain operations, such as reading a file, are invoked.

Component architectures. With this technique, the host application exposes its internal component structure to third-party developers. Other techniques, in contrast, view the host application as a monolithic entity whose internal structure is opaque and immutable. Each component exposes an interface and may use the interfaces exposed by other components. Add-ons alter the behavior of the host application by adding new components that interact with existing components or by replacing an existing component with one that exposes a compatible interface. Examples of component architectures include Microsoft's Media Player (Microsoft DirectShow 1998) and Unix's filters.

Event architectures. With this technique, the host application exposes two distinct interfaces to third-party developers: an incoming event interface, which specifies the set of messages it can receive and act on; and an outgoing event interface, which specifies the set of messages it generates. Events are exchanged between the host application and add-ons by passing messages through an event mechanism. Add-ons alter the behavior of the host application by sending messages to it or by acting on the messages they receive from it. Event mechanisms commonly provide a message broadcast facility, which sends a message to every add-on attached to the event mechanism. Event architectures are unique in that the event mechanism mediates the communication between the host application and add-ons. As an intermediate, the event mechanism encapsulates and localizes program binding and communication decisions. As a result, these decisions can be altered independently of the host or add-on programs. This is in sharp contrast to the other techniques, in which programs directly reference, bind to, and use each others interfaces.

Implicit invocation (Garlan et al. 1992), in which events are broadcast anonymously, is a subclass of event architectures. Implicit invocation has been formalized (Garlan and Notkin 1991), and several techniques for incorporating it into programming languages have been explored (Garlan and Scott 1993, Notkin et al. 1993).

Source code-based. With this technique, the host application exposes its source code to third-party developers. Third-parties alter the host application's behavior by changing its source code, and redistributing their modified version. Mozilla (www.mozilla.org), Linux (www.linux.org), and Apache (www.apache.org), all of which use the open-source software development model, are popular examples of this technique.

Host applications developers can combine these techniques. For example, operating systems typically use the API technique for third-party application developers and the plug-in technique for third-party device driver developers. Emacs (Stallman 1984) effectively combines several techniques. It provides a scripting language called elisp, source code for Emacs itself implemented in elisp and C, a plug-in mechanism based on Lisp-style function hooks, and an extensible documentation mechanism.

3.2 Adaptability

How do DSE techniques differ in terms of adaptability? What aspects of the host application can a third-party change and how? In this section, we propose a technique for answering these questions.

Our general strategy for evaluating the adaptability of a DSE technique relies on enumerating the *types of open points* that it supports *at a particular level of abstraction*.

Open points. Open points are documented, behavioral parameters of programs. The concept of open points was introduced by Kurt Nørmark (1996), who coined the phrase when devising a more flexible version of Lisp's function hook mechanism. Here, we generalize the concept by considering any behavioral change as an open point (i.e., not just function hooks), and use it to evaluate the adaptability of DSE techniques. For example, Netscape Communicator's plug-in mechanism provides a behavioral placeholder that third-party program modules can fill. The third-party module acts as a behavioral parameter of the host application, and the third-party's ability to add modules constitutes a type of open point.

Level of abstraction. Adaptability can be measured at various levels of abstraction. For example, we could imagine techniques that let third-parties change a program's machine language instructions, its source code, or its module bindings. Each level of abstraction facilitates some types of change and confounds others. A fair comparison of the types of change DSE techniques provide must evaluate their open points at a particular level of abstraction.

Here, we compare techniques at the module-level of abstraction even though the concepts we describe can be applied at any level of software description. We do this so that we may present concepts in concrete terms using specific examples instead of abstract intangibles. Although this choice helps make the presentation accessible, it runs the risk of alienating readers interested in software customization at a different level of abstraction. We attempt to remedy this dilemma, in part, by discussing how the fundamental concepts presented here apply at the level of programming language functions and event systems in Appendix A. Unfortunately, restrictions prevent a comprehensive treatment across a broad range of levels.

We have chosen the module level because preliminary evidence from research indicates that *components*—modules that encapsulate units of functionality and data with minimal coupling to other components—represent a highly leveraged basis for understanding, reasoning about, implementing, reusing, and evolving software systems (Perry and Wolf 1992, Shaw and Garlan 1996, ISAW 1995-2000, IWCSE, ICOP].

Open points at the module level. At the module-level, an application consists of a collection of modules and bindings (see Figure 3-1). Modules encapsulate black box units of functionality and are depicted as rectangles. Each module exposes an interface—a collection of functions, types, and variables—that other modules can use and is depicted as a filled circle on the left border of a module. A module referencing another module's interface element is

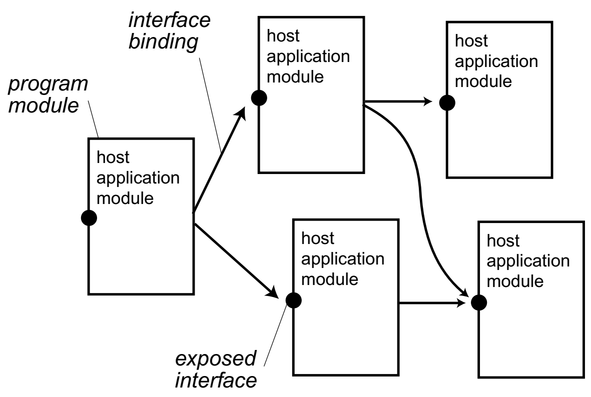


Figure 3-1. A module-level depiction of a program.

called a binding. Bindings represent one-way dependencies and are depicted as directed lines from dependent (source) modules to referenced (target) modules.

At the module-level, we define six types of open points: (1) adding a new module that exposes an interface usable by others and binds to the interfaces of any existing modules; (2) replacing one module with another; (3) removing a module, including all the bindings to and from it; (4) adding a new binding between two existing modules; (5) redirecting an existing binding by specifying a new target module; and (6) removing an existing binding. Since modules are black boxes, we assume that their interfaces are immutable. Changing a module's interface can be considered in our discussion as replacement of one module with another.

Also of interest is whether or not the host application's module structure is apparent to third-party developers. If it is not, third-parties must infer or reconstruct the overall structure by examining each module to determine which modules and bindings are impacted by a change.

The following subsections evaluate each of the DSE techniques described in Section 3.1. Figures 3-2 through 3-8 graphically depict these DSE techniques. These figures use the same graphical elements as Figure 3-1, and use shaded boxes to designate the portions of the host application's module structure that third-parties cannot modify. A table in each subsection summarizes the open point types supported by that technique. A filled circle (●) indicates full support for the open point type, an unfilled circle (○) indicates partial or restricted support for an open point type, and an empty cell indicates that the open point type is unsupported.

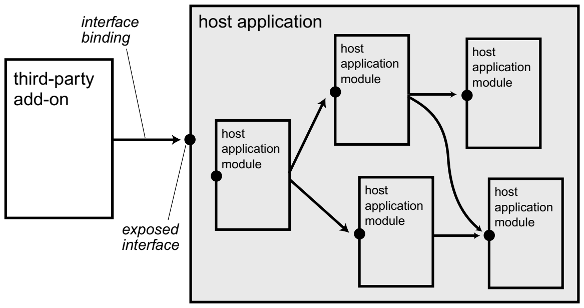


Figure 3-2. API module diagram.

3.2.1 APIs

An API is an interface exposed by the host application, consisting of a subset of the host's internal module interface elements. Third-parties develop add-on modules that use this API (see Figure 3-2). Thus, third-parties can add new modules, but the host application (and other modules) lack a canonical means of discovering and binding to interfaces of new modules. The host application is a monolithic entity whose internal modules, interface elements, and bindings are opaque to third-parties. Thus, system integrators can only replace and remove third-party modules. No explicit mechanism is provided to add, redirect, or remove bindings.

DSE technique	add module	replace module	remove module	add binding	redirect binding	remove binding	internal structure mutable	internal structure apparent
API	○	○	○					

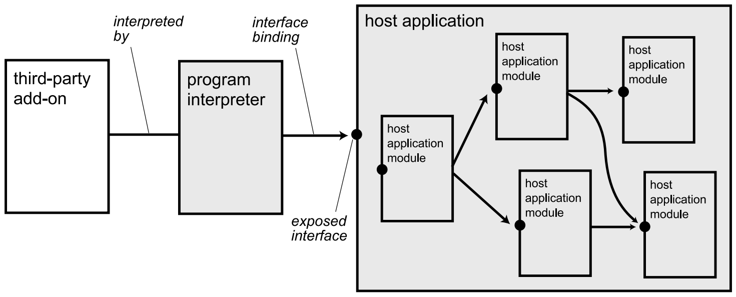


Figure 3-3. Scripting language module diagram.

3.2.2 Scripting languages

A scripting language provides a domain specific language for implementing add-ons. Yet scripts use the same mechanisms as an API to alter the host application (see Figure 3-3). Therefore, scripting languages share the same set of open points as an API.

DSE technique	add module	replace module	remove module	add binding	redirect binding	remove binding	internal structure mutable	internal structure apparent
scripting language	O	O	O					

3.2.3 Software plug-ins

The host application expects a third-party plug-in module to expose a predefined interface, which the host uses in a predetermined way (see Figure 3-4). This is in sharp contrast to an API, where the host exposes an interface for add-ons. Plug-ins can expose additional interface elements, but the host application (and other plug-ins) lack a canonical means of discovering and binding to these elements. The host application is a monolithic entity whose internal modules, interface elements, and bindings are opaque to third-parties. Thus, system

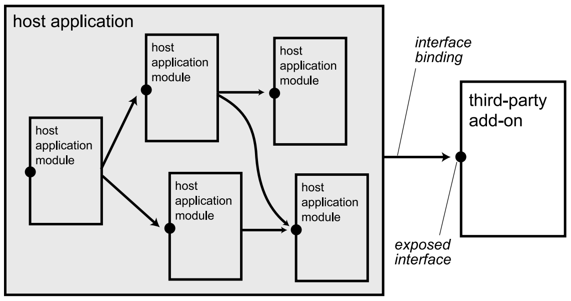


Figure 3-4. Software plug-in module diagram.

integrators can only replace and remove third-party modules. No explicit mechanism is provided to add, redirect, or remove bindings.

DSE technique	add module	replace module	remove module	add binding	redirect binding	remove binding	internal structure mutable	internal structure apparent
plug-in	○	○	○				○	

3.2.4 Event architectures

An event architecture exposes an event interface, which describes the set of messages that the host application sends to and accepts from third-party modules. All communication among the host application and third-party modules occurs indirectly through an event mechanism (see Figure 3-5). Third-parties can add new modules that communicate with the host application (or other add-on modules) by sending and receiving messages. The host application is a monolithic entity and its internal modules, interfaces, and bindings are opaque to third-parties. Thus, system integrators can only replace and remove third-party modules. Depending on the capabilities of the event mechanism, third parties may also be able to add, redirect, or remove some bindings. For example, third-party modules can intercept all messages sent over FIELD's message bus using the Policy Tool (Reiss 1996), and redirect or

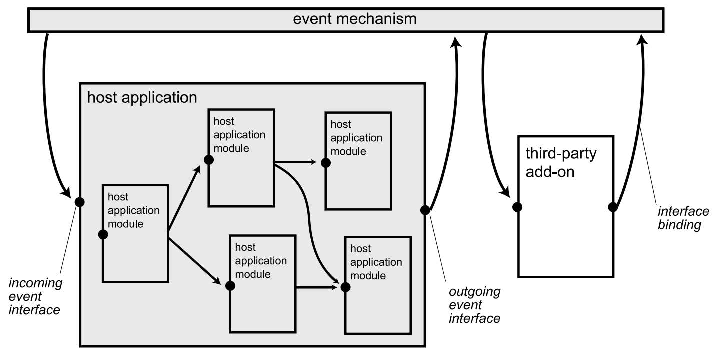


Figure 3-5. Event architecture module diagram.

eliminate them as desired. This enables third-parties to add, redirect, and remove bindings established through the event mechanism.

DSE technique	add module	replace module	remove module	add binding	redirect binding	remove binding	internal structure mutable	internal structure apparent
event architecture	●	○	○	○	○	○		

3.2.5 Component architectures

The key property that distinguishes component architectures from the other techniques is that it exposes the host application's internal module *structure* to third-parties, allowing them to change certain aspects of it (see Figure 3-6). Implementation technologies and frameworks that support component architectures, such as COM (Platt 1999), CORBA (OMG 1999), and JavaBeans (Chan and Lee 1997), provide canonical ways for modules to expose interfaces and to discover and bind to interfaces of other modules. Third-parties can add new modules and replace existing modules, but can only reliably remove a module once references to it have been removed. Since a component's implementation often references the other components is depends on (by name or using a unique identifier), third-parties can only add, change, or remove a binding by replacing the component that initiates it.

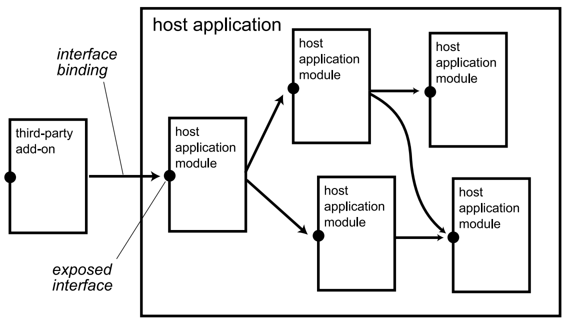


Figure 3-6. Component architecture module diagram.

Several modern component implementation technologies and frameworks provide special-purpose mechanisms for altering a component's bindings without modifying its implementation. For example, the "interception" mechanisms in COM+ (Platt 1999) and CORBA (OMG 1999) allow distinguished modules to intercept function calls to a specific module's interface as a means of extending its behavior. This approach has several limitations. One, interceptor modules are distinguished modules and unlike the host application's modules. As a result, they themselves cannot be extended using the interception mechanism. Two, the overall architecture of the host application is not apparent to third-parties. This makes certain changes fragile, since a third-party may, for example, replace a module that inadvertently disables critical interceptors.

DSE technique	add module	replace module	remove module	add binding	redirect binding	remove binding	internal structure mutable	internal structure apparent
component architecture	●	●	○	○	○	○	●	

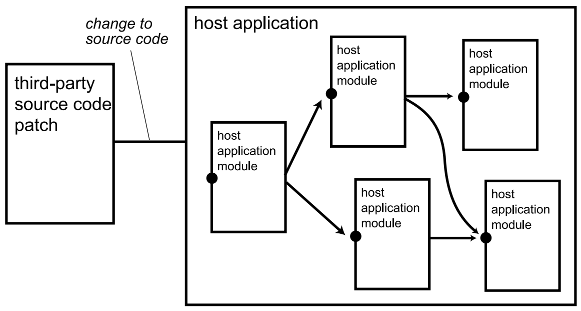


Figure 3-7. Source code module diagram.

3.2.6 Source code-based

Source code-based techniques, such as open-source software (Raymond 1999), expose the host application's behavioral specification to third-parties (see Figure 3-7). Thus, third-parties can modify every aspect of the host application, including the six open point types.

DSE technique	add module	replace module	remove module	add binding	redirect binding	remove binding	internal structure mutable	internal structure apparent
source code-based	●	●	●	●	●	●	●	

3.2.7 Summary

Table 3-1 summarizes the adaptability of the above techniques. APIs, scripting languages, and plug-ins support three open point types (with different restrictions). Only component architectures and source code techniques allow third-parties to modify the internal structure of the host application, but neither technique makes this structure explicit. Instead, third-parties must infer or reconstruct the overall structure by examining each module.

3.3 Consistency

Unrestricted change by third-parties may compromise the integrity of the host application. This usually occurs in one of two ways. One, a third-party, either out of negligence or malice,

DSE technique	Add module	Replace module	Remove module	Add binding	Redirect binding	Remove binding	Internal structure mutable	Internal structure apparent
API	○[a]	○[b]	○[b]					
plug-in	○[ac]	○[b]	○[b]				○[d]	
scripting language	○[a]	○[b]	○[b]					
component architecture	●	●	○[e]	○[f]	○[f]	○[f]	●	
event architecture	●	○[b]	○[b]	○[b]	○[b]	○[b]		
source code	●	●	●	●	●	●	●	

Table 3-1: Summary of open point types supported by the six characteristic DSE techniques at the module-level.

a. other modules cannot discover the interface of the added modules
b. confined to third-party modules and bindings only, not those within the host application
c. cannot bind to the functional interfaces of other modules
d. only the plug-in module of the internal structure is mutable
e. only if the module is not a destination of any binding
f. only by modifying the module that is the source of the binding

implements an add-on that violates a property of the host application. For example, a poorly written add-on for the Apache Web server may compromise the security of a Web site by serving private files. Two, third-party developers, as a consequence of working independently of one another, may change the host application in incompatible ways. In this case, each add-on works correctly when used independently of the others, but they interfere with one another when used together. For example, if two Netscape Communicator plug-ins try to register for the same document type, only one will be used as the renderer.

Anomalies of the first type involve two self-aware parties— a third-party developer creates an add-on for a particular host application—whereas anomalies of the second type involve independent parties. This difference has lead to two classes of techniques for addressing inconsistency. The next two subsections discuss and evaluate these techniques in turn.

3.3.1 Host application/add-on inconsistency

Host application developers offer a variety of resources to aid third-parties in creating add-ons. Commonly provided resources include technical documentation covering pertinent portions of the host application's design and functional behavior, cookbook code and sample add-ons, discussion groups and mailing lists, and test scripts. Such resources not only help third-parties implement add-ons, but to implement them "correctly." Clearly, this approach does not ensure application consistency. Host application developers rely solely on the good intentions and thoroughness of third-party developers in developing and testing their add-ons using traditional testing and analysis techniques. Even so, this represents the most common approach to avoiding application inconsistency in shrink-wrapped software applications.

When the intentions of third-parties are in doubt or when preserving integrity is paramount, two general strategies are used: (a) restricting the language used to implement add-ons, and (b) restricting the runtime environment of the add-on. We consider each of these in turn below.

Restricting the implementation language amounts to circumscribing the set of operators and operands available to add-on developers. This avoids certain classes of inconsistencies all together by eliminating language constructs that lead to those errors. For example, "type safe" programming languages like Java and Modula-3 do not allow pointer arithmetic, thereby preventing dangling pointers and misuse of direct memory access. Examples of more restricted implementation languages include spreadsheets formula languages and database query languages.

But restricting the add-on implementation language is not a panacea. If the language is Turing-complete, no amount of automated analysis can prove that the add-on terminates (this would be equivalent to the halting problem). Hence, only certain aspects of consistency may be guaranteed and the host developers must still rely in part on the good intentions of third-party developers.

Restricting the add-on runtime environment is achieved in one of two ways: program interpretation or software fault isolation. Program interpretation prevents certain types of inconsistencies by verifying program instructions before executing them. Examples of this technique include Java's virtual machine interpreter. Although interpretation usually occurs on an intermediate executable format such as bytecodes or p-code or at the source code level, modern virus checkers interpret native machine instructions to detect and prevent malicious operations by software viruses (Nachenberg 1997).

Software fault isolation (SFI) techniques (Wahbe et al. 1993), such as sandboxing, and proof-carrying code techniques (Necula and Lee 1996) achieve similar results with non-interpreted languages. In sandboxing, critical points of the compiled program are "instrumented" with logic that verifies particular constraints on the program's runtime environment. MisFIT (Small 1997), for example, instruments assembly language programs by inserting code before every instruction that references memory to verify that only permissible memory locations are addressed. Balzer (1998) uses SFI techniques to instrument common communication mechanisms (such as Unix sockets, RPC, and file I/O) as a way of augmenting an application's behavior. Other SFI mechanisms include the invalid instruction interrupts of modern hardware processors, the memory and file system protection mechanisms of operating systems, and Java's virtual machine security manager.

3.3.2 Interference add-on inconsistency

Although it is reasonable to expect a third-party developer to thoroughly test their add-on with the host application before releasing it to the public, such testing cannot reveal inconsistencies that may arise when it is used in conjunction with other third-party add-ons. Given a host application with n add-ons, there are unique host/add-on configurations (including the degenerate case of the host application without add-ons). Thus, an application with 20 add-ons has over one million unique configurations. Obviously, this makes exhaustive testing of configurations infeasible.

Software producers use a variety of strategies to combat the problem. We have generalized these into three categories: defensive design, standardized interfaces, and external tools.

A defensive design strategy avoids inconsistencies altogether by having the host application developers design and implement the host application in such a way that add-ons are forced to be mutually independent—they do not depend on or interact with one another directly. Instead, the host application mediates all interactions among add-ons.
Szyperski (1996) refers to such designs as independently extensible as they "can cope with the late addition of extensions without requiring a global integrity check." Krishnamurthi and Felleisen (1998) have formalized a more restricted notion of independent extensibility. Many shrink-wrapped applications adopt this strategy. For example, Netscape's Communicator Web browser supports a build-in set of audio and graphical media types, and allows third-parties to incorporate new media types using its plug-in mechanism. When Communicator begins executing, each plug-in associates itself with at least one media type. Communicator invokes the plug-in whenever it retrieves a Web page containing that content type. If a single Web

page contains multiple different content types, each plug-in is provided with an independent means of retrieving its content and its own private region of the display screen for rendering. With this strategy, two plug-ins will not interfere with one another so long as they follow the host application's direction. Although a defensive design strategy can be used to prevent many types of inconsistency, it has its drawbacks. The host application must mediate all the interaction between add-ons, which precludes the use of novel and useful interactions that were not foreseen by the host developers. Additionally, consistency is preserved so long as the add-ons do not inadvertently interact through some shared resource, such as an operating system device or file.

A standardized interface strategy, used by component-based and event-based architectures built on CORBA (OMG 1999), COM (Brockschmidt 1994), and FIELD (Reiss 1996), assumes that interface compatibility implies behavioral compatibility. For example, FIELD's debugger understands 34 command messages and broadcasts 22 event messages, each of which has its own ASCII string representation. Other tools in the environment (i.e., add-ons) can invoke debugger commands by sending it appropriately formatted messages. As a result, independently developed add-ons can interact so long as their interfaces are compatible. But interface compatibility is a poor measure of consistency since it does not prevent two (or more) add-ons from using the same component interface in incompatible ways. In the debugger example, two add-ons that simultaneously send a command to the debugger to 'single step' will end up advancing the program by two instructions instead of the intended one.

An external tool-based strategy uses tools to detect and resolve inconsistencies. The "system extension" mechanism of Apple Computer's Macintosh System 7 operating system uses such a strategy. With System 7, inconsistencies arise when two or more add-ons, called extensions or "INITs", intercept invocations to the same operating system function (called "patching a trap") in incompatible ways (e.g., by returning an unexpected value) (Zobkiw 1995, page 253).

Since there is no built-in mechanism for avoiding "trap conflicts," the only way to determine if two or more add-ons will work together is by trying them together. System integrators and end-users use utility programs to help detect and resolve inconsistencies in a semi-automated way. For example, Casaday & Greene's Conflict Catcher utility attempts to resolve an inconsistency by repeatedly rebooting the machine with different combinations of system extensions until it narrows the list of (potential) culprit extensions down to one. Such tools rely on the end-user to determine if the anomaly persists after each reboot—an arduous, time-consuming, and an error-prone process. The Complete Conflict Compendium (www.mac-conflicts.com) and MacFixIt (www.macfixit.com/archive.html) are examples of related strategies. These Web sites catalog the conflicts reported by end-users. End-users manually search these catalogues to find remedies to extension conflicts.

Strategies that detect and resolve inconsistencies for source code-based techniques are severely limited. This is because determining whether or not two arbitrary source code changes conflict requires careful analysis of the source code, and typically cannot be automated. The problem is analogous to merging several branches of a revision tree in a software configuration management system (see Buffenbarger 1995, Horwitz et al. 1989, Yang et al. 1992).

In general, these strategies assure a certain aspect of consistency in the host application by governing third-party changes. We consider possible ways to improve upon these techniques in Section 4.2.2

3.4 Related Work

Related work has largely focused on customization and extension in particular application domains, such as operating systems and databases, or through the use of special-purpose programming language constructs. In some instances, the proposed techniques imply extension and preclude more general forms of software evolution.

Notkin and Griswold's (1988) "extension interpreter" can dynamically load and link components into a running system. Their dynamic linker is novel in that it resolves function bindings using a separate "arbitrator" component. The arbitrator uses an explicit model of the relationships among application components to resolve function names into entry points. By replacing the arbitrator's lookup algorithm, one can constrain the set of allowable program structures. This mechanism could potentially be generalized to detect other types of program inconsistencies.

Research in "extensible operating systems" is concerned with incorporating mechanisms in operating systems that allow application programs, which are written by independent software vendors, to change underlying operating system services. Hence, the operating system and the applications that extend it are analogous to our notion of host application and third-party add-ons. Small and Seltzer (1996) survey operating system kernel extension techniques, where issues of consistency, performance, security, and trust dominate over the other issues on our list. They observe that operating system extensions generally fall into three basic structural patterns—replacing default system policies like paging, filtering data streams as in a network protocol stack, or other generic extensions supported by the operating system—and examine the performance overhead of using several different host application/add-on inconsistency strategies in those contexts. Seltzer et al. (1996) examine the range of choices made by different operating systems with regard to inconsistencies, security and trust, and granularity of change. These surveys have informed our more general results, which are one of the contributions of this paper. All of the issues and strategies we have discussed apply to domain-specific systems; but their inherent properties and invariants can simplify or confound particular issues or strategies. The dominance of performance and trust in the operating systems domain is a good example.

The programming language Beta (Malhotra 1994) provides a unique construct for allowing third-parties to extend application behavior without modifying or recompiling its source code. Beta's "virtual pattern" construct allows the host application to invoke a (subsequently added) third-party method if one is provided (i.e., the inverse of a super method invocation in object-oriented languages). Lisp's advise facility supports "insertive" programming (Sandewall 1978), whereby programmers can extend the host by writing functions that are executed before, after, or "around" its functions. Heineman (1998) has proposed a similar technique for Java Bean components. Although quite powerful, extensive use of insertive programming techniques can obfuscate the host application's design, making future extension more cumbersome. Hölzle (1993) proposes "type adaptations," which allow object-oriented classes to be augmented "in place" by adding and renaming existing class methods without accessing the application's source code. Many of these techniques where designed for specific programming languages, but could be incorporated into high level services such as component object models. Although these techniques offer fine-grained adaptability, they rely on overly simplistic mechanisms for preventing inconsistency, such as type or name compatibility.

Szyperski's (1996) and Krishnamurthi and Felleisen (1998) notions of "independently extensible software" is similar to ours, but more restrictive—both assume mutual independence among add-ons, which we do not.

3.5 Discussion

Systems development has an established history of providing source code as a means of achieving the ultimate degree of third-party adaptability. In describing the Interlisp system, Teitelman & Masinter (1981) observe:

> *"The most straightforward way of allowing users to modify or tailor system tools … is simply to make sources available and allow the users to edit and modify tools as they wish."*
> *— Teitelman & Masinter 1981*

In order for this work, users must be programmers and must understand enough about the system—how its source code is organized, program invariants, interactions between modules, etc.—before making changes. For systems consisting of several thousand lines of source code, this burden can be negligible. In large systems, this is a significant burden. Furthermore, some systems do not provide source code.

Source-based techniques also lack effective strategies for dealing with application inconsistency. Remedies to this problem will likely remain elusive since the low-level of abstraction at which the system must be examined confounds many whole-program analyses.

API, plug-in, scripting language, component architecture, and event architecture techniques support much more effective strategies for dealing with inconsistency, but sacrifice a significant degree of adaptability to attain it. The limitations of adaptability in component architecture techniques stems from the high coupling among components. In

API, plug-in, scripting language, and event architecture techniques, the lack of adaptability stems from the fact that the host application's internal structure is not exposed to, or malleable by, third-party developers.

The trade-off between adaptability and consistency is apparent. Intuitively, we expect this—the more changes we allow, the more opportunities we provide for making them incorrectly. What we do not as yet known is precisely where the trade-off occurs and to what degree new techniques can improve upon them. In other words, which aspects of the trade-off are intrinsic and which are not?

One strategy for increasing adaptability is to combine complementary techniques such that each masks the shortcomings of the other(s). Combining component architectures with event architectures is promising—an event mechanism would reduce the coupling among components, while at the same time exposing the host application's internal module structure. Early examples of this, such as the SPIN operating system (Bershad et al. 1995), have demonstrated improved adaptability.

We can similarly imagine devising more effective techniques for detecting add-on inconsistencies. Current strategies use simplistic models of compatibility (e.g., name, interface, or type compatibility). With the exception of tool-based strategies, the compatibility models are encoded into and intertwined with the application's implementation, making them difficult to verify and change independently.

We consider these and other alternatives in the next chapter.

CHAPTER 4

Problem Statement, Analysis, and Motivation

The purpose of this chapter is three-fold. First, it concisely states the problem that this dissertation tackles (its thesis). Second, it demonstrates that prior work in the area has not addressed this problem. Third, it discusses why it is beneficial to find a solution to this problem. The next chapter presents one particular solution to this problem.

4.1 Adaptability versus Consisting

Available DSE techniques are bipolar—they either offer a high degree of adaptability with little or no assurance over consistency or severely restrict change to achieve consistency. Figure 4-1 abstractly depicts this dichotomy.

Source code, depicted in the extreme lower-right corner, offers complete adaptability because it permits third-parties to change the application's source code, its principal behavioral specification. In theory, any change is possible. In practice, the burdens of

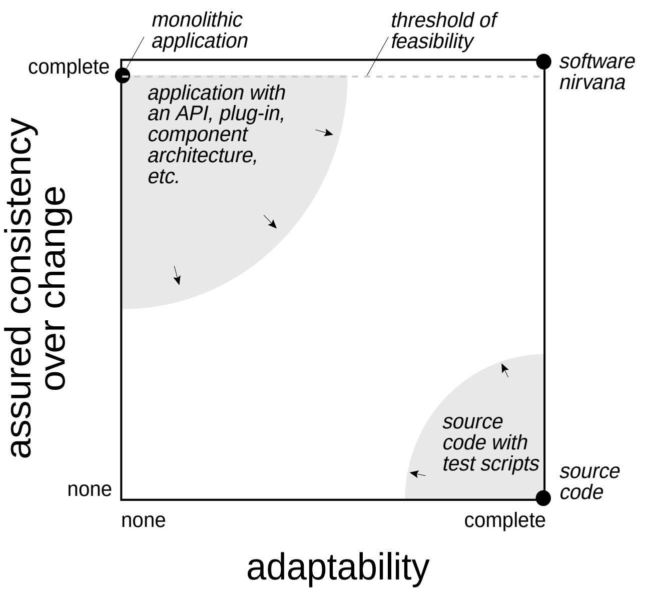

monolithic application
threshold of feasibility
complete
software nirvana
application with an API, plug-in, component architecture, etc.
assured consistency over change
source code with test scripts
none
source code
none
complete
adaptability

Figure 4-1. An abstract depiction of the trade-off between adaptability and consistency offered by current DSE techniques.

understanding the source code well enough to make changes coupled with verifying the correctness of those changes poses significant hurdles, especially for large, intricate software systems. While any change is possible, few if any assurances are provided over those changes. In response, various strategies have been devised that help raise confidence in source code changes, and hence an application's consistency. These include confining change to certain source files and verifying application invariants using embedded assertion statements or test scripts. The consistency gained using these measures cuts off certain avenues of adaptability.

Monolithic applications, which lack DSE mechanisms altogether, represent the opposite extreme. By prohibiting third-party change, application developers can guarantee a high degree of consistency. Even so, large monolithic applications cannot achieve complete consistency due to the fundamental limits of program analysis (Young & Taylor 1986). Incorporating DSE techniques into these applications, by utilizing APIs or component architectures for example, increases adaptability by permitting third-parties to change certain aspects of the application, but simultaneously introduces the possibility that those changes could violate consistency.

Software nirvana, the mythical "software ideal" in which third-parties can change all aspects of an application without sacrificing its consistency, is represented in the extreme upper-right corner of this space.

The thesis of this dissertation is *not* that this adaptability/consistency trade-off can be eliminated. In truth, any externally perceptible change to a program's behavior could be regarded as an inconsistency. In other words, what one constituent regards as an erroneous change to a program, a second constituent may regard as a desirable feature. This does not mean that consistency is a meaningless or indiscriminate concept, just that notions of consistency are relative to an application domain and must be consciously and explicitly stated. This is especially true in the context of decentralized software evolution where application developers cannot anticipate the imaginative changes that third-parties may make to their applications, including tailoring it to different application domains.

The thesis of this dissertation *is* that:

> *An application's architectural system model can provide a basis for decentralized software evolution that offers greater degrees of adaptability while at the same time supporting more assurances over consistency.*

We claim that it is possible to devise a new DSE technique that offers more assurances over consistency than source-code techniques while simultaneously exposing more adaptability than non-source-code techniques. More precisely, this dissertation demonstrates that the open-architecture software technique can expose all six open point types defined in Section 3.2 without revealing source code, and govern how those open points are used by third-parties. As a result, this new DSE technique offers a unique adaptability/consistency trade-off that is more desirable in some situations than currently available techniques.

4.2 Limitations of Previous Techniques

Whereas Chapter 3 answers the question "what is and is not adaptable and assured by available DSE techniques?", this section answers "why?" and "how come?" Why do certain DSE techniques engender greater degrees of third-party adaptability than others? What limits the degree to which DSE techniques can assure consistency over third-party changes? This section answers these questions.

4.2.1 Limited adaptability

Section 3.2 characterized the adaptability offered by six DSE techniques using the concept of open points. Figure 3-1 on page 45 summarizes their key differences. Here, we examine "why" these DSE techniques fail to support all six open point types. Source code-based techniques are not considered in our analysis since they inherently support all open point types without restrictions.

A simple example elucidates the conspiring factors that limit third-party adaptability. Consider the partial implementation of the calculator application in Figure 4-2. It is comprised of three interacting modules: `calculator`, `stack`, and `display`. In the excerpt shown here, the `calculator` module references several of the exposed interface

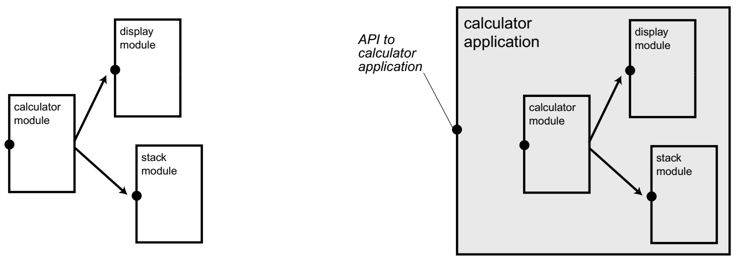


Figure 4-3. On the left, the internal modules of the calculator example. On the right, an API for the calculator application.

functions of the `stack` and `display` modules.[3] The module diagram for the calculator example is shown in Figure 4-3 (left).

One factor limiting third-party adaptability stems from not exposing the host application's internal module structure to third-parties. We call this *opaque structure*. Consider the result of using the API technique on the calculator application. No matter which or how many interface functions get exposed to third-parties—assume all functions in all modules are exposed as in Figure 4-3 (right)—third-parties will perceive the application as a

```
module calculator;

import module stack,
display;

public function add() {
  ...
  result = stack.pop() +
           stack.pop();
  ...
  display.print(result);
}
```

```
module stack;

public function pop()
returns integer {
  ...
}

public function
push() returns
integer {
  ...
}
```

```
module display;

public function
print(String s) {
  ...
}

public function
print(int i) {
  ...
}
```

Figure 4-2. The partial implementation of a calculator application comprised of three modules, a calculator module (left), a stack module (middle), and a display module (right). Here, the calculator module explicitly references interface functions in the stack and display modules.

3. In this example, the calculator module explicitly references the stack module's `pop` function using the construct `stack.pop()` as is common in programming language like Java (Gosling et al. 1996). In programming languages such as C (Kernighan and Ritchie 1988), references are implicit. In such cases, the `calculator` module would reference a function named `pop` (eliding the reference to stack), and a program linker would resolve this under-specified reference to an actual function by scanning the exposed interfaces of other modules for a function implementation named `pop`; multiple matching functions would result in a linkage error.

monolithic entity, whose internal module structure is opaque and impervious to change[4]. This acts as a barrier to change since third-parties cannot modify those modules (or the bindings among them). To understand why this is so, let us compare identical changes to an application, one performed on its source code form, the other performed on its compiled, monolithic form.

In the program's source code form, replacing the `stack` module with another module is a simple matter of text editing and recompilation. We can either alter the `stack` module's definition or replace the `calculator` module's references to the `stack` module with another (named) module. In the program's compiled form, making a similar change is arduous. Consider just some of the potential impediments:

- *difficult to locate behavior*: In the program's compiled machine processable form, symbolic, human comprehensible function names such as `push` and `pop` have been replaced with machine addresses that reference the function's implementation. Without a symbol table relating names to addresses, locating the function references of interest in order to change them is a laborious and error-prone process.
- *replicated behavior*: Certain performance optimizations replicate the implementation of a function or module, for example to specialize a generic or parameterized behavior for an often used data type. This complicates changes since all replicated copies of the behavior must be located and changed together.
- *blurred module boundaries*: Inline function optimizations, which replace a function call with its definition, may have eliminated some of the `calculator` module's

4. The application's deployed form may consist of multiple discrete parts, but third-parties cannot *reliably* change the application by manipulating these parts unless the host application developers have documented how the parts operate and interact (in which case, it is a component architecture). Irrespective of documentation, segmented applications are more amenable to invasive software customization techniques.

references to the `stack` module's functions. Further optimizations on such inlined functions may have intertwined the two implementations such that they can no longer be independently recognized or modified.

- *abridged behavior*: Control-flow optimizations may have detected unused functions and modules and removed them from the compiled form, thus precluding the possibility of redirecting bindings to them.

The compilation process, as a by-product of translating the program's source code description into a machine executable specification, petrifies the malleability inherent to the source code. Compiler directed program transformation aimed at improving the program's runtime efficiency blur module boundaries, further complicating change.

Since host application developers can artificially impose such barriers on their application, i.e., by confining their use of DSE techniques to a subset of their application and deploying the rest as a monolithic entity, no DSE technique is immune to these issues. But this problem is inherent to APIs, scripting languages, and event-based systems since these techniques, by themselves, do not segment the host application.

Plug-ins and component architectures, in contrast, do expose an application's internal module structure, though to different degrees. Plug-ins expose a single module and allow third-parties to provide alternate implementations of that module. Component architectures go further by exposing all modules—a technique that is conceptually akin to making every application module a plug-in. Plug-ins and component architectures achieve this by segmenting the host application along module boundaries. Each segment can then be compiled and distributed separately from the rest. And, if a segment (i.e., module) is adequately documented, third-parties can reliably replace it.[5]

Although segmentation allows plug-ins and component architectures to support module addition and replacement open points, module removal and all three binding open points are either unsupported or their use is severely restricted. This is because module bindings are over-specified and cannot be changed without replacing the module in which the binding originates. Consider just some of impediments to changing bindings in plug-in and component architectures:

- *replicated bindings*: Similar to the replicated behavior problem of modules, this problem occurs if compiler optimizations replicate a module reference, for example to select the most optimal function implementation based on the particular data types passed as parameters in a function reference. This complicates change since all replicated copies of the behavior must be located and changed together.
- *difficult to locate bindings*: Module bindings, which can be readily identified in a module's source code, are difficult to locate once the module has been compiled. This makes locating and changing the binding of interest a difficult, ad hoc process. Some of the invasive customization techniques can locate and modify such bindings. These techniques often rely on subtle characteristics of the implementation technology or eavesdrop on the module as it executes, looking for and replacing bindings of interest before they are de-referenced.
- *over-specified bindings*: Module bindings are often over-specified, directly naming the destination module and its interface element, or worse, naming the destination

5. The particular implementation mechanisms used to segment host applications are beyond the scope of this thesis, but can be readily found as a part of popular operating systems and component object technologies. For example, the Microsoft Windows operating system's Dynamic Link Libraries (DLLs) mechanism allows a program module to be compiled, packaged, and distributed separately from applications that use it. Applications can then use the mechanism to locate, load, and dynamically link to these modules. Component object technologies, such as COM (Brockschmidt 1994) and CORBA (OMG 1999), provide similar services.

module and substituting a memory offset that points to the element's implementation.

All of these factors conspire to diffuse and rigidify module bindings. This ultimately forces third-parties to change module bindings by replacing the modules in which the bindings originate.

Programming language and design approaches provide numerous strategies for re-factoring source code to localize and encapsulate the specification of behaviors and bindings. Ultimately, mechanisms that preserve this locality need to be available to third-parties so that they can make similar changes just as easily. What is needed is a mechanism that enables bindings to change as easily as segmentations enables modules to change.

4.2.2 Fragility

Section 3.3 reviewed the characteristic techniques for gaining assurance over third-party changes. Each technique has its place in the tool chest of DSE, offering a unique trade-off with respect to the errors detected, runtime overhead, etc.

The techniques that attempt to detect inconsistencies (as opposed to preventing them in the first place) rely solely on either the program's source code specification or its compiled, machine executable specification. While these executable specifications can accommodate certain inconsistency checks, they do not favor whole-program analyses such as deadlock detection, schedulability, resource allocation, and dependency checking. The immensity and uniformity of detail in the program's executable specification, coupled with its low-level abstractions, confound whole-program analyses. We refer to this as the *confounded analysis* problem.

To overcome the confounded analysis problem, researchers have developed modeling languages specially suited for whole-program analyses. Examples of such modeling languages include MetaH/ControlH (Binns et al. 1996), Rapide (Luckham and Vera 1995), Darwin (Magee and Kramer 1996), Wright (Allen and Garlan 1997), Instress (Perry 1996), and Armani (Monroe 1998). These and other modeling languages overcome the confounded analysis problem by (a) capturing a subset of the program's overall behavior, (b) using language constructs that are not necessarily executable, and (c) using language constructs that facilitate analysis as opposed to efficient translation into a machine executable form.

Often, these high-level program models are not mapped to corresponding implementation concepts, making it impossible to determine whether a program's implementation adheres to the properties ascribed to its high-level model. We refer to this as the *implementation mapping* problem.

4.2.3 Composition

Add-ons can interference with one another if their behaviors overlap. We refer to this as the *change composition problem*. In the calculator example, consider a scenario in which two add-ons interpose themselves between the `calculator` and `display` modules. The first add-on logs calculator actions to a persistent file and the other add-on magnifies the output text for visually impaired users. Different behaviors emerge depending on the order in which the add-ons are invoked. Consider the behaviors that can emerge using four different composition strategies:

1. magnified text in the persistent log and on the display, which occurs when the magnification feature executes first, then passes its modified (i.e., magnified) input to the logging feature.

2. regular sized text in the persistent log and magnified text on the display, which occurs when the logging feature executes first, then passes its unmodified input text to the enlarging feature.
3. regular sized text in the persistent log and on the display, which occurs when the magnification feature executes first, then passes its modified (i.e., magnified) input text to the logging feature. The logging feature then unknowingly strips the 'text size' attribute from its input because its developers assumed that it would always be of regular size. This combination mysteriously disables the enlarging feature.
4. regular sized text in the persistent log and magnified text on the display, which occurs when the modules run in parallel instead of serially, and only the enlarging feature has its output "wired" to the inputs of the `display` module.

In this example, only one of the four resulting behaviors is clearly undesirable (#3) as one feature unintentionally masks a second. Outcomes #2 and #4 produce the same external outputs, at least for these two particular features. Which is the "correct" outcome? There is no clear best choice—it depends on the needs of an application and its customizations. In light of this, it is surprising that some programming languages and class frameworks encourage (or even require) developers to use the composition policy they provide, e.g., CLOS's call-next-method mechanism (Kiczales et al. 1991), BETA's inner method construct (Malhotra 1994), AP's traversal algorithm (Lieberherr 1996) and CORBA's interceptors (OMG 1999).

4.3 Protean Software

The ultimate goal of software engineering is the systematic, principled design of systems that fulfill the original promise of *soft*ware: protean applications that are just as easy to modify once implemented as they are on the drawing board. Decentralized software evolution extends this tradition to third-parties, empowering them to modify applications in-the-field and independently of their original developers.

Much progress remains to be made in the area of decentralized software evolution. Currently available off-the-shelf software applications fall far short of the goal. Many commercial software packages are still distributed as opaque, monolithic programs that lack sanctioned mechanisms for third-party change. Nearly all commercial applications that do incorporate DSE mechanisms confine change to a small subset of the application's behavior. Most freeware and shareware applications rely solely on source-code DSE techniques, and are plagued by its limitations.

Substantial benefits will accrue if these problems are remedied by. For example, broadening the scope of change gives third-parties more leverage to reuse and adapt existing software as opposed to custom building a solution from scratch. Offering greater assurances over change raises the level of confidence in the resulting application. These efficiencies ultimately benefit society by:

- shortening development cycles, since more opportunities for reuse can lead to greater productivity as compared to custom development;
- reducing application development costs, since more opportunities for reuse can lead to greater productivity and hence less development cost as compared to custom development; and
- increasing production quality, since more opportunities for reuse can lead to products tailored to user needs.

CHAPTER 5 Open-architecture Software

Every problem in computer science can be solved by adding a layer of indirection.
– unknown

This chapter describes the Open-architecture Software approach to decentralized software evolution. Unlike previous techniques, open architectures support all six open point types at the module-level without revealing source code and offer new opportunities for consistency checking. Chapters 6 and 7 describe three case studies in which the approach has been used. Chapter 8 evaluates the approach.

5.1 Description of the Approach

The open-architecture software approach consists of three integral and interacting elements:

1. An *architectural system model* that abstractly represents the application.
2. An *implementation* that independent third-parties evolve by altering the application's architectural system model.
3. An *architectural evolution manager* that validates changes to an application's architectural sys-

tem model and makes corresponding changes to its implementation.

An application's architectural system model is causally connected to its implementation—changes to one affect the other. This relationship is maintained by the architecture evolution manager. In the parlance of software reflection, this approach would be characterized as *declaratively reflective* (Maes 1987) since the implementation is changed indirectly using a declarative representation, in this case, an architectural system model.

Figure 5-1 abstractly depicts these elements in relation to one another. The following subsections describe each of these elements in further detail.

5.1.1 Architectural system model

A system model describes an abstracted, high-level representation of an application's implementation. System models typically capture an application's implementation as a collection of *components,* each of which represents a unit of functional behavior, and a *configuration*, which represents the binding relationships among components. In most instances, components correspond to a collection of program modules, and the configuration corresponds to the control and data dependencies among modules.

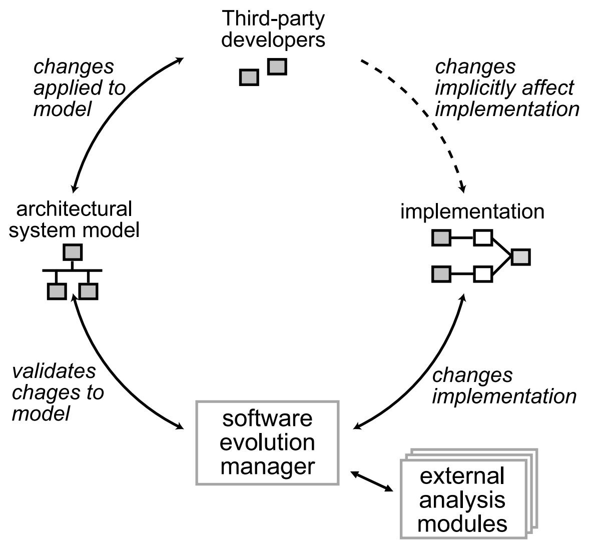

Third-party developers
changes applied to model
changes implicitly affect implementation
architectural system model
implementation
validates chages to model
changes implementation
software evolution manager
external analysis modules

Figure 5-1. An abstract depiction of the open-architecture software approach.

An architectural system model enriches this representation by recognizing interaction as a concept distinct from functional behavior (Perry and Wolf 1992). An interaction between two or more components may range from a simple procedure call to a complex protocol for organizing nodes in an ad hoc distributed network. Traditionally, every component describes a portion of the interactions it participates in—the portion it is responsible for. As such, each interaction's implementation is intertwined with the others and with the module's functional behavior. This has the effect of diffusing an interaction's description, making it difficult to understand, reason about, and change. This frustrates third-party changes to component interactions. By representing interactions as distinct elements called *connectors*, an architectural system model efficiently circumvents these problems.

An architectural system model can also capture and associate arbitrary data with its elements, called *annotations*. Below, we elaborate upon each element of an architectural system model—components, connectors, a configuration, and annotations.

One novel aspect of the open-architecture approach is that it regards the architectural system model as an integral part of an application. The approach deploys this model with the application's implementation, exposes the model to third-party developers, and permits third-parties to evolve the application's implementation by changing the model. Permissible changes to the model effect corresponding changes to the application's implementation.

Components. A component represents a unit of software, a locus of computation and state (Shaw and Garlan 1996, Shaw et al. 1995). A component exposes one or more interfaces, each of which consists of a collection of functions, types, and variables. A component actuates another component's behavior solely through its exposed interface.

A component's specification is also *context neutral:* it does not reference other components or connectors (explicitly or implicitly) and is independent of the application's configuration. This permits binding decisions among components and connectors to change independently of component models. A configuration specifies the actual bindings among components and connectors (see below).

Connectors. A connector represents a unit of interaction, the "glue" that holds the system together (Perry and Wolf 1992). Traditionally, a component has intertwined its functional and interaction specifications. A connector extricates interaction specifications from components and models them as distinct elements. This permits the specifications to change independently of one another (Perry and Wolf 1992, Purtilo 1994).

A connector's specification is also *context neutral,* thereby permitting binding decisions to change independently of connector models.

Configuration. A configuration describes the bindings between an application's components and connectors. As such, it is used as instructions for assembling the application out of its component and connector parts.

Annotations. An annotation describes arbitrary information related to a component, connector, or configuration. Annotations are a form of meta-data, i.e., data about data. Annotations make system models more useful. For example, an annotation that links the elements of a system's implementation to its associated design documentation can help third-parties better understand the system. Annotations that describe system invariants can be used to prevent erroneous system configurations.

5.1.2 Implementation

An implementation refers to an executable form of an application's functional behavior. The particular form and representation of an implementation is inconsequential—it can be specified in any compiled or interpretable language, implement an entire system or a small part of one, be packaged as a single binary image or as many small parts, etc.

An application's implementation reflects its architectural system model—if one changes in a valid way, the other is changed in a corresponding manner. In order to achieve this, a *mapping* between these two forms is necessary. We assume a one-to-one mapping between the elements in an application's architectural system model and the elements in its implementation. This is done for three reasons. First, it greatly simplifies the task of mapping between the two, a task which the architectural evolution manager must perform. Second, it permits us to clearly and concisely assign responsibilities to implementation elements. Third, a simple, intuitive mapping is easier for people to understand than a complex one. Although more complex and flexible mappings could be devised, their benefits must be weighed against the cognitive burden they place on people. Third-parties "see," comprehend, reason about, and change the application's implementation through its architectural system model. As such, a mapping should strive for intuitiveness and fidelity.

Below, we elaborate upon each element of an implementation—components, connectors, a configuration, and annotations.

Implementation components. A component implements an encapsulated unit of computation and state of arbitrary complexity. Its internal structure is regarded as a black box. A component's implementation is also *context neutral:* it has no preconceptions regarding its bindings to other implementation elements and it does not establish bindings (implicit or explicit) on its own volition. This permits binding decisions to change independently of component implementations and permits the architectural system model's configuration to determine all binding decisions.

A component is explicit and distinct in the implementation. It is packaged in a form that allows the underlying environment to add, replace, and remove it from/to a deployed application. Popular operating systems have provided such services for many years. For example, the Dynamic Link Library mechanism of Microsoft's Windows operating systems provides a programming language-independent module package and a late-binding mechanism for loading and linking components into an application during runtime (Petzold 1996). Most popular implementations of the Unix operating system provide a similar dynamic library mechanism. Component object models and technologies, such as CORBA (OMG 1999), Microsoft's COM (Brockschmidt 1994), and JavaBeans (Chan and Lee 1997), can also be used. These technologies augment dynamic linking facilities with mechanisms that can locate components given an interface description, enumerate the interfaces of a component, perform reference counting, etc.

Implementation connectors. A connector implements an encapsulated unit of interaction of arbitrary complexity. Its internal structure is regarded as a black box. A connector's implementation, like a component, is *context neutral* and provides a mechanism for adding, redirecting, and removing bindings.

Traditionally, connectors have been implemented as indiscrete entities, e.g., UniCon reifies connectors as linker instructions (Shaw et al. 1995). This, in effect, entangles interactions with a components functional behaviors, making them difficult to change.

In the open-architecture approach, a connector is a discrete, explicit, reconfigurable entity in the implementation. The practical utility of explicit connectors in the implementation has put them to a myriad of uses, including specifying communication mechanisms independent of functional behaviors and thereby enabling components written in different programming languages and executing on different processors to transparently interoperate (Purtilo 1994); visualizing and debugging system behavior by monitoring messages between components (Purtilo 1989); and integrating tools by using a connector to broadcast messages between them (Reiss 1990).

Implementation configuration. A configuration encapsulates the binding relationships among components and connectors in an application.

Implementation annotations. Annotations associate arbitrary data with components, connectors, or a configuration. Annotations commonly describe mappings between an architectural system model and an implementation, and application invariants that help analysis tools determine whether or not a particular change to the architectural system model is valid. The architecture evolution manager ensures that the two descriptions correspond.

5.1.3 Architecture Evolution Manager

The architectural evolution manager is an agent with two responsibilities:

1. it validates all changes to an application's architectural system model; and
2. it changes an application's implementation to reflect its architectural system model.

A third-party changes the architectural system model by adding, replacing, and removing components and connectors, altering its configuration, and altering any annotations on these elements.

To support task #1, the architectural evolution manager uses a combination of analysis tools external to the approach and annotations on the architectural system model. If a change is judged valid after analysis, the architectural evolution manager makes the corresponding changes to the application's implementation. Otherwise, some appropriate reaction is initiated, such as denying the change or notifying a responsible party.

To support task #2, the architectural evolution manager uses the model-to-implementation mapping to translate changes specified in terms of the architectural model to corresponding changes in the implementation.

5.2 Evaluation

As a consequence of exposing an explicit architectural model for the application and allowing third-parties to change the implementation by changing the model, open-architectures support all six open point types. As before, a filled circle (●) indicates full support for the open point type, an unfilled circle (○) indicates partial or restricted support for an open point type, and an empty cell indicates that the open point type is unsupported.

DSE technique	add module	replace module	remove module	add binding	redirect binding	remove binding	internal structure mutable	internal structure apparent
open-architecture	●	●	●	●	●	●	●	●

CHAPTER 6

Case Study: KLAX and Cargo Router

This chapter describes two case studies that used the open-architecture approach on two small applications: KLAX, a clone of the game by Atari, and Cargo Router, a simple logistics planning application. The case study in the next chapter describes the use of the approach on a complex, real-world, commercial software application. Chapter 8 validates and evaluates the open-architecture approach based, in part, on these case studies.

This chapter begins by describing the systems and tools used by the KLAX and Cargo Router applications. We then describe each application, its architecture, and the third-party changes that were made to it.

6.1 Implementation Apparatus

Both KLAX and Cargo Router are implemented in the C2 architectural style with the help of an object-oriented class framework. The C2-style and the class framework are described in Section 6.1.1 and 6.1.2, respectively.

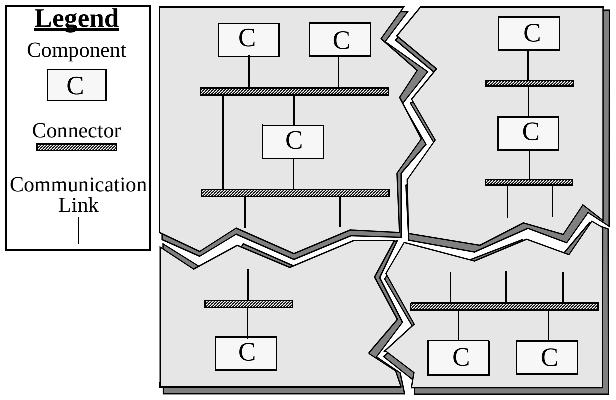


Figure 6-1. An abstract C2 architecture. Jagged lines represent portions of the architecture not shown.

6.1.1 C2 architectural style

The C2 architectural style[6] (Taylor et al. 1996) can be informally summarized as a network of concurrent components bound together by connectors, i.e., message routing devices, in accordance with a set of style rules. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. No direct component-to-component links are allowed. A connector may be connected to an unbounded number of other components and connectors. When two connectors are attached to one another, it must be from the bottom of one to the top of the other (see Figure 6-1).

Components implement application behavior and may encapsulate functionality of arbitrary complexity, maintain state information, and utilize multiple threads of control. The style does not place restrictions on the implementation language or granularity of the components. It does require that all component communication occur by asynchronous message exchange through connectors[7]. Furthermore, components cannot assume that they

6. The description of the C2 architectural style presented here is summarized from (Taylor et al. 1996)—a more detailed description of the style and its benefits can be found therein.

will execute in the same address space as other components or share a common thread of control.

Central to the architectural style is a principle of limited visibility or *substrate independence*: a component within the hierarchy can only be aware of components “above” it and is completely unaware of components which reside at the same level or “beneath” it. Notions of above and below are used to support an intuitive understanding of the architectural style. A component explicitly utilizes the services of components “above” it by sending a *request* message. Communication with components below occurs implicitly; whenever a component changes its internal state, it announces the change by emitting a *notification* message, which describes the state change, to the connector below it. Connectors broadcast notification messages to every component and connector connected on its bottom side. Thus, notification messages provide an implicit invocation mechanism, allowing several components to react to a single component’s state change.

6.1.2 C2 class framework

We have developed an extensible class framework to facilitate the implementation of C2-style applications (Taylor et al. 1996, Medvidovic et al. 1997). The framework provides abstract classes for C2 concepts such as components, connectors, and messages, and implements default behavior for interconnecting components and connectors, message passing, and component initialization and termination (see Figure 6-2). Application components (and connectors) subclass from the appropriate framework classes and override the default

7. While the style does not forbid synchronous communication, the responsibility for implementing synchronous message passing resides with individual components. Ideally, the most common synchronous communication patterns would be implemented in the C2 class framework and reused across applications.

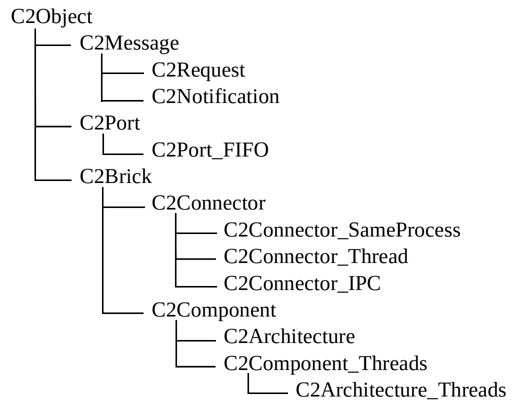


Figure 6-2. C2/Java class framework for building C2-style architecture.

behavior as needed. Like other frameworks, the C2/Java framework eliminates most repetitive programming tasks associated with developing C2-style applications, and allows developers to focus on application-specific concerns. The class framework has also been implemented for C++ and partially for Ada.

6.1.3 ArchStudio 2.0 environment

ArchStudio 2.0 is an extensible, integrated environment for architecture-based software development. It comprises a number of tools, including:

- a graphical design environment called Argo (Robbins and Redmiles 1998)
- support for runtime evolution (Oreizy and Taylor 1998)
- static analysis and code generation (Medvidovic et al. 1999)
- hypertext linking to external artifacts using Chimera (Anderson et al. 2000)
- a Web browser for downloading new software components
- a shared XML-based data store (Khare et al. 2000)

ArchStudio was implemented in collaboration with Nenad Medvidovic, Jason Robbins, Rohit Khare, Michael Guntersdorfer, Kari Nies, Eric Dashofy, and Yuzo Kanomata.

The tools comprising ArchStudio 2.0 are implemented in the Java programming language, and can modify C2-style applications written with the C2/Java class framework. Figure 6-3 depicts a conceptual view of ArchStudio 2.0's architecture. The lightly shaded tools in the figure represent integrations intended to assess feasibility; they have not yet been added to the production environment. The more heavily shaded tools are part of the current environment.

All the tools in the environment use a shared model of the application-under-design that is maintained by the *ArchADT* component. ArchStudio 2.0's tools query and modify the model by sending C2 request messages to *ArchADT*. If a tool's request changes the model (e.g., adds a new component), *ArchADT* emits a C2 notification describing the change to the connector below it, which, in turn, broadcasts it to the tools attached below it. Each tool receives and reacts to the state change independently; typical reactions include making a

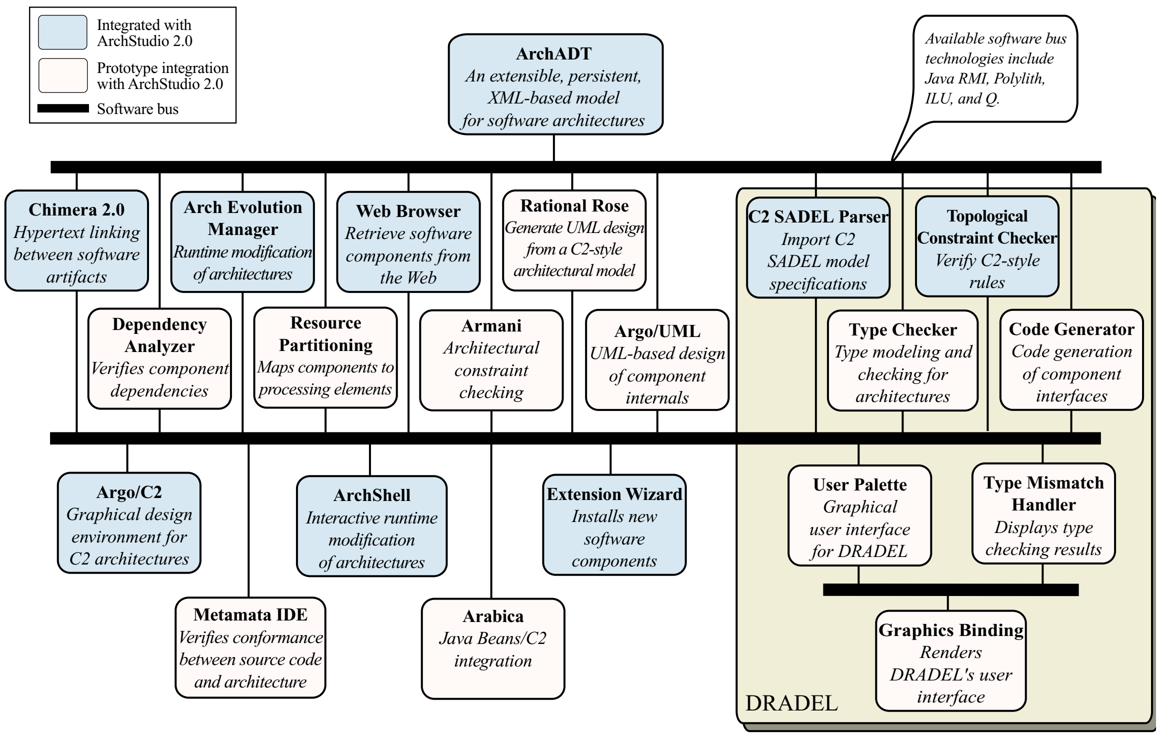

Integrated with ArchStudio 2.0
Prototype integration with ArchStudio 2.0
Software bus
ArchADT An extensible, persistent, XML-based model for software architectures
Available software bus technologies include Java RMI, Polylith, ILU, and Q.
Chimera 2.0 Hypertext linking between software artifacts
Arch Evolution Manager Runtime modification of architectures
Web Browser Retrieve software components from the Web
Rational Rose Generate UML design from a C2-style architectural model
C2 SADEL Parser Import C2 SADEL model specifications
Topological Constraint Checker Verify C2-style rules
Dependency Analyzer Verifies component dependencies
Resource Partitioning Maps components to processing elements
Armani Architectural constraint checking
Argo/UML UML-based design of component internals
Type Checker Type modeling and checking for architectures
Code Generator Code generation of component interfaces
Argo/C2 Graphical design environment for C2 architectures
ArchShell Interactive runtime modification of architectures
Extension Wizard Installs new software components
User Palette Graphical user interface for DRADEL
Type Mismatch Handler Displays type checking results
Metamata IDE Verifies conformance between source code and architecture
Arabica Java Beans/C2 integration
Graphics Binding Renders DRADEL's user interface
DRADEL

Figure 6-3. Architecture of the ArchStudio 2.0 development environment in the C2-style.

corresponding change to its internal representation, updating affected graphical views, or ignoring state changes that fall outside its domain of interest.

The primary tools used to support decentralized software evolution are described in the following subsections.

6.1.4 ArchADT

The *ArchADT* component encapsulates the architectural system model of the system-under-design. Its implementation represents the model as an abstract data type (ADT) whose public interface is shown in Figure 6-4. The ADT provides operations for querying and modifying the application's architectural system model as well as storing a representation of it in the file system. Additionally, the model may be extended by adding new attributes to existing elements or by adding new sub-hierarchies of elements. The model is stored in XML (Harold 1999), a hierarchical structured ASCII format.

ArchADT represents all four basic elements of the architectural system model: components and connectors (lines 24-29 in Figure 6-4), a configuration (lines 45-54), and annotations (lines 31-43). ArchADT augments the basic model by associating type information (lines 9-14) and interface specifications (lines 16-22) with each component and connector.

6.1.5 Architecture Evolution Manager

Architecture Evolution Manager (or AEM) maintains the correspondence between the architectural system model and the implementation. Attempts to modify the architectural model invoke the AEM, which determines if the modification is valid. The AEM may utilize external analysis tools, such as an architectural constraint mechanism, to determine if a

```
public interface IArchADT {
    // category: architecture
    public void defineArchitecture(String archName);
    public void setArchProperty(String archName, String name, Object value);
    public Object getArchProperty(String archName, String name);
    public Vector enumArchProperties(String archName);
    public Vector enumArchitectures();

    // category: component/connector types
    public void defineType(String archName, String type, String name);
    public boolean isType(String archName, String type, String name);
    public Vector enumTypes(String archName, String type);
    public String getTypeKind(String archName, String typeName);
    public void removeType(String archName, String type, String name);

    // category: interface elements
    public void addMethod(String archName, String name, String portName,
                          String returnType, String msgName, Vector paramList);
    public Vector enumMethods(String archName, String name);
    public Vector enumMethodsAtPort(String archName, String name, String portName);
    public void removeMethod(String archName, String name, String portName,
                             String returnType, String msgName, Vector paramList);

    // category: component/connector instances
    public void addInstance(String archName, String typeName, String name);
    public boolean isInstance(String archName, String type, String name);
    public String getInstanceKind(String archName, String name);
    public Vector enumInstances(String archName, String type);
    public void removeInstance(String archName, String name);

    // category: properties (types and instances)
    public void addProperty(String archName, String type, String name,
                            StringpropName, IPropertyValue value);
    public void removeProperty(String archName, String type, String name,
                               String propName, IPropertyValue value);
    public Vector enumProperties(String archName, String type,
                                 String name, String propName);
    public Vector enumProperties(String archName, String type, String name);

    // category: attributes
    public void setAttribute(String archName, String type, String name,
                             String attrName, String attrValue);
    public String getAttribute(String archName,String type,String name,String attrName)
    public Vector enumAttributes(String archName, String type, String name);

    // category: configuration
    public void weld(String archName, String fromInstance, String fromPort,
                     String toInstance, String toPort);
    public void unweld(String archName, String fromInstance, String fromPort,
                       String toInstance, String toPort);
    public boolean areWelded(String archName, String fromInstance, String fromPort,
                             String toInstance, String toPort);
    public Vector areWelded(String archName, String fromInstance,String toInstance);
    public Vector enumWelds(String archName);
    public Vector enumWeldsAtPort(String archName, String instanceName, String portName

    // category: file I/O
    public void load(String filename);
    public void save(String filename);
}
```

Figure 6-4. The public interface for ArchStudio's ArchADT in Java. Tools in the ArchStudio environment use this interface to manipulate the software architecture for the system-under-design. Third-party developers use the same interface for evolving the application in-the-field. Java exception clauses have been elided for brevity.

particular change is acceptable. The current implementation of the AEM uses implicit knowledge of C2-style rules to constrain changes. If a change violates the C2-style rules, the AEM rejects the change. Otherwise, the architectural model is altered and its implementation mapping is used to make the corresponding modification to the implementation.

6.1.6 ArchShell

ArchShell (Oreizy 1996) provides a textual, command-driven interface for describing and modifying an application's architectural model, even as it is running. Commands are provided for adding and removing components and connectors, reconfiguring the architecture, and displaying a textual representation of the architecture. Architects can also send arbitrary C2 messages to any component or connector, which facilitates debugging.

6.1.7 Extension Wizard

The Extension Wizard is a simple end-user tool for installing and removing software add-ons that is also deployed as a part of the host application. End-users use a Web browser to display a list of downloadable software add-ons, e.g., provided by the software vendor on their Web site. When the user selects the file representing the add-on, the Web browser downloads the file and invokes the Extension Wizard. The software add-on file is a compressed archive containing new implementation modules and an installation script. The Extension Wizard uncompresses the file, locates the installation script it contains, and executes it. The software add-on's installation script may query and modify the architectural model as necessary. As the installation script queries and alters the architectural model, the AEM ensures that application invariants are preserved. If the installation script violates any application invariants, the AEM prevents the change and throws an exception to the installation script. If the installation succeeds, the Extension Wizard notifies the end-user and provides an option to un-install the add-on. A similar approach for deploying system updates is used by Hall et al. (1999)

6.1.8 Armani

Armani (Monroe 1998) is a language and tool set for specifying and checking constraints over a system's architectural topology. ArchStudio's integration with Armani enables architects to specify constraints over how an application's topology may be changed. If any tool (e.g., ArchShell) or third-party add-on installation script modifies the architectural system model in a way that violates a constraint, Armani detects a constraint violation and undoes the change.

6.1.9 Other tools

Argo/C2. Argo/C2 (Robbins et al. 1996) is an interactive, graphical design environment for software architectures. Architects construct a software system by dragging-and-dropping components and connectors from a palette onto a design canvas. Argo/C2s critics continuously examine the system under design and non-intrusively suggest design alternatives and identify errors.

DRADEL. The DRADEL tools supports analysis and code generation capabilities for C2-style architectures using a heterogeneous type system for software architectures (Medvidovic et al. 1999). This allows dradel to establish (the degree of) behavioral conformance among interacting components and among different versions of a single component.

Web browser. An off-the-shelf Web browser, e.g., Netscape Communicator or Internet Explorer, allows architects to browse the Web for new components. Clicking on a C2 component (identified by a unique MIME type and file extension) causes the browser to execute an external program to emit a C2 notification message announcing a successful download of a new C2 component.

Resource Partitioning. One of *Argo/C2's* graphical views allows architects to assign system components and connectors to operating system processes and machine hosts. The *Resource Partitioning* tool retrieves these attributes and generates initialization and startup code for executing the system in the specified configuration.

Dependency Analyzer. The *Dependency Analyzer* tool examines the interface of every component in the system (consisting of the messages understood and potentially emitted by the component) and the architectural topology to reveal dependencies between components. This information helps architects evaluate how components are used and the consequences of adding, removing, replacing and reconnecting components.

Argo/UML. *Argo/UML* (Robbins and Redmiles 1999) is a design environment for UML, much like *Rational Rose*. Our prototype integration allows architects to diagram a component's internal design, initializing the diagram with a component's interface specification. Since *Argo/UML* was developed independently of ArchStudio, it stores its diagrams in individual files, not within *ArchADT*. To relate the UML diagrams with the architectural model, our integration stores the file name of each component's UML diagram as an annotation on the component's model within *ArchADT*.

Arabica. *Arabica* (Natarajan and Rosenblum 1998) provides interoperability between Sun Microsystems' JavaBean components (Chan and Lee 1997) and C2 components by automatically translating JavaBean events into C2 messages, and visa versa. Additionally, *Arabica* ensures C2's topological rules among connected JavaBean components.

Metamata IDE. After generating template code for a component's interface using DRADEL's *Code Generator* (Medvidovic et al. 1999), developers can edit the code using *Metamata Inc.'s Audit* (www.metamata.com). Using Metamata's style checking feature, our integration detects source code changes that diverge from the component's interface and notifies the architect. This assures a certain degree of fidelity between the architectural model and its implementation.

6.2 Case Study: KLAX

6.2.1 Host application

Figure 6-5 illustrates the C2-style architecture for the KLAX game application. The user interface and the game rules are illustrated in Figure 6-6. The connectors, $Connector_{1..6}$, are responsible for routing messages between components. For our example, we assume that the connectors use an inter-process communication mechanism.

The components that make up the KLAX game can be divided into three logical groups. The *game state* components are at the top of the architecture. These components receive no notifications, but respond to requests and emit notifications of internal state changes. The *game logic* components request changes of game state in accordance with game rules and interpret the resulting notifications to determine the state of the game in progress. The *artists* also receive notifications of game state changes, causing them to update their depictions. Each artist maintains a set of abstract graphical objects which, when modified, send state change notifications in hope that lower-level graphics components (in this case, *GraphicsBinding*) will render them. *GraphicsBinding*, in turn, translates user events, such as a key press, into requests to the artist components.

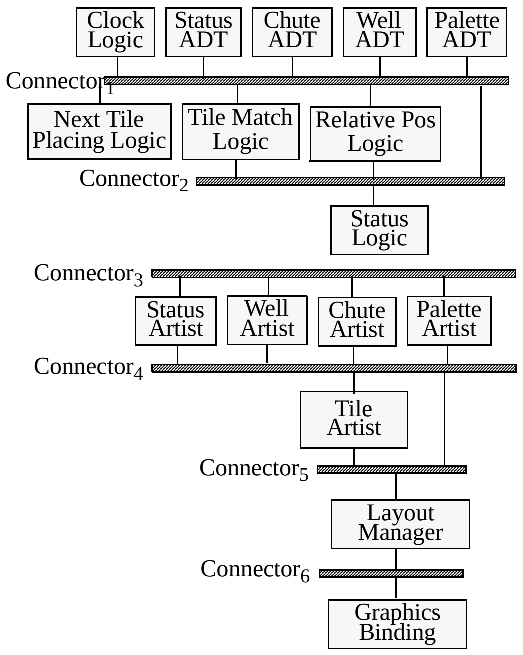


Figure 6-5. KLAX's C2-style architecture.

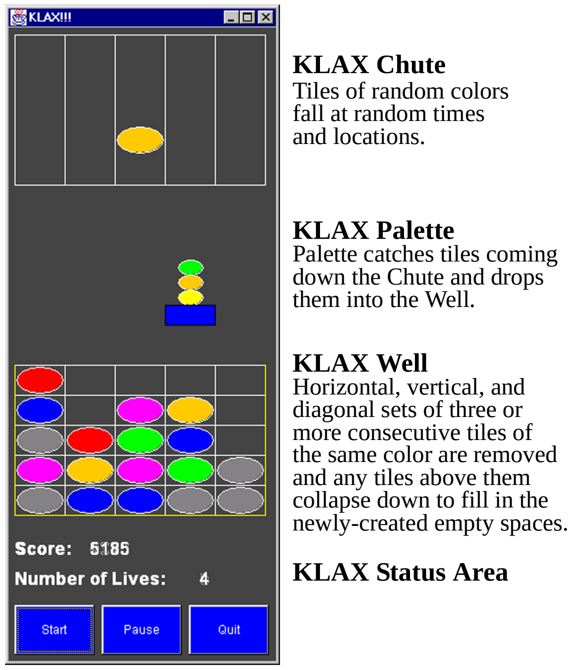

KLAX!!!
Score: 5185
Number of Lives: 4
Start
Pause
Quit
KLAX Chute
Tiles of random colors fall at random times and locations.
KLAX Palette
Palette catches tiles coming down the Chute and drops them into the Well.
KLAX Well
Horizontal, vertical, and diagonal sets of three or more consecutive tiles of the same color are removed and any tiles above them collapse down to fill in the newly-created empty spaces.
KLAX Status Area

Figure 6-6. KLAX's user interface and game rules.

6.2.2 Third-party changes demonstrated

6.2.2.1 Spelling KLAX

The team of graduate students that implemented the original version of the KLAX game using the C2 class framework for C++ also implemented a variation of the game called Spelling KLAX. This game variation involved replacing the original *Tile Matching*, *Next Tile Placing Logic*, and *Tile Artist* components with components that instead matched, placed, and displayed letters, respectively. This transformed the objective of the game from matching the colors of tiles to spelling words. Each time a word was spelled correctly, it would be removed from the well. The spelling logic component wrapped an existing spell-checker, written in C. Figure 6-7 depicts the architecture for Spelling KLAX and Figure 6-8 shows the updated game interface with letters instead of tiles.

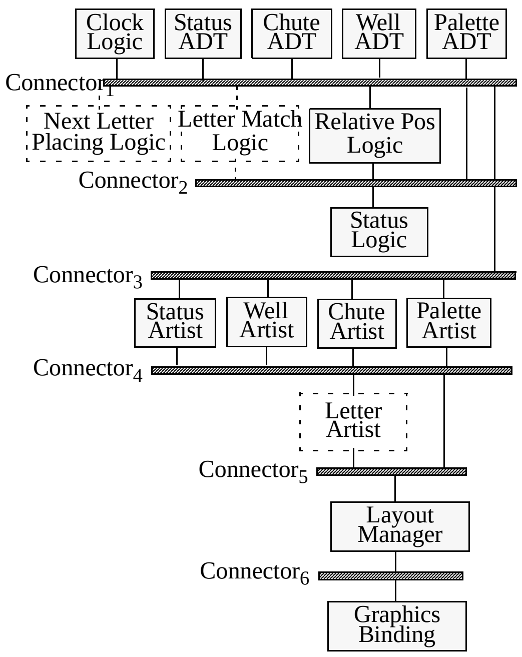


Figure 6-7. Spelling KLAX's C2-style architecture, highlighting the three components that were replaced from the original.

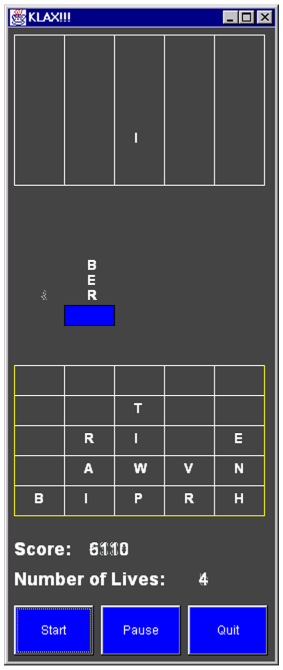


Figure 6-8. Spelling KLAX's user interface.

6.2.2.2 High score list

A team of four undergraduate students from a UC Irvine project class in software design implemented a high score list feature for KLAX. These students had access to one of the two developers of the KLAX application. The students also had access to KLAX's source code and architecture diagram. Their implementation of the high score list feature does not modify any of KLAX's pre-existing components. Figure 6-9 depicts KLAX's architecture after the addition of the high score list feature and Figure 6-10 depicts its user interface.

The high score list feature adds three new components to KLAX's architecture: a *High Score ADT* component that maintains a persistent list of the top ten player scores and names, a *High Score Logic* component that decides when the high score list needs to be changed, and a *High Score Artist* component that provides a user interface for displaying the list. The interaction between these new components and KLAX's other components are minimal. The *High Score Logic* component waits until the *Status ADT* emits an *endOfGame()* notification, at which time it queries the *Status ADT* for the final game score and the *High Score ADT* for the list of high scores. If the final game score is greater than the lowest score in the high score list, it sends a *setNewHighScore(finalScore)* request to *High Score ADT*, causing the ADT to update its internal state and broadcast a state change notification on *Connector$_1$*. *High Score Artist* receives this notification (since the notification is also broadcast on *Connector$_3$*) and responds by creating a dialog box for retrieving the players name and subsequently sending a *setPlayerName(name)* request to the *High Score ADT*.

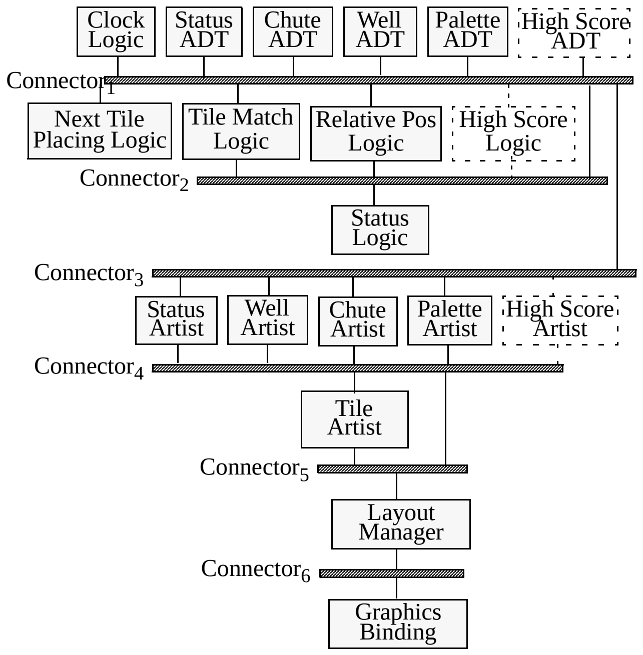
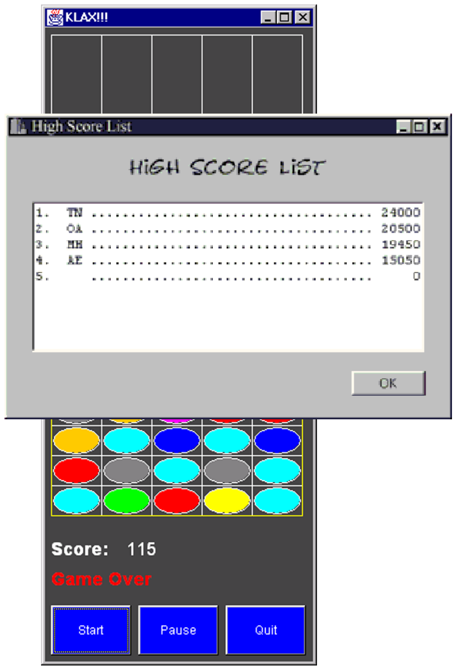


Figure 6-9. KLAX's high score list architecture.



HiGH SCORE LiST

```
1.  TN ...................................... 24000
2.  OA ...................................... 20500
3.  MH ...................................... 19450
4.  AE ...................................... 15050
5.     ......................................     0
```

Score: 115

Game Over

Figure 6-10. KLAX's high score list user interface.

6.2.2.3 Multi-player support

The team of undergraduate students that developed the high score list feature also implemented multi-player support for KLAX. In this variation of the game, several players of the game compete to score points. A separate display reveals the scores of the other contestants, and each time a player makes a match in the well, he gains a tile that he can "throw" onto another player's chute (see Figure 6-12). The architecture for each game client

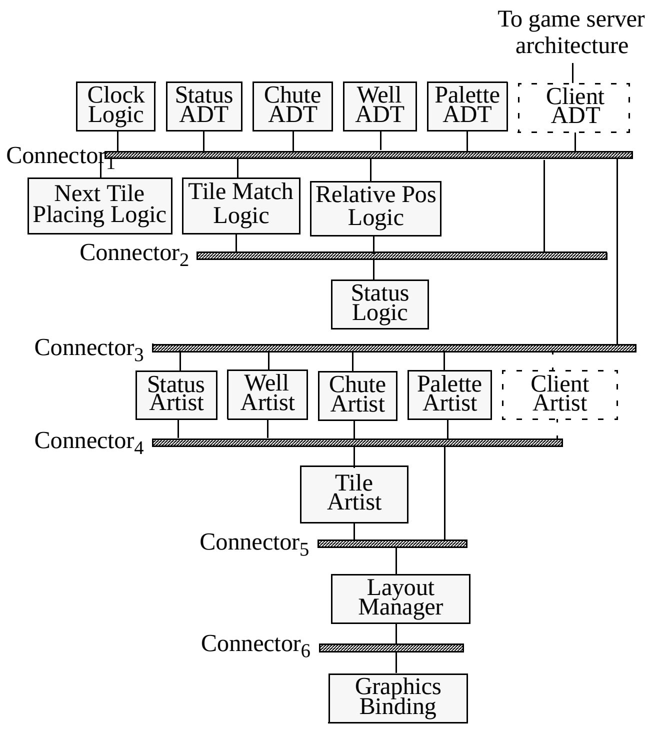
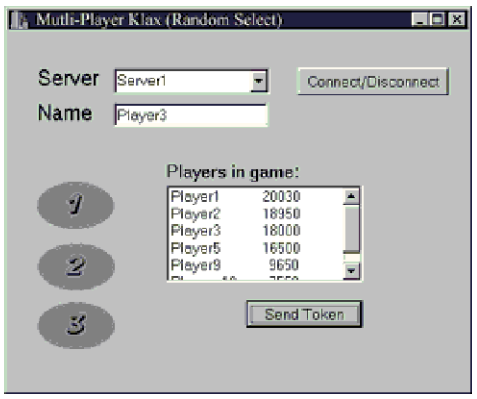


Figure 6-11. Multi-player KLAX's architecture for each game player.



Server: Server1

Name: Player3

Players in game:

```
Player1    20030
Player2    18950
Player3    18000
Player5    16500
Player9     9650
```

Figure 6-12. Multi-player KLAX user interface for each game player.

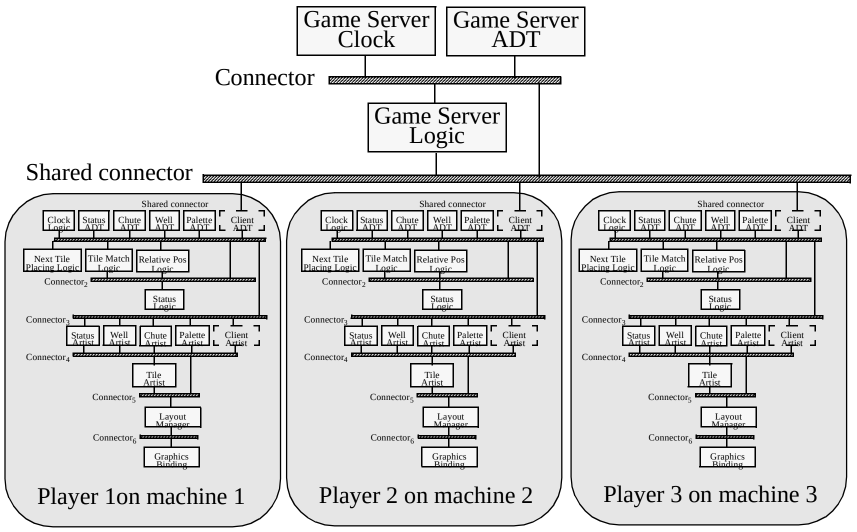


Figure 6-13. KLAX's architecture for the game server with 3 connected clients. All client architectures share the connector labeled "Shared connector".

is depicted in Figure 6-11, and the architecture for the game server with three connected clients in depicted in Figure 6-13.

Each participant executes a separate instance of the game. All instances communicate with a single shared game server component via a shared connector in each of their architectures. This connector can span multiple machines and uses Java/RMI (Chan and Lee 1997), an inter-process communication mechanism provided with the Java runtime environment.

6.3 Case Study: Cargo Router

6.3.1 Host application

The Cargo Routing application is a simple logistics system for routing incoming cargo to a set of destination warehouses. The Cargo Router's C2-style architecture is depicted in Figure 6-14 and its user interface is depicted in Figure 6-15. The three list boxes on the top represent three incoming cargo delivery ports, in this case two airport runways and a train station. When cargo arrives at a port, an item is added to the port's list box. The system keeps track of each shipment's content, weight, and the amount of time it has been sitting idle at the port. The text box in the center displays available vehicles for transporting cargo to destination warehouses. The system displays the vehicle's name, maximum speed, and maximum load. The bottom most text box displays a list of destination warehouses. The system displays each warehouse's name, maximum capacity, and currently used capacity. End-users route cargo by selecting an item from a delivery port, an available vehicle, and a destination warehouse, and then clicking the "Route" button.

Figure 6-14 depicts the architecture of the Cargo Routing system. The *Ports*, *Vehicles,* and *Warehouses* components are ADTs which keep track of the state of ports, the

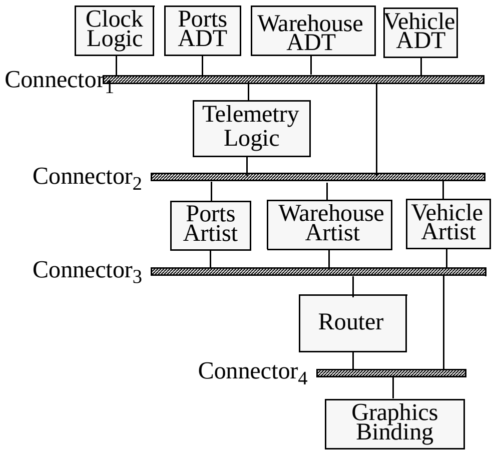
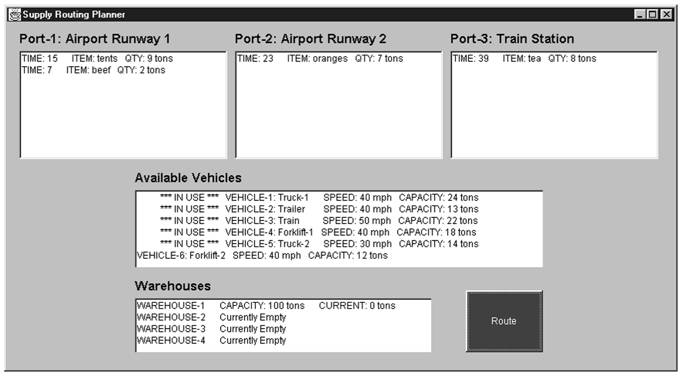


Figure 6-14. Cargo Router's C2 architecture.



Port-1: Airport Runway 1

```
TIME: 15     ITEM: tents   QTY: 9 tons
TIME: 7     ITEM: beef   QTY: 2 tons
```

Port-2: Airport Runway 2

```
TIME: 23     ITEM: oranges   QTY: 7 tons
```

Port-3: Train Station

```
TIME: 39     ITEM: tea   QTY: 8 tons
```

Available Vehicles

```
      *** IN USE ***   VEHICLE-1: Truck-1     SPEED: 40 mph   CAPACITY: 24 tons
      *** IN USE ***   VEHICLE-2: Trailer     SPEED: 40 mph   CAPACITY: 13 tons
      *** IN USE ***   VEHICLE-3: Train       SPEED: 50 mph   CAPACITY: 22 tons
      *** IN USE ***   VEHICLE-4: Forklift-1  SPEED: 40 mph   CAPACITY: 18 tons
      *** IN USE ***   VEHICLE-5: Truck-2     SPEED: 30 mph   CAPACITY: 14 tons
VEHICLE-6: Forklift-2   SPEED: 40 mph   CAPACITY: 12 tons
```

Warehouses

```
WAREHOUSE-1     CAPACITY: 100 tons     CURRENT: 0 tons
WAREHOUSE-2     Currently Empty
WAREHOUSE-3     Currently Empty
WAREHOUSE-4     Currently Empty
```

Figure 6-15. Cargo Router's user interface.

transportation vehicles, and the warehouses, respectively. The *Telemetry* component determines when cargo arrives at a port, and tracks the cargo from the time it is routed until it is delivered to the warehouse. The *Port Artist*, *Vehicle Artist*, and *Warehouse Artist* components are responsible for graphically depicting the state of their respective ADTs to the end-user. The *Router* component sends a message to the telemetry component when the end-user presses the "Route" button and provides the end-user's last selected port, vehicle, and warehouse. The *Graphics* component renders the drawing requests sent from the artists using the Java AWT graphics package.

6.3.2 Third-party changes demonstrated

We now describe how ArchShell and Extension Wizard were used to add new functionality to the application. ArchShell was used to add a new graphical visualization of cargo routes, and an Extension Wizard script was used to add an automated planning component that assists users in making optimal routing decisions. In this case, both of these changes can be

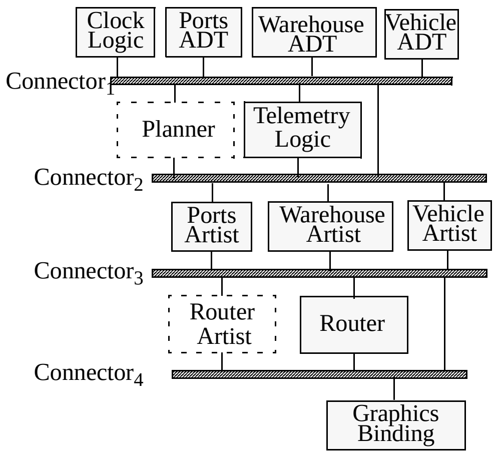
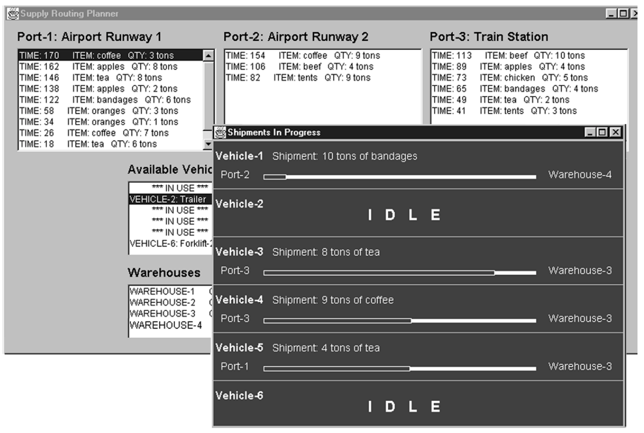


Figure 6-16. Cargo Router's C2 architecture with graphical artist and AI planner


```
Supply Routing Planner

Port-1: Airport Runway 1
TIME: 170   ITEM: coffee   QTY: 3 tons
TIME: 162   ITEM: apples   QTY: 8 tons
TIME: 146   ITEM: tea   QTY: 8 tons
TIME: 138   ITEM: apples   QTY: 2 tons
TIME: 122   ITEM: bandages   QTY: 6 tons
TIME: 58   ITEM: oranges   QTY: 3 tons
TIME: 34   ITEM: oranges   QTY: 1 tons
TIME: 26   ITEM: coffee   QTY: 7 tons
TIME: 18   ITEM: tea   QTY: 6 tons

Port-2: Airport Runway 2
TIME: 154   ITEM: coffee   QTY: 9 tons
TIME: 106   ITEM: beef   QTY: 4 tons
TIME: 82   ITEM: tents   QTY: 9 tons

Port-3: Train Station
TIME: 113   ITEM: beef   QTY: 10 tons
TIME: 89   ITEM: apples   QTY: 4 tons
TIME: 73   ITEM: chicken   QTY: 5 tons
TIME: 65   ITEM: bandages   QTY: 4 tons
TIME: 49   ITEM: tea   QTY: 2 tons
TIME: 41   ITEM: tents   QTY: 3 tons

Available Vehic
*** IN USE ***
VEHICLE-2: Trailer
*** IN USE ***
*** IN USE ***
*** IN USE ***
VEHICLE-6: Forklift-2

Warehouses
WAREHOUSE-1
WAREHOUSE-2
WAREHOUSE-3
WAREHOUSE-4

Shipments In Progress
Vehicle-1   Shipment: 10 tons of bandages
Port-2                                   Warehouse-4
Vehicle-2
            I D L E
Vehicle-3   Shipment: 8 tons of tea
Port-3                                   Warehouse-3
Vehicle-4   Shipment: 9 tons of coffee
Port-3                                   Warehouse-3
Vehicle-5   Shipment: 4 tons of tea
Port-1                                   Warehouse-3
Vehicle-6
            I D L E
```

Figure 6-17. Cargo Router's user interface with graphical artist and AI planner.

made while the system is executing. Figure 6-16 and Figure 6-17 depict the updated architecture and user interface, respectively, after both modifications have been made.

6.3.2.1 Alternative visualizations

Adding the new visualization requires adding a *Router Artist* component to the architecture. We add the new router artist between Connector 1 and Connector 4 because it uses notification messages provided by the *Port*, *Warehouse*, and *Vehicle* ADTs and utilizes the *Graphics* component for drawing graphics. The architect uses ArchShell to add the component using the "add component" command, connect it to buses using the "weld" command, and signal that the component should receive execution cycles using the "start" command (see Figure 6-18).

6.3.2.2 Automated planner

Adding the automated planner involves adding a *Planner* component to the architecture. The new planner component is added below Connector 1 because it monitors the state of the

```
try {
      if (model.architectureName().equals(“CargoSystem”)) {
            Connector above = model.connectorBelow(“Ports”);
            Connector below = model.connectorAbove(“PortArtist”);
            model.addComponent("Planner", "planner");
            model.weld(above, "planner");
            model.weld("planner", below);
            model.startEntity("planner");
            return true;
      } else return false;
} catch (ArchitectureModificationException e) {
      return false;
}
```

Figure 6-19. A portion of the Extension Wizard script used to add the Planner component into the running system. The “model” represents the ADT interface to the system’s architectural model.

ADTs to determine optimal routes. Figure 6-19 shows the critical portion of the modification script the Extension Wizard executes when installing the change. The script determines if the architectural model is that of the cargo routing system, then queries the model to determine the names of the connectors to which the planner component must be attached. If any of these operations fail, an exception is thrown which aborts the installation. An operation may fail if the architectural elements on which the change relies have been previously altered by other architectural modifications.

Supporting decentralized software evolution requires the deployment of the Architecture Evolution Manager, the Extension Wizard, and a portion of the cargo routing system’s architectural model. The Architecture Evolution Manager and the Extension Wizard consist

```
> add component
ClassName: c2.planner.RouterArtist
Name? RouterArtist
> weld
Top entity: Connector1
Bottom entity: RouterArtist
> weld
Top entity: RouterArtist
Bottom entity: Connector4
> start
Entity: RouterArtist
```

Figure 6-18. The ArchShell commands used to add the Router Artist component. Commands are denoted using bold text and command arguments are denoted using italicized text.

of 38 kilobytes of compiled Java code. The cargo routing system's architectural model consumes 2 kilobytes of disk space. The Planner system update, which consists of the modification script and the compiled Planner component, is 6 kilobytes.

6.4 Discussion

In addition to the adaptability demonstrated in the above scenarios, two types of consistency checks were employed. The first, as discussed in Section 6.1.5, prevents third-party changes that violate the constraints of the C2-style. The second, as discussed in Section 6.1.8, prevents third-party changes that violate constraints specified in Armani's architectural constraint language.

CHAPTER 7 Case Study: Mozilla Web Browser

This chapter describes a case study in which the open-architecture approach was used on a commercial software application: the Mozilla Web browser. Mozilla is the open source version of Netscape Communication Inc.'s popular Communicator Web browser.[8]

Mozilla was chosen for this case study for several reasons. One, unlike the previous case studies, Mozilla is a large, complex, legacy software system developed by an independent commercial software company under the "real-world" conditions imposed by the competitive shrink-wrapped software industry. Two, Mozilla implements several traditional third-party extension mechanisms, thus serving to demonstrate how the open-architecture approach can co-exist with other DSE techniques as well as to permit direct comparison with them. Third, Mozilla represented an opportunity to inject a minimal implementation of the open-architecture approach into a large, legacy system that did not necessarily conform to a component-based software engineering methodology. Fourth, most users of Web browsers

8. Netscape Inc. is now part of America Online, Inc.

can readily suggest new features they would like to have, but which cannot be implemented using Mozilla's traditional DSE techniques. Additionally, our selection of Mozilla was not biased by its implementation, or any other factors that would necessary lead to success.

This case study examines how well the open-architecture approach performs on such an application. Although the results of this study, by its very nature, cannot be generalized to other systems, the study does demonstrate that the open-architecture approach can be successfully applied to *a particular* large, complex legacy system in an incremental manner, and within a reasonable amount of time and effort.

7.1 Implementation Apparatus

The implementation of the open-architecture approach for Mozilla is quite minimal. Why? This was the first time the approach had been used on an large, legacy system that did not (a) have an explicit architectural model, (b) whose components did not communicate using message passing, (c) whose modules did not respect the principles of information hiding, and (d) whose implementation assumed a single address space. These differences warranted a cautionary approach. Instead of attempting to implement an elaborate, general-purpose infrastructure for open architectures on the first try, an ad hoc, experimental strategy was used. This permitted quick exploration of issues. Ultimately, the experience gained here will guide an implementation of a general-purpose infrastructure for legacy systems similar to the C2/Java framework and environment (see Section 6.1).

7.2 Host Application

Mozilla is by any measure a large, complex software system. It consists of 1.2 million lines of source code (SLOC) implemented in a mixture of the C and C++ programming languages, and it supports multiple operating system platforms (Microsoft Windows, Apple Macintosh, and many varieties of Unix), and has undergone considerable evolution over multiple releases spanning several years (four major releases since 1994).

7.2.1 Original architecture

Mozilla's existing third-party adaptability mechanisms included:

- *helper tool*: The helper tool mechanism can execute other applications in response to retrieving a particular media type, such as an application that can display postscript files on the screen.
- *plug-in mechanism*: The plug-in mechanism permits third-parties to provide new or replacement handlers for media types and render them within the Web browser's user interface window.
- *Java applets and JavaScript*: These two mechanisms allow third-party applications and scripts embedded within a Web page to execute whenever it is retrieved. If these programs are granted special privileges, they can execute as ordinary programs in the same runtime environment as the Web browser.
- *proxy server*: Mozilla can optionally redirect all Web-related (HTTP) network request to an intermediary server called a proxy server. Proxy servers are commonly used as bridges between private corporate networks and the public Internet, or as a shared cache of frequently requested resources.

Each of these mechanisms can be used to extend Mozilla in different ways. Yet, the changes we consider could not be made using any of these mechanisms because none of these mechanisms provides adequate access to Mozilla's internal data structures and behaviors.

7.2.2 Rearchitecting Mozilla

Our goal was to expose a small number of key bindings and modules as open points. These few open points could then potentially be used in a different ways to support the development of numerous add-ons. Several factors constrained the selection of open points:

Externalizing existing adaptability. Our intent was not to modify Mozilla's source code arbitrarily to add new features. This is certainly possible and desirable in certain situations as it permits any feature to be added, but it would not further our claim that open-architectures engender greater adaptability than previous techniques. Instead, we sought flexibility in the existing implementation that could only be used through source code modification. We then considered the potential benefits of externalizing that flexibility for third-party use as well.

Mozilla's existing design. Several aspects of Mozilla's implementation constrained our choice over open points. For example, an assumption that all of the modules execute in a single address space pervades much of the implementation. This is evidenced by the ubiquitous use of pointers to objects, object interfaces, variables, and data structures that straddle module boundaries.

Complexity. Mozilla's large size and complexity prevents a complete understanding of its implementation. For example, the implementation makes it difficult to elicit module dependencies and design assumptions. As such, it becomes difficult to know exactly how a change to its source code will affect other modules.

Excessive compile time. It can take anywhere from one to forty minutes to recompile Mozilla on a high-end workstation after a change depending on how much of the system is affected. This worked against interactive programming.

After examining and evaluating Mozilla's source code, we selected three points of flexibility that we wanted to expose to third-parties. Each of these is described below.

7.2.2.1 Network module open point

The network module open point provides a standard means for third-party add-ons to interpose behavior between Mozilla and its built-in network module, called *netlib*. Mozilla uses its *netlib* module whenever it has to retrieve a resource on the network, such as an HTML page or a GIF file. This open point permits third-party add-ons to augment or replace *netlib's* behavior or the data exchanged between it and Mozilla. Add-ons can, for example, support a new networking protocol (e.g., one that begins with "iiop://") or modify the contents of an HTML page as it passes from *netlib* to Mozilla's HTML parser and renderer.

This open point was implemented in three steps. First, a NetworkOpenPoint connector was constructed with the same functional interface as Mozilla's *netlib* module. Second, the name of Mozilla's *netlib* module was changed to *netlib.orig*. Third, the NetworkOpenPoint connector was assigned the name *netlib*. This was the simplest way to ensure that all of Mozilla's numerous references to the *netlib* module were properly redirected to the NetworkOpenPoint connector. Third-party add-ons register with the connector if they want to use the open point.

The NetworkOpenPoint connector acts as a proxy to *netlib*. Whenever Mozilla's code invokes a *netlib* function, the connector is invoked instead. This permits the connector to allow any third-party add-ons interested in altering or replacing the function's behavior to execute before Mozilla does. A third-party add-on can optionally return a special value that informs the connector not to invoke Mozilla's implementation of the function or that of any other yet-to-be-invoked third-party add-ons.

One aspect of Mozilla's implementation complicated the connector's ability to serve as an effective proxy: *netlib* functions in which clients pass an object reference as a parameter instead of a simple data type. For instance, when clients invoke the *netlib* function *opNetlibService::OpenStream*, they pass a URL specification (i.e., "http://www.ics.uci.edu/") and a reference to an object of type *nsIStreamListener*. *Netlib* stores this object reference with the URL, issues an asynchronous network read request, and immediately returns. Later when the network read request completes, a separate thread of execution within Mozilla invokes several interface functions on the associated *nsIStreamListener* object. Implemented this way, the connector's purview into the data exchanged between clients and *netlib* is incomplete—it does not witness the data exchange between *netlib* and the client it is occurs via the *nsIStreamListener* object. The NetworkOpenPoint connector addresses this problem by dynamically creating a proxy for the *nsIStreamListener* object passes as a parameter to *opNetlibService::OpenStream*. This way, *netlib* stores a reference to the dynamically created proxy object instead, which permits the NetworkOpenPoint connector to witness all of the communication between these modules.[9]

7.2.2.2 Content tree open point

The content tree open point provides a standard means for third-party add-ons to query and modify the abstract data type (ADT) Mozilla uses to represent the currently displayed HTML

9. Note that although this solution is adequate for this particular case, it can still be foiled. The solution assumes that the connector witnesses all attempts at indirect forms of communication, such as the *nsIStreamListener* object. But other forms of communication may be implicitly agreed upon in advance, such as a shared file or shared memory, and therefore are not discoverable by the proxy.

page, what Mozilla calls the *content tree*. As Mozilla's HTML parser populates this data structure, Mozilla's HTML render queries the ADT and renders the content in the browser window. Mozilla's built-in HTML editor provides an interactive user interface for changing this content tree.

The content tree ADT provides a functional interface for querying and changing the content tree data structure. Built-in to the browser is an incremental display update facility that can intelligently redraw the content tree as it changes. Mozilla's HTML editor uses this capability.

This open point was implemented by modifying a function in Mozilla's *nsNativeBrowserWindow* module called *EndLoadURL* and by introducing a ContentTreeOpenPoint connector to Mozilla's architecture. Third-party add-ons register with the connector if they want to use the open point. *EndLoadURL* was modified to notify the ContentTreeOpenPoint connector that Mozilla had just finished loading a new HTML page and, consequently, updated the content tree ADT. This permits the connector to notify any interested third-party add-ons and invokes them with a reference to the content tree ADT. Third-party add-ons use this reference to query and modify the content tree as needed.

7.2.2.3 Menu open point

The menu open point provides a standard means for third-party add-ons to add menu items to Mozilla's main menu and be notified when any menu item (including their own) is selected by the user.

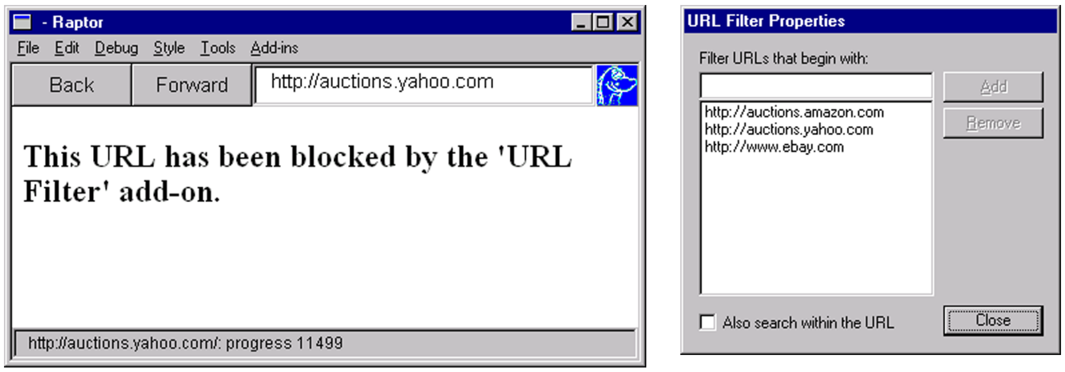
This open point was implemented by modifying two functions in Mozilla's *nsNativeBrowserWindow* module, one which loads Mozilla's main menu called *CreateMenuBar* and another that handles menu events called *DispatchMenuItem*, and by introducing a

MenuOpenPoint connector to Mozilla's architecture. Third-party add-ons register with the connector if they want to use the open point. *CreateMenuBar* was modified to notify the MenuOpenPoint connector that Mozilla had just loaded its main menu. This permits the connector to notify any third-party add-ons interested in this event and permit them to add their own menu items to the menu. *DispatchMenuItem* was modified to notify the MenuOpenPoint connector that the user had selected a menu item before Mozilla handles the event. This permits the connector to allow any third-party add-ons interested in the menu selection event to handle it before Mozilla does. After a third-party add-on has handled the event, it can optionally return a special value that informs Mozilla that the menu selection event has already been handled.

7.3 Third-party Changes Demonstrated

7.3.1 URL Filter add-on

The URL Filter add-on prevents an end-user from visiting a Web site whose URL matches a an entry in a list of "blocked" URLs. This add-on was implemented by the author and uses both the network module and menu open points. Figure 7-1 depicts the output produced by



http://auctions.yahoo.com

This URL has been blocked by the 'URL Filter' add-on.

http://auctions.yahoo.com/: progress 11499

Filter URLs that begin with:

http://auctions.amazon.com
http://auctions.yahoo.com
http://www.ebay.com

☐ Also search within the URL

Figure 7-1. URL Filter add-on. The result of attempting to visit a filtered URL on the left, and the add-on's properties for changing the list of blocked URLs on the right.

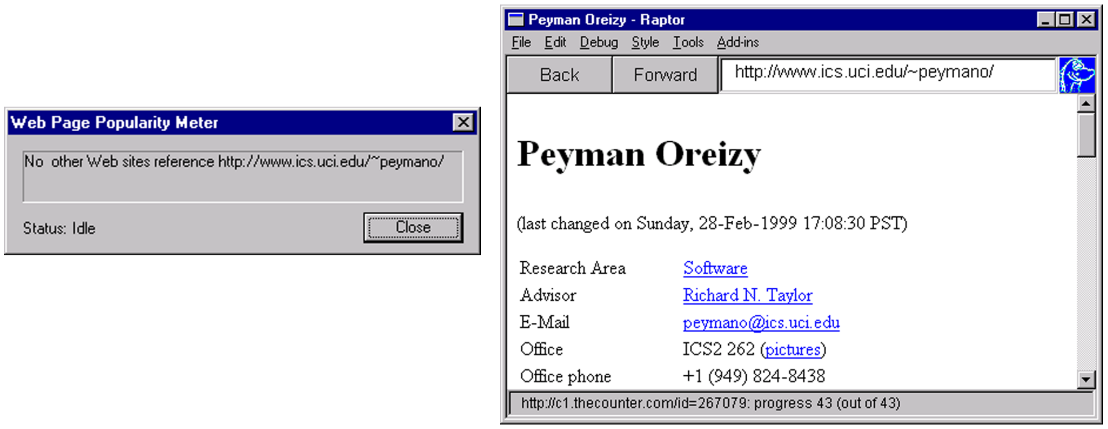
the add-on when the user attempts to visit a filtered URL (left) and the user interface for changing the list of "blocked" URLs (right).

7.3.2 Popularity Meter add-on

The Popularity Meter add-on, when activated, monitors an end-user as they traverse from one URL to another and, in a separate thread of execution, queries AltaVista's search engine (www.altavista.com) for the number of Web pages that refer to that URL. This add-on was implemented by the author and uses both the network module and menu open points. Figure 7-2 depicts the user interface of the add-on (left), which reflects the currently viewed Web page (right).

7.3.3 Content Filter add-on

The Content Filter add-on scans the body of an HTML page looking for HTML tags that define background colors, audio, and images, custom fonts, or images that have the same proportions as most banner ads (468 pixels wide by 60 pixels high). Based on end-user settings, the add-on can remove these HTML tags from the HTML page, and hence remove those attributes from the rendered Web page.


Web Page Popularity Meter
Close
Peyman Oreizy - Raptor
File Edit Debug Style Tools Add-ins
Back Forward

No other Web sites reference http://www.ics.uci.edu/~peymano/

Status: Idle

http://www.ics.uci.edu/~peymano/

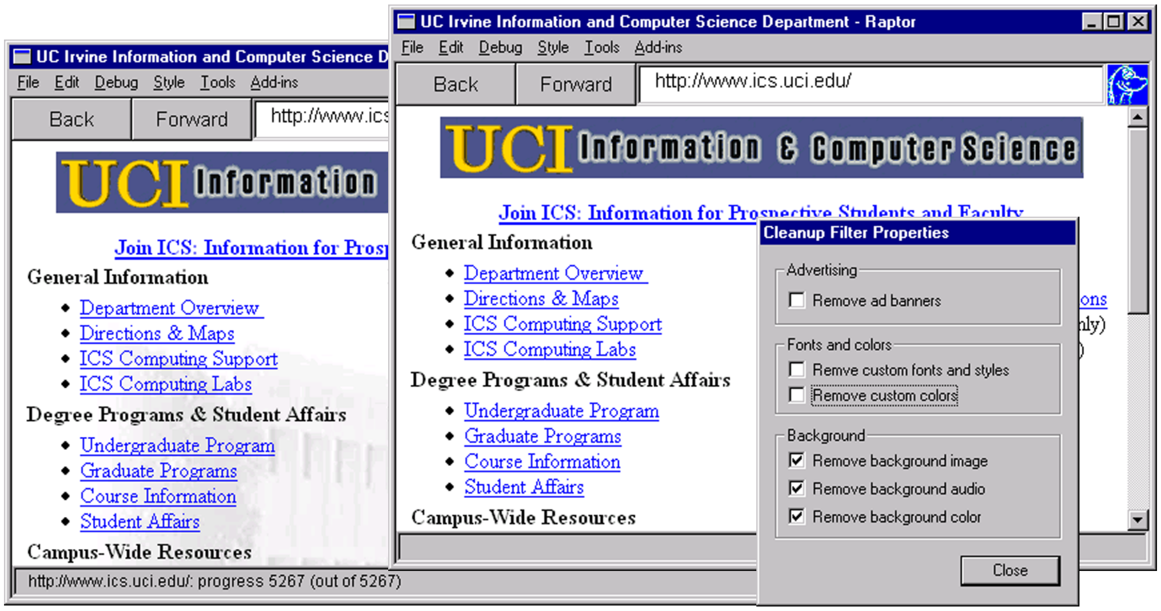
Peyman Oreizy

(last changed on Sunday, 28-Feb-1999 17:08:30 PST)

Research Area	Software
Advisor	Richard N. Taylor
E-Mail	peymano@ics.uci.edu
Office	ICS2 262 (pictures)
Office phone	+1 (949) 824-8438

http://c1.thecounter.com/id=267079: progress 43 (out of 43)

Figure 7-2. Popularity Meter add-on. The add-on's user interface on the left automatically updates to reflect the currently viewed Web page on the right.


UC Irvine Information and Computer Science D

File Edit Debug Style Tools Add-ins

Back Forward http://www.ics

UCI Information

Join ICS: Information for Pros

General Information

- Department Overview
- Directions & Maps
- ICS Computing Support
- ICS Computing Labs

Degree Programs & Student Affairs

- Undergraduate Program
- Graduate Programs
- Course Information
- Student Affairs

Campus-Wide Resources

http://www.ics.uci.edu/: progress 5267 (out of 5267)

UC Irvine Information and Computer Science Department - Raptor

File Edit Debug Style Tools Add-ins

Back Forward http://www.ics.uci.edu/

UCI Information & Computer Science

Join ICS: Information for Prospective Students and Faculty

General Information

- Department Overview
- Directions & Maps
- ICS Computing Support
- ICS Computing Labs

Degree Programs & Student Affairs

- Undergraduate Program
- Graduate Programs
- Course Information
- Student Affairs

Campus-Wide Resources

Cleanup Filter Properties

Advertising
- [ ] Remove ad banners

Fonts and colors
- [ ] Remve custom fonts and styles
- [ ] Remove custom colors

Background
- [✓] Remove background image
- [✓] Remove background audio
- [✓] Remove background color

Close

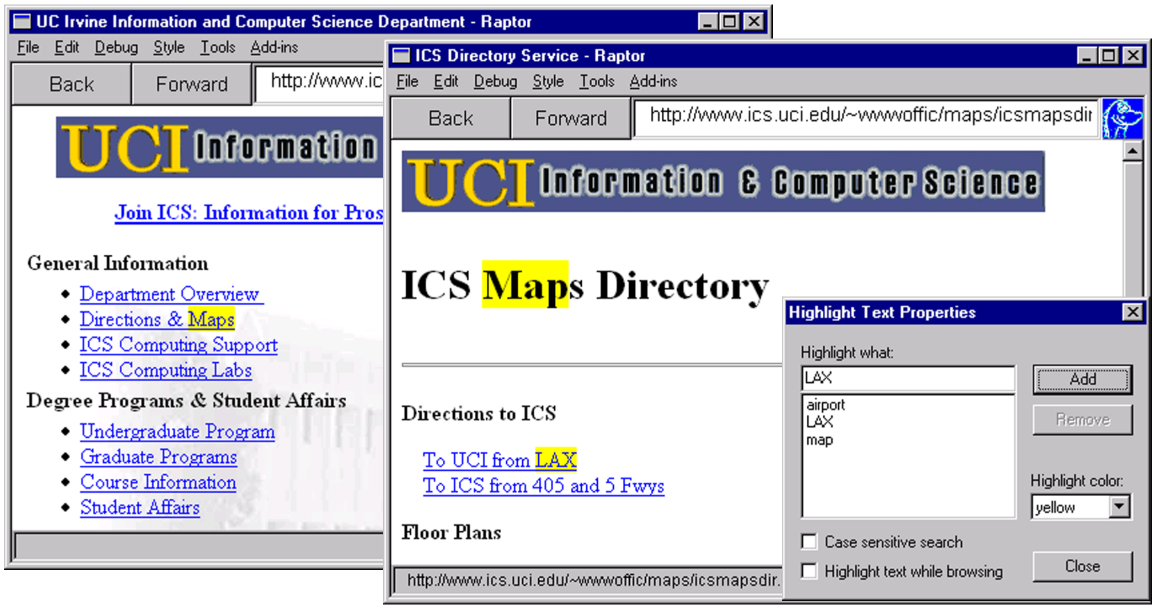
Figure 7-3. Content filter add-on. The background image in the Web page (at left) is removed (at middle) after the user has enabled the 'remove background image' check box (at right).

This add-on was implemented by the author and uses both the network module and menu open points. Figure 7-3 depicts how the background image in the Web page (at left) is removed (at middle) after the user has enabled the "remove background image" check box (at right).

7.3.4 Text Highlighter add-on

The Text Highlighter add-on scans the body of an HTML page and highlights phrases specified by an end-user. This add-on was implemented by the author and uses both the content tree and menu open points. Figure 7-4 depicts two pages (at left and at middle) with words highlighted in yellow based on the phrases specified by the user (at right).

This third-party add-on and the content tree open point were implemented in approximately 29 person hours. A majority of this time (18 person hours) was spent discovering how to properly use the content tree's functional interfaces to add HTML elements.


UC Irvine Information and Computer Science Department - Raptor
File Edit Debug Style Tools Add-ins
Back Forward
ICS Directory Service - Raptor
Highlight Text Properties
Add
Remove
Close

http://www.ic

UCI Information

Join ICS: Information for Pros

General Information

- Department Overview
- Directions & Maps
- ICS Computing Support
- ICS Computing Labs

Degree Programs & Student Affairs

- Undergraduate Program
- Graduate Programs
- Course Information
- Student Affairs

http://www.ics.uci.edu/~wwwoffic/maps/icsmapsdir

UCI Information & Computer Science

ICS Maps Directory

Directions to ICS

To UCI from LAX
To ICS from 405 and 5 Fwys

Floor Plans

http://www.ics.uci.edu/~wwwoffic/maps/icsmapsdir.

Highlight what: LAX

airport
LAX
map

Highlight color: yellow

☐ Case sensitive search
☐ Highlight text while browsing

Figure 7-4. Text Highlighter add-on. The add-on's user interface on the lower-right permits the user to specify words that are subsequently highlighted as the user surfs the Web. Two sample Web pages with highlights are shown on the left and in the middle.

7.4 Discussion

Overall, it took approximately 100 person hours to examine Mozilla's source code, implement all three open points, and implement all four add-ons. This was done without the aid of a class framework (such as C2/Java) or special tools. With these three open points in place, third-parties can implement a large variety of add-ons.

CHAPTER 8

Validation and Evaluation

This chapter validates the claim made back in Chapter 4, that:

> *An application's architectural system model can provide a basis for decentralized software evolution that offers greater degrees of adaptability while at the same time supporting more assurances over consistency.*

Does open-architecture software, which is an example of the architectural system model approach, offer greater degrees of adaptability and consistency? We validate this claim using a two-pronged approach. First, a conceptual argument is given that compares the adaptability and consistency offered by open-architectures with that of previous DSE techniques. Second, experience using the open-architecture approach on three case studies is summarized and used to demonstrate that the approach can be implemented in a practical manner.

8.1 Conceptual Validation

8.1.1 Evaluating adaptability using open points

Section 3.2 introduced open points as a measure for adaptability and used it to compare several characteristic DSE techniques at the module-level of abstraction. Table 8-1 summarizes that comparison and adds to it the open-architecture software approach proposed in Chapter 5. As before, a filled circle (●) indicates full support for the open point type, an unfilled circle (○) indicates partial or restricted support for an open point type, and an empty cell indicates that the open point type is unsupported.

Unlike previous techniques, open architectures fully support all six open point types without revealing source code. This is due in part to the use of multiple flexible connectors and an explicit architectural model.

DSE technique	Add module	Replace module	Remove module	Add binding	Redirect binding	Remove binding	Internal structure mutable	Internal structure apparent
API	○[a]	○[b]	○[b]					
plug-in	○[ac]	○[b]	○[b]				○[d]	
scripting language	○[a]	○[b]	○[b]					
component architecture	●	●	○[e]	○[f]	○[f]	○[f]	●	
event architecture	●	○[b]	○[b]	○[b]	○[b]	○[b]		
source code	●	●	●	●	●	●	●	
open-architecture	●	●	●	●	●	●	●	●

Table 8-1: Summary of open point types supported by DSE techniques at the module-level.

a. other modules cannot discover the interface of the added modules
b. confined to third-party modules and bindings only, not those within the host application
c. cannot bind to the functional interfaces of other modules
d. only the plug-in module of the internal structure is mutable
e. only if the module is not a destination of any binding
f. only by modifying the module that is the source of the binding

This validates the claim that an application's architectural system model can provide a basis for decentralized software evolution that engenders novel types of adaptability. The argument that the technique is practical is made in Section 8.2.

8.1.2 Evaluating assured consistency

Assuring consistency for any non-trivial program is difficult and in general undecidable. Young and Taylor (1986, page 54) observe that "since the presence of faults is generally an undecidable property, it is not even theoretically possible to devise a completely accurate technique which is applicable to arbitrary programs." In the context of decentralized software evolution, this problem is aggravated by (a) independent change and often (b) limited or no access to source code.

The open-architecture approach neither claims to prevent inconsistencies, nor does it advocate a particular technique for doing so. Instead, the open-architecture approach offers a means for using traditional model-based analyses in the context of decentralized software evolution. Specifically, analyses that operate upon abstract, high-level representations of an application's implementation can use the approach's architectural system model as a surrogate.

Consider, for example, consistency analysis with reuse contracts (Steyaert et al. 1996). Reuse contracts can codify some of the assumptions a host application developers makes regarding the manner their application is customized by third-party developers. A reuse contract can, for example, state that add-ons cannot override or mask certain host application functions. Reuse contracts such as this can annotate an application's architectural system model. Whenever a third-party add-on is installed, the reuse contract can be rechecked to prevent stated assumptions from being violated.

This same strategy can be applied to other system model-based analyses, such as configuration graphs (Hiltunen 1998), grammars (Batory and Geraci 1998), architectural type theory (Medvidovic et al. 1999), and architectural constraints (Monroe 1998). Oreizy and Taylor (1999) suggests another possible strategy based on the resource contention mechanisms used by operating systems. Annotations can also be used to support less automated, possibly human-directed, analyses.

This validates the claim that an application's architectural system model can provide a basis for decentralized software evolution that engenders novel types of consistency.

8.2 Empirical Validation

Empirical validation of new software techniques is notoriously difficult. Proper empirical evaluation requires multiple, carefully controlled studies that have been adequately randomized to achieve a certain degree of confidence or external validity. However, research in software concerns large-scale, collaborative development efforts that involve multiple developers and diverse stakeholders. Project scale plays a critical factor. Certain software development techniques, such as Stepwise Refinement (Wirth 1971), have a positive impact on well-understood or small-scale projects, but breed disaster on novel or large-scale projects. The reverse is also true. On a large-scale project, the inherent overhead of some software techniques, such as information hiding (Parnas 1972), are negligible and overshadowed by the benefits produced. On a small-scale project, the benefits may never accrue to outweigh the technique's overhead.

In most cases, running a large-scale empirical study is infeasible. The cost, time, and risks often cannot be substantiated, and even if they could, it is difficult to control the study. As a result, empirical validation of new software techniques relies on alternative methods, such as case studies. Although case studies do not prove that a technique is better in all situations, it does provide some evidence that the technique is beneficial and practical.

		Open point types used					
Application	Third-party change	Add module	Replace module	Remove module	Add binding	Redirect binding	Remove binding
KLAX	high score list	×			×		
	spelling KLAX	×		×	×		×
	multi-player KLAX	×			×		
Cargo Planner	alternate visualizations	×		×	×		×
	automate planner	×			×		
Mozilla	URL filter	×	×		×	×	
	content filter	×	×		×	×	
	popularity meter	×	×		×	×	
	text highlighter	×			×		

Table 8-2: Summary of case studies showing the open point types used by each change.

Table 8-2 summarizes the three case studies presented in Chapter 6 and Chapter 7 by relating the third-party add-ons of each with the open point types used to make the change. Collectively, the changes demonstrated by the three case studies have used all six open point types and done so without source code modification. Furthermore, the effort expended to make these changes was acceptable.

It should be noted that the time and effort spent rearchitecting Mozilla to introduce open points is amortized by every third-party add-on that uses the open points. This is because each third-party add-on developer that uses the open points is relieved of the burden of having to understand, reason about, modify, and retest Mozilla's source code.

8.3 Evaluation

Open points. Are "open points" a good measure of adaptability? Although we cannot offer a rigorous proof of their utility, we argue that open points are useful because:

1. they provide a useful characterization of existing techniques based on what can and cannot be changed by third-parties;
2. they measure an essential property of third-parties adaptability, namely "what can third-parties change and what can they not?"
3. they identify several shortcomings of existing techniques and suggest ways of combining techniques to overcome them;
4. they help identify the types of adaptability that are not provided by available techniques.

Several challenges and risks remain.

Intellectual property rights. Exposing an application's system model to third-parties reveals an aspect of its software design, and can be used by competitors to reverse engineer the application's implementation, eroding the host application vendor's "competitive advantage." While this is a legitimate concern in certain competitive markets, the host application developer has some recourse. For example, host application developers can encrypt the system model and selectively disclose it trusted third-party developers, e.g., those developers that have agreed not to build competitive products.

Runtime performance. The use of connectors in the implementation can degrade runtime performance. The greatest performance impact occurs for primitive connectors, such as procedure calls, since an additional level of indirection is introduced. For more complex connectors, such as RPC and software buses such as FIELD (Reiss 1990), the functionality required can usually be integrated without a significant runtime performance penalty. Recent developments in the area of dynamic linking attempt to reduce or even eliminate the runtime overhead associated with altering binding decisions (e.g., see (Franz 1997, Massalin 1992, Pardyak and Bershad 1996)).

Ultimately, host application developers determine which connectors are used based on application requirements. In situations where performance outweighs third-party adaptability, connectors may be replaced in the implementation with direct component-to-component bindings such as is done in UniCon (Shaw et al. 1995).

CHAPTER 9

Conclusion

9.1 Contributions of this Dissertation

The contributions of this dissertation are:

1. a measure for evaluating and comparing the flexibility of DSE approaches;
2. a system model-based DSE approach that supports a broader class of changes than previously feasible and that can be applied in an incremental manner;
3. a conceptual architecture for implementing system model-based approaches, such as open-architectures;
4. a policy neutral mechanism for governing third-party changes based on system-models;
5. a more effective characterization of the relationships between system models, the types of third-party changes they support, and the assurances they provide.

9.2 Further Research in the Area

There are a number of directions for further work in the area of decentralized software evolution. These are:

Developing applications using their own DSE techniques. The most customizable applications are generally those that implement a significant portion of their functionality using their own DSE mechanisms. This helps in two ways: (1) it forces the host application developers to exercise the DSE mechanisms, which reveals limitations sooner, possibly in time to remedy them, and (2) third-parties can learn from and use the host application's own add-ons. Systems that use component architectures or source code are intrinsically built in this fashion, but other techniques may be similarly used. For example, a large portion of Emacs is implemented using its own scripting language and hook mechanism, and add-ons use the same mechanism as built-in features to document themselves.

Supporting composition of non-behavioral aspects. Numerous strategies guide us in composing add-on behaviors and dealing with inconsistencies. But there are few guidelines for composing non-behavioral aspects of add-ons, such as documentation, user interface elements, and data models, and for dealing with their inconsistencies. For example, if an add-on augments the printing capabilities of a word processor, its documentation should be presented along with the host application's built-in documentation on the subject.

Detecting inconsistencies. Deploying an architectural system model as an integral part of an application creates new opportunities for detecting inconsistencies in-the-field. One promising area is suggested by the resource contention models used by operating systems (Oreizy and Taylor 1999). Operating systems effectively prevent multiple, independently developed applications from interfering with one another even though the applications must share common functional interfaces and resources such as processors, memory, the file system, and hardware devices. Operating systems essentially do this by managing the resources accessible to applications. For example, the speaker device is usually reserved for a single application. Applications must request exclusive access to the device before using it; attempts to use the device without permission are prevented. More complex resource sharing policies are used for the processor, memory, and file system.

9.3 Final Remarks

Although numerous methodologies, techniques, and tools aid the task of designing and implementing evolvable software, i.e., centralized software evolution, we know relatively little about its decentralized counterpart. Successful uses of decentralized software evolution, especially post-deployment, have demonstrated increased opportunities for software reuse and offered solutions that would have otherwise been infeasible.

This dissertation takes the first steps toward improving our understanding of decentralized software evolution. We have defined and contrasted decentralized software evolution from other facets of software evolution and identified a number of unique issues that it raises. We have also presented a practical and general-purpose technique for measuring the adaptability of software customization techniques and have used our technique to compare six commonly used, characteristic techniques.

We have also proposed a novel software customization technique, called open-architecture software, that offers greater degrees of adaptability and consistency than previous approach. We have described our experience in using the approach on two small examples and on the Mozilla Web browser, the open-source version of a large, legacy commercial software application.

The contributions of this dissertation lay the groundwork for further work in the area.

References

AHO ET AL 1986. Aho, Sethi, Ullman. *Compilers: Principles, techniques, and tools*, 1986.

ALLEN AND GARLAN 1997 R. Allen, D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, July 1997.

ANDERSON ET AL 2000 Kenneth M. Anderson, Richard N. Taylor, E. James Whitehead, Jr. Chimera: Hypermedia for Heterogeneous Software Development Environments, Accepted for publication in *ACM Transactions on Information Systems.* Anticipated Publication date: March 2000.

APPLE COMPUTER, INC. 1996. Apple Computer. *AppleScript Language Guide: English Dialect.* July 1996.

BALZER 1998. Robert Balzer. Instrumenting, monitoring, & debugging software architectures. *Unpublished.* Available at http://www.isi.edu/software-sciences/papers/instrumenting-software-architectures.doc, January 28, 1998.

BARRETT ET AL 1996. Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, Alexander E. Wise, A Framework for Event-Based Software Integration. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 4, October 1996.

BATORY AND GERACI 1998. Don Batory, Bart J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, vol. 23, no. 2, February 1997.

BERSHAD ET AL 1995 B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, and others. Extensibility, safety and performance in the SPIN operating system. *Fifteenth Symposium on Operating Systems Principles.* Copper Mountain Resort, CO, USA, 3-6 Dec. 1995.

BINNS ET AL 1996 P. Binns, M. Engelhart, M. Jackson, S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control. *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, 1996.

BOEHM 1988 B.W. Boehm. A spiral model of software development and enhancement. *Computer*, vol.21, no.5, May 1988.

BOEHM AND SCHERLIS 1992. Barry W. Boehm and William L. Scherlis. Megaprogramming. In *Proceedings of the Software Technology Conference 1992*, p63-82, Los Angeles, June 1992. DARPA.

BOOCH 1994. Grady Booch, *Object-oriented analysis and design*. Second edition. Benjamin/ Cummings Publishing Company, Inc. 1994.

BOOCH ET AL 1998. Grady Booch, Ivar Jacobson, James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley Pub. Co. October 1998.

BROCKSCHMIDT 1994. Kraig Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.

BROOKS 1987. Fredrick P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. *Computer*, vol.20, no.4, April 1987. Pages 10-19.

BROOKS 1995. Fredrick P. Brooks Jr. 1995. *The Mythical Man Month*. Anniversary Edition. Addison-Wesley Publishing Company, Inc. 1995.

BUFFENBARGER 1995. J. Buffenbarger. Syntactic Software Merging. *Proceedings of the Fifth Workshop on Software Configuration Management*. 1995. Pages 153-172.

CATALANO 1997 Frank Catalano. Components roll over as consumers snap up suites. *Computer Retail Week*, November 03, 1997.

CHAN AND LEE 1997 Patrick Chan and Rosanna Lee. *The Java Class Libraries, Second Edition, Volume 2: java.applet, java.awt, java.beans*. Addison-Wesley Pub Co., 2nd edition, vol 2. October 1997.

CUTKOSKY ET AL 1996. M. R. Cutkosky , Tenenbaum, J. M., and Glicksman, J. Madefast: Collaborative engineering over the Internet. *Communications of the ACM*, vol. 39, no. 9, September 1996.

CYPHER ET AL 1993 A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers and A. Turransky, eds. *Watch What I Do: Programming by Demonstration*. Cambridge, MA. The MIT Press, 1993.

EWELL 1998. Maranda Ewell, An Interview with Alvin Toffler—A New World Of Knowledge, *San Jose Mercury News*, March 23, 1998.

FAYAD AND CLINE 1996 Mohamed Fayad and Marshall P. Cline. Aspects of software adaptability. *Communications of the ACM*, Oct. 1996, vol.39, no.10.

FIELDING AND KAISER 1997. Roy Fielding and Gail Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, July-August 1997.

FRANZ 1997. M. Franz. Dynamic linking of software components. IEEE Computer, vol 30, no 3, pp 74-81, March 1997.

GAMMA ET AL 1995. Eric Gamma, Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley, 1995.

GARLAN AND NOTKIN 1991. David Garlan, David Notkin. Formalizing design spaces: implicit invocation mechanisms. *Proceedings of VDM'91: Formal Software Development Methods*. October 1991.

GARLAN ET AL 1992. David Garlan, Gail E. Kaiser, David Notkin. Using tool abstraction to compose systems. *Computer*, June 1992, vol.25, no.6. Pages 30-38.

GARLAN AND SCOTT 1993. David Garlan, Adding implicit invocation to traditional programming languages. *Proceedings of the 15th Internation Conference on Software Engineering*, May 1993. Pages 447-455.

GERETY 1990. C. Gerety. HP SoftBench: A new generation of software development tools. Hewlett-Packard Journal, vol. 31, no. 3, June 1990. Pages 48-59.

GHEZZI ET AL 1991. C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering.* Englewood Cliffs, NJ : Prentice Hall, 1991.

GOSLING ET AL 1996. James Gosling, Bill Joy, Guy L. Steele. *The Java Language Specification.* Addison-Wesley Pub. Co., September 1996. Available online at http://www.javasoft.com/docs/books/jls/html/index.html

HALL ET AL 1999 Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *Proceedings of the 1999 International Conference on Software Engineering*, IEEE Computing Society. May 1999.

HAROLD 1999 Elliotte Rusty Harold. *XML Bible.* IDG Books Worldwide. July 1999.

HEINEMAN 1998 George T. Heineman. Composing Software Systems from Adaptable Software Components. *Proceedings of the OMG-DARPA Workshop on Compositional Software Architectures*, Monterey, CA, January 6-8, 1998.

HERBSLEB AND GRINTER 1999 J. D. Herbsleb and R.E. Grinter. Splitting the Organization and Integrating the Code: Conway's Law Revisited. *21st International Conference on Software Engineering.* Los Angeles, USA, May 1999.

HILTUNEN 1998 M.A. Hiltunen. Configuration management for highly customisable software. *IEE Proceedings-Software*, vol.145, no.5, IEE, Oct. 1998. p.180-8.

HÖLZLE 1993. Urs Hölzle. Integrating Independently-developed Components in Object-oriented languages. *Proceedings of ECOOP `93*, Kaiserslautern, Germany, July, 1993.

HORWITZ ET AL 1989. S. Horwitz, J. Prins, T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems.* vol. 11, no. 3, July 1989, Pages 345-388.

ICCDS 1992-1998 The Proceedings of the International Conference on Configurable Distributed Systems.

ICOP The International Workshop on Component-Oriented Programming.

ISAW 1995-2000 The International Software Architecture Workshop.

IWCSE The International Workshop on Component-based Software Engineering

IWPSE 98 The Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1). Kyoto, Japan. April 20-21, 1998.

IWPSE 99 Proceedings of the Second International Conference on the Principles of Software Evolution (IWPSE 2). Fukuoka, Japan. July 16-17, 1999.

KADIA 1992. R. Kadia. Issues envountered in building a flexible software development environment: Lessons learned from the Arcadia project. *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments.* Tyson's Corner, Virginia, December 1992.

KERNIGHAN AND RITCHIE 1988 Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, June 1988.

KHARE ET AL 2000 Rohit Khare, Michael Guntersdorfer, Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor. xADL: Enabling Architecture-Centric Tool Integration With XML. *Unpublished draft*, 2000.

KICZALES ET AL 1991. Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

KICZALES ET AL 1997A. Gregor Kiczales, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J. Aspect-oriented programming. *Xerox PARC Technical Report, SPL97-008 P9710042*. February 1997.

KICZALES ET AL 1997B. Gregor Kiczales. Open Implementation Design Guidelines. *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA. May 1997. Pages 481-490.

KRAMER AND MAGEE 1985. Jeff Kramer and Jeff Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 11, no. 4, April 1985.

KRISHNAMURTHI AND FELLEISEN 1998. Shriram Krishnamurthi and Matthias Felleisen. Toward a Formal Theory of Extensible Software. *Proceedings of the Sixth Symposium on the Foundations of Software Engineering*, 1998

KRUEGER 1992. C. W. Krueger. Software reuse. *Computing Surveys.* vol. 24, no. 2. June 1992.

LEHMAN AND BELADY 1985. M.M. Lehman and L.A. Belady. *Program evolution : processes of software change.* Academic Press, 1985.

LIEBERHERR 1996. Karl J. Lieberherr. *Adaptive object-oriented software–the Demeter method.* PWS Publishing Company. 1996.

LIENTZ AND SWANSON 1980 B. P. Lientz and E. B. Swanson. *Software Maintenance Management*, Addison-Wesley, Reading, Mass., 1980.

LIPPMAN 1991. Stanley B. Lippman. *C++ Primer.* 2nd Edition. Addison-Wesley Publishing Company. 1991.

LUCKHAM AND VERA 1995 D.C. Luckham, J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, vol. 21, no. 9, September 1995.

MACKAY 1990 W.E. Mackay. *Users and Customizable Software: A Co-Adaptive Phenomenon*, PhD Thesis, Massachusetts Instititute of Technology. 1990.

MAES 1987 P. Maes. PhD Thesis. *Computational Reflection*. Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Technical Report 87-2.

MAGEE AND KRAMER 1996. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *Fourth SIGSOFT Symposium on the Foundations of Software Engineering.* San Francisco, October 1996.

MALHOTRA 1994. Jawahar Malhotra. On the construction of extensible systems. *Technology of Object-Oriented Languages and Systems, TOOLS 13*. Versailles, France, March 7-11, 1994. Pages 255-270.

MASSALIN 1992 Henry Massalin. *Synthesis: An Ecient Implementation of Fundamental Operating System Services.* PhD Thesis. Columbia University. 1992.

MCLELLAN ET AL 1998. S. G. McLellan, A. W. Roesler, J. T. Tempest, C. I. Spinuzzi. Building more usable APIs. *IEEE Software*, May-June 1998, vol.15, (no.3). Pages 78-86.

MEDVIDOVIC ET AL 1997 N. Medvidovic, P. Oreizy, R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. *Symposium on Software Reusability*, Boston, May 1997.

MEDVIDOVIC ET AL 1999 Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 16-22, 1999.

MICROSOFT DIRECTSHOW 1998. Microsoft DirectShow SDK, October 1998. Available online at http://www.microsoft.com/directx/dxm/help/ds/c-frame.htm#default.htm

MICROSOFT VBA 1997. *Microsoft Office 97/Visual Basic Programmer's Guide.* Microsoft Press. December 1996. Available online at http://www.microsoft.com/officedev/articles/Opg/001/001.htm

MONROE 1998. Robert T. Monrow. Capturing Software Architecture Design Expertise with Armani: The Armani Language Reference Manual version 1.0. *CMU Technical Report CMU-CS-98-163*, School of Computer Science, Carnegie Mellon University, October 1998.

MOORE 1965 G.E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, vol. 38, no. 8, April 19, pp. 114-117.

MØRCH 1998 A. Mørch. Tailoring Tools for System Development. *Journal of End User Computing.* Spring 1998.

NACHENBERG 1997. Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, vol. 40, no. 1, Jan. 1997. Pages 46-51.

NATARAJAN AND ROSENBLUM 1998 R. Natarajan and D. S. Rosenblum. Merging Component Models and Architectural Styles, *Proc. Third Int'l Software Architecture Workshop*, Lake Buena Vista, FL, Nov. 1998.

NECULA AND LEE 1996 George Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington. October 28-31, 1996.

NETSCAPE COMMUNICATIONS INC., 1998. Netscape Corporation. *Plug-in Guide for Communicator 4.0*, January 1998. Available online at http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm

NØRMARK 1996. Kurt Nørmark. Hooks and open points. Published in *Quality Software - Concepts and Tools*, edited by Jan Stage, Kurt Nørmark, and Kim Guldstrand Larsen, The Software Engineering Programme, Institute for Electronic Systems, Aalborg University, Denmark.

NOTKIN AND GRISWOLD 1988. David Notkin, William G. Griswold. Extension and Software Development. *Proceedings of the 10th International Conference on Software Engineering*, 1988.

NOTKIN ET AL 1993. David Notkin, David Garlan, William G. Griswold, Kevin Sullivan. Adding implicit invocation to languages: three approaches. In S. Nishio and A. Yonezawa Eds., *Object Technoogies for Advanced Software: Proceedings of the First JSSST International Symposium*, Kanazaw, Japan, Nov. 1993. Springer-Verlag. Pages 489-510.

OMG 1999 Object Management Group. *CORBA/IIOP 2.3.1 Specification.* July 1999. Available online at http://www.omg.org/corba/cichpter.html

OREIZY 1996 Peyman Oreizy. Issues in the Runtime Modification of Software Architectures. *Technical Report UCI-ICS-96-35*, Department of Information and Computer Science, University of California, Irvine, August 1996.

OREIZY 1998 P. Oreizy. Decentralized Software Evolution. *Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1).* Kyoto, Japan. April 20-21, 1998.

OREIZY ET AL 1998. Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. Proceedings of the *International Conference on Software Engineering 1998*, Kyoto, Japan. April 1998.

OREIZY AND TAYLOR 1998. Peyman Oreizy and Richard N. Taylor. On the Role of Software Architectures in Runtime System Reconfiguration. Proceedings of the *International Conference on Configurable Distributed Systems (ICCDS 4).* Annapolis, Maryland, May 4-6, 1998.

OREIZY AND TAYLOR 1999 P. Oreizy and R. N. Taylor. Coping with Application Inconsistency in Decentralized Software Evolution. *Second International Conference on the Principles of Software Evolution (IWPSE 2)*. Fukuoka, Japan. July 16-17, 1999.

OUSTERHOUT 1994. John K. Ousterhout. *Tcl and the Tk Toolkit*, Addison-Wesley Professional Computing, May 1994. See also http://www.scriptics.com/

PARDYAK AND BERSHAD 1996 P. Pardyak, B. N. Bershad. Dynamic Binding for an Extensible System. *Proceedings of the Second Symposium on Operating Systems Design and Implementation*. OSDI II. ACM SIGOPS/USENIX Association Symposium on Operating System Design and Implemenation. October 1996.

PARNAS 1972. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of ACM*. vol. 15, no. 12, December 1972.

PERRY 1996 Dewayne E. Perry. System Compositions and Shared Dependencies. *6th Workshop on Software Configuration Management*, ICSE18, Berlin Germany, March 1996.

PERRY ET AL 1998 D. E. Perry, G. S. Gil, L. G. Votta. A Case Study of Successful Geographically Separated Teamwork. *SPI'98*, December 1998.

PERRY AND KAISER 1991. Dewayne E. Perry and Gail E. Kaiser, Models of software development environments. *IEEE Transactions on Software Engineering*, vol 17, no. 3. pp 283-295, March 1991.

PERRY AND WOLF 1992. Dewayne E. Perry and Alex L. Wolf, Foundations for the study of software architecture. *Software Engineering Notes*, vol. 17, no. 4, October 1992.

PETZOLD 1996. Charles Petzold. *Programming Windows 95*. Microsoft Press. March 1996.

PITAC REPORT 1999 *President's Information Technology Advisory Committee Report to the President, Information Technology Research: Investing in Our Future*.
Available at http://www.ccic.gov/ac/report/

PLATT 1999 David S. Platt. *Understanding COM+*. Microsoft Press. June 1999.

POUNTAIN AND SZYPERSKI 1994 Dick Pountain and Clemens Szyperski. Extensible Software Systems. *Byte Magazine*. vol. 19, no. 5, May 1994.

PURTILO 1989 J.M. Purtilo. MINION: An environment to organize mathematical problem solving. *Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation*, July 1989.

PURTILO 1994. James M. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*. vol 16, no 1. January 1994. Pages 151-174.

RAN AND XU 1996. Alexander Ran and Jianli Xu. Structuring Interface. *Proceedings of the 2nd International Software Architecture Workshop*, 1996. Held in conjunction with the Foundations of Software Engineering Conference.

RAYMOND 1999 Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly & Associates. October 1999.

REISS 1990. Steven P. Reiss. Connecting tools using message pasing in the FIELD environment. *IEEE Software.* vol. 7, no. 4. July 1990. Pages 57-67.

REISS 1996. Steven P. Reiss. *The FIELD programming environment: A friendly integrated environment for learning and development.* Kluwer Academic Publishers. 1996.

RHEINFRANK 1995. John Rheinfrank, A conversation with Don Norman, *interactions*, vol. 2, no. 2, April 1995, pp 47-55.

ROBBINS ET AL 1996 J. E. Robbins, D. F. Redmiles, D. M. Hilbert. Extending design environments to software architecture design. *11th Knowledge-Based Software Engineering Conference (KBSE'96).* Syracruse, New York. Sept. 1996.

ROBBINS AND REDMILES 1998 Jason E. Robbins, David F. Redmiles. Software Architecture Critics in the Argo Design Environment. *Knowledge-Based Systems.* Sept. 1998. vol. 5. no. 1.

ROBBINS AND REDMILES 1999 J. E. Robbins, D. F. Redmiles. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. Invited paper to appear in the *Journal of Information and Software Technology*, Special issue: The Best of COSET'99.

SANDEWALL 1978. Erik Sandewall. Programming in an interactive environment: the "Lisp" experience. *Computing Surveys*, Vol. 10, (No. 1), March 1978, Pages 35-71.

SELBY 1988 R. Selby. Empirically Analyzing Software Reuse in a Production Environment, In *Software Reuse: Emerging Technology*, W. Tracz (Ed.), IEEE Computer Society Press, 1988., pp. 176-189.

SELTZER ET AL 1996. Margo I. Seltzer, Yasuhiro Endo, Christopher Small, Keith A. Smith. Issues in Extensible Operating Systems. *Unpublished draft 1996.* Available at http://www.eecs.harvard.edu/vino/vino/papers/cacm-96.ps

SHAW ET AL 1995 Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, April 1995.

SHAW AND GARLAN 1996. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

SMALL 1997. Christopher Small. A tool for constructing safe extensible C++ systems. *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS).* Portland, OR, USA, 16-20 June 1997. Berkeley, CA, USA. 1997. p. 175-84.

SMALL AND SELTZER 1996. Christopher Small and Margo Seltzer. A Comparison of OS Extension Technologies. *Proceedings of the USENIX Technical Conference 1996*, New Orleans, LA, Pages 41-54, January 1996.

STALLMAN 1984. Richard M. Stallman. Emacs: An extensible, customized, self-documenting display editor. *Interactive Programming Environments*, D.R. Barstow, H. E. Shrobe, and E. Sandewall, editors. McGraw-Hill, 1984. Pages 300-325.

STEYAERT ET AL 1996. Patrick Steyaert, Carine Lucas, Kim Mens and Theo D' Hondt. Reuse Contracts : Managing the Evolution of Reusable Assets. *Proceedings of OOPSLA 1996*. Also published in ACM SIGPLAN Notices, 31(10), pp.268-286. ACM Press, 1996.

SULLIVAN AND NOTKIN 1992. Kevin Sullivan, David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology.* vol. 1, no. 3, July1992.

SZYPERSKI 1996. Clements Szyperski. Independently extensible systems–software engineering potential and challenges. Proceedings of the *19th Australasian Computer Science Conference*, Melbourne, Australia, January 31- February 2, 1996.

TAYLOR ET AL 1996. Richard N. Taylor, Nenad Medvidovic, Ken M. Anderson, Jim E. Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A Component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, June 1996.

TEITELMAN AND MASINTER 1981. W. Teitelman and L. Manister. The InterLisp Programming Environment. *IEEE Computer*, vol. 14, no. 4, April 1981. Pages 25-33.

THOMAS AND NEJMEH 1992. Thomas and Nejmeh. Definitions of tool integration for environments. *IEEE Software*, vol. 9, no. 2, March 1992. Pages 29-35.

TIVOLI SYSTEMS INC. 1998. Applications Management Specification. http://www.tivoli.com/

VAN DER HOEK 1997. André van der Hoek, Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. Software Release Management. *Proceedings of the 6th European Software Engineering Conference,* (held jointly with SIGSOFT '97: Fifth International Symposium on the Foundations of Software Engineering), Lecture Notes in Computer Science 1301, Springer, Berlin, 1997. Pages 159-175.

WAHBE ET AL 1993. R. Wahbe, S. Lucco, T. Anderson, S. Graham. Efficient Software-based Fault Isolation. *Proceedings of the 14th ACM Symposium on Operating Systems Principles.* Asheville, NC. p203-216, December 1993.

WHITEHEAD ET AL 1995 E. J. Whitehead Jr., J. E. Robbins, N. Medvidovic, R. N. Taylor. Software architecture: foundations for a component marketplace. 1st International Workshop on Architectures for Software Systems. April 24-25, 1995.

WING 1990 J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, vol. 23m no. 9. Pages 8-24, September 1990.

WIRTH 1971. Nicholas Wirth. Program development by stepwise refinement. *Communications of the ACM*. vol. 14, no. 4, April 1971.

WIRTH 1995 Niklaus Wirth. A Plea for Lean Software. *Computer*, vol. 28, no. 2, February 1995.

YANG ET AL 1992. Wuu Yang, Susan Horwitz, and Thomas Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology.* vol. 1, no. 3, July 1992. Pages 310-354

YOUNG AND TAYLOR 1986 Michael Young, Richard N. Taylor. Rethinking the taxonomy of fault detection techniques. *Proceedings of the Eleventh International Conference on Software Engineering.* Pittsburgh, May 1989.

ZARMER 1992 C. Zarmer, B. Nardi, J. Johnson, J.R. Miller. ACE: Zen and the Art of Application Building. *Proceedings of the 25th Hawaii International Conference on System Sciences*, Kola, HI, January 7-10, 1992. Vol. II, pp687-698.

ZOBKIW 1995. Joe Zobkiw. *A Fragment of Your Imagination: Code Fragments and Code Resources for Power Macintosh and Macintosh.* Addison-Wesley Pub. Co., August 1995.

APPENDIX A: Programming Language and Event-based Open Point Comparison

During the course of system design and implementation, numerous design decisions are made. The principles of separation of concerns and modularity tell us that localized design decisions are easier to change than those that permeate the system. A design must ultimately be implemented using abstractions provided by implementation frameworks, such as programming languages, event mechanisms, and/or component infrastructures. Thus, the fidelity with which a design can be implemented depends on the abstractions provided by available implementation frameworks.

Programming languages and event mechanisms are commonly used implementation frameworks whose abstractions localize changes to the system implementation[1]. For example, a programming language's function abstraction encapsulates a behavioral specification that may be reused in other parts of the system implementation using function invocation. Because the function abstraction localizes a behavioral specification, changing the specification in one place affects all the parts of the system that invoke the function. Similarly, an event mechanism's message routing abstraction encapsulates how events propagate from participant to participant.

APIs, scripting languages, plug-ins, component-based, and source code DSE techniques are function-based and adopt programming language abstractions. We cover these in Section A.1 and summarize their open and closed points in Table A-1. Event-based techniques, in contrast, adopt event abstractions. We cover these in Section A.2 and summarize their open and closed points in Table A-2.

A.1 Function-based Approaches to Flexibility

Since different programming languages provide different abstractions, we focus on two popular object-oriented programming languages of the day: C++ and Java. The process we use here could just as easily be followed to evaluate other programming languages.

Given the C++ and Java programming languages, we enumerate the abstractions they provide below. Since we are only interested in changes that support behaviorally open DSE, we only enumerate programming language abstractions that would allow third-parties to add new program behaviors. Global constants and variables, name spaces, default function parameters, access modifiers, and the like are non-behavioral abstractions—they only support selection and combination of built-in behaviors. We also restrict ourselves to abstractions at or above the level of abstraction of functions—we do not consider intra-function

1. Of course, tools may be built that "localize" changes that would otherwise be distributed across the system. For example, the HTML language does not provide a construct for specifying common headers and footers across a collection of Web pages. Yet, Web site design tools let users specify common headers and footers that are systematically appended to every Web page when the site's HTML files are generated. Program restructuring and transformation tools provide similar capabilities for source code. For third-parties to make the same changes as easily as the original developers, a similar mechanism must be provided for them as well.

abstractions such as statement labels, critical sections, etc.—since none of the available DSE techniques support such fine-grained changes to program abstractions.

The behavioral abstractions in C++ and Java are:

- macro substitutions, template functions and classes[2], and file inclusion (C++ only): A C++ header file encapsulates function, type, and macro substitution definitions; changes to these definitions affect all the source files that include the header file using the "#include" directive of the C++ preprocessor. An approach supports *file inclusion open points* if changes to include files are propagated to all the modules that include the file. If changes are not thoroughly propagated, inconsistencies resulting from incompatible definitions would arise.
- function definition: Function definitions encapsulate behavioral specifications that may be reused throughout the system via function invocation. Changes to a function definition affect all invocations of the function. An approach supports *function definition open points* if changes to the executable behavioral specification of functions can be made, and if new functions can be defined.
- function binding: Function bindings determine the function definition that is bound to a function invocation. Changes to a function binding affect which function is invoked. An approach supports *function binding open points* if changes to a function binding can be made.
- function overloading: Function overloading determines which function definition is bound to a particular invocation based on the context in which it is used (Aho et al

2. Template functions are instantiated (i.e., expanded for each actual type) during compile-time in C++. The compiler treats them more like macro expansions than functions. See Lippman (1991, pages 197-198). Template classes appear to be handled in a similar fashion.

1986). Introducing or removing an overloaded function may alter function bindings if existing function invocations resolve to a different function definition. An approach supports *function overloading open points* if changes to overloaded functions cause existing function bindings to be reevaluated and rebound.

- variable-type binding: Variable-type bindings determine to which type a variable is bound. Changes to the type of a variable affect the set of operations that can be performed on the variable. An approach supports *variable-type binding open points* if changes to a type propagate to affect all the variables instantiated with that type. In C++, for example, if the definition of a type is changed to reorder existing data members or add new data members, all the source modules that declare variables of that type must be recompiled since the offsets for accessing each data member have changed.
- inheritance: Class inheritance relationships determine what behavior a class acquires from its parent classes. Changes to a class, such as adding or removing methods or altering the classes it inherits from, may affect the behavior of descendant classes. An approach supports *inheritance open points* if changes to a class are propagated to inherited classes.
- virtual functions: Virtual, or dynamically bound, functions allow function bindings to be determined during runtime based on the type of object being invoked. In C++, functions are statically bound unless they are declared "virtual". In Java, functions are dynamically bound unless they are declared "final". An approach supports *virtual function open points* if the introduction of new types or changes to existing types cause existing function bindings to be reevaluated and rebound.

- class definition: Class definitions encapsulate a collection of related behavioral specifications that modify a shared state. Changes to a class definition affect all uses of that class. An approach supports *class definition open points* if behavioral changes to classes can be made, and if new classes can be defined.
- class binding: Class bindings determine the class definition that is bound to a class reference. Changes to a class binding affect which class is referenced. An approach supports *class binding open points* if changes to this binding can be made.
- dynamic class loading (Java only): Dynamic class loading allows the definition of new classes during runtime. Java classes may optionally execute behavior when loaded (Gosling et al 1996). Hence, changing the set of dynamically loaded classes affects behavior. An approach supports *dynamic class loading open points* if changes to the set of dynamically loaded classes can be made.
- package location (Java only): Package locations are determined by Java's runtime system, which uses a class loader to retrieve class definitions. The default implementation of the class loader looks in each of the file system directories specified in the host machine's CLASSPATH environment variable. Changing the set or relative order of directories in the environment variable affects which class definitions are retrieved by the class loader. An approach supports *package location open points* if the set of classes retrieved by the class loader can be changed.
- interfaces (Java only): Interfaces determine which functions a class implements. Changes to the set of interfaces implemented by a class changes the type of class. An approach supports *interface open points* if the set of interfaces that a class implements can be changed.

- synchronized modifier (Java only): Synchronized modifiers determine whether or not access to data and function invocations are serialized in a multi-threaded environment. Changes to the synchronized modifier affect the behavior of execution threads that attempt to access the data or function simultaneously. An approach supports *synchronized modifier open points* if changes to the synchronized modifier of a function affect all invocations of the function.
- reflection (Java only): Reflection in Java allows classes, functions, and data fields to be inspected and allows new class, function, and variable bindings to be established. An approach supports *reflection open points* if such bindings can be established during runtime using reflection.

Note that these abstractions are particular to C++ and Java. Different abstractions may be provided in other programming languages or as a part of the runtime environment. For example, although C++ does not offer dynamic class loading, C++ programs utilizing CORBA can use CORBA's dynamic linking mechanism to achieve dynamic class loading.

Table A-1 summarizes the open and closed points of different DSE techniques. Since we are comparing approaches, not particular implementations, we take the most liberal interpretation of each approach. In this way, we convey *what is possible*, not what has been realized. Consequently, existing implementations of these approaches may or may not demonstrate all of these open points. In the case of C++, we assume the presence of a linker. The differences between C++ and Java in Table A-1 can be attributed to the fact that Java is a more dynamic language than C++.

Source code-based technique, such as open source, expose the entire behavioral specification of the application. Hence, all behavioral abstractions are open points.

	APIs	scripting languages	plug-ins	component-based	source code
macro substitution and file inclusion					✓
function definitions			only those exported by the add-on	yes, at the granularity of a component	✓
function bindings	strict addition from add-on to host only	strict addition from add-on to host only		add by adding new components, change and remove by replacing originating component	✓
function overloading				change by replacing referencing components	✓
variable-type bindings	strict addition from add-on to host only	strict addition from add-on to host only	no in C++; only with interface compatibility in Java	change by replacing referencing components in C++; only with interface compatibility in Java;	✓
inheritance			no in C++; yes in Java under certain circumstances	change by replacing referencing components in C++; yes in Java under certain circumstances	✓
virtual functions			no in C++; yes in Java under certain circumstances	change by replacing referencing components in C++; yes in Java under certain circumstances	✓
class definitions			no in C++; only those exported by the add-on in Java	change by replacing referencing components in C++; with interface compatibility in Java	✓
class binding	strict addition from add-on to host only	strict addition from add-on to host only		can change by replacing components only	✓
dynamic class loading (Java only)			only those initiated by the add-on	change by replacing components only	✓
package location (Java only)			✓	✓	✓
interface implementation (Java only)			yes under certain circumstances	yes without replacing components under certain circumstances, yes without restrictions by replacing components	✓
synchronized method modifiers (Java only)			✓	✓	✓
reflection (Java only)			✓	✓	✓

Table A-1: The open and closed points of different function-based DSE approaches. Empty cells represent closed points, checked cells represent open points, and cells with text describe restrictions on open points.

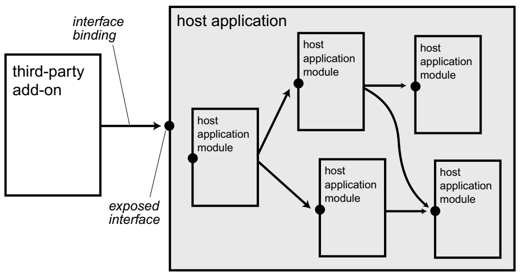


Figure A-1. API approach to decentralized software evolution.

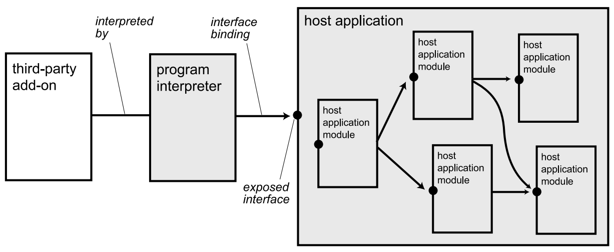


Figure A-2. Scripting language approach to decentralized software evolution.

APIs allow add-ons to bind to functions, types, and classes exposed by the host application and support the addition of bindings to those abstractions, where these bindings originate from the add-on and terminate at the host application (see Figure A-1). Thus, APIs support function binding, variable-type binding, and class binding open points; all other behavioral abstractions are closed points in APIs.

Scripting languages and APIs share the same set of open and closed points. This is because scripting languages just add a special-purpose language that is more suitable for end-users and integrate the development environment with the host application (see Figure A-2). Although this lowers the entry-barrier for writing add-ons, it does not introduce additional open points.

We contrast the plug-in and component-based approaches by examining each type of abstraction in turn. Abstract depictions of plug-in and component-based approaches are shown in Figure A-3 and Figure A-4, respectively. These figures will be used in the discussions that follow.

Function definitions. Plug-ins are the inverse of APIs. The host application expects a plug-in to implement a specific, predetermined interface, which the host invokes in a predetermined way (see Figure A-3). Plug-ins can implement and expose additional functions,

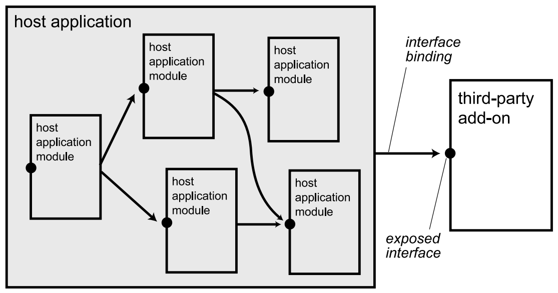


Figure A-3. Plug-in approach to decentralized software evolution.

but the host application (and other plug-ins) have no way to invoke such functions directly. Hence, plug-ins allow a restricted form of function definition open points. In component-based approaches, add-ons can replace function definitions and introduce new functions that other components can invoke. But since components are the smallest unit of change, the component containing the redefined function must be replaced. Thus, function definitions are open points in component-based approaches at the component-level.

Function binding and function overloading are closed points in plug-ins since function bindings, which always originate in the host application and terminate at the plug-in's interface, cannot be changed by plug-ins—they are bound to the abstract interface of the plug-in when the host application is compiled. In the component-based approach, function bindings are open points since new bindings are added when new components are added (in Figure A-4, bindings b_1 and b_2 are added when component w is introduced), and an existing binding can be changed or removed by replacing the component that originates the binding (in Figure A-4, binding b_3 can be changed by replacing component z). Function overloading open points are also supported if all the components that invoke the overloaded function are replaced (in Figure A-4, if a function in component z is overloaded by a function in component w, function bindings from components x and y to component z must be rebound).

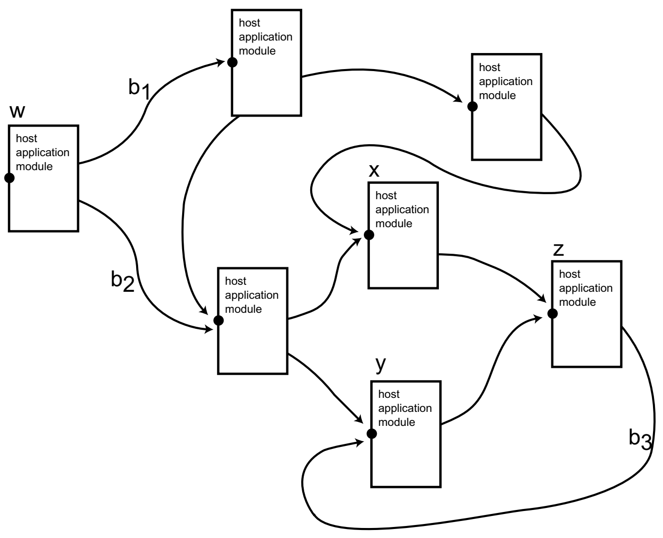


Figure A-4. Component-based approach to decentralized software evolution.

Variable-type bindings occur when an add-on exports a type that is bound to a variable declared in the host application. Since C++ calculates a type's object size, data member offsets, and function member offsets at compile time, a plug-in cannot change the type because it cannot re-compile the host application. Hence, variable-type bindings are closed points in plug-ins with C++. Similar problems plague component-based systems, though a variable-type binding can be changed if components that reference it are replaced. Hence, variable-type bindings are open points in components with C++. Java, on the other hand, stores symbolic references to public data and function members, and resolves them during runtime. Hence, plug-in and component-based approaches can change the exported type as long as type compatibility with the original type is maintained.

Inheritance and *virtual functions* in C++ are closed points for plug-ins and open points for component-based approaches when components that implement descendant classes are replaced. The reasoning is identical to variable type bindings. In Java, inheritance and virtual function bindings are also resolved at compile time with one exception.[3] Therefore, these abstractions are open points for plug-ins and component-based approaches in those situations where Java dynamically resolves inheritance bindings; otherwise they are closed points.

Class definitions in C++ are closed points for plug-ins and open points for component-based approaches when components that reference the class are replaced. The reasoning is identical to variable type bindings. In Java, class definitions that are exported by the plug-in are open points since bindings to the exported class are stored symbolically and resolves during runtime. Class definitions in component-based systems are open points as long as type compatibility with the original type is maintained.

Class bindings in plug-ins are closed points since class bindings, which always originate in the host application and terminate at the plug-in's interface, cannot be changed by plug-ins—they are bound to the abstract interface of the plug-in when the host application is compiled. In component-based systems, class bindings can be changed by replacing the components where the binding originates, and thus are open points.

Dynamic class loading (Java only) is an open point for plug-ins if the add-on initiates the dynamic loading of classes. Likewise, in component-based approaches, every class that is dynamically loaded can be changed by replacing the add-on that initiates it.

Package location (Java only) is an open point for both plug-ins and component-based systems since Java's class loader is localized and may be changed through the runtime system without modifying the host application or its add-ons.

Interface implementation (Java only) is similar to inheritance in Java—bindings are resolved at compile time with one exception. Therefore, interface implementations are open

3. Inheritance bindings to direct superclasses using "super" are resolved dynamically, otherwise they are static. For details, see binary compatibility in the Java Language Reference Manual (Gosling et al 1996, §13.4).

points for plug-ins and component-based approaches in those situations where Java dynamically resolves inheritance bindings; otherwise, they are closed points for plug-ins and open points for component-based systems when referencing components are replaced.

Synchronized method modification (Java only) is an open point for both plug-ins and component-based approaches because the behavior that implements it is localized to the function it modifies.

Reflection (Java only) is an open point for both plug-ins and component-based approaches because it is provided by Java's runtime system, making it independent of the host application and its add-ons.

A.2 Event-based Open Point Comparison

Barrett et al (1996) develop a generic event-based integration (EBI) framework that models the key components of event-based integration mechanisms and, consequentially, the set of abstractions provided by event-based systems (see Figure A-5). We use the abstractions they identify as the basis of our comparison of event-based approaches. Of the seven abstractions they identify, one, "messages", does not affect program behavior since messages represent units of information exchange by programs. For our evaluation, we assume that messages do not contain executable specifications.[4] Therefore, we can eliminate the message abstraction

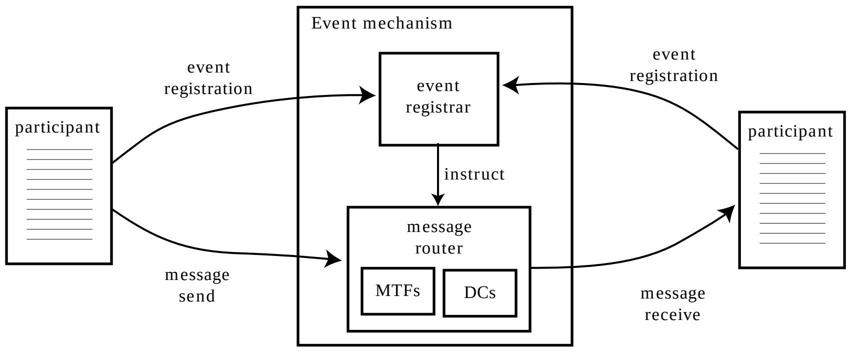


Figure A-5. Event-based integration framework, adapted from Barrett et al [1996] from our discussion since we are only interested in behavioral changes to programs. The behavioral abstractions provided by event-based systems are:

- Participants: Participants represent software modules that encapsulate a collection of related behavioral specifications. Participants use the event-based communication mechanism to exchange messages with other participants. When a participant receives a message, it may execute some behavior that may result in messages being sent to other participants. Changes to a participant affects all uses of that participant. An approach supports *participant open points* if behavioral changes to participants can be made and if the set of participants in an application can be changed.
- Event registration: Before a participant engages in message exchange, it must register its intent to do so with a registrar. As a part of this step, the participant may register its interest in receiving messages from particular participants or in receiving messages that match a particular pattern. Changes to a participant's event registration information may affect which messages it receives and how the messages it sends are processed by the event mechanism. An approach supports *event registration open points* if changes to the event registration information can be made.
- Message routing: Message routing involves delivering messages sent from one participant to one or more participants. The message routing agent that performs this task encapsulates decisions regarding who receives a sent message, how a message is transmitted, and whether or not any transformations or delivery constraints should be applied to a message. Presumably, message routing is informed by the event registration information provided by participants. Changes to message routing affects

4. We consider situations in which messages contain executable specifications as a combination of the event-based and scripting language approaches.

message transmission and message delivery, both of which may alter program behavior. An approach supports *message routing open points* if changes to message routing can be made.

- Groups: A group represents a collection of participants, an event registrar, and a message router such that all group members can be logically treated as a single entity. Changes to a participant's group memberships may affect the messages it receives. An approach supports *group open points* if changes to group memberships can be made.
- Message transformation functions (MTFs): Message transformation functions modify the content and/or the routing of messages within the message routing agent. For example, a MTF may filter a participant's outgoing messages. Adding, removing, or modifying MTFs may affect message contents and message delivery. An approach supports *message transformation open points* if MTFs may be added, removed, or modified.
- Delivery constraints (DCs): Delivery constraints define rules governing message delivery within the message routing agent. For example, a constraint may state that a message must be delivered to at most one participant, or that a message must be delivered in a certain amount of time. Changes to delivery constraints affect properties of message delivery, which may alter program behavior. An approach supports *delivery constraint open points* if changes to the delivery constraints within the router can be made.

abstractions	FIELD	Polylith	CORBA 2.0
participants	✓	✓	✓
event registration	yes by replacing add-ons or changing policies via the Policy tool	only implicitly by changing the add-on to channel mapping in the MIL	yes by changing the add-on's interface definition
message routing	change by replacing add-ons or changing policies via the Policy tool	yes by changing the add-on to channel mapping in the MIL	change by replacing add-on
groups	change by replacing add-ons	only implicitly by changing the add-on to channel mapping in the MIL	
message transformation functions	change by replacing add-ons, or by changing policies via the Policy tool	only implicitly by changing the add-on to channel mapping in the MIL	
delivery constraints	change indirectly by changing policies via the Policy Tool		change by replacing add-ons

Table A-2: The open and closed points of different event-based DSE mechanisms. Empty cells represent closed points, checked cells represent open points, and cells with text describe restrictions on open points.

Table A-2 summarizes the open and closed points engendered by three different event-based DSE mechanisms: FIELD (Reiss 1990, Reiss 1996), Polylith (Purtilo 1994), and CORBA 2.0 (OMG 1999). We evaluate the flexibility that these mechanisms engender in applications that employ them, not the mechanism itself. Consequently, we take the most liberal interpretation of each mechanism since we are interested in conveying *what is possible*, not necessarily what has been realized in applications built using the mechanism.
We contrast the three different mechanisms by examining each type of abstraction in turn.

Participants are open points with all three mechanisms since participants may be added, removed, or replaced. Since participants are the smallest unit of change, finer-grained changes require that the entire participant to be replaced.

Event registration is an open point with all three mechanisms, though each requires that different software artifacts be modified to effect change. FIELD offers two ways to change a participant's event registration: (1) use FIELD's Policy tool to add or remove message transformation functions that change the messages a participant receives; or (2) replace the participant since the participant's event registration is specified programmatically. In Polylith, a participant's event registrations are derived from the binding instructions that involve it in the application's MIL specification. In CORBA, a participant's event registrations are specified in the participant's IDL specification. Unlike FIELD, Polylith and CORBA can accommodate event registration open points without replacing participants.

Message routing is an open point with all three mechanisms, though, as before, each requires that different artifacts be modified. FIELD again offers two ways to change message routing: (1) use FIELD's Policy tool to add or remove message transformation functions that change how messages are routed among participants; or (2) replace all the participants affected by the change in message routing since message routing is specified programmatically. In Polylith, message routing is determined by aggregating all of the explicit binding instructions in the application's MIL specification. Hence, Polylith supports message routing open points without replacing participants. In CORBA, message routing is specified programmatically in the executable specification of participants and can evolve during runtime. Thus, message routing open points are supported in CORBA when all the participants affected by the change are replaced.

Groups and *message transformation functions* (MTFs) are open points in FIELD and Polylith. As before, FIELD offers two ways to change groups and MTFs: (1) FIELD's Policy tool can be used to add and remove MTFs, and MTFs can be changed such that the participant's event registration matches that of a group; or (2) replace the participant since it programmatically joins a group when it registers with the event mechanism and can programmatically add and remove MTFs. In Polylith, a participant's group membership and

MTFs are implicitly determined based on the binding instructions that involve it in the application's MIL specification. Hence, Polylith supports group open points without replacing participants. CORBA does not provide any mechanistic support for groups or MTFs.

Delivery constraints are open points in FIELD and CORBA, though to different degrees. FIELD's Policy tool only allows the priority of message delivery to be specified using MTFs. In CORBA, participants can programmatically select between two delivery constraints: "best effort", which does not cause an exception if the message cannot be delivered, and "at most once", which ensures that the message is delivered to at most one participant or an exception is generated. Hence, delivery constraints are open points in CORBA when participants are replaced. Polylith does not provide any mechanistic support for delivery constraints.