Software Architecture
Many faces, many places, yet a central discipline

Richard N. Taylor
University of California, Irvine
Ken Anderson, 1997
University of Colorado
Greg Bolcer, 1998
Keroseneandamatch.com
Peyman Oreizy, 2000
Launch21
Jim Whitehead, 2000
UC Santa Cruz
Roy Fielding, 2000
Day Software
Peter Kammer, 2004
Google, Inc.
Eric Dashofy, 2007
The Aerospace Corporation
Hazel Asuncion, 2009
UC Irvine (post-doc)
Justin Erenkrantz, 2009
The Apache Foundation
Lee Osterweil
UMass

Alex Wolf
Imperial

André van der Hoek
UCIrvine
Outline

How did I get here?

Just what IS architecture?

Some REST-ful thoughts

A note about research and publications

“Just one more thing”
How Did I Get Here?
I Love Design

Painting
Photography
Music
Software
Aiding Design
Aiding Design

First focus: aiding in the design of concurrent programs
Aiding Design

First focus: aiding in the design of concurrent programs
Aiding Design

First focus: aiding in the design of concurrent programs

A Concurrency Analysis Tool Suite for Ada Programs: Rationale, Design, and Preliminary Experience

A. Young
Purdue University
and
R. Taylor, D. Levine, K. A. Nies
University of California

Concurrency Analysis (Tool Suite) is designed as a toolkit of algorithms to analyze concurrent programs. It is intended to aid in the development of concurrent programs by analyzing potential synchronization errors. The tool suite includes four main components: automatic detection of synchronization errors, concurrency detection, parallelism detection, and concurrency verification. The tool suite is available as open-source software, and the authors encourage its use and feedback from the community.

Complexity of Analyzing the Synchronization Structure of Concurrent Programs

R. Taylor
Department of Information and Computer Science, University of California, Irvine, CA 92697, USA

Summary: Analyzing the complexity of concurrency structures is a fundamental problem in concurrent programming. This paper presents a model for analyzing the complexity of synchronization structures and discusses the implications of this analysis for concurrent programming.

Introduction

Verification and Concurrency Systems

Concurrency systems which employ multiple processors and tasks are used in a wide variety of applications. A concurrency system can be considered to be "embodied" when the computer system is a natural part of a larger physical system. For example, an embedded system may be a control system for a manufacturing process, or a distributed system may be a network of computers connected by a high-speed network. In these cases, the system must be designed to work well in the physical environment, and the tasks must be designed to be efficient and reliable.

* This work was supported in part by the National Science Foundation under grant NCR 75-1018, the U.S. Army Research Office under joint DAAG 04-01-C-0075, and by the National Science and Engineering Research Council of Canada under grant NSERC 20-02-18.
Environments
Environments

First focus: analysis and testing environments

An Integrated Verification and Testing Environment

RICHARD N. TAYLOR
Department of Information and Computer Science, University of California, Irvine
Irvine, California 92717, U.S.A.

SUMMARY
A verification and testing environment that includes static analysis, symbolic execution, and
dynamic analysis capabilities is presented. Tool integration and co-operation are promoted
through use of an intermediate program representation and a global system manager. A
substantially user interface aids application of the tools. Their use is guided by a verification
and testing methodology on which the system's design is based. The environment has been
engineered to support the production of flight control software written in HRSIS. The
environment itself is written in Pascal and is designed to be portable. Several development
experiences are described. The environment demonstrates that a strong, unified verification
and testing environment can be built; it serves as a basis for future investigations.

INTRODUCTION
During the past decade, a variety of software verification and testing techniques have
appeared, each seeking to aid the process of producing reliable code. Of course no
single technique is able to satisfy all the needs in this area; different techniques are
suitable to different aspects of the job. Study of the various techniques has led to some
important understandings, however. First, their relative efficacies and efficiencies have
become known. Second, it has become clear that many of the techniques have
complementary characteristics. That is, the strengths of one technique are able to
compensate for the weaknesses of another. These investigations have led to the
formulation of a verification and testing (V & T) methodology: a way of applying the
techniques to obtain their maximum benefit. Indeed their co-operative application
can result in a synergistic effect.

Study of the methodology and how it may be applied practically has led to another
understanding: not only do the techniques functionally complement each other, but
significant potential also exists for their physical integration. This paper reports,
therefore, on a V & T environment designed to provide the functionality of the useful
techniques, enable application of the V & T methodology, and promote efficient
practical application of both. The components and integration scheme of the
environment are described and a summary of the methodology is given. Some advances
in fundamental technology are noted also.

This environment stands in distinct contrast with early collections of testing and
verification tools. Foremost is the provision of direct support for a definite V & T
methodology. This methodology enables aspects of a program’s correctness to be

Received 7 June 1982
© 1983 by John Wiley & Sons, Ltd. Revised 3 November 1982 and 21 March 1983
First focus: analysis and testing environments

Second: programming environments
Environments

First focus: analysis and testing environments

Second: programming environments

Third: software development environments

Steps to an Advanced Ada Programming Environment

 Foundations for the Arcadia Environment Architecture

Richard H. Taylor, Francis C. Bolje, Larry A. Charles, Louis Ostrovsky
Richard W. Selby, Jack C. Winkels, Alexander L. Wolf, Michel Young

Department of Information and Computer Science
University of California, Irvine

Department of Computer Information Science
University of Massachusetts, Amherst

TRW
Redondo Beach, California

EATX Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ

Abstract

Early software environments have supported a narrow range of activities (programming environments) or else been restricted to a single "home" software development process. The Arcadia research project is investigating the construction of software environments that are tightly integrated, yet flexible and adaptable enough to support experimentation with alternative software processes and tools. This has led us to view an environment as being composed of two distinct, cooperating parts. One is the caretaker part, consisting of process frameworks and the tools and objects used and defined by these programs. The other is the used part, or infrastructure, supporting creation, execution, and change to the constituents of the caretaker part. The major components of the infrastructure are a process programming language and interpreter, object management system, and user interface management system. Process programming facilitates precise definition and automated support of software development and maintenance activities. The object management system provides typing, environment polymorphism, distributions, and concurrency control capabilities. The user interface management system mediates communication between human users and executing processes, providing (nearly) total access to all facilities of the environment. Research in each of these areas and the interactions among them is described.

1 INTRODUCTION

The purpose of a software environment is to support users in their software development and maintenance activities. Past attempts to do this have neglected the full scope and complexity of this problem. The Arcadia project is a consortium research effort aimed at achieving an unusually broad range of software environment issues. In particular, the Arcadia project is concerned with simultaneously investigating and developing prototype demonstrations of:

- environment architectures for organizing large collections of tools and facilitating their interaction with users as well as with each other,
- tools to facilitate the testing and analysis of concurrent and sequential software, and
- frameworks for environment and tool evaluation.

This paper describes the research rationale and approach being taken by Arcadia researchers in investigating environment architecture issues. Although details concerning the test suite and the evaluation framework are outside the scope of this paper, attempting to assemble these components into a coherent environment will provide a non-trivial test case for experimentally evaluating this architecture.

The remainder of this section presents a high-level overview of our proposed environment architecture. The major components of the architecture are described. Each of these represents a major research focus, some of whose component parts are often raised by the interactions among others. Indeed, it is the importance and complexity of these interactions that requires research in the subareas be pursued cooperatively. One of the novel features of the Arcadia project is that it is synergistically exploring many of these issues.

©1983 ACM 089791-256X/83/0011-0001 $1.50

1

Recommended by: Stuart Boyd

environment are described and a summary of the methodology is given. Some advances in fundamental technology are noted also.

This environment stands in distinct contrast with early collections of testing and verification tools. Foremost is the provision of direct support for a definitive V&T methodology. This methodology enables aspects of a program’s correctness to be verified. 1

0026-0664/83/0006-0007$01.75
Received 7 June 1982
©1983 by John Wiley & Sons, Ltd. Revised 3 November 1982 and 21 March 1983
First focus: analysis and testing

Second: programming environments

Third: software development environments

Abstract

The Ada project was a typical software development environment. This paper presents some lessons learned about developing effective software development environments. We describe some of the preconditions and constraints that need to be satisfied in the construction of an Ada-like development environment, and present lessons learned from developing Ada.

1. Introduction

The Ada project was a typical software development environment. This paper presents some lessons learned about developing effective software development environments. We describe some of the preconditions and constraints that need to be satisfied in the construction of an Ada-like development environment, and present lessons learned from developing Ada.

2. Ada Overview

Ada's architecture is the result of an extensive and intensive effort to design and build a highly productive software development environment. The Ada language was designed to support a wide range of programming styles and paradigms. It includes features such as operators, types, and type conversions, which are all supported in Ada. The language also includes features such as exception handling, which are not supported in Ada. Ada is a programming language, and the Ada environment is the set of tools and facilities that support Ada.

3. Lessons from the Ada Environment

The Ada environment is a typical software development environment. This paper presents some lessons learned about developing effective software development environments. We describe some of the preconditions and constraints that need to be satisfied in the construction of an Ada-like development environment, and present lessons learned from developing Ada.

4. Conclusions

In conclusion, we have presented some lessons learned about developing effective software development environments. We describe some of the preconditions and constraints that need to be satisfied in the construction of an Ada-like development environment, and present lessons learned from developing Ada.

R. Kato

References


Environments

First focus: analysis and test environments

Second: programming environments

Third: software development environments

Especially UI aspects

Steps to an Advanced Ada1 Programming Environment

Foundations for the Arcadia Environment Architecture

Issues Encountered in Building a Flexible Software Development Environment
Lessons from the Arcadia Project

R. Kadia

This paper presents some of the insights we have gained while building and experimenting with these components. We begin by briefly describing our goals and the principal components of Arcadia. In Sections 4 through 8 we discuss several key lessons that seem to have general applicability to a wide range of environment efforts. Many of these lessons were learned by repeatedly encountering similar problems and devising similar solutions in diverse technical areas. Finally, we summarize our lessons.

2 Arcadia Overview

Arcadia believes an effective software development environment (SDE) is a collection of capabilities effectively integrated to support software developers and managers. For us, to be effective an SDE must be extensible, incrementally improvable, flexible, fast, and efficient. Its components must be interoperable; it must be able to support multiple users and user classes; it must be easy to use, able to support effective product and process visibility, able to support effective management control, and it should be proactive.

Through the years, Arcadia has evolved an architecture that addresses these objectives simultaneously. We have learned, however, that these various design objectives are not orthogonal and often conflict. Much of the most challenging work of Arcadia has been concerned with understanding the various tensions between these diverse desires and devising strategies for supporting adjustable compromises between conflicting SDE objectives. The focus of this paper is on the tensions that arose and the lessons we learned in our attempts to alleviate these tensions.

Arcadia's architecture is the result of our efforts to simultaneously achieve all of the above objectives in the presence of the various tensions. Several of the devices used to mediate these tensions are described later in this paper. In this section we briefly describe the principal components that form the basis for our architecture and indicate why we believe they are important to the structure of any SDE that attempts to meet the...
Software Environment Architectures and User Interface Facilities

MICHAEL YOUNG, RICHARD N. TAYLOR, AND DENNIS B. TROPP

Abstract: User interface facilities are a crucial part of the interface structure of a software environment. We discuss the particular design and construction of a user interface management system for a software environment, and the interrelationships between facilities provided by the system and the environment in which it is embedded. We describe the design and implementation of the interface management system for an environment that supports a large-scale software development system.

I. INTRODUCTION

SOFTWARE environments provide automated support for groups of activities performed by software producers. An environment architect is the set of basic facilities underlying the capabilities of a spectrum of environments and the roles for assembling a specific environment out of these and other facilities. Environment architectures typically deal with issues such as:

- the storage and manipulation of persistent objects,
- user interface conventions,
- access to internal information and tools, and
- interactions with the user.

It is the last item which concerns us here: the basic facilities provided for interacting with the user and how these are bound up with other environment capabilities to produce interactive tools.

The user interface management system (UMIS) is an environment architecture in its own right. This paper focuses on the relationship between UMIS and environment architecture; the particular design and construction of software environments, and the interplay between UMIS issues and the other issues dealt above. This paper is about the architecture of UMIS: it is not about physical interaction techniques per se, nor about human factors in interactive programs. It describes a design which integrates several key ideas from UMIS research, adapting and extending experiences from earlier work.

Manuscript received January 15, 1988. This work was supported in part by the National Science Foundation under Grant CCR-87043, and in part by the Defense Advanced Research Projects Agency through the Computer Software Laboratory. Equipment and systems were developed at the Computer Research Laboratory of the University of California, Irvine. CA 92717.

II. CONTEXT: SOFTWARE ENVIRONMENT ARCHITECTURES

Over the past decade the term "environment" has been applied to a wide variety of systems. These have ranged from highly-complex combinations to systems intended to support group processes and projects. We use the term to denote a set of facilities designed to aid the user in the performance of some job, where the "job" here is a task which requires the use of one or more pieces of computer software. These tasks have come to be understood as software development and maintenance activities.

As such, the number of abstractions that can be supported is very large, as are the number and types of objects manipulated by the environment. The concrete system which drive the following discussion, therefore, are much broader than those associated, for example, with ordinary editor environments.

Furthermore, we get increased in integrated environments. From a user's perspective this means that certain properties of the environment hold. While this is true, and is examples in this paper, there can be much more to the notion of integration. From a tool (or environment) builder's perspective an environment is integrated if there are uniform schemes for tool invocation, and combining communication, data sharing, persistent object management, constraint maintenance, addition of new objects and facilities, and so on.

Another pertinent issue of ours is in extendible environments. In our context the most useful environments are ones in which it is possible to add new capabilities to address new needs, as in supporting new jobs. New development and analysis techniques appear regularly, it is only sensible to integrate them into an environment. Doing so can be difficult, however, unless an en-
Software Environment Architectures and User Interface Facilities

MICHAEL YOUNG, RICHARD N. TAYLOR, and DENNIS B. TROUP

INTRODUCTION

The user interface management system (UIMS) is an important aspect of an application program's user interface. This paper presents an overview of the user interface management system (UIMS) and discusses its role in software development and maintenance. The UIMS supports the user interface, providing a consistent and persistent user interface throughout the system.

The UIMS is designed to handle the user interface tasks of creating, modifying, and deleting user interfaces. It also manages the user interface components, such as buttons, menus, and dialog boxes. The UIMS is responsible for coordinating the activities of various components in the user interface, including the user interface editor, the user interface debugger, and the user interface animator.

The UIMS also supports the development of user interface components. It provides a framework for defining user interface components and their behavior. The UIMS supports the creation of user interface components, such as buttons and menus, and provides a mechanism for defining their behavior.

The UIMS is designed to be extensible, allowing new components to be added to the user interface. The UIMS also supports the creation of custom user interface components, allowing developers to create components that meet their specific needs.

The UIMS is designed to be portable, allowing it to be used on a variety of platforms. The UIMS is designed to be compatible with a variety of programming languages, including C, C++, and Java.

CONCLUSION

The user interface management system (UIMS) is an important aspect of an application program's user interface. The UIMS supports the development of user interface components, and provides a framework for defining user interface components and their behavior. The UIMS supports the creation of custom user interface components, allowing developers to create components that meet their specific needs. The UIMS is designed to be extensible, allowing new components to be added to the user interface. The UIMS is designed to be portable, allowing it to be used on a variety of platforms. The UIMS is designed to be compatible with a variety of programming languages, including C, C++, and Java.
A Component- and Message-Based Architectural Style for GUI Software

Richard Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Member, IEEE, Jason E. Robbine, Karl A. Nies, Peyman Oraizi, and Deborah L. Dubrow

Abstract—While a large fraction of application code is devoted to graphical user interfaces (GUI) functions, support for reuse in these domains has largely been confined to the reuse of "shell" widgets. We present a novel architectural style designed to support larger, granular reuse and flexible system composition. Moreover, the style supports design of distributed, concurrent applications. Dynamic, notification-driven messages and asynchronous required messages are the data links for inter- and intra-component communication. A key aspect of the style is that components are not built with any dependencies that are typically associated with conversational or event-driven components, such as user interface look and feel. Instead, all components are defined in the presence of any constraints that must be satisfied in order for a client to use them.

1 Introduction

Software architecture is a key design discipline [4], [10], [14], [16]. Many GUI programming tools are more than 20 years old, and the AI-architecture community has long been concerned with the use of user interface (UI) components. User interface software has typically used one of two primary architectural patterns: the event-driven or the model-view-controller (MVC) style. This paper presents a new architectural style, called the component-based style, that is designed to support the reuse of UI components by providing a graphical user interface metaphor, but the style does not support other types of applications. The style differs from the MVC in that it is more focused on the style’s usage. A key aspect of the MVC is that the style supports design of distributed, concurrent applications. Dynamic, notification-driven messages and asynchronous required messages are the data links for inter- and intra-component communication. A key aspect of the style is that components are not built with any dependencies that are typically associated with conversational or event-driven components, such as user interface look and feel. Instead, all components are defined in the presence of any constraints that must be satisfied in order for a client to use them.

The style supports a paradigm in which UI components, such as dialogs, structured graphic models of various levels of abstraction, and form-based menus, can more readily be re-used. A variety of other goals are potentially supported as well. These goals include the MVC to compose systems in which components may be written in different programming languages. Components may also be run concurrently on a distributed, heterogeneous environment without shared address space. Architectures may be changed at runtime. Multiple user interfaces may be interacting with the system. Multiple dialog boxes may be active and described in different terminologies, and multiple media types may be involved. We have not yet determined that all goals can be achieved at the same time.

The style is designed to be used as a network of concurrent components hooked together by message passing. The architectural style is based on the architectural style of service-oriented architecture (SOA) for component-based development (CBD) [17, 18]. A key motivating factor behind definition of the SOA style is the emerging need for the user interface community to use a more component-based development economy (CBD). User interface software frequently addresses a wide variety of application software, yet reuse in the UI domain is typically limited to similar design objects. The architectural style presented in this paper supports a paradigm in which UI components, such as dialogs, structured graphic models of various levels of abstraction, and form-based menus, can more readily be re-used. A variety of other goals are potentially supported as well. These goals include the MVC to compose systems in which components may be written in different programming languages. Components may also be run concurrently on a distributed, heterogeneous environment without shared address space. Architectures may be changed at runtime. Multiple user interfaces may be interacting with the system. Multiple dialog boxes may be active and described in different terminologies, and multiple media types may be involved. We have not yet determined that all goals can be achieved at the same time.

The style supports a paradigm in which UI components, such as dialogs, structured graphic models of various levels of abstraction, and form-based menus, can more readily be re-used. A variety of other goals are potentially supported as well. These goals include the MVC to compose systems in which components may be written in different programming languages. Components may also be run concurrently on a distributed, heterogeneous environment without shared address space. Architectures may be changed at runtime. Multiple user interfaces may be interacting with the system. Multiple dialog boxes may be active and described in different terminologies, and multiple media types may be involved. We have not yet determined that all goals can be achieved at the same time.

The style supports a paradigm in which UI components, such as dialogs, structured graphic models of various levels of abstraction, and form-based menus, can more readily be re-used. A variety of other goals are potentially supported as well. These goals include the MVC to compose systems in which components may be written in different programming languages. Components may also be run concurrently on a distributed, heterogeneous environment without shared address space. Architectures may be changed at runtime. Multiple user interfaces may be interacting with the system. Multiple dialog boxes may be active and described in different terminologies, and multiple media types may be involved. We have not yet determined that all goals can be achieved at the same time.

The style supports a paradigm in which UI components, such as dialogs, structured graphic models of various levels of abstraction, and form-based menus, can more readily be re-used. A variety of other goals are potentially supported as well. These goals include the MVC to compose systems in which components may be written in different programming languages. Components may also be run concurrently on a distributed, heterogeneous environment without shared address space. Architectures may be changed at runtime. Multiple user interfaces may be interacting with the system. Multiple dialog boxes may be active and described in different terminologies, and multiple media types may be involved. We have not yet determined that all goals can be achieved at the same time.

The style supports a paradigm in which UI components, such as dialogs, structured graphic models of various levels of abstraction, and form-based menus, can more readily be re-used. A variety of other goals are potentially supported as well. These goals include the MVC to compose systems in which components may be written in different programming languages. Components may also be run concurrently on a distributed, heterogeneous environment without shared address space. Architectures may be changed at runtime. Multiple user interfaces may be interacting with the system. Multiple dialog boxes may be active and described in different terminologies, and multiple media types may be involved. We have not yet determined that all goals can be achieved at the same time.
Software Architecture as a Field

If you look at the history of the research field, from Perry & Wolf and Shaw & Garlan onwards:

- lots and lots of activity
- mostly concerning the structure of software systems
- but also addressing other views of systems
Some of My Emphases
Some of My Emphases

Structure & Style
Some of My Emphases

Structure & Style
A Classification and Comparison Framework for Software Architecture Description Languages

Nenad Medvidovic and Richard N. Taylor, Member, IEEE Computer Society

Abstract—Software architecture analysis shifts the focus of developers from lines of code to coarse-grained architectural elements and their overall interaction structure. Architecture description languages (ADLs) have been proposed as modeling notations to support architectural analysis and development. These notations for architectural representation include such paradigms as object-oriented architecture (TOAST), software component architecture (SCA), and component-based software architecture (COBRA). This paper locates the attention on architectural languages that support the representation and configuration of software components and their interaction. Architecture description languages are compared in terms of their support for component modeling, component configuration, and graphical representation of architecture. The notations are classified as instance-based representation; they are: context modeling, component configuration, and collaborative tool support. The paper provides a classification framework for architectural languages (ADLs) and their strengths and weaknesses.

1 INTRODUCTION

Software architecture analysis shifts the focus of developers from lines of code to coarse-grained architectural elements and their overall interaction structure. Architecture description languages (ADLs) have been proposed as modeling notations to support architectural analysis and development. These notations for architectural representation include such paradigms as object-oriented architecture (TOAST), software component architecture (SCA), and component-based software architecture (COBRA). This paper locates the attention on architectural languages that support the representation and configuration of software components and their interaction. Architecture description languages are compared in terms of their support for component modeling, component configuration, and graphical representation of architecture. The notations are classified as instance-based representation; they are: context modeling, component configuration, and collaborative tool support. The paper provides a classification framework for architectural languages (ADLs) and their strengths and weaknesses.

Some of My Emphases

Structure & Style

Modeling (ADLs)
Some of My Emphases

Structure & Style
Modeling (ADLs)
Support environments
Abstract: Reuse of large-grain software components offers the potential for significant savings in application development cost and time. Successful reuse of components and component architectures depend both on the quality of the components reused as well as the software context in which the reuse is attempted. Disciplined approaches to the structure and design of software applications offer the potential of providing a comprehensive setting for each reuse.

The authors present the results of a series of exercises designed to determine how well off-the-shelf constraint solvers could be reused in applications designed in accordance with the C2 software architectural style. The exercises involved the reuse of SkyBlue and Amulet's one-way constraint solver Amulet's one-way constraint solver and constructed numerous variations of a single application, i.e., an application family. The family summarises the style and presents the results from the exercises. The exercises were successful in a variety of dimensions; one conclusion is that the C2 style offers significant potential for the development of application families.

1 Introduction

Software architecture research is directed at reducing the costs of developing applications and increasing the potential for reusability between different members of a closely related product family. One aspect of this research is the development of software architectural styles, canonical ways of organizing the components in a product family [1, 2]. Typically, styles reflect key properties of one or more application domains, and recurring patterns of application design within those domains. As such, architectural styles have the potential for providing structure for off-the-shelf (OTS) component reuse.

However, all styles are not equally well equipped to support reuse. If a style is too restrictive, it will exclude the world of legacy components. Conversely, if the set of style rules is too permissive, developers may be faced with all of the well documented problems of reuse in general [3-6]. Therefore achieving a balance, where the rules are strong enough to make reuse tractable but broad enough to enable integration of OTS components, is a key issue in formulating and adopting architectural styles.

Our experience with C2, a component- and message-based style for GUI software [7, 8] indicates that it provides such a balance. In a series of exercises, we were able to integrate several OTS components of various granularities into architectures that adhere to the rules of C2. This paper focuses on a subset of these exercises, in which we successfully integrated two externally developed OTS constraint solvers into a C2 architectural style.

In doing so, we were able to create several constraint maintenance components in the C2 style, enabling the construction of a large family of applications. We describe the details of these exercises and the lessons we learned in the present.

2 Overview of C2 architectural style

C2 is an architectural style designed to support the particular needs of applications that have a graphical user interface aspect. The style supports a paradigm in which UI components, such as dialogue structured around windows, can be managed by constraint solvers. As this paper will show, constraint managers, can more readily be reused. A variety of other goals are potentially supported as well. These goals include the ability to compose systems in which components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple access may be independent with the system, multiple toolkits may be employed, multiple dialogues may be active, and multiple media types may be involved.

2.1 Style rules

A more detailed exposition on the style is given in [8]. The C2 style can be informally summarized as a network of concurrent components linked together by connections, i.e., message passing devices. 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. There is no bound on the number of component or

Some of My Emphases

Structure & Style

Modeling (ADLs)

Support environments

Domain-specific software architectures
Some of My Emphases

Structure & Style

Modeling (ADLs)

Support environments

Domain-specific software architectures

Exploiting architectural style to develop a family of applications

N. Musshchikov
R. N. Taylor

Abstract: Reuse of large-grained software components offers the potential for significant savings in application development cost and time. Successful reuse of components and component architectures depends both on the quality of the components reused as well as the software content in which the reuse is attempted. Disciplined approaches to the structure and design of software applications offer the potential of providing a systematic setting for such reuse. This paper presents the results of a series of exercises designed to determine how well off-theshelf constraint solvers could be reused in applications designed in accordance with the C2 software architectural style. The exercises involved the use of SkyBlue and SkyBlue’s extension, SkyBlue/Artist, to develop various exercises. These exercises constructed numerous variations of a single application, i.e., an application family. The paper summarizes the style and presents the results from the exercises. The exercises were successful in a variety of dimensions, one conclusion is that the C2 style offers significant potential for the development of application families.

1 Introduction

Software architectural research is directed at reducing the costs of developing applications and increasing the potential for compatibility between different members of a closely related product family. One aspect of this research is developing of software architectural styles, contextual ways of organizing the components in a product family [1, 2]. Typically, styles reflect and leverage key properties of one or more application domains, and recurring patterns of application designs within those domains. As such, architectural styles have the potential for providing structure for off-the-shelf (OTS) component reuse.

However, all styles are not equally well equipped to support reuse. If a style is too restrictive, it will exclude the world of legacy components. Conversely, if the set of styles is too permissive, developers may be faced with all of the well documented problems of reuse in general [3-4]. Therefore achieving a balance where the styles are strong enough to make reuse tractable but broad enough to enable integration of OTS components is a key issue in formulating and adopting architectural styles.

Our experience with C2, a component- and message-based style for GUI software [7-8] (indication that it provides such a balance). In a series of exercises, we were able to integrate several OTS components of various granularities into architectures that adhered to the rules of C2. This paper focuses on a subset of those exercises, in which we successfully integrated two externally developed OTS constraint solvers into a C2 architecture.

In doing so, we were able to create several constraint maintenance components in the C2 style, enabling the construction of a large family of applications. We describe the details of these exercises and the lessons we learned in the process.

2 Overview of C2 architectural style

C2 is an architectural style designed to support the particular needs of applications that have a graphical user interface aspect. The style supports a paradigm in which UI components, such as drop-down menus, message boxes, and buttons, are seen as separate components as described in this paper we show, constraint managers, can be readily reused. A variety of other goals are potentially supported as well. These goals include the ability to compose systems in which components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple toolkits may be employed, multiple displays may be active, and multiple media types may be involved.

2.1 Style rules

A more detailed exposition of the style is given in [8]. The C2 style can be informally summarized as a network of concurrent components connected together by connections, i.e., message passing devices. 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. There is no bound on the number of components or
Some of My Emphases

Structure & Style

Modeling (ADLs)

Support environments

Domain-specific software architectures

Dynamic adaptation

Exploiting architectural style to develop a family of software

**An Architecture-Based Approach to Self-Adaptive Software**

Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf

**SELF-ADAPTIVE SOFTWARE**

Self-adaptive software requires high dependability, robustness, adaptability, and availability. This article describes an infrastructure supporting two simultaneous processes in self-adaptive software: system evolution, the consistent application of change over time, and system adaptation, the cycle of detecting changing circumstances and planning and deploying responsive modifications.

Consider the following scenario. A fleet of unmanned air vehicles undertakes a mission to disable an enemy airfield. Pre-mission intelligence indicates that the airfield’s one defended, and mission planning proceeds accordingly. While the UAVs are en route to the target, new intelligence indicates that a mobile surface-to-air missile launcher now guards the airfield. The UAVs autonomously replan their mission, dividing into two groups—a SAM-suppression unit and an airfield-suppression unit—and proceed to accomplish their objectives. During the flight, specialized algorithms for detecting and recognizing SAM launchers automatically upload and are integrated into the SAM-suppression unit’s software.

In this scenario, new software components are dynamically inserted into fielded, heterogeneous systems without requiring system restart, or indeed, any downtime. Mission replanning relies on analytics that include feedback from current performance. Furthermore, each replanning can take place autonomously, can involve multiple, distributed, cooperating platforms, and where major changes are demanded and require human approval or guidance, can cooperate with mission analysts. Throughout, system integrity requires the assurance of consistency, correctness, and combination of changes. Other applications for fleets of UAVs might include environmental and land-use monitoring, disaster traffic management, fire fighting, airborne cellular telephone relay stations, and damage surveys in times of war and disaster. How wasteful to construct, which a specific software platform for each new UAV application! Far better if software architectures can simply adapt the platform to the application at hand, and better yet, if the platform itself adapts to demand even while serving some other purpose. For example, an airborne sensor platform designed for environmental and land-use monitoring could prove useful for damage surveys following an earthquake or hurricane, provided someone could change the software quickly enough and with sufficient assurance that the new system would perform as intended.

Software engineering aims for the systematic, principled design and deployment of applications that fulfill software’s original promise—applications that retain full plasticity throughout their lifecycle and that are as easy to modify in the field as they are on the drawing board. Software engineers have pursued many techniques for achieving this goal: specification languages, high-level programming languages, and object-oriented analysis and design, among others. However, while each contributes to the goal, the core need still falls short.

Self-adaptive software will provide the key. Many disciplines will contribute to its progress, but wholesale advances require a system...
Architecture: A Matur(ing) Field
Architecture: A Matur(ing) Field

Plenty of fuzzy-headed notions in the ether, however

Awareness, understanding, and uptake uneven

Hence a personal challenge: draw the results together, present the field comprehensively, technically, deeply
Architecture: A Matur(ing) Field

Plenty of fuzzy-headed notions in the ether, however

Awareness, understanding, and uptake uneven

Hence a personal challenge: draw the results together, present the field comprehensively, technically, deeply

AND DON’T FORGET ABOUT THE SOURCE CODE!
A Comprehensive Textbook

732 pages
Designing
Modeling
Connectors
Implementation
Analysis
NFPs, Deployment, DSSAs, ...
A Comprehensive Textbook

732 pages
Designing
Modeling
Connectors
Implementation
Analysis
NFPs, Deployment, DSSAs, ...
But What IS Architecture?
But What IS Architecture?

Definition: A software system’s architecture is the set of principal design decisions made about the system.
But What IS Architecture?

Definition: A software system’s architecture is the set of principal design decisions made about the system.

Principal decisions
Whenever
Whoever
Wherever
But What IS Architecture?

Definition: A software system’s architecture is the set of principal design decisions made about the system.

Encompasses structural, functional, UI decisions
Encompasses internal and external architecture

Principal decisions
Whenever
Whoever
Wherever
If Architecture is the set of principal design decisions - by whomever, wherever, whenever, then it is the proper central focus of software development.

When explicit, carries value not only for the present, but the system’s future
So What?

SA becomes the basis for long-term intellectual control

It is the focal point for ensuring conceptual integrity

It is an anchor for supporting reuse: of knowledge, experience, designs, and code

It is a centerpiece for effective project communication

It provides an adequate technical basis for development and management of related variant systems
Opportunities and Challenges
Opportunities and Challenges

When explicit, carries value to all

But priorities vary, hence different models, visualizations, and tools needed

Interesting demands on tool support
Opportunities and Challenges

When explicit, carries value to all

But priorities vary, hence different models, visualizations, and tools needed

Interesting demands on tool support

The special value of explicit connectors

as a way of thinking; as a way of building
Opportunities and Challenges

When explicit, carries value to all

But priorities vary, hence different models, visualizations, and tools needed

Interesting demands on tool support

The special value of explicit connectors

as a way of thinking; as a way of building

Going to code, and with code

demands support for implementation mapping, frameworks, generation, …

and hence provides basis for dynamic adaptation
A Little Case Study
A Little Case Study

Where have architectures really mattered?
A Little Case Study

Where have architectures *really* mattered?
In what way have they actually mattered?
A Little Case Study

Where have architectures *really* mattered?

In what way have they actually mattered?

Case study: the World Wide Web
What is the WWW?
What is the WWW?
What is the WWW?
What is the WWW?
What is the WWW?

What is its structure “right now”?
What is the WWW?

What is its structure “right now”?
What source code do I read?
What is the WWW?

What is its structure “right now”?
What source code do I read?
Who is in control?
What is the WWW?

The architecture of the Web is wholly separate from the code that implements it.
What is the WWW?

The architecture of the Web is wholly separate from the code that implements it.

There is no single piece of code that implements it.
What is the WWW?

The architecture of the Web is wholly separate from the code that implements it.

There is no single piece of code that implements it.

There are multiple equivalent pieces of code that implement various parts of the architecture.
What is the WWW?

The architecture of the Web is wholly separate from the code that implements it.

There is no single piece of code that implements it.

There are multiple equivalent pieces of code that implement various parts of the architecture.

The stylistic constraints that constitute the Web’s style are not readily apparent in the code.
Hypertext Transfer Protocol -- HTTP/1.1

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers [47]. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1", and is an update to RFC 2068 [33].
Hypertext Transfer Protocol -- HTTP/1.1

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers [47]. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1", and is an update to RFC 2068 [33].
This document specifies a Uniform Resource Locator (URL), the syntax and semantics of formalized information for location and access of resources via the Internet. These representations are sometimes referred to as "HTTP/1.1", and is an update to RFC 2068 [33].
This document specifies an Internet standards track protocol for the
Internet community, and requests discussion and suggestions for
improvements.  Please refer to the current edition of the "Internet
STD 1" for the standardization state

This document specifies an Internet standards track protocol for the
Network Working Group
Request for Comments: 1738
obsoletes: 2068
December 1994

T. Berners-Lee
MIT/LCS
R. Fielding
U.C. Irvine
L. Masinter
Xerox Corporation
August 1998

RFC 2616
HTTP/1.1
Request for Comments: 2616
June 1999

T. Berners-Lee
MIT/LCS
R. Fielding
U.C. Irvine
L. Masinter
Xerox Corporation
August 1998

RFC 2396
URI
Request for Comments: 2396
August 1998

T. Berners-Lee
MIT/LCS
R. Fielding
U.C. Irvine
L. Masinter
Xerox Corporation
August 1998
ICSE 2000

Network Working Group
Request for Comments: 2616

Principled Design of the Modern Web Architecture

Roy T. Fielding and Richard N. Taylor
Information and Computer Science
University of California, Irvine
Irvine, CA 92618-4525 USA
(949)482-4121
(fielding@ics.uci.edu)

Abstract

The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an Internet-scale distributed hypertextual system. The modern Web architecture emphasizes scalability of component interaction, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and maintain legacy systems. This paper introduces the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the Hypertext Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to align those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit responsibilities between the existing protocols and the applications they are intended to support.

Keywords: software architecture, software architectural style, WWW

Introduction

At the beginning of our efforts within the Internet Engineering Task Force to define the existing Hypertext Transfer Protocol (HTTP/1.0) [5] and design the extensions for the new standards of HTTP/1.1 [10] and Uniform Resource Identifiers (URI) [6], we recognized the need for a model of how the World Wide Web (WWW, or simply Web) should work. This abstract model of the interactions within an overall Web application, what we refer to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which those interactions are shaped.

A software architecture determines how system elements are identified and allocated, how the elements interact to form a system, the amount and generality of communication needed for interaction, and the interfaces protocols used for communication. An architectural style is an abstraction of the key aspects within a set of potential architectures (implementations of the style), encapsulating important decisions about the architectural elements and the allowed relationships among these elements within any architecture that conforms to the style.

REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and modifiability of component implementations. This is achieved by plugging constraints on connector semantics where other styles have focused on component semantics. REST satisfies the caching and reuse of interactions, dynamic scalability of components, and processing of events by intermediation, thereby meeting the needs of an Internet-scale distributed hypertextual system.

The modern Web is one instance of a REST-style architecture. Although Web-based applications can include access to other styles of interaction, the central focus of its protocol and performance concern is distributed hypertext. REST elaborates only those portions of the architecture that are considered essential for Internet-scale distributed hypertextual interaction. Areas for improvement of the Web architecture can be seen where existing protocols fail to express all of the potential semantics for component interaction, and where the details of syntax can be replaced with more efficient forms without changing the architectural model. The modern Web architecture is intended to allow the Web to REST to see if they fit within the architecture; if not, it is more efficient to replace them with more applicable architectural styles.

This paper presents REST after the completion of six years’ work on architectural standards for the modern (post-1992) Web. It does not present the details of the architecture itself, since these are found within the standards. Instead, we focus on the functionality for URI. To understand what is a valid URI, both the grammar and an associated description have to be studied. Some of the functionality described is not applicable to all URI schemes, and some operations are only possible when certain media types are retrieved using the URI, regardless of the scheme used.

A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource. This document defines the generic syntax of URI, including both absolute and relative forms, and guidelines for their use; it revises and replaces the generic definitions in RFC 1738 and RFC 1630.

This document defines a grammar that is a superset of all valid URI, such that an implementation can parse common components of a URI reference without knowing the scheme-specific requirements of every possible identifier type. This document does not define a generative grammar for URI; that task will be performed by the individual necessities of each URI scheme.

Apache

The Apache Server Project

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires specific permission and/or fee. C 1998 ACM 1-58113-250-3/98/0007...

407

Abstract

A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource. This document defines the generic syntax of URI, including both absolute and relative forms, and guidelines for their use; it revises and replaces the generic definitions in RFC 1738 and RFC 1630.

This document defines a grammar that is a superset of all valid URI, such that an implementation can parse common components of a URI reference without knowing the scheme-specific requirements of every possible identifier type. This document does not define a generative grammar for URI; that task will be performed by the individual specifications of each URI scheme.

T. Berners-Lee
MIT/LCS

R. Fielding
UC Irvine

L. Masinter
Xerox Corporation
August 1998

Uniform Resource Identifiers (URI): Generic Syntax

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and comments for its

Apache

Saenir Protocol (HTTP)

Protocol, collaborative
version, stateless, pr
is use for hypertext
management systems,
or codes and header
ation of data repr
ently of the data b

by the World-Wide

Berners-Lee, et. al.

Standards Track

RFC 2396

URI Generic Syntax

August 1998

?/1.1", and is an update to RFC 2068 [33].
ABSTRACT

The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an Internet-scale distributed hypermedia system. The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and interoperability to reduce interaction latency, enforce security, and accommodate legacy systems. In this paper, we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the HyperText Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to satisfy those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.

Keywords: software architecture, software architectural style, WWW

1 INTRODUCTION

At the beginning of our efforts within the Internet Engineering Task Force to define the existing HyperText Transfer Protocol (HTTP) [1], and design the extensions for the new standards of HTTP/1.1 [10] and Uniform Resource Identifiers (URI) [15], we recognized the need for a model of how the World Wide Web (WWW, or simply Web) should work. This idealized model of the interactions within an overall Web application, in which we refer to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which the Web scales to an Internet-scale distributed hypermedia system. This model has been identified and validated prior to deployment.

A software architecture describes how system elements are identified and allocated, how the elements interact to form a system, the amount and granularity of communication needed for interaction, and the interaction constraints used for communication. An architectural style is an abstraction of the key aspects within a set of potential architectures (instantiations of the style), encompassing important decisions about the architectural elements and the relationships among those elements within any architectures that conforms to the style. REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time minimizing the independence and flexibility of component implementations. This is achieved by placing constraints on computer semantics where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of events by intermediation, thereby meeting the needs of an Internet-scale distributed hypermedia system.

The modern Web is one instance of a REST-style architecture. Although Web-based applications can include other styles of interaction, the central focus of its protocol and performance concerns is distributed hypermedia. REST elaborates only those portions of the architecture that are considered essential for Internet-scale distributed hypermedia systems. For improvement of the Web architecture to be seen where existing protocols fail to express all of the potential semantics for component interaction, and where the details of syntax can be trivialized for other styles of interaction without changing the architecture. The REST architectural style is intended to help avoid these mismatches, while at the same time allowing other architectural styles to fit well within the Web architecture framework. In this paper, we introduce REST to see if they fit within the Web architecture. If so, it is more efficient to reduce that functionality to a system running in parallel with a more applicable architectural style.

This paper presents REST after the completion of six years’ work on architectural standards for the modern (post-1993) Web. It does not present the details of the architecture itself, since those are found within the standards. Instead, we focus on the general principles and foundations that we believe will underlie any future Web architecture, and we compare and contrast the properties of REST to other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.
Principled Design of the
Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding
CHAPTER 5

Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

5.1 Deriving REST

The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web’s constraints. Additional constraints can then be applied to form a new architectural style that better reflects the desired properties of a modern Web architecture. This section provides a general overview of REST by walking through the process of deriving it as an architectural style. Later sections will describe in more detail the specific constraints that compose the REST style.
Principled Design of the

Roy T. Fielding
Information and
University of
Irvine, CA 92
1-3490
(fielding@uiuc)

ABSTRACT

The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an Internet-scale distributed hypermedia system. The modern Web architecture emphasizes scalability, component interaction, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. In this paper, we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the Hypermedia Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit relationships between the existing protocols and the applications they are intended to support.

Keywords
software architecture, software architectural style, WWW

1 INTRODUCTION

At the beginning of our efforts within the Internet Engineering Task Force to define the current Hypermedia Transfer Protocol (HTTP) [47] and design the extensions for the new standards of HTTP 1.1 [5] and Uniform Resource Identifiers (URI) [5], we recognized the need for a model of how the World Wide Web (WWW or simply Web) should work. This idealized model of the interactions within an overall Web application, what we refer to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which it is to be used in the deployment of Web applications. The architectural style consists of a set of constraints that must be satisfied to implement a REST system.

5.1 Deriving REST

The design rationale behind the Web architectural style consisting of the set of constraints applied to the URI and HTTP by examining the impact of each constraint as it is identified the properties induced by the Web’s constraints, and be applied to form a new architectural style that better reflects Web. This section provides more detail the specific constraints that compose RESTful Web Services.

CHAPTER 5

Representational State Transfer

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles.

76
Themes
Themes

Technical disciplines having an influence

Networking

Hypertext

Open hypermedia

Software architecture

Information retrieval
Themes

Technical disciplines having an influence

Networking

Hypertext

Open hypermedia

Software architecture

Information retrieval

The movement of people
Themes

Technical disciplines having an influence

Networking

Hypertext

Open hypermedia

Software architecture

Information retrieval

The movement of people
Themes

Technical disciplines having an influence

Networking

Hypertext

Open hypermedia

Software architecture

Information retrieval

The movement of people

“When I ask you a question, you give me the right answer” -- Roy Fielding to Dick Taylor, 1994
Themes

Technical disciplines having an influence

Networking

Hypertext

Open hypermedia

Software architecture

Information retrieval

The movement of people

The social milieu Roy was in
Themes

Technical disciplines having an influence

Networking

Hypertext

Open hypermedia

Software architecture

Information retrieval

The movement of people

The social milieu Roy was in
Themes

Technical disciplines having an influence

Networking

Hypertext

Open hypermedia

Software architecture

Information retrieval

The movement of people

The social milieu Roy was in
Themes

- Technical disciplines having an influence
  - Networking
  - Hypertext
  - Open hypermedia
  - Software architecture
  - Information retrieval
- The movement of people
- The social milieu Roy was in
Themes

Technical disciplines having an influence

Networking

Hypertext

Open hypermedia

Software architecture

Information retrieval

The movement of people

The social milieu Roy was in
The REST Principles

R1. Information is a resource, named by a URL
R2. Representation of a resource is a set of bytes, plus meta-data describing them
R3. All interactions are context-free
R4. Only a few operations available, and act in accordance with the other REST principles
R5. Idempotent operations and representation meta-data support caching
R6. Presence of intermediaries is promoted
The REST Principles

R1. Information is a resource, named by a URL
R2. Representation of a resource is a set of bytes, plus meta-data describing them
R3. All interactions are context-free
R4. Only a few operations available, and act in accordance with the other REST principles
R5. Idempotent operations and representation meta-data support caching
R6. Presence of intermediaries is promoted

These are the principal design decisions. This* is the core architecture of the WWW.
The as-built architecture varies from the as-designed
The as-built architecture varies from the as-designed
The principles do not suffice to describe today’s Web applications
The as-built architecture varies from the as-designed

The principles do not suffice to describe today’s Web applications

Crystallizing these principles (largely by Justin) came while writing the textbook
The as-built architecture varies from the as-designed

The principles do not suffice to describe today’s Web applications

Crystallizing these principles (largely by Justin) came while writing the textbook

“Hypertext” doesn’t appear in the REST Principles, per se

If the representation is a hypertext then ...
Research, Funding, Pubs
Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others
Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others.

Should ease of publication direct what you do?
Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

The first paper on HTML and the Web was rejected by the 1991 ACM Hypertext conference
Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

The first paper on HTML and the Web was rejected by the 1991 ACM Hypertext conference

DARPA continually challenged my funding of Roy’s and Jim’s work (until it was a success!)
Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

The first paper on HTML and the Web was rejected by the 1991 ACM Hypertext conference

DARPA continually challenged my funding of Roy’s and Jim’s work (until it was a success!)

Roy took 9 years to get his Ph.D
Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

The first paper on HTML and the Web was rejected by the 1991 ACM Hypertext conference

DARPA continually challenged my funding of Roy’s and Jim’s work (until it was a success!)

Roy took 9 years to get his Ph.D

Exploratory development in SE research should be valued
“Just One More Thing”

Isn’t all this stuff about the REST principles just trying to dress up development work that happened over a decade ago? Make “architecture” look good? Just trying to get more mileage out of old work?
What Can You Do If...

You think through the crystallized REST principles,
See where they fail in modern applications, and then
Make an elegant generalization?
Computational REST (CREST)

C1. A resource is a locus of computations, named by an URL.

C2. The representation of a computation is an expression plus metadata to describe that expression.

C3. All computations are context-free.

C4. Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.

C5. The presence of intermediaries is promoted.
Demo

A dynamic, customizable news reader
Exemplary CREST peer

Weak CREST peer
Demo FAQ’s

What’s going on in the browser?

“Just graphical rendering” Dojo and Javascript for drawing only

What are those visible widgets?

(1) Artists (2) Highly restricted CREST computations on restricted peers

What’s being transported across the network?

HTML and JSON as weak CREST expressions describing state changes in the underlying computations

What’s the transport protocol? HTTP/1.1

How many CREST computations were there? ≈12

How many nodes/servers were active? 2 laptops, 2 phones, Mac mini

How was time fudged? Mini had stored one month of RSS feed data

What kind of compatibility w/ existing infrastructure?

Completely backwards compatible with all WWW browsers & servers
The Demo, redux

The point: analysis of the essential architectural decisions of the WWW, followed by generalization, opens up an entirely new space of decentralized, Internet-based applications based on computations as the fundamental entity.
CREST Credits

Justin Erenkrantz
(Final defense: next Thursday)

Michael Golick

Demo support:

Yongjie Zheng

Alegria Baquero
Thank you!