Dagstuhl Seminar 9508 on

Software Architecture

Feb. 20 - 24, 1995, Schloß Dagstuhl, Germany

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213
USA

Walter Tichy
School of Informatics
Univesity of Karlsruhe
D-76128 Karlsruhe
Germany

Frances Paulisch
Siemens AG
ZFE T SE 2
D-81730 Munich
Germany

Table of Contents

1. Introduction **4**

Position Papers::

2. Architectural Modelling and Representation **5**
3. Requirements and Architecture **9**
4. Patterns **11**
5. Domain-specific Software Architectures **14**
6. Architecture Extraction and Views **15**
7. Architectural Evolution **16**
8. Tools **18**

Working Groups:

9. Architecture and Implementation **20**
10. What are the Limitations of Architecture? **21**
11. Commercial Aspects of Software Architecture **22**
12. Heterogeneity, Incompleteness, and Inconsistency **24**
13. Dynamic Aspects of Software Architecture **25**
14. Balance between Formal and Informal Descriptions **26**

15. Closing Remarks **27**

Extended Abstract

For large, complex software systems the design of the overall system structure — or software architecture — emerges as a central problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; scaling and performance; and selection among design alternatives.

The architectural level of software design has become the focus of an increasingly important body of research and development in areas such as graphical design notations, codification of common architectural patterns, module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, architectural handbooks, and formal models of component integration mechanisms.

While there is broad-based agreement on the importance of software architecture, there have been few opportunities for researchers in the area to meet specifically on this topic. This international seminar on software architecture provided such a forum. The goal was to bring together both industrial and academic workers on the leading edge of software architecture, critically assess the state of the art, discuss novel approaches to leveraging architecture in software development and maintenance, and to set out promising directions for future research.

More specifically the workshop focused on the following topics:

Definitions: Discussion of what software architecture is and what it is not, with the goal of exploring the dimensions of the field and understanding where there is agreement on core issues.

Description: How do/should we describe and represent architectural designs?

Analysis: What kinds of properties do we care about and how can we determine them from our descriptions?

Design Guidance: Given set of non-functional requirements, how can we choose an appropriate software architecture? How can we take advantage of existing design experience?

Methods: How can we incorporate architectural design into the broader arena of software development practices? How can we accommodate different paradigms in software architecture?

Domain-Specific Software Architectures (DSSA): What can we learn from achievements in the area of DSSA, in particular from application frameworks, reference architectures, and component generators.

Formal Underpinnings: To what extent are/should formal models of architectures, mathematical foundations for modularization and system composition, formal characterizations of non-functional properties, theories of architectural connection used? What is the right level of formalism appropriate for software architecture?

Bridges: How can we link architectural abstractions with implementation structures and with problem requirements? What is the relationship between software architecture and evolution? What is the relationship between software architecture and design?

Tools and Environments for Architectural Design: What is the current state-of-the-art/state-of-the-practice for analysis tools, design environments, and component and application generators?

Extraction of Architectural Design from Existing Systems: How can we capture architectural knowledge and convey it to people in a useful way? In particular, addressing descriptions of exemplary architectural designs, reinterpretation of architectural descriptions in simple terms, and unification of related architectural designs.

Component Integration Mechanisms: What component integration mechanisms are available, e.g. module interconnection, mediation, wrappers, other novel composition techniques?

Software Architecture Case Studies and Experience Reports: What can we learn from retrospective analyses of industrial architectural development and model problems?

Cost/Benefit Trade-offs: What are the cost/benefit trade-offs in using architectural notations, formalisms, patterns, tools? How can we get the most "out" for reasonable effort "in" ?

Education: How can we teach architectural skills to others? How can we learn from practitioners? Can we lessen the gap between industry and academia?

The remainder of this document presents short position statements of most of the participants as well as reports from the working groups. In both cases the major points made in the plenary discussions are listed after the contribution in typewriter font.

List of Participants

Bruce Anderson ..... University of Essex, Colchester, UK
Robert Balzer ..... USC - Marina del Rey, CA, USA
Barry Boehm ..... Univ. California - Los Angeles, CA, USA
Lou H. Coglianese ..... Loral Federal Systems. - Owego, NY, USA
Joëlle Coutaz ..... IMAG - Grenoble, FR
David Garlan ..... CMU - Pittsburgh, PA, USA
William Griswold ..... Univ. California - San Diego, CA, USA
Richard Helm ..... DMR Group, Montreal, CAN
Larry Howard ..... SEI - Pittsburgh, PA, USA
Paola Inverardi ..... IEI-CNR - Pisa, IT
Michael Jackson ..... MAJ Consulting - UK
Jeff Kramer ..... Imperial College - London, UK
Balachander Krishnamurthy ..... AT&T Bell Laboratories - Murray Hill, NJ, USA
Peter Löhr ..... Freie Universität Berlin - Berlin, FRG
David Lamb ..... Queens University - Kingston, CAN
Hausi A. Müller ..... University of Victoria - Victoria, B.C., CAN
Oscar Nierstrasz ..... Universität Berne - Berne, CH
David Notkin ..... University of Washington - Seattle, WA, USA
Frances Paulisch ..... Siemens - München, FRG
Dewayne Perry ..... AT&T Bell Laboratories - Murray Hill, NJ, USA
Rubén Prieto-Díaz ..... Reuse Inc. - Fairfax, VA, USA
Michael D. Rice ..... Wesleyan University, Middleton, CT, USA
Hans Rohnert ..... Siemens - München, FRG
Bo Sandén ..... George Mason Uni. - Fairfax, VA, USA
William Scherlis ..... CMU - Pittsburgh, PA, USA
Robert Schwanke ..... Siemens Corp. Res. - Princeton, NJ, USA
Mary Shaw ..... CMU - Pittsburgh, PA, USA
Dilip Soni ..... Siemens Corp. Res. - Princeton, NJ, USA
Walter Tichy ..... Universität Karlsruhe - Karlsruhe, FRG
John Vlissides ..... IBM - Yorktown Heights, NY, USA
Bernhard Westfechtel ..... RWTH Aachen - Aachen, FRG
Reinhard Wilhelm ..... Universität Saarbrücken - Saarbrücken, FRG
Alexander Wolf ..... University of Colorado - Boulder, CO, USA

1. Introduction

Introductory Remarks

David Garlan
Carnegie Mellon University, Pittsburgh, PA, USA
garlan@cs.cmu.edu

While it is unlikely to be productive at this point to argue for a specific definition of software architecture, it is generally recognized that as the size of software systems increases, the design problem goes beyond the algorithms and data structures of the computation. Designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include the gross organization and global control structure; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; scaling and performance; and selection among design alternatives. This is the architectural level of design.

While the application of good architectural design is becoming increasingly important to software engineering practice, the fact remains that much of common practice leads to architectural designs that are informal, ad hoc, unanalysable, unmaintainable, and handcrafted. This has the consequence that architectural designs are only vaguely understood by developers; that architectural choices are based more on default than solid engineering principles; that architectural designs cannot be analysed for consistency or completeness; that architectures are enforced as a system evolves; and that there are virtually no tools to help the architectural designers with their tasks.

One of the key ingredients of an improved discipline of software architecture is the ability to accurately model architectural descriptions. Ideally these descriptions should take us above the level of implementation concerns typically found with existing module interconnection formalisms. That is, rather than portray the code artifacts needed to build a given system, an architectural description should expose the significant computational elements and their interactions. Moreover, the description should be able to inform the reader of the intended constraints on the design and their rational for selection. Finally, it is important to be able to model not only the architectures of specific systems, but the architectural styles and patterns (such as "pipeline", "client-server", and "blackboard") that form the emerging vocabulary of software architects.

Balzer: That definition is right, but it's missing a lot. It gives an impression that it's all worked out. Are the problems any different than what we would have said about software design 5-10 years ago? Are we in the "same soup"? Garlan: yes, but points out progression assembler, high-level languages, MIL, and that SWA, Patterns, Design Reuse is next step.

Lamb: During my experience in teaching industry, I have noticed a strong resistance to "formal" approaches. The industrial customers prefer solutions which can be done using whiteboards by "mortals" (i.e. not experienced software architects).
Notkin: I agree. You have to "buy into" the idea and the trade-off between the “startup” cost of the formal method and the associated benefit will be examined.
Wolf: But at some point, one must be precise and formal (like building architect who is required by law to put the design into a physical form eventually).

2. Architectural Modelling and Representation

A Hybrid Architecture for Tools that Manipulate Complex Representations

William G. Griswold
University of California, San Diego, La Jolla, CA, USA
wgg@ccs.ucsd.edu

Maintaining the consistency of multiple program representations — such as Abstract Syntax Trees and Program Dependence Graphs — in a program manipulation tool is difficult. I designed a hybrid software architecture for a meaning-preserving restructuring tool. Layering is one architectural paradigm, successively providing increasingly constrained and easier to use interfaces to implement restructuring transformations. To provide separation of representations, I also adopted the paradigm of keeping the implementations of data abstractions separate throughout the layering. To further increase the separation of representations, a mapping abstraction was placed between the program abstractions to provide the mappings between isomorphic data elements.

This hybrid architecture was crucial in separating the issue of achieving the basic function from later overcoming performance problems, which are common layered in layered systems, in particular, two of the data abstractions could be independently extended to fix the performance problem. The lower layers — and one data abstraction — were completely unaffected by the extension. The upper layers are now a little harder to use to implement a new transformation. However, the guidance and separation provided by the rules of the layers and data abstractions minimizes the negative impact of this trade-off.

Wolf: points out that this change has violated the original "clean" layering scheme. Griswold agrees, but thinks the benefit is worth the "cost" in uncleanliness (because the trade-off of better performance is thus realized and the "unclean" interface was not exported). Of Garlan's list of Problems, Griswold finds the ad hoc derivation one to be of greatest concern. Furthermore he states that without a good architecture, the remaining issues are irrelevant. Griswold, however, questions whether formalism is the right/only answer.

Industrial Software Architecture with Gestalt

Robert Schwanke
Siemens Corporate Research, Princeton, NJ, USA
rschwanke@scr.siemens.com

Practising architects tell us that their most pressing architectural concern is maintaining consistency between the architecture and the code. To our customers, "consistency" means structural, or topological, consistency. To fill the need for maintainable architectures, the Gestalt project seeks to develop a module architecture language that:

- captures the structure of large, complex systems
- can be rigorously, automatically checked for consistency with the code
- supports those kinds of abstraction that are most needed by current practitioners
- is general enough to support most types of communication mechanisms that are used in industrial software
- is suitable for re(verse) engineering as well as forward engineering
- provides an evolution path toward more sophisticated architecture languages and methodologies in the future.

The principle features of the Gestalt language are:

- multi-level decomposition into components, where each leaf component represents a set of language modules, as in Ada, C++, or Chill
- connectors between components, representing paths of communication
- multiple ports on components, representing different interfaces to other components.
- composite connectors, made out of components and connectors, to conceal implementation details of complex communication mechanisms
- mechanisms to support virtual machine layering, structural exceptions, incompleteness, and separate sub-architectures

Kramer and Inverardi: question the lack of dynamic aspects.
Schwanke: responds by saying he agrees that dynamics aspects are important and that they are at least partially reflected in the structure which is "good enough".

Notkin: Is this for forward or reverse engineering? How much can be automated?
Schwanke: It is for both. It is not an automatic process, but a lot of the shallow exhaustive searches can be done with simple tools once you know what you are looking for (e.g. mailboxes usually have some keywords associated with them and not just a number).

Components and Design in Concurrent Systems Architecture

Klaus-Peter Löhr
Freie Universität Berlin
lohr@info.fu-berlin.de

My interest is in understanding the architecture of concurrent systems, in particular as resulting from object-oriented design. I have the feeling that not even sequential systems architecture is well understood. We know that these are vastly different designs even for toy problems, yet we have no scientific framework for comparing them, let alone for speaking about design quality and giving guidelines for good design. I believe that without a firm understanding of small-scale design issues (patterns) and medium-scale design issues (component interconnection paradigms and topologies) progress in large systems architecture may remain limited.

I brought to Dagstuhl a few ideas about accommodating heterogeneous interconnection paradigms and about restricting the dependencies among components of a subsystem. Regarding interconnection, I would like to maintain that may of the so-called client/server interactions actually are manifestations of imperative glue code acting on a component — which helps to simplify architectures by differentiating refinement from component parameterization.

It turned out that the spectrum of interests among the seminar participants was much wider than I had anticipated. So it was intriguing to see the quite different connotations people associated with terms like "architecture", "dynamics", "protocol" or "view", and I learned a lot from this. On the other hand, most participants seemed to shun detailed technical discussions on patterns, object-orientation, design quality, etc., even in the subgroups. Subjects like legacy code, reverse engineering, living with heterogeneity, were more prominent.

Conclusion: I am going home with a much broader perspective than I brought along, better equipped for pursuing the topics I brought along — and feeling a more urgent need to pursue them.

```
Garlan: points out that the problem is often that the "glue" (as opposed to the components)
changes. This suggests that one needs to be able to reason about the interactions between
components in an explicit way.

Anderson: claims that components are not valuable on their own; only within a framework.
"Reusable components as such is a red herring".

Balzer: sees it as a "binding issue".  What we have done incorrectly is to separate the
specification of an application and architecture, and allow one to change and not the other
```

A Composition Language

Oscar Nierstrasz
University of Bern
oscar@iam.unibe.ch

We are working towards the development of a composition language that will support the specification of flexible, open applications as compositions of plug-compatible software components. The key characteristic of open systems is that they should easily accommodate changing requirements. We propose that, from the point of view of the software technology, open systems evolution can best be supported by the use of *component frameworks* to define flexible application architectures, and the use of a *composition language* to make component frameworks and applications explicit and manipulable.

A component framework is a generalization of an object-oriented framework, defining the kinds of components, collaborations, interactions and generic behaviours for a class of applications. A composition language can be used to specify both the abstractions of a component framework as well as the actual compositions of specific applications. The high-level specification of an application as a composition of components makes application architectures explicit and manipulable. The key features of a composition language are *objects* and *components*. An object is a potentially concurrent run-time entity that provides a set of services through some well-derived interface (possibly entailing adherence to a protocol), whereas a component is an abstraction over the object space that can be used to compose object systems. These features must be integrated in a composition language according to a common semantic foundation. We are simultaneously developing such a foundation and developing prototypes to test the practical application of these ideas.

```
Wolf/Notkin: Reuse is rarely systematic. Reused software rarely fits.
```

Helm: Frameworks provide "hooks" to extend the system, i.e. the ways to extend are designed in ahead of time, but in fixed ways.
Several: There exists a trade-off between generality and power; maybe we need a special-purpose composition language.
Balzer: Another place for leverage is the tool infrastructure. It's hard to use application frameworks because their use imposes lots of constraints, but that is also the source of their power.

What is Style?

David Garlan
Carnegie Mellon University, Pittsburgh, PA, USA
garlan@cs.cmu.edu

One of the central aspects of software architecture is the use of *styles* to define certain classes of architecture. In general, we view an architectural style as having four key properties. A style:

- defines a *vocabulary* of design elements
- imposes *configuration* constraints on its systems
- determines a *semantic interpretation* that gives meaning to the system descriptions
- enables a collection of *analyses* that can be performed on its systems.

An important question is what kind of *models* should we use to characterize and exploit architectural styles. Three plausible answers are:

- Style is a formal language
- Style is a system of types
- Style is a theory

Each of these has certain advantages and we conclude that they represent complementary aspects of the problem.

Balzer: You are talking about style in the sense of a formal set of rules, but not "preferences". This is "all" (i.e. the bounds of the allowable space).
Garlan: Yes, there is a difference between "legal" and "good" elements, but this is not a level of distinction I want to make at this point.
Balzer: There is a notation of describing styles. Behind that notation needs to be analysis tools. These tools require a set of formalisms that those analyses might make use of.

Notkin: What leverage do we get from what we write down. We often ask the question in the wrong order.

Formal Aspects of Software Architecture

Michael D. Rice
Wesleyan University, Middletown, CT, USA
mrice@uts.cs.wesleyan.edu

My lecture described a model for the incremental construction of hierarchical and modular software systems based on a family of Z schema type definitions which are invariant across all applications. A particular application is described by specifying values for generic parameters and adding application-specific declarations and constraints to the basic schemas. The schema types provide an "architectural semantics" that

- specifies the meaning of interfaces between system components, and
- specifies the meaning of the encapsulation process for constructing a new component from a system of components.

Additional schema types specify fundamental operations, so the model; supports an incremental design process and the reconfiguration of a software architecture. In addition, the model permits the description of each component's execution semantics using either process algebra or other specifications based on the interface port structure. Further research issues to be investigated are

- the effects of local changes of components and connections
- the accommodation of multiple views
- consistent maintenance of global system information

Lamb: Why do you care about the method of design strategies?
Rice: Was trying to model systems that had that kind of thing in the system.

Lamb: The problem with design strategy is that the system is complete and some axioms don't apply.

Garlan: There are aspects of architecture different in modelling than with respect to traditional programming languages; difference between module interconnection language and architectural description language. In programming language the glue is fixed; In architecture world the glue is not fixed.

Inverardi: The problem of static and dynamic specifications is well known. From our point of view the dynamic aspects are very important for software architecture.

Concurrent Software Architecture

Bo Sandén
George Mason University
bsanden@gmu.edu

There is more to software architecture than components and interconnections. In many real-time systems, concurrent processes largely define the architecture. A premise of my work is that the concurrent software structure can be based on intuitive concurrency in the problem domain. However, such concurrency is undefined, especially if the problem domain is viewed in terms of discrete events.

A definition of concurrency in discrete-event environments is proposed that is both intuitive and suitable to be modelled in software: An *occurrence* is a unique happening at a specific time. A *thread* is a rule defining a set of occurrences that are separated by $\delta>0$ time units. Given a *trace* of occurrences, the goal is to find a *thread model* by attributing each occurrence to exactly one thread. A thread model is *minimal* if all the threads have simultaneous occurrences.

A thread maps onto a task (process) in the software. Each task will have time to process each occurrence in its thread. A set of tasks based on a minimal thread model contains no redundant tasks, that is no tasks execute in lockstep or sequentially, one after the other.

Balzer: if see set of traces from operating environment there exists no way of seeing concurrency, but clearly want this to understand the problem. Concurrency inherently exists, and you want to be able to express it.
Sanden: Given a timed trace there is no inherent way to do this. Time is a real number so no two different things can happen at exactly the same time. Instead "simultaneous" is defined as "arbitrarily close in time"

Chemical Abstract Machine

Paola Inverardi Universita di L'Aquila, Italy inverardi@smaq20.univaq.it
Alexander Wolf, University of Colorado, Boulder, CO, USA alw@cs.colorado.edu

We are exploring a novel approach to formally specifying and analysing software architectures that is based on viewing software systems as chemicals whose reactions are controlled by explicitly stated rules. This powerful metaphor was devised in the domain of theoretical computer science by Banâtre and L Métayer and then reformulated as the Chemical Abstract Machine, or CHAM, by Berry and Boudol. The CHAM formalism provides a framework for developing operational specifications that does not bias the described system toward any particular computational model and that encourages the construction and use of modular specifications at different levels of detail. We illustrate the use of the CHAM for architectural description and analysis by applying it to two different architectures for a simple, but familiar, software system, the multi-phase compiler.

Shaw: In selecting a formalism the overriding consideration is to figure out what one is trying to express. You chose a different way.
Wolf: We want to see if existing ones are useful before developing a new one.
Shaw: An alternative ordering would be to determine what is necessary to express and then evaluate on that basis.
Wolf: The driving choice is the dynamics. We want to be able to express different architectural styles in one formalism. Our experience is that the chemical abstract machine is a flexible and powerful operational formalism.
Shaw: How far should we be taking this analogy with molecules and chemical reactions?
Wolf: You can go too far. Making an analogy between membranes and interfaces for example is OK. But trying to make sense of valence is probably going too far.
Shaw: What do you get out of the chemical analogy that is not just from term rewriting? What is different?
Inverardi: The CHAM gives an operational model, in terms of expressions (the molecule/solutions) and of rules to transform them. Term rewriting is just a way to implement the transformation.

Through the notion of molecule we can for example define the notion of "granularity". There is a semantics of the dynamics. The set of rules can be applied concurrently.

Separation of component and interaction design

Reinhard Wilhelm (with Martin Alt)
Universität des Saarlandes
wilhelm@cs.uni-sb.de

In the CoSy model, compiler components called engines are designed independently of the way they interact. Different potential interaction schemata can be used to coordinate the engines constrained by the way the engines access the compiler's data structures. These goals are achieved using a generative approach. Several specification languages specify different aspects of the compiler:

- FSDL (full structure definition language) defines the compiler's data structure and possible access patterns (views) of engines.
- EDL (engine description language) specifies the type of access of the different engines to the data structures.
- CCL (compiler configuration language) defines platform characteristics and other constraints on the compiler configuration.

The FSDL-compiler generates the program managing the data structures and the access functions to the data structures. The EDL-compiler derives a dependency graph from the EDL specification. In the EDL-specification, the interaction of the different engines is specified by composing engines with interaction schemata (pipeline, fork, data parallel, speculative). Each of these interaction schemata has a precise semantics. Hence, each use of a schema can be statically checked against the data dependencies. For each interaction schema, synchronization code is generated by the EDL compiler depending on whether a sequential or a shared memory parallel compiler organization is wanted. CoSy was developed in the ESPRIT project Compare.

3. Requirements and Architecture

Multi-view Implementation

Michael Jackson
jacksonma@attmail.com, mj@ic.ak.uk

The problems we tackle are multi-faceted, and must be decomposed into many parallel subproblems, parallel architectures, parallel views, parallel patterns. These parallel abstractions are pinned together by common phenomena, of which each abstraction requires a different description.

Quite separately, it has been observed that the architecture, and patterns, should be explicitly preserved in the implementation.

My conclusion is that we should work towards the kinds of implementation infrastructure that would support multiple, superimposed, architectures, and the multiple, superimposed, typing of elementary phenomenon that pin these architectures together.

Balzer: There's a lot of information not shown in the diagram. Do you have a notation for that rest? Jackson: No such notation exists.

Shaw: Concerning the problem decomposition — does there exist a calculus to show the correspondence? Jackson: There is no calculus. Different and parallel abstractions of the problem exist and in each of these abstractions you are concerned with some subset.

Howard: Is the way of putting the pieces back together inherent in how they were carved up? Jackson: You have a full description of the problem. There is no need to put the problem frame back together in any sense at all. That is, problem frames are not hierarchical. Furthermore, the decomposition into a hierarchy of procedures is a very poor way to go about solving a problem.

Balzer: There are two reasons for decomposition.
1) purely in problem space — how do I get assurance that all parts of the problem description is specified.
2) if all parts of the subproblem have been resolved then all parts of the entire problem have been resolved.
Is that a valid conclusion in that problem calculus?

Shaw: When we look at problems and recognize subproblems, we must consider both 1) the completeness question, and 2) the possibility of interference or incompatibility between subproblems. Jackson: There is no compatibility requirement between subproblems.

Müller: There is a desire to make these problem frames disjoint. You say they overlap in one form or another. What are the advantages of the overlap?
Jackson: One reason is that the subproblems are known to be solvable; decomposition into subproblems that are not solvable is pointless.

Balzer: Concerning putting it all together into a consistent whole. There may not be a calculus, but you must believe that it works somehow.
Jackson: You are making the assumption that you make a chunk of software for each of the boxes, but it is not like that. Implementation as a hierarchy of procedures is not the right way. Procedures can't be combined with conjunctions.
Shaw: Sometimes decomposition allows you to decompose into solvable chunks which can further break down the structure.

Architectures as a Key to Reasoning About Software System Attributes

Barry Boehm
University of Southern California, Los Angeles, CA, USA
boehm@sunset.usc.edu

The software field has been wrestling for over 20 years with the slippery set of issues involved with reasoning about software system attributes or non-functional properties (portability, modifiability, reliability, efficiency, cost, etc.). Up to now, the bases for reasoning have been source code (frustrating in its lack of big-picture information) and requirements (frustrating in their lack of implementation specifics).

Software architecture provides the prospect of capturing both the big-picture and the implementation specifics, but also the risk that it wouldn't provide enough of either to be helpful. I was encouraged that several architectural frameworks (styles, patterns, unit operations, domain architectures) give evidence that the optimistic outcome is at least relatively achievable, even though full knowability of attribute trade-offs will never be achievable.

Balzer: Are they [the trade-offs?] knowable in advance?
Boehm: They are risk-reducable in advance. Some organizations have institutionalized such "pathfinders" to smoke out such issues (e.g. benchmark scalability).

Garlan: Few commercial systems tell you realistically what non-functional attributes are supported.
Boehm: Yes, and there may be a difference between the in-house developers and the marketing people.

Notkin: What about the granularity? You are talking about unit operations, but what can you say about these properties as they relate to the structure of the whole system?
Boehm: The mapping is not so reliable. It is a complex computation.
Notkin: It's not clear to me what the algebraic structure of that joint calculation is and do any of those pluses and minuses apply any more?
Boehm: It is very complex; maybe within a specific domain it may be possible.

Boehm: Shows Strawman Attribute Engineering Process
Tichy: It's naive to think that people will evaluate three architectures when they have difficulty finding one!
Garlan: We teach a course on software architecture at CMU and characterize two or three architectures to a particular problem and evaluate the trade-offs. It is effective at raising the level of expertise in architectural design.
Shaw: We grade them by evaluating the quality of the reasoning.
Anderson: When I discuss design with my students, I ask that they tell me two alternatives that they considered but did not choose.

Domain Models and Software Architectures

Rubén Prieto-Díaz
Reuse Inc., Fairfax Virginia, USA
70410.3014@compuserve.com

Research in software architectures has focused mainly on identifying, classifying, and characterizing the solution space. My position is that if we identify, classify, and characterize the problem space we will be able to improve the engineering process of matching problems to solutions. Engineers will characterize new problems in terms of

smaller problems for which there are solutions making the engineering process one of composition and integration rather than one of creation from scratch.

The goal of this proposal is to make a case for the need to develop domain models for varieties of problem domains and characterize their solutions in the software domain (i.e., solution space) utilizing software architectures and/or existing solutions. These domain models and the suggested solutions could be presented in a catalog.

It is at the core of finding suitable matches between problems and solutions that the proposed catalog would bring the most benefit. One of the main problems of such catalog is scoping the multitude of problems. There may be thousands of problems that map into one solution. Classification of problem domains is a suggested first step to manage complexity. By analysing shared attributes among application areas, one can identify commonalities among them. Areas of common functionality are good candidates for investigating possible generic architectures. Reuse Inc. has a prototype tool for domain analysis (DARE) that can be used for identifying generic architectures. This tool is based on the principle that domain experts can propose (i.e. postulate) an architecture (DARE provides an architecture editor for this task) which is revised and validated with information extracted and abstracted by DARE from existing systems.

```
Nierstrasz: What is the granularity of your architecture and how do you deal with various
granularities?  E.g. office information systems is too big; payroll and accounting may have many
similarities.
Shaw: It's very useful to define domains so that they match the structure of the problem and not
just those that match the marketplace.  Power to leverage across product lines.
Lamb: Many systems consist of things that would be categorized in terms of multiple domains e.g.
switching systems both routing and billing.
```

4. Patterns

A Pattern-Oriented Approach to Software Architecture

Frances Paulisch
Siemens AG
paulisch@zfe.siemens.de

Software architecture is a growing area within software engineering. Traditionally, (building) architecture involves a mix of form and function and in software architecture we see an analogous trend of increasing concern with the structure (i.e. form) of the software as opposed to solely its function. Also, there is a trend towards explicit consideration of non-functional aspects such as performance, extensibility, fault-tolerance,... to aid in determining a suitable structure of the system (based on the trade-offs).

A pattern describes a particular recurring design problem and presents a pre-defined scheme for its solution by specifying its different parts as well as their collaboration and responsibilities. In addition, a pattern comprises rules for when and how to create it. Patterns for software architecture exist in various ranges of scale and abstraction. An *architectural framework* expresses a fundamental paradigm for structuring a system (e.g. the model-view-controller to separate the functional core from its control and from its presentation to the user). A *design pattern* describes a basic scheme for structuring subsystems and components of a software architecture as well as their relationships (e.g. Mediator-Worker Pattern which facilitates the decoupling of otherwise strongly connected components (the "workers") by introducing a special agent model to act as a "mediator").

We collect useful patterns from existing systems and describe them in a uniform way (structured prose and sketches). We find this to be a valuable way to communicate useful design knowledge among software architects.

Proxy-Original: an example of a design pattern

Hans Rohnert
Siemens AG, Munich, Germany
rohnert@zfe.siemens.de

We are writing a book on 'Pattern-oriented Software Architecture' which will, among other things, present a collection of software design patterns. In the following I will give a very succinct description of one particular pattern, the Proxy-Original Pattern.

Rationale: In a software system it is not always possible to access a component directly. Additional actions are to

be performed in order to control the access to a component or its functionality, e.g., checking access rights or handling multiple references to it. These actions, however, are neither related to the primary subject matter of the original component nor to the ones of its clients. Thus, there is a need for a structure allowing to handle such actions separately and independently from the responsibilities of the original components and its collaborators.
Applicability: The Proxy-Original design pattern is applicable whenever there is a need for a representative for a component in a different address space (remote proxy), for a component that should be loaded on demand (virtual proxy), or for a protected access to a component (protected proxy).
Description: Basically, the Proxy-Original design pattern separates special operations necessary for a correct access of a component from its implementation as well as from its clients. The design pattern consists of three participants. The first component, the proxy, represents a client's interface to the original component and is responsible to ensure a correct access to it. In other words, a proxy can be seen as a placeholder for the original component within a software system. The second participant, the original, represents the original component which is now guarded by the proxy. Each proxy component refers to a single original and in many cases for each original exactly one proxy exists.
Dynamic Behaviour: The typical behaviour of a Proxy-Original structure starts with a client's request for a service from a proxy. The proxy itself checks the correctness of this request, e.g., the access rights of the client, and may perform additional actions to access the original safely, e.g., creating the original or loading it from a data base. Then, the request is forwarded from the proxy to the original. The results of the service are returned from the original to the client via the proxy. However, before passing the control back to the client, the proxy component may perform further actions on the original, e.g., calling its destructor or storing it into a data base.
Examples:

• The OMG-CORBA uses the Proxy-Original design pattern for two purposes. So- called Client-Stubs or IDL-Stubs guard clients against the concrete implementation of their servers and the Object Request Broker. IDL-Skeletons are used by the Object Request Broker itself to forward requests to concrete remote server components.

• The Proxy-Original design pattern can be used for the implementation of reference counting for shared resources. In this case, it is often useful to introduce a separate proxy for each client in order to be explicitly independent from the existence of further clients.

Patterns in Practice

Richard Helm
DMR Group, Montreal, Canada
Richard.Helm@dmr.ca

Patterns describe recurring solutions to problems in a context. Patterns have been applied to telecommunications, banking, network management, O/S kernels. Our experience in using patterns are:

- Patterns do work. They seem to apply across domains of application, and across technologies. This is heartening.
- Patterns need to be applied carefully. You must realize that they provide solutions to particular problems but if these are not your problems, then solutions you may create may not be solutions to your problems.
- Patterns provide a common vocabulary that increases design velocity, and spans teams and management boundaries.
- A common vocabulary, however, can be exclusionary
- Patterns help find the "key objects/abstractions" for reuse earlier than they would otherwise be.
- Methodologies have much to say about what to do but little to say about what has been done. Patterns say much about what has been done but little about what to do. How does one exploit experience in the face of method? How does method take into account experience? Experience and method must be reconciled.

Finally, the study of software architecture, the description of software architecture, the evolution of software architecture can not be done isolated from practice. Practice guides theory and description. Theory and description feed back to practice.

Reverse Architecture

John Vlissides
IBM T. J. Watson Research
vlis@watson.ibm.com

Expert architects, be they traditional building architects or software architects, have ingrained skills that non-experts lack. Recovering, characterizing, and recording these skills can have a profound impact on the productivity

of experts and non-experts alike.

Design patterns are a medium for capturing such skills. But distilling experience and writing it up as design patterns is and to do well. It involves studying a variety of systems and finding snippets of design that have in common. It also involves characterizing these recurring designs and expressing them in a succinct and understandable way. Put simply, design pattern development involves generating *experience* from *designs*. This is just the opposite of what an architect does — namely generating *designs* from *experience* — hence the term "reverse architecture", of which design pattern development is just one example.

Here is a brief summary of lessons I've learned in doing this kind of reverse architecture:
Discovering patterns in software involves:

- incremental reflection and archiving
- examining and learning from other systems

These produce the raw materials of patterns. Once you have the raw materials, you should:

- decide on a structure
- be concrete early and often
- keep patterns distinct and complementary
- create an effective presentation (good typography, writing style)
- iterate (i.e. no closed-form solution to pattern development)
- get and incorporate feedback
- use your patterns, and have others do likewise (the acid test)

Community and Technology in Software Architecture

Bruce Anderson
University of Essex, Colchester, Essex CO4 3SQ, England
bruce@essex.ac.uk

We are faced with the need to produce software under demanding requirements such as low cost, short time-to-market and high reliability. We must do this in a competitive environment. A powerful way to make these improvements is to shift from a product focus to a product-family focus, and from a design focus to an architecture focus. An architecture is a generic product, or product generator, that is specific enough to be useable without much further design, but general enough to cover a product range.

There are barriers. We have to recognise that systems are delivered, not finished, and get to grips with realistic lifecycle models. The technical and organizational cost of generating excellent architectures is high, and is paid early in the product-family cycle, so all levels of a software-producing organisation will have to work in a new way. A key move is to integrate practice, research and learning. We need to move from "technology transfer" to "research integration" in the development of software practice. The very phrase "technology transfer" implies some chasm dividing "ideas" from "work". Researchers can never produce the required results while their work is separated from practice. Practitioners can never reap the full potential of change if they leave things to researchers. A few years ago I began a project to explore ideas at the architectural level by working with others in an attempt to create an Architecture Handbook. We were soon led to realise that a handbook is an expression of the knowledge, understanding and agreement of a community, that the knowledge does not exist in the abstract. Valuable handbooks currently exist at the project and division level, but we do not yet have a wider community of agreement.

A handbook can be a vehicle for creating understanding and consensus - the writing of the recent book on OO Design Patterns was catalysed by the Architecture Handbook effort, and is already encouraging the use of a common vocabulary, and the creation of local pattern collections.

There is more than just a shift of focus to architecture happening - there is a shift to valuing and exploring the knowledge of practitioners, and a shift to learning, so that we constantly extend that knowledge, leveraging our experience into new products, new architectures, and new conceptions of architecture.

General Discussion on "Patterns"

Shaw: How nicely will patterns interact with each other, which interact with which, mix and match or used in creating specific orders and combinations. The matrix of pattern-combinations for Alexander is fairly sparse? Does the same apply to software design patterns?

Löhr: How much is tied to OO?
Vlissides: The ideas are not, but the book is.

Westfechtel: How large is the set of patterns?
Vlissides: There are three broad categories (creational, structural, behavioural) and various kinds of cross indexes between them.

Notkin: I observed the patterns people discussing the OO-design of Bill Griswold's application and was impressed with the speed of communication among those who use patterns. And for 80% of the terms I could associate a reasonably accurate picture of what was meant — even though I had not read the book. Were able to quickly identify key questions like "if the machine were to go down, would this be the one piece of data you would save".
Shaw: It is a good idea to write down the folklore in a way that people can easily understand.

5. Domain-Specific Software Architectures

Advantages and Challenges of Orthogonal Views in Software Architectures

Lou Coglianese
Loral Federal Systems, Owego, New York
lou@lfs.loral.com

The world of software architecture research encompasses a broad range of activities from the specification of narrowly defined languages, to the identification of broadly used patterns, to the creation of end-user or professional kits, and to the establishment of formalisms that support domain-specific system development environments. Our work has been focused on the use of architecture formalisms and models to create environments for the specification of systems from reusable parts. The challenge in this arena is to be able to specify the minimum needed to create the system while providing the maximum flexibility and information to the end user.

As we have been concerned with domain-specific architectures, we were directly confronted with the issue of orthogonal views. In complex systems, each person, or at least each discipline, involved with the development brings a set of unique perspectives and concerns. Each of these views into a system is equally valid; each requires appropriate design and analysis; each uses it's preferred notations; and each interacts with the others in non-trivial ways. In our work, we have noted several orthogonal views that need to be reconciled. The most important set of orthogonal views, however, are those that describe the functionality of the system and those that describe the organization of the software that implements those functions. It is from this starting point that the non-functional and the performance oriented views derive their meaning.

Modelling the Architecture of Interactive Systems Using PAC-Amodeus

Joëlle Coutaz
LGI-IMAG (campus), BP 53, 38041 Grenoble Cedex, France
Joelle.Coutaz@imag.fr

PAC-Amodeus is a domain-specific architecture model applicable to the software design of interactive systems. The underpinning principles that guided the definition of this model include software engineering factors such as modifiability, modularity, portability, and scalabilility, as well as HCI-centered properties such as multithreading, immediate and informative feedback. PAC-Amodeus blends together the Arch and PAC conceptual models. Arch defines the overall functional decomposition of an interactive system as a five component structure (the Functional Core, the Functional Core Adapter, the Dialogue Controller, the Logical and Physical Interaction components). PAC-Amodeus populates the Arch Dialogue Controller with PAC agents, i.e., a set of computational units that operate in parallel and in coordination. A PAC agent is a three facet interactor. Its Presentation defines the behaviour of the agent as perceived by the user; its Abstraction denotes the competence of the agent in a media-independent way, and the Control part expresses the dependencies (e.g., constraints) between the abstract and the presentation facets. It also maintains relationships with other agents.

Conceptual models like PAC-Amodeus are good frameworks for reasoning and assessing a particular architecture against specific criteria. However, they are hard to use by inexperienced designers or by designers who are newcomers to a specific domain.I have observed from users of PAC-Amodeus that the primary difficulty is not to allocate function to structure but to identify the structure per se. In particular the refinement of the dialogue

controller in terms of agents depends on the development tools used for the implementation of the Logical and Physical components.Another difficulty is how to plug the code generated by an interface builder into the conceptual architecture. In particular, are generic mechanisms such as constraint and fusion engines, part of the architecture or do they "sit behind the scene"? Are they explicit components of the architecture or, alternatively are they implicitly inherited from the environment? How to conciliate multiple styles in one architectural design? How to distribute a function (say dialogue control) at multiple levels of abstraction? Clearly, general guidelines are needed in order to make conceptual models more operational.

6. Architecture Extraction and Views

Dimensions of Software Architecture for Program Understanding

Hausi Müller
hausi@csr.uvic.ca

Software architecture is usually considered in terms of software construction rather than software understanding. Architecture for construction typically embody design patterns based on software engineering principles. In contrast, architectures for understanding represent change patterns and business rules based on software engineering conceptual models. This paper presents three dimensions of software architecture for program understanding. in each dimension the user of the architecture plays a central role.

Software architectures for construction and understanding cater to different audiences and are therefore inherently different. However, the interpretation of multiple, user-oriented architectures is equally applicable in both domains. From the perspective of software evolution, more effort is needed to address software understanding concerns during the design and construction phases of software systems.

During system construction, requirements and business rules are readily available. Thus, it would be easier for example to build and record hierarchical decompositions based on business rules during the design stage than during long-term maintenance when the unearthing of business rules is a slow and tedious process. Ideally, design and understanding architectures would be built concurrently during the early phases of the software life cycle and evolve concurrently during the later stages.

Evolution and Engineering

David Notkin
University of Washington, Seattle, WA USA
notkin@cs.washington.edu

I am centrally interested in improving our ability to effectively change software systems. Just like software development alone, software evolution is an engineering activity — the situation, the people, the available resources, etc. are key. Therefore, determining appropriate approaches to simplifying evolution cannot be done in the pure abstract.

Any approach to software architecture (for me, as an aid to evolution) has a cost of entry (based on technical difficulty, based on culture, etc.) and a potential benefit with respect to reducing the risk of making poor decisions. Perhaps costly decisions is a better word, since the question is a mixture of how hard it is to identify that a bad choice has been made, of the ease of overcoming an identified bad choice, and of the risk of not identifying that a bad choice has been made.

Oh yes, about architecture. To me it's most critical in context with system realizations. Associations between architecture and code are central to both forward and reverse engineering (change, of course). A focus on explicit associations — or perhaps explicit manipulation of the associations — is central.

Separation of Architectural Concerns

Dilip Soni
Siemens Corporate Research Inc.755 College Road East, Princeton, NJ, USA
dsoni@scr.siemens.com

A case study of large, complex software systems revealed that software architecture represents critical design decisions which affect quality, reconfigurability, reuse, cost, and time to market [1]. Separation of concerns played a major role in the success of these systems (such as improved reuse and reconfigurability). Their software

architectures were described from four different perspectives: application architecture, module interconnection architecture, execution architecture, and code architecture. These architectural perspectives were characterized by different needs and goals. They represented design decisions that are relevant at different times in the development process, and are modified at different rates.

While these categories are not the final answer to the design of software architecture, we believe that separation of architecture into more than one perspective is crucial. We also believe that such separation should not be limited to just design, even the source code corresponding to these concerns must be separately developed and maintained. The separated source code can be combined at build time to produce the complete source code for the system. Software development based on such a separation of architectural concerns and source code will

- focus expertise appropriately,
- localize the impact of changing a particular architectural decision,
- facilitate portability, reconfigurability, and extensibility by localizing and reducing the number of required modifications, and
- improve productivity and reuse.

We invite the software engineering research community to investigate techniques and tools to make separation of concerns a practical reality.

General Discussion on Architecture Extraction and Views

```
Anderson: Sometimes we have to deviate from the architecture for some pragmatic reason, often
performance. Such deviations are unavoidable.
Notkin: I agree. Furthermore, a good way to look at this is also that it is important to recognize
the pertinent differences, not just any differences.
```

7. Architectural Evolution

How can software architecture facilitate change?

Walter Tichy
University of Karlsruhe
tichy@ira.uka.de

In the past, software architecture descriptions were used primarily for software development and for preventing architectural drift. Recently, the focus has shifted to maintenance. What help can we get from architectural information and techniques to maintain software? What architectural information has to be captured and what techniques and tools can use this information to simplify the actual change process? I wish to propose the following approaches:

1. Interface change assistant: Many interface changes involve simple alterations such as adding or changing a parameter, renaming, or extending a public data structure. A straightforward solution would be a program that hunts down all the affected code pieces and presents them to the user or applies a simple update script.

2. Anticipated changes: Changes anticipated by the designer can be handled relatively painlessly, provided the designer used information hiding principles to protect the rest of the system from the changes. In this case, the architecture description should record the anticipated changes, as in the module guides proposed by Parnas. Indexing techniques could then be used to find the modules relevant to the changes.

3. Nearly anticipated changes: Unanticipated changes that could never the less be incorporated by applying some "glue code" or "interface adapters" to some already existing components also occur often. Here it seems necessary to classify changes of this sort and find manageable transformations of one interface to another.

4. The most difficult changes are those that involve a design breakdown, i.e. ones that affect the structural architecture in a significant way. Examples are breaking up a large module, encapsulating an unanticipated change, morphing a prototype architecture into a production architecture, or designing an application programming interface for an existing system. Here the only approach possible seems to be to build tools that help to understand the software or restore consistency.

Fluid Architecture

William L. Scherlis
Carnegie Mellon University, Pittsburgh, PA, USA
scherlis@cs.cmu.edu

Structural architectural designs are rarely optimal and even more rarely are they durably so. In highly conventionalizable domains or product lines, where there may be general agreement on pertinent abstractions and associated software representations, internal system interfaces and abstractions may nonetheless need to evolve. And designers still wrestle with whether to sacrifice performance and use general-purpose standard components that are too general or too handcraft specialized instances.

This suggests that there would be benefit in increasing the extent to which the internal interface structure of systems, as abstracted into structural designs, can be made fluid or pliable. Fluid architecture techniques address this need to evolve structure through two kinds of mechanisms, boundary transformations and specialization transformations. Boundary transformations adjust abstraction/encapsulation interfaces, but generally without affecting computations. Specialization transformations exploit local context to tailor components. They include path transformations, which alter representations within program encapsulations.

Tacit in the Fluid architecture approach is its emphasis on the particular architectural facet of structural abstraction and its value in the evolution of systems. Structural "fluidity" may also be useful in obtaining conventionalized structure in product lines, because it permits more rapid migration towards common structural elements such as interfaces, components, and patterns.

A Perspective on the Innovations called Software Architecture

Larry Howard
Software Engineering Institute, Carnegie Mellon University
lph@sei.cmu.edu

We innovate. And then we look at the assimilation of our innovations as a *transition* problem.

Our innovations become self-focused and -motivated, resulting in breakdowns and blindness in the application. We "cook it" and they "eat it". Right? Of course not. The "cooking" and the "eating" must be framed and motivated by the nutrition and health and growth of the *whole*.

As we attend to the incorporation of the innovations called "software architecture", we would do well to look with *new* eyes at the ways these innovations will both change their worlds and reflect their changing worlds.
We must look not only at the *formative* roles such innovations will play in one or more software artifacts, but also at the situation of designing, building, using, and evolving these artifacts in the context of the existence and evolution of the complex socio-technical systems in which they are embedded.

As these systems become more complex and dynamic, master planning may be impossible; but, so too may be emergent formation from pre-fabricated, quasi-indigenous component artifacts and technologies.

My contribution to this Dagstuhl Seminar attempted to motivate and consider a synthesis of holistic and organic outlooks as the basis for the contribution of software architecture — a speculation born in reflection on our experience with an architectural device called structural models.

Open Research Questions in Software Architecture

Mary Shaw
Carnegie Mellon Univeristy, Pittsburgh, PA USA
mary.shaw@cs.cmu.edu

I am interested in software architecture as a means to improving practical software. To that end my research involves design methods, architectural representation languages, models of system organization, and supporting tools. I wish to record four open research problems, together with ideas about approaches:

1. In software architecture, structural or stylistic heterogeneity is inevitable. That is, we will always have to deal with mismatched parts. How can we cope?
Idea: Classify current ad hoc techniques and provide guidance for choosing which one(s) apply best in a given situation.

2. Architectural specifications are intrinsically incomplete. It is not possible to anticipate all the properties that might be relevant to some future use. Even if that were possible, it is too large a task to specify everything, every time. What are the implications for tools, methods, models, etc.?
Idea: Introduce *credentials* or partial specifications that record known properties. Provide support that recognizes them to be partial, incremental, an evolving.

3. How can higher-level architectural abstractions be connected to their underlying realizations in lower-level abstractions or code?
Idea: Create explicit abstraction mapping similar to Hoare's A for abstract data types.

4. Complex designs often involve multiple issues. Different views in different notations are often required to express those different issues. How can we manage the problem of maintaining consistency among those views?
Idea: Improve the precision with which we identify the elements of each view so that the interactions among view are more explicitly and precisely represented.

General Discussion on Architectural Evolution

```
Helm: What about the glue code?  The hard intellectual work is in defining the glue code as in
an application framework.

Lamb (to Tichy): You briefly mentioned the RCS/API problem.  Can you tell us more about happened
there?
Tichy: The first step of partitioning the commands is easy.  But it is very difficult to determine
how the user is likely to use the API in a real system and this has a big affect on what functions
should be internal and which should be part of the API.
```

8. Tools

Exoskeletal Software — Making Structure Explicit

Jeff Kramer
Imperial College, London, England
jk@doc.ic.ac.uk

The humble crab provides us with an example of a working system which explicitly flaunts its structure, both in terms of its parts and interconnections. One can readily comprehend its operation from its exoskeletal "architecture". Each part provides at its interface a clear indication of the allowable interactions with other parts yet hides its internal workings. Like the crab, it is possible for software systems to benefit from an explicit and visible architecture, not only during design and analysis but also in the constructed system itself. Structuring/ binding (configuration) languages such as Darwin provide a means of making general and instantiated software architectures explicit and precise.

Architecture Styles and Services

Balachander Krishnamurthy
AT&T Bell Laboratories, Murray Hill, NJ, USA
bala@research.att.com

Software architecture is a description of a set of components and relationships among them, how such components can be put together to build some class of structures, and implications on the resulting structures' usability, integrity, overall performance and construction cost. Another way to view architecture is as a collection of architectural styles and services. An architectural style consists of rules, constraints and a collection of characteristics that are both common to application architectures that are designed to solve a family of problems and are sufficient to identify such a collection. The architectural styles range in complexity from simple pipelines, to complex transaction processing and real time applications. For example, the ACID (atomicity, consistency, isolation, and durability) characteristics identify a system in the transaction processing style. Application systems must also satisfy a number of non-feature requirements. Among these are fault tolerance, visualization, and security. These capabilities are crucial to most software products, and make up a significant part of the software in application systems. These are referred to as architectural services and allow the architecture, or platform, to support applications with these generic, non-feature services; relieving the application programmer of the need to

write them. Since architecture styles enforce constraints on the architecture and inter-style interactions are better understood, the evolution of each subsystem can be controlled and made consistent. We have identifies a set of styles and services that span a large number of applications in certain domains. We believe that a focused look at architectural styles and services can help in structuring new applications and aid in improving reuse.

```
Garlan: Were the styles acknowledged by the designers?
Krishnamurthy: About half definitely were; other half not so clear.

Griswold: You "partition" into styles.  In my system I "overlay" with styles.
Krishnamurthy: We have legacy code that needs to be carved into multiple styles.
```

Legacy Architecture

Dewayne E. Perry
AT&T Bell Laboratories, Murray Hill, NJ, USA
dep@research.att.com

There are a very large number of existing systems which have been built without our research and insights into software architecture. These continue to evolve and their architecture continues to erode.

- Where is the architecture in this code?
- Where is the architecture in this design?
- Can we recover anything more that just boxes and arrows?

How does the architecture of a system (an existing one) get to the state it's in in the first place?

- It begins with an ad hoc architecture
- Organizational structure defined and continues to define the architectural structure
- Local optimization is done at the expense of global structure
- There is no global structure — certainly not a coherent one.

How do we get from here to a better, more suitable architecture?

Software architectures: Languages and Tools

Bernard Westfechtel
RWTH Aachen
bernhard@i3.informatik.rwth-aachen.de

I am engaged in a research project which is devoted to the construction of syntax-aided, incremental, integrated, and graph-based software development environments. Software architectures have been a major concern from the very beginning in multiple respects.

First, a module interconnection language has been developed in order to describe software architectures. This language distinguishes between different types of modules -- function, data object, and data type modules --, supports inheritance and import relations, covers coarse design as well as detailed design of module interfaces, and provides for subsystems in such a way that modules and subsystems behave identically from the clients' point of view.

Second, this module interconnection language has been applied (among others) to the description of the IPSEN architecture itself. Since an IPSEN environment comprises 400,000 lines of code, this application has given evidence to the applicability of the module interconnection language.

Third, tools for software architectures have been incorporated into the IPSEN environment. An editor which provides for both textual and graphical representations of an architecture and supports incremental context-sensitive analysis, has been implemented with the help of the IPSEN generator environment. Furthermore, we have built tools for putting the software architecture into context. More specifically, tools have been constructed to integrate software architectures with requirements definitions, software architectures and documentation. All of these tools maintain fine-grained units. They perform consistency analyses, incremental transformations, and navigation. All tools are driven by analysis and transformation rules which embody syntactic knowledge (both context-free and context-sensitive).

9. Working Group: Architecture and Implementation (mod. Müller)

Summary Points

- Architecture consists of multiple views, which are related to purposes, and which are not necessarily refinements of one another.
- Leverage is obtained from recording, managing, and manipulating explicit connections among the views.

Discussion topics

1. Need for explicit connection between architecture and implementation.
2. There is no simple mapping between problem and solution domain.
3. What to do about change patterns not encapsulated in source.
4. Can encapsulation be done after the fact?
5. We should better understand the relationship of mapping and inverse mapping.
6. Topology and dynamics of mappings are important.
7. Cannot define (order) "levels" among layers.
8. Many useful perspectives.
9. No fixed set of layers of abstractions.
10. What do we need to capture/record about design to ease change?
11. Change patterns in practice — list was primarily implementation oriented.
12. Utility of recording design rationale.

Discussion

Rohnert: There will be no grand unified view of software architecture.
Shaw: Maybe no single view, but a set of views that interlock and jointly provide "global truth"

Helm: What are the n views, how are they related, and what are they used for? We should be describing architecture, not describing descriptions of architecture. Don't ignore the fact that architecture is a verb as well as a noun.

Schwanke: Starting at the code level gives us a lot of potential. Maintainers already have a hard time dealing with two views of the system (i.e. code and architecture) much less with large numbers of multiple views.
Griswold: Human beings can move between levels of abstraction easily, but the tools we currently have do not support this adequately.
Shaw: Look at other domains (e.g. HVAC, electric, structural). No single person is expert at the different views but different people specialize in different views. There is an analogy to building architecture where different people know a great deal about each view and it is the role of the architect to coordinate these views. Don't be afraid of different specialized notations as long as someone (i.e. the architect) can coordinate them.

Jackson: With reverse engineering you don't have an arbitrarily unbounded number of views, but with forward engineering you do have this problem. There is an issue of designing small perturbations to existing architectures (i.e. existing set of views and you can reuse the composition) as opposed to designing something entirely new (where even the composition itself is new).

Helm: Is consistency between views really necessary? Views have to admit inconsistency, but one should be aware of it. Inconsistency in the design process is a source of information. I want to be able to build software not just how I have built it before but how people in the community have built it before.
Garlan: In other disciplines it is common to live with a dissonance between what is built and the design.
Balzer: How important is consistency? Inconsistency is the driver of focus of what to concentrate on. There exist no tools to help us manage that inconsistency. They all presume a consistent world and don't identify clashes. If we are going to provide tool support, then we must deal with partial information or we will do all the "hard stuff" outside of the tool.
Kramer: I agree that inconsistencies are a valuable source of information.

Inverardi: Actually, what you want is to be able to reason about incomplete knowledge.
Wolf: Views let you study particular aspects of a system.

10. Working Group: What are the limitations of architecture and what important things can't we do with architecture? (mod. Balzer)

Summary

We defined three categories as follows:

A. Bounding design space

B. Making choices — understanding analysis

C. Aiding construction/realization

We identified the following architectural issues (the associated category level is given in the right-hand column):

1. Trade-off Aids (pre-construction analysis, statement of invariants)	B
2. Map existing system onto an intrinsic architecture	C-1
3. Statement of environment context	A
4. Multi-level reasoning of architectural abstraction	B
5. Relation between software architecture and application domainA	A
6. Style guide	B
7. Aid to system construction	C
8. Design rationale	B
9. Mechanisms for families	B
10. Architecture-cognizant configuration management	C
11. Generation, Integration, Customization	C
12. Multiple Views	B
13. Testing Aid	C
14. How do we train architects, how do we train customers	(orthogonal)
15. Heterogeneity	B (some C)
16. Incompleteness, Inconsistency	B (some C)

Discussion

Helm: Architectural archetypes are tangible and achievable. Instead of talking about describing architectures, we should do it. This should be taken as a challenge for the next year.

Balzer: Describes Software Process Improvement Networks (SPIN).
Müller supports idea for architectural SPIN groups. This would give us leverage and lead to progress.

Anderson: You have to have very high-level commitment to get the agreements that are required for an architecture to be widely used in a company.
Balzer: similar as product to product line development (i.e. a family of systems).

11. Working Group: Commercial aspects of software architecture (mod. Anderson)

1. Trajectory of the work
text text
text text

	Client	Industry
Sell Arch. to	(2)	
Develop Arch. with	(1)	(3)
	[private asset]	[public asset]

2. How things will change
text text
text text

Entity	Initial	Steady	
		Shared	Distinguishing
Company (ARCHS 'R' US)	enabler champion • convincing • experience	paradigm	leader multiple industries
Client in an Industry (Nations Bank)	(projects ---------->	product line) architecture	• extension of architecture • how to use components in architecture • own components
A Client's Industry (Banking)	(ad hoc ----------> (company focus --->	architecture) industry focus)	• structure for learning • interoperability
Servers to Industries	specialized markets	industry arch. tools expertise	components

3. Issues

- Level of agreement about architecture in the industry
- Interoperability of architecture between industries (e.g. health-care and insurance)
- Final role of the company
- Banking industry - yes; TV-making industry?

Discussion

Notkin: The strategy of Microsoft was to create a marketplace rather than selling the architecture directly. They take advantage of the subsequent development of 3rd party tools. Sometimes selling the architecture itself directly is not allowed due to problems like trade secrets.
Shaw: Notes an interesting article in the Harvard Business Review on open vs. proprietary systems. The advantage of openness still being achieved through the leadtime in development and expertise.

Scherlis: There is a general evolution toward commonality, with a natural process of moving up to higher levels of capability, introducing commonality in the layers below. The trick is to time it right.
Rohnert: Notes the book "How to survive in a post-IBM world".
Lamb: This shows that technologists don't matter. Technology does.

12. Working Group: How can we cope with heterogeneity, incompleteness, and inconsistency? (mod. Shaw)

Table 1: Heterogeneity

	Mismatched Parts	Views on System	Implementation
Identify	• Catalog kinds of mismatch and repair	• How to identify critical/sensitive parts • When is one view a refinement of another?	*(This area is well worked-out, so we did not spend much time here.)*
Asses	• How much can we tolerate? What's minimum granularity? • Asses existing fixup methods • Interoperability simulation/ testbed • I/F consistency OK	• Views for other property sets	
Compen-sate for/ Repair	• Synthesize strategies for fixup and provide help for choosing right one • Integrators expendable toolkit	• Interaction checks to identify problems (e.g. via data dictionary)	
Avoid/ Reduce/ Mitigate	• Generality vs. power (arch. frameworks)	• Separate corre-spondence from consistency • Reduce cross-product explosion • "ity" budgets for other resources	

(Undesired) Incompleteness/Inconsistency:

It's REALLY hard to anticipate everything that might ever matter:

- Expand scope of specification based on experience
- Help propagate new information to places already analyzed
- Make resource issues visible as early as possible (checklists)
- How to do this globally, not pairwise
- Simulation, interoperability testbed

Discussion

Müller: It is important to note that views are not just refinements. One should have a "road map" of views.
Jackson: You have presented heterogeneity and incompleteness as "defects" to be coped with. Was that your intention?
Shaw: We're concerned with "undesirable" heterogeneity and incompleteness.

13. Working Group: Dynamic Aspects of Software Architectures (mod. Kramer)

Goals

- **Which dynamic aspects of software architectures are important to capture/formalize?**
- **Give an illustrative example.**
- **Which mechanisms/formalisms would be best to express these aspects?**

The group concentrated on the first two points, hoping to progress to the third in the future, after rationalising the list below.

Aspects

1. Dynamic creation and deletion of components
Example: Supervisor/Worker

2. Architectural evolution (in perpetually running systems)
Distinguish between: (i) replacement of parts; (ii) replacement of protocols
Examples: telephone system ; adding new customer line to DATAPAC, change of software in a space station , change in tax laws , change in environment, operating system ...

3. Data dynamics
Requires extension to — or modification of — system interface?
Examples: introduction of new data formats; connection to new database, schema updates

4. Dynamic connection/disconnection
Example: Mobile telephones

5. System start-up/shutdown/recovery protocols
Requires a global ordering of system phases.
Examples: Air traffic control system, telephone switch

6. Process creation/administration/replication
performance reasons, fault tolerance, load balancing, etc.
Example: multiple process system

7. Process migration
Application to load balancing...
Example: telephone systems

8. Instrumentation
Examples: monitoring communications between components, introduction/removal/control of probes

9. Reconfiguration by selective encapsulation
Example: reallocation on a multiprocessor

10. Timing
real-time constraints and characteristics of components
Example: composing multimedia presentation from standard parts

11. Concurrency abstractions
Especially to bridge architectural styles
Example: David Garlan’s gnuzip example. (Nierstrasz)

12. Protocols
When can protocol-sensitive components be substituted?
Examples: multimedia pipes and filters, functional and protocol-compatible APIs

13. Recording dataflows
I.e., dynamic aspects not captured by static structure
Example: Bob Balzer’s Aegis system.

14. Architecture-level simulation
Esp. (i) blocking/deadlock, (ii) interaction, (iii) abstraction of behaviour

Grouping

A. Dynamic configuration — changes in structure

- Dynamic creation and deletion of components
- Process creation/administration/replication
- Dynamic connection/disconnection
- Process migration
- Protocols
- System start-up/shutdown/recovery protocols
- Architectural evolution (in perpetually running systems)
- Reconfiguration by selective encapsulation

B. Analysing system behaviour

- Data dynamics
- Instrumentation
- Timing
- Concurrency abstractions
- Recording dataflows
- Architecture-level simulation

It may be useful to consider which of these aspects has *local* significance and which has *global* significance.

Discussion

Müller: I see a cross-fertilization of strands across architecture and yours is of a different flavour and apparently quite advanced. What cross-fertilization do you see and is it perhaps due to domain-specificity?
Kramer: I would not presume to tell you what lessons can be learnt. To discuss further, I would need to understand your concerns more. I don't know if the same things apply, but it certainly would be interesting to pursue.

14. Working Group: What is an effective balance between formalism and informal descriptions? (mod. Notkin)

There is a spectrum of degree of formalism and informalism. The answer to the question "what is an effective balance of formalism and informalism" is that it depends on:

- Which systems
- Which activities
    - Suggesting alternatives
    - Evaluating alternatives
    - Generating realizations,...
- Other Dimensions
    - Team vs. Individual
    - Computer vs. person
    - How many choices?
    - Creative activity?
    - Cost of entry, ...

Discussion

Shaw: This casting as an economic leverage problem is interesting, but I note that it is silent on the actual issue of formal vs. informal.
Notkin: The decision depends on the case at hand and many variables. For example, in the case of an OODB described by Helm, due to hard performance considerations it was useful to walk through all the use-cases and do the clustering analysis. And in a second case of doing a dependency analysis for assigning people to jobs an informal approach was chosen for this less-critical task.
Boehm: This example captures the business model well. Similar experience at TRW where formalism was used when there was a large need for security.

15. Closing Remarks

Software architecture is a very broad field and the people working in this area come from diverse backgrounds with various reasons for participating in the workshop. It was no easy task to channel such diversity in a productive manner. Especially, since this was the first meeting of its kind and no "established community" as such exists, a significant investment had to first be made into getting to know and understand each other before delving into the technical discussions. On the whole, we all profited from the joint time spent working with and learning from each other and we forsee that some seeds were sown here which will lead to future advances in the area of software architecture. We thank the administration of Schloß Dagstuhl for providing such a stimulating environment for such seminars. We hope that through this document summarizing the workshop, we will be able to share some of what we have learned with the software engineering community.

In closing, a selection of two of the more humorous closing remarks from the last day of the seminar:

Top 10 reasons why software architecture will make a difference (from Larry Howard):

10. "We stand on the shoulders of giants."
9. We don't oversell.
8. Our goals are modest.
7. We've learned from the past what hasn't worked and why.
6. We're practical...really. Trust us.
5. We don't invent, we *reflect*.
4. Our research focuses exclusively on recognized and quantified problems.
3. An ounce of *cure* is worth a pound of *prevention*.
2. God is *not* in the details, but in the abstractions.
1. "Every day, in every way, we're getting better and better."

Four kinds of people and their reaction laws (from Paola Inverardi)

The semiformalist ::= "Only semiformal methods are possible: MINE!"	The formalist ::= "Everything can be formalized, otherwise it does not exist!"
The pragmatist ::= "Only software exists, therefore only I exist!"	The informalist ::= "Everything should not be formalized, otherwise it does not exist!"

Reaction Law

Reaction Law

Reaction Law

Reaction Law