CONSTRUCTING SYSTEMS FROM PARTS: WHAT STUDENTS SHOULD LEARN ABOUT SOFTWARE ARCHITECTURE

M Shaw

Rapporteur: Ian Welch

Constructing Systems from Parts:
What Students Should Learn
about Software Architecture

Mary Shaw
Carnegie Mellon University
Pittsburgh PA
http://www.cs.cmu.edu/~shaw

"Point of view is worth 40 IQ points"
-attr to Alan Kay

**Software architecture provides
new points of view on
integrating components into systems**

Outline

- Software architecture in context
- *Abstractions:* style, components, and connectors
- *Decisions:* choosing among alternatives

≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈

- *Architectural mismatch* and its amelioration
- *Credentials:* partial, evolving specification
- *Resource coalitions:* prospects for an informatics marketplace

Topics of these Seminars

Focus	Human Client	Problem	Solution
Topic	**UI design**	**Engineering design**	**Software architecture**
	Product design	**Problem frames**	**Patterns**

Design for Real People

- Challenge:

  Disintermediation -- direct connection of non-experts to software -- raises the bar for usability
- Successes:

  Spreadsheets, the Web, integrated office suites, (perhaps) Visual Basic.

  Note: these largely originated outside computer science
- Gentle-slope systems:

  Learning curve should match the reward curve -- a little effort should generate some useful results, more effort should be rewarded proportionally
- People who study CS know woefully little about this

Engineering Design → Decisions

- You must discriminate among
  - > different kinds of problems
  - > different kinds of solutions
- You must make informed choices to match solutions with problems
  - > different kinds of problems match different kinds of solutions

The decisions with the most impact are the ones that deal with overall system organization

Engineering Design

Engineering is ...

Creating cost-effective solutions to practical problems by applying scientific knowledge when possible, building things, in the service of mankind

- Warning --

  No amount of technique, method, tooling, or other rote or automatic support can substitute for actually understanding the problem -- and the limitations of the possible solutions

- Distinction: innovative vs routine design

  Virtuosos are required for novel applications; variations on familiar themes can be carried out by more ordinary folk

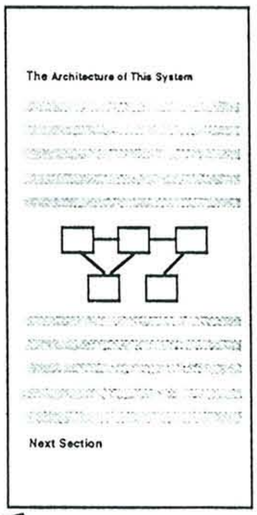
Problems and Solutions

- Problem
  - > Requires context -- the subject matter of the system
    - » Relevant parts of real world, their properties & relations
    - » Often nonformal and in heterogeneous notations
  - > Software designer must be able to learn about domain
    - » phenomenology, technology of description, formalization
  - > *Problem frame* == principal parts + solution task
- Solution
  - > characteristics of the machine that solves the problem
  - > draw on standard parts, templates, ..
  - > *software architecture, patterns, programming clichés*

Patterns

- Structured form for explanation of design element
  - > Uniform structure similar to handbooks
  - > Provides rationale and terms of use as well as definition
- Strength in families
  - > Collections of related patterns intended to work together
  - > Often called "pattern languages"
- Not limited in subject matter
  - > Principally found in object-oriented design at present
  - > Also suitable for system-level design elements

Typical Descriptions of Software Architectures



- Descriptions of software systems often include a section on "the architecture of this system"
- Usually informal prose plus box-and-line diagram
- Lots of appeal to intuition
- Little precision, rarely formal

[[sample architectural diagrams]]

Typical Descriptions of Software Architectures

> "Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers." [Spector 87]

> "We have chosen a distributed, object-oriented approach to managing information." [Linton 87]

> "The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors." [Seshadri 88]

> "The ARC network [follows] the general network architecture specified by the ISO in the Open Systems Interconnection Reference Model." [Paulk 85]

Observations about Designers

- They freely use informal styles (idioms)
  > Very informal, imprecise semantics
  > Diagrams as well as prose, but no uniform rules
  > Communication takes place anyhow
- Their vocabulary uses system-level abstractions
  > Overall organization (styles)
  > Kinds of components, distinguished by role in system
  > Kinds of interactions among components
- They compose systems from subsystems
  > Tend to think about system structure statically
  > Often select organization by default, not by design

Component Properties

- Functional <---- *What many folks mean by "specification"*
  > Type, signature, pre/post conditions
- Structural <---- *What many folks mean by "architecture"*
  > Packaging: component type & allowed interactions
- Extra-functional<---- *What often appears in product spec*
  > Performance, capacity, environment, global properties
- Family <---- *What often is included in "config mgt"*
  > Shared assumptions, constraints on aggregates, envelope of allowable variation

The Architecture of a Software System ...

- Defines the system structurally, in terms of components and interactions among components
  - > We believe the interactions are so critical that they should be recognized explicitly as design elements
- Shows correspondence between requirements and elements of the constructed system
- Addresses system-level properties such as scale, capacity, throughput, consistency, compatibility
- Captures and preserves designer's intentions about system organization and structure

Software Architecture Topic Areas

- Descriptive techniques
  - > Notation (ADLs), descriptive formalism, graphical tools
  - > Design methods focused on use of a notation
- Abstractions about architectures
  - > Classification and comparison of styles
  - > Interactions of multiple views
  - > Patterns as descriptive vehicle
- Homogeneous (fixed-style) architectures for particular targets
  - > Product families
  - > Domain-specific architectures
  - > Specific styles/frameworks (o-o, event, dynamic, ...)

Software Architecture Topic Areas

- Heterogeneous (multi-style or ad hoc) architectures as seen in practice
  - > Design, including use of COTS, reuse
  - > Interoperability and mismatch resolution
  - > Evolution
- Analysis
  - > Quality attributes
  - > Behavior, performance
  - > Reverse engineering, architecture recovery
- Formal methods
  - > Specifications and formalisms
  - > Models

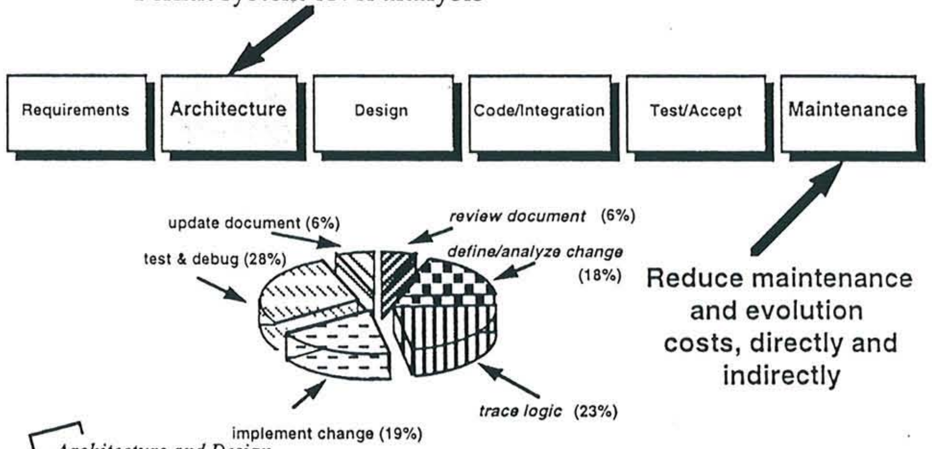
Key Ideas

- Structure matters -- overall organization, kinds of components, how they interact
- Interaction matters -- how a component must interact, not just what it computes
- Abstraction matters -- the designer's view of interaction, not just the procedure calls
- Decisions matter -- different kinds of problems require different solutions

Provide a solid basis in models, notations, and tools to support developers' intuitions

Anticipated Benefits

- Clarify intentions
- Make decisions and implications explicit
- Permit system-level analysis



Reduce maintenance and evolution costs, directly and indirectly

Outline

- Software architecture in context
- *Abstractions:* style, components, and connectors
- *Decisions:* choosing among alternatives

≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈

- *Architectural mismatch* and its amelioration
- *Credentials:* partial, evolving specification
- *Resource coalitions:* prospects for an informatics marketplace

Building Systems from Parts

- The hype:

  "... and then we'll be able to construct software systems by picking out parts and plugging them together, just like Tinkertoys ..."

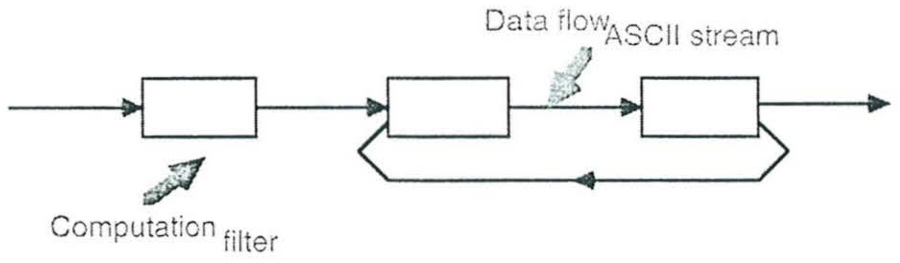
- The hard cold truth:

  It's more like having a bathtub full of Tinkertoy, Lego, Erector set, Lincoln logs, Block City, and six other incompatible kits -- picking out parts that fit specific functions and expecting them to fit together

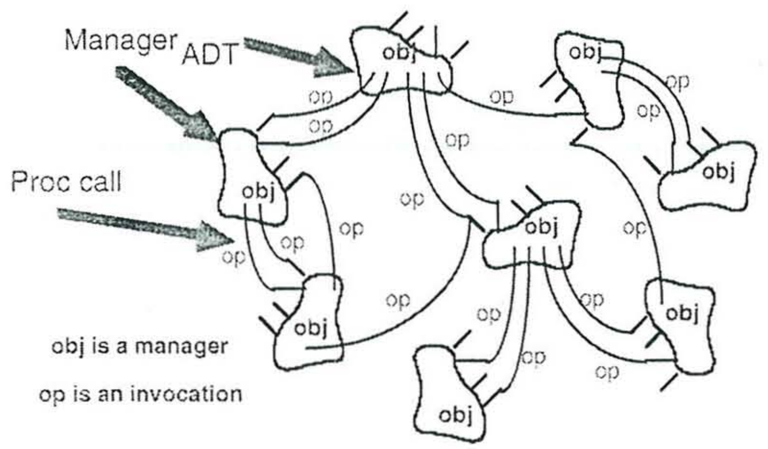
Elements of Architectural Style

- Patterns of system composition
  - > Constraints on component and connector topologies
  - > Example: pipeline
- Family of compatible parts
  - > Selection of interoperable components and connectors
  - > Example: clients and servers
- Conventions about the meaning of architectural descriptions
  - > Semantic interpretations
  - > Example: lines mean pipes, boxes mean filters

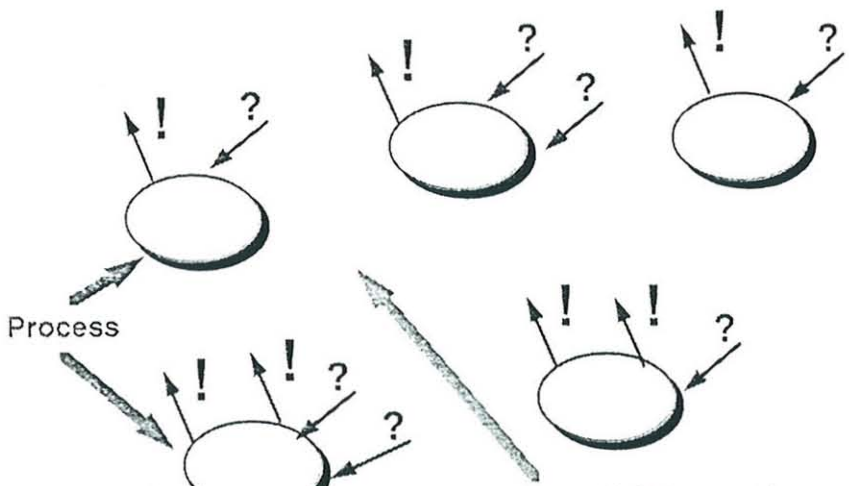
Pipeline



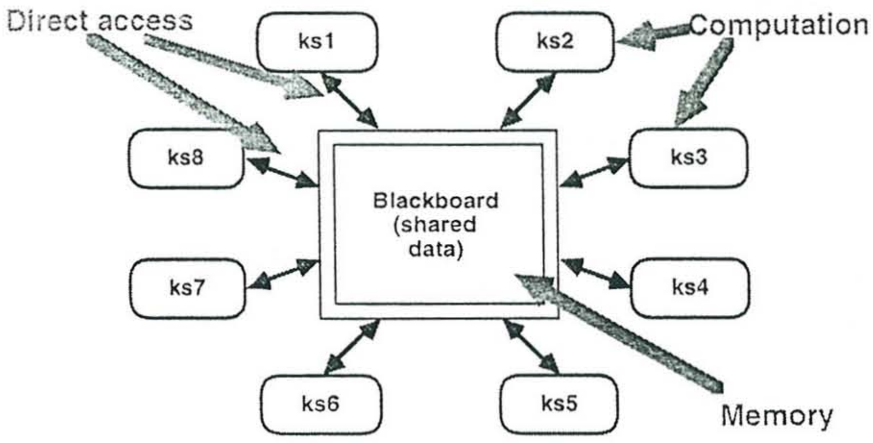
Data Abstraction (Classical Objects)



Events



Repository (Blackboard)



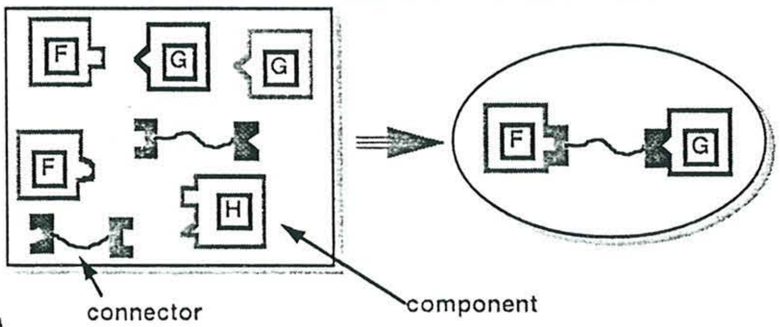
Comparison of System Patterns

System Model	Components	Connections	Control Struct
Pipeline			
stream -> stream	filters (local processing)	data flow ASCII streams	data flow
Data abstraction (object-oriented)			
localized state maint	servers (ADTs, objs)	procedure call	decentralized, single thread
Event			
spontaneous reaction	independent processes	implicit invocation	asynchronous processes
Repository			
central database	1 memory N processes	direct access or proc call	internal or external

Connectors are First Class

Beyond the module level:

How a component interacts with its environment is as important as what it computes.



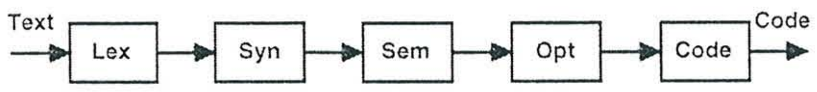
Why distinguish connectors?

- *Locality:* Information about component interaction is usually distributed through the application code and hence hard to find and change
- *Separation of concerns:* Different expertise and design considerations are involved in designing application and inter-component interactions.
- *Abstraction:* Designers use abstractions ("RPC") that don't appear in code; code contains calls on the procedure libraries that implement the abstractions
- *Choice:* Explicit design elements are more likely to receive focused consideration and less likely to be ignored or defaulted,

Architecture Description Languages

- **Provide text and/or graphical notation for system descriptions using these abstractions**
- **May support one or many styles**
- **Support analysis, code generation, compilation**
- **Most focus on a few aspects of the problem**
- **Organization compatible with unix-style environments, less so with PC development**

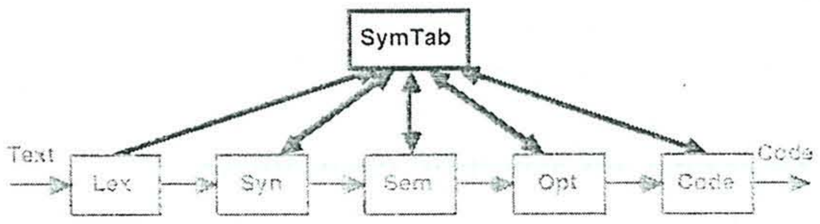
Example: Canonical Compiler



This is the textbook version

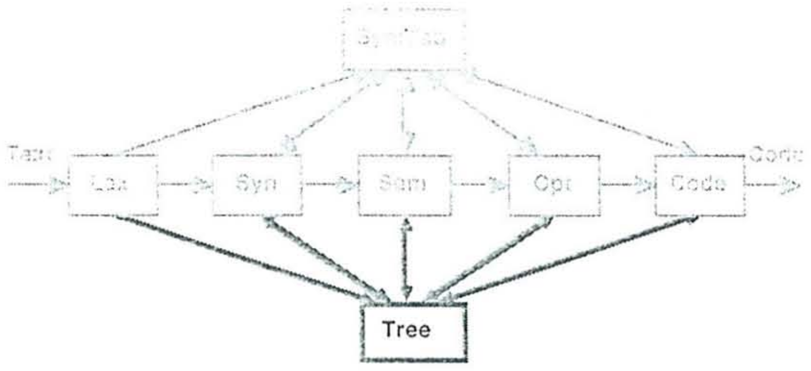
Changing your point of view
can lead to more useful view of system

Canonical Compiler: Troublesome Details

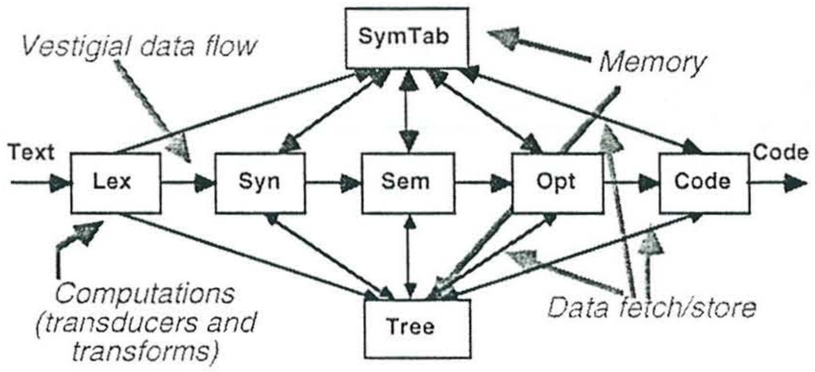


Pipeline?
No, Batch Sequential

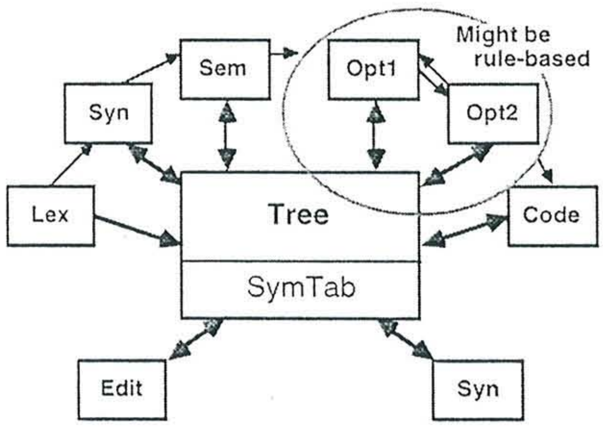
Example: Modern Canonical Compiler



Example: Modern Canonical Compiler



Canonical Compiler, Revisited



Outline

- Software architecture in context
- *Abstractions:* style, components, and connectors
- *Decisions:* choosing among alternatives

≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈

- *Architectural mismatch* and its amelioration
- *Credentials:* partial, evolving specification
- *Resource coalitions:* prospects for an informatics marketplace

Building Systems from Parts

- The hype:

  If you just use the XYZZY method, everything will work out fine
- The hard cold truth:

  Different problems require different organizations; each organization or method has strengths and weaknesses; there is no "silver bullet" or universal solution. Careful, deliberate choices are required.

How Do You Choose an Architecture?

- Organize the next system like the last one
- Adhere to company coding guidelines
- Follow the latest fad
- Use a prescriptive methodology or tool
- . . .
- Use the definitive architecture for your application domain
- Evaluate alternatives on the basis of
  > characteristics of the application requirements
  > constraints of the operating environment

Status of Results

- Not all CS results are established scientific truths; we do have interesting observations and generalizations.
- Brooks proposes recognizing three kinds of results, with individual criteria for quality:
  - > findings -- well-established scientific truths -- judged by truthfulness and rigor
  - > observations -- reports on actual phenomena -- judged by interestingness
  - > rules-of-thumb -- generalizations, signed by an author but perhaps not fully supported by data) -- judged by usefulness

  with freshness as criterion for all
- Discriminations among styles and classification are observations, based on examining system descriptions
- Suggestions about style selection are rules of thumb
- Lane's design guidance is properly validated finding

Taxonomy of Architectural Styles

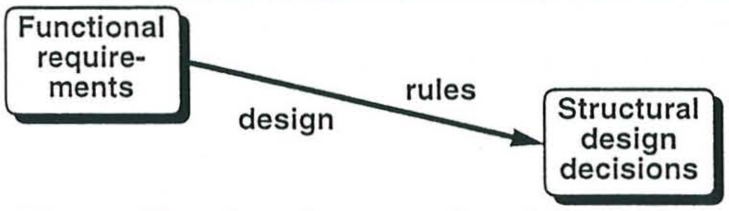
- Data Flow
  - Batch sequential
  - Dataflow network
    - acylic, fanout, pipeline, unix
  - Closed loop control
- Call-and-return
  - Main program/subroutines
  - Information hiding
    - ADT, object, naive client/srvr
- Interacting processes
  - Communicating processes
    - LW processes, distrib objects,
  - Event systems
- Data-oriented repository
  - Transactional databases
    - True client/server
  - Blackboard
  - Modern compiler
- Data-sharing
  - Compound documents
  - Hypertext
  - Fortran COMMON
  - LW processes
- Hierarchical
  - Layered
    - Interpreter

Classification Basis for Taxonomy

- Constituent parts
  > Components, connectors
- Control issues
  > Topology, synchronicity, binding times
- Data issues
  > Topology, continuity, mode, binding times
- Control-data interaction
  > Isomorphism of topology, flow directions
- Type of reasoning supported

Software Design Decisions

- Design spaces for function and structure
- Some functional requirements favor or disfavor certain structures
  > capture these as a set of preference rules
  > develop prototype designer's advisor



- PhD thesis (Tom Lane) on user interface decisions
- Similar problems for other software decisions

Rules of Thumb Re Data Flow

- If your problem can be decomposed into sequential stages, consider batch sequential or pipeline architectures.
  > If in addition each stage is incremental, so that later stages can begin before earlier stages finish, consider a pipeline architecture.
- If your problem involves transformations on continuous streams of data (or on very long streams), consider a pipeline architecture.
  > However, if your problem involves passing rich data representations, avoid pipelines restricted to ASCII.
- If your system involves controlling continuing action, is embedded in a physical system, and is subject to unpredictable external perturbation so that preset algorithms go awry, consider a closed loop architecture.

Rules of Thumb Re Objects and Repositories

- If a central issue is understanding the data of the application, its management, and its representation, consider a repository or abstract data type architecture. If the data is long-lived, focus on repositories.
  > If the representation of data is likely to change over the lifetime of the program, abstract data types or objects can confine the changes to particular components.
  > If you are considering repositories and the input data is noisy (low signal-to-noise ratio) and the execution order cannot be predetermined, consider a blackboard.
  > If you are considering repositories and the execution order is determined by a stream of incoming requests and the data is highly structured, consider a database management system.

Rules of Thumb Re Processes, Virtual Machines

- If your task requires a high degree of flexibility/ configurability, loose coupling between tasks, and reactive tasks, consider interacting processes
  - > If you have reason not to bind the recipients of signals to their originators, consider an event architecture.
  - > If the tasks are of a hierarchical nature, consider a replicated worker or heartbeat style.
  - > If the tasks are divided between producers and consumers, consider a client-server style (naive or sophisticated).
  - > If it makes sense for all of the tasks to communicate with each other in a fully connected graph, consider a token passing style.
- If you have designed a computation but have no machine on which you can execute it, consider an interpreter architecture.

Outline

- Software architecture in context
- *Abstractions:* style, components, and connectors
- *Decisions:* choosing among alternatives

≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈≈

- *Architectural mismatch* and its amelioration
- *Credentials:* partial, evolving specification
- *Resource coalitions:* prospects for an informatics marketplace

DISCUSSION

Rapporteur: Ian Welch

Lecture One

A participant asked if Professor Shaw could clarify what she meant by programmed system. She replied that she meant a software system in general. He queried whether she meant a system composed of one or many components and whether complex systems ever had an overall architectural view. In his experience complex systems evolved over time, perhaps over many years, without anyone actively designing an overall architecture. She responded that research into how people spent their time maintaining systems had revealed that a significant proportion was spent rediscovering the underlying architecture of the system. If, when changes were to be made to a system, an overall architecture was available then considerable productivity increases could be expected. For those real life systems where no architectural design had taken place there are tools being developed that discover the architecture by examining the software artefacts.

Professor Randell asked if she was talking about the virtual or the real architecture. A virtual architecture may be compiled into a real architecture that no longer reflected the virtual architecture used by the designers to structure the solution. She replied that this was related to the documentation problem. If there is no real gain to programmers in keeping documentation up to date then they won't. One answer is to integrate the architectural design into the system build process - actually compile from the architecture. Professor Randell suggested that you may need additional information relating to the environment to make sense of errors when compiling - for example memory model problems. A participant from ICL suggested that although direct compilation from the system architecture may be unfeasable at present they have gained from using the system architecture to drive system testing. This has given a direct payoff for keeping the system architecture up to date. Mr Jackson commented that the key is ensuring that the payoff is to the person who creates the problem in first place - so if the programmer is the one changing the system then there must be some direct benefit to him/her of maintaining the system architecture or documentation.

Professor Shaw was describing the bathtub problem - in a world of reuse we are faced not with a set of tinkertoys that can be easily assembled together, we have a bathtub of many types of toys from Lego to Meccano. We want to be able to use apparently incompatible toys together. To this one participant asked whether this wasn't being solved by technologies such as CORBA that allow interconnection of different types of components. She replied that this wasn't the whole story, as besides interconnection you have to decide on the suitability of the type of toy for the problem and the problem of interrelating two quite different types. Lego bricks are good for walls and Meccano is good for trusses not vice versa. How do you reconcile their different innate "types" when using them together?

Professor Randell asked if components and connectors had internal structure. Professor Shaw replied that connectors don't but components do. There was no reason why connectors shouldn't and in fact people were working on this problem but there where a few open questions about the mapping of the interface to internal states for connectors.

Professor Randell asked if all connectors were binary. Professor Shaw replied that only in her diagram shown they are, but in fact in their tool connectors they can be symmetric, asymmetric, many-tendriled etc. Also connectors can be static or dynamic. However, people find it easier to reason about systems built from static connectors.

Dr de Lemos asked if there was really any difference between components and connectors - were connectors not in fact specialised components. Professor Shaw replied that in her view components were localised points of computation and connectors were message passing conduits. Dr de Lemos replied that this was not precise enough and could easily be blurred. Professor Brooks suggested that the reason people used this component and connector model was that generally people had drawn informal diagrams of components joined by lines. A connector was essentially a formalisation of those informal lines.

Professor Shaw concluded by saying that if you wished you could certainly have a model that was just components.

But the distinction between components and connectors was useful because, as Professor Brooks had suggested, people were already familiar with the idea of representing systems in terms of components and connections between them. Also the really important thing was that the interactions between components were represented as first class entities that exposed the choices people made about interactions and made reasoning about them possible. In the past interactions were buried in the code which made this very difficult especially as interactions are really the heart of how a computer system works.

Professor Randell queried a comment about current ADLs being oriented more to Unix like development environments rather than PC development environments. What made PCs so different? Professor Shaw explained that unlike Unix development environments where compilation units were separately crafted essentially from scratch PC development revolved around the use of wizards that generated code that was modified. Wizards meant high productivity but make it difficult to make major changes to say the front end of the system. ADLs don't have any concept of this type of development.

In response to Professor Shaw's example of a compiler architecture not really being a pipeline as usually imagined, Mr Jackson made the point that architectural change is often driven by technological change. His example was the availability of more RAM made online processing possible resulting in fewer batch architectures being used. So old compilers were batch like but the advent of cheap memory allowed other architectures to be developed. Professor Shaw agreed that changes in technology led to changes in architectures used to solve problems.

A participant asked what was Professor Shaw's definition of design and how did it differ from architecture. To him her lifecycle blurred design of solution and architectural phases. She suggested that she wasn't defending her lifecycle or phase definitions to the death - they were more to break away from the waterfall model and open up new views. Indeed design means many things to many different people with architecture overlapping with high level design.