CSEP504:
Advanced topics in software systems

- Overview of course, expected work, staff, …
  - Differences from CSEP503 (Spring 2009)
  - Most administrivia just before break
- Introduction to software architecture
- Who are you – companies, background, interests, etc.?
- Discussion: possible third course topic

David Notkin ● Winter 2010 ● CSEP504 Lecture 1

Software engineering in the PMP

- Collectively and individually, you have designed, developed, tested, shipped and maintained orders of magnitude more software than I have
- Collectively and individually, you continue to make design decisions, write code, test code, fix bugs, etc. on a daily basis; I don't
- Few of you are aware of much ongoing research in software engineering; I am
- Few of you are able to separate quickly the good from the bad in software engineering research; I am good (although imperfect) at this

Course goals

- To expose you to key approaches in software engineering research, with the hope that one or more of them can help you in your daily work – perhaps immediately, perhaps in the longer term
- Without ignoring your day-to-day issues, try to look deeper into the issues of engineering quality software than day-to-day pressures usually allow
- To increase your ability to communicate with software engineering researchers and other software engineers

But wait! That was CSEP503!

- Well, yes… the underlying principles and course goals are the same as in 503
- However, 504 will focus more narrowly
  - software architecture, software tools, and a topic to be named later
- In contrast, 503 was more breadth-oriented: model checking, bounded model checking, design, information hiding, layering, patterns, aspect-oriented programming, cooperative bug isolation, test prioritization, mutation testing, concolic testing, system summarization, …
- Some limited overlap

Staff

Reid Holmes

Sai Zhang

Yuriy Brun

“Software architecture”

- **Free association**

“Software architecture”

- Bing (12/25/09): 1-10 of 89,800,000 results
- Google (12/25/09): Results 1 - 10 of about 1,720,000
- Searches related to software architecture
  - Bing: free architecture software, 3D architecture software, architect design software, free home design software, floor plan design software, …
  - Google: software architecture diagram, software architecture document, software architecture patterns, software design, enterprise architecture, system architecture , …

Software architect (job title):
monsters.com posted 60 days up to 12/29/09

- Component Software
- CYBER-SECURITY
- E-Commerce
- Ethernet Switch
- GUI Software
- Perl Developer
- SDR
- Federal Health IT
- Senior Java Developer
- C#- C++ - P2P -...
- C/C++ - Algorithms
- Embedded Systems
- .Net - SQL Server
- Secure Design
- VMWare
- Team Lead/Manager
- Cognos TM1 Technical Architect
- Enterprise Architect
- Enterprise Software Architect - C# ..
- IT Architect - Software
- J2EE Architect
- .NET Architect
- Software Development & Testing Architect
- Enterprise Architect
- Sharepoint Architect
- Software User Interface Architect
- Solution Architect w/Business Services
- Application Architect
- System/Software Engineer-Architect
- Web Architect

Some (wildly incomplete) history

- Peter Naur @ 1968 NATO conference
- *"…*software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem. As one single example of such a source of ideas I would like to mention: Christopher Alexander: *Notes on the Synthesis of Form (Harvard Univ. Press, 1964)"*

R. Fennell. Multiprocess Software Architecture for AI Problem Solving

- Carnegie-Mellon University, May 1975
- This dissertation describes the design and development of a knowledge-based artificial intelligence problem-solving organization that is suitable for efficient implementation on a closely-coupled multiprocessor computer system. The method is a result of formulating the problem-solving organization in terms of the hypothesize-and-test paradigm for heuristic search, with communication between the various hypothesizers and testers being effected by writing intermediate results in a shared blackboard-like data base. These hypothesizers and testers are expressed in terms of knowledge sources which represent bodies of suitably organized subject-matter knowledge pertinent to the task domain of the problem being solved. The various system organization problems connected with such a multiprocessing scheme are discussed, and solutions to these problems are presented. The major contributions of this work lie in the analysis and solution of the various multiprocessing problems that have arisen in the course of specifying this problem-solving organization.

Wulf, W. A. Reliable hardware-software architecture

- *SIGPLAN Notices* (1975).
- This paper deals with the problem of reliability in a hardware/software system. More specifically it deals with the strategy used to achieve reliability in a particular hardware/software system built by the author and his colleagues at Carnegie-Mellon University. Rather than dealing with the myriad details of the reliability aspects of this systems, the paper focuses on the design philosophy which aims at keeping the system operational even though the underlying hardware may be malfunctioning. This philosophy is essentially an extension of the 'modular' programming methodology, advocated by Parnas and others, to include dynamic error detection and recovery.

The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture

- Peter Deutsch, *Smalltalk-80: Bits of History, Words of Advice*, 1983

- Describes a mostly-microcode implementation of the ST-80 VM.

White, J. R., Booth, T. L., Towards an engineering approach to software design

- Proceedings of the 2nd International Conference on Software Engineering (1976).
- The software design process is discussed from an engineering point of view. Initially, a distinction is made between software design and program design. Software design is then described in terms of evolving a system architecture independently of implementation considerations. Computation structures are introduced as a means of modeling the dynamic behavior of a software architecture. Functional completeness, quality, machine independence, and performance completeness of a design are then used as criteria for engineering design decisions. Finally, the basic elements of a system to support the software design process are described

Scott, L. R., An engineering methodology for presenting software functional architecture.

- Proceedings of the 3rd international Conference on Software Engineering (1978)
- A method of presenting software architecture has been developed which is useful for reviewing and documenting software designs. Diagrams showing all software levels as well as inputs and outputs are systematically developed to provide an understanding of the construction and operation of software systems and programs. The manner in which the software architecture is displayed provides an effective way for managers to understand and review software designs.

D. J. Mishelevich et al.. Application development system: The software architecture of the IBM Health Care Support/DL/I-Patient Care System

- *IBM Systems Journal 19*, 4 (1980)
- Application development productivity is a broad-based concern. A system answering this concern is the IBM Health Care Support/DL/I-Patient Care System announced by IBM in late 1977. The system is of general importance because its application development system architecture is not application specific and thus can be used for the rapid development of many types of on-line systems. It has an elegant simplicity, and it uses the standard facilities of such operating system components as CICS/VS and DL/I. The application productivity has been clearly and successfully demonstrated in the real working environment of the Dallas County Hospital District (Parkland Memorial Hospital) and other sites. This paper provides an architectural overview followed by a description with an example of CRT (cathode ray tube) screen and print format design and coding and an examination of a data collection list to demonstrate the power of that facility.

D.L. Weller et al. Software architecture for graphical interaction

- *IBM Systems Journal 19*, 3 (1980)
- Pointing at items on a graphics display is one of the most useful methods of interacting with a system graphically. This paper examines existing graphical support and lists requirements for high-level support of graphical interaction. The architecture of a prototype system with high-level support for graphical interaction is presented. This includes database support for manipulating graphical data and device-independent graphical support based on a proposed standard for graphical interaction. Algorithms are presented for identifying items selected from a display by the user. Inclusion of a database management system in graphical software support is shown to be helpful in meeting the requirements of interactive graphical application programs.

Sandewall, E., et al., Software architecture based on communicating residential environments.

- Proceedings of the 5th international Conference on Software Engineering, 1981.
- This paper describes an alternative approach to software architecture, where the classical division of responsibilities between operating systems, programming languages and compilers, and so forth is revised. Our alternative is organized as a set of self-contained environments which are able to communicate pieces of software between them, and whose internal structure is predominantly descriptive and declarative. The base structure within each environment (its diversified shell) is designed so that it can accommodate such arriving software modules. The presentation of that software architecture is done in the context of an operational implementation, the SCREEN system (System of Communicating REsidential ENvironments).

Lawson, D., A New Software Architecture for Switching Systems

- *IEEE Transactions on Communications 30*, 6 (1982)
- Electronic switching systems were introduced in the mid1960's, and have since undergone constant modifications to provide new features, expand into new applications, and adapt to new hardware technologies. The increasing costs of software development have led to the need to reexamine the design concepts of present systems and define more cost-effective architectures. The specifications and derivation of a new modular switching system architecture which addresses the problems of constant change are discussed.

P. E. Satterlee, Jr., H. L. Martin, J. N. Herndon. CONTROL SOFTWARE ARCHITECTURE AND OPERATING MODES OF THE MODEL M-2 MAINTENANCE SYSTEM

- American Nuclear Society Topical Meeting on Robotics and Remote Handling in Hostile Environments, April 1984
- The Model M-2 maintenance system Is the first completely digitally controlled servomanipulator. The M-2 system allows dexterous operations to be performed remotely using bilateral force-reflecting master/slave techniques, and its integrated operator interface takes advantage of touch-screen-driven menus to allow selection cf all possible operating modes. The control system hardware for this system has been described previously. This paper describes the architecture of the overall control system. The system's various modes of operation are identified, the software implementation of each Is described, system diagnostic routines are described, and highlights of the computer-augmented operator interface are discussed.

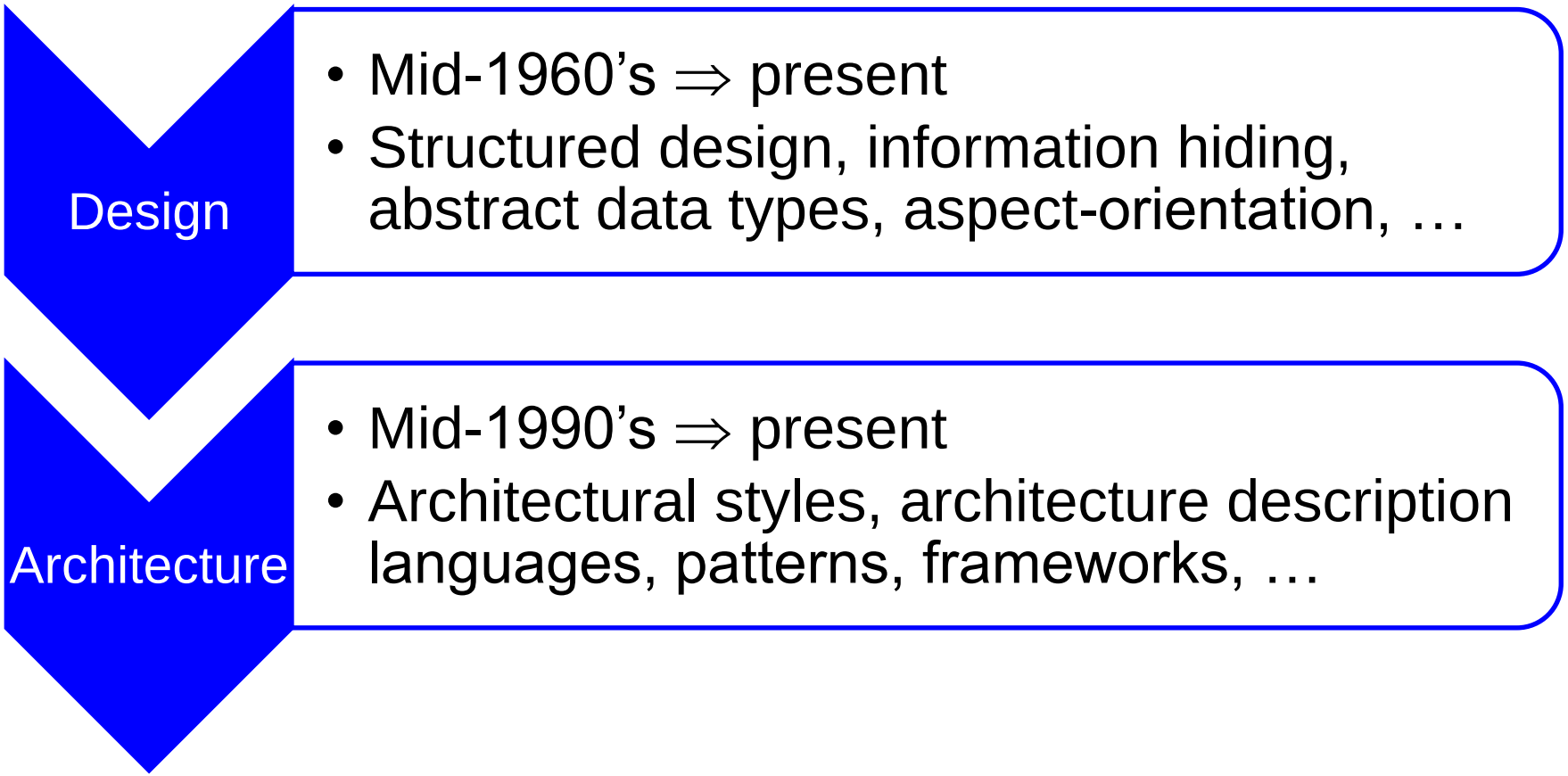
How can anyone govern a nation that has two hundred and forty-six different kinds of cheese?

-- Charles de Gaulle

Two categories: *very* soft distinction

- Software architecture: design-oriented
  - Based in software design, in defining taxonomies based on experience, etc.
- Software architecture: property-oriented
  - Based on a desire to design software systems with a particular property – such as autonomic systems, fault-tolerance, privacy, etc.

Software architecture: design-oriented



Software design: a fast history

- "design" – in OED
  - Noun: nine definitions, 1462 words
  - Verb: 16 definitions, 2165 words
- Brooks' 1993
  - rationalism — the doctrine that knowledge is acquired by reason without resort to experience [WordNet]
  - empiricism — the doctrine that knowledge derives from experience [WordNet]

Rational	Empirical
Aristotle	Galileo
France	Britain
Descartes	Hume
Roman law	Anglo-Saxon law
Prolog	Lisp
Algol	Pascal
Dijkstra	Knuth
Program proofs	Program testing

Computing examples due to Wegner

Characteristics of software design

- Complexity
- Multi-level, continuous, iterative
- Broad potential solution space
- Relatively unclear criteria for selecting solution

Complexity

- “Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one… In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.”

  —Brooks, 1986

Continuous & iterative

- High-level (“architectural”) design
  - What pieces?
  - How connected?
- Low-level design
  - Should I use a hash table or binary search tree?
- Very low-level design
  - Variable naming, specific control constructs, etc.
  - About 1000 design decisions at various levels are made in producing a single page of code

Almost never a key part of architecture

Broad solution space

- How do we select a design?
  - We determine the desired criteria
  - We select a design that will achieve those criteria
- In practice, it's hard to
  - Determine the desired criteria with precision
  - Tradeoff among various conflicting criteria
  - Figure out if a design satisfies given criteria
  - Find a better one that satisfies more criteria
- In practice, it's easy to
  - Build something designed pretty much like the last one
  - This has benefits, too: understandability, properties of the pieces, etc.

Almost always a key part of architecture

Criteria for success include…

- Correctness
- Readability
- Usability
- Modifiability
- Robustness
- Security
- Safety
- Performance
- Cost
- Time-to-market
- Profit
- Adoption
- Compatibility
- Extensibility
- Reusability
- Fault-tolerance
- Conceptual integrity
- …

Conceptual integrity

- Brooks and others assert that conceptual integrity is a critical criterion in design
  - “It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.” —Brooks, MMM
- Such a design often makes it far easier to decide what is easy and reasonable to do as opposed to what is hard and less reasonable to do

“Internal” criteria for success include

- Cohesion
- Coupling
- Complexity
- Correctness
- Correspondence

Cohesion

- The reason that elements are found together in a module (coincidental, temporal, functional, …)
- During maintenance, one of the major structural degradations is in cohesion
- Hard to measure quantitatively

Coupling

- Strength of interconnection between modules
- Hierarchies are touted as a wonderful coupling structure, limiting interconnections
- Coupling also degrades over time
- Many quantitative measures – of questionable utility

Complexity

- Simpler designs are better, all else being equal
- But, few useful measures of design/program complexity exist
- There are dozens of such measures
  - McCabe's cyclomatic complexity = E - N + p
    - E = the number of edges of the CFG
    - N = the number of nodes of the CFG
    - p = the number of connected components
  - Function points, feature points, …
- My understanding is that, to the first order, most of these measures are linearly related to "lines of code"

Correctness

- Even if you “prove” modules are correct, composing the modules’ behaviors to determine the system’s behavior is hard
- Leveson and others have shown clearly that a system can fail even when each of the pieces work properly – this is because many systems have “emergent” properties
- Arguments are common about the need to build “security” and “safety” and … in from the beginning

Correspondence

- “Problem-program mapping”
- The way in which the design is associated with the requirements
- The idea is that the simpler the mapping, the easier it will be to accommodate change in the design when the requirements change
- M. Jackson: problem frames
  - In the style of Polya

Defining structure

Almost always a key part of architecture

- The focus of most design approaches is structure
- What are the components and how are they put together?
- Behavior is important, but often largely indirectly

The technique of mastering complexity has been known since ancient times: Divide et impera (Divide and Rule). —Dijkstra, 1965

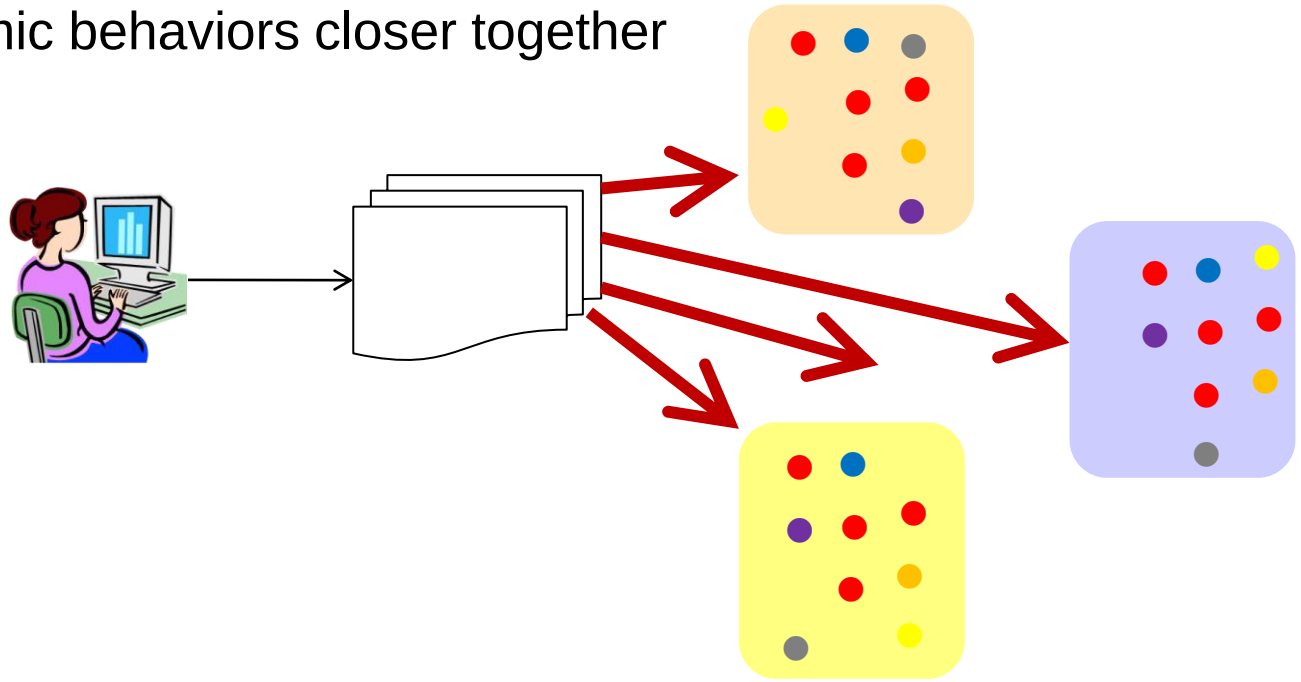
…as soon as the programmer only needs to consider intellectually manageable programs, the alternatives he is choosing from are much, much easier to cope with. —Dijkstra, 1972

The complexity of the software systems we are asked to develop is increasing, yet there are basic limits upon our ability to cope with this complexity. How then do we resolve this predicament?

—Booch, 1991

Structured programming

- Dijkstra observed that a programmer manipulates source code to modify program behaviors – obvious, perhaps, but oft-ignored
- He noted that any gap between the static structure of the program and its dynamic behaviors can cause confusion as people are better able to reason about static than dynamic structures
- Using one-in-one-out control structures brought the static structure and dynamic behaviors closer together

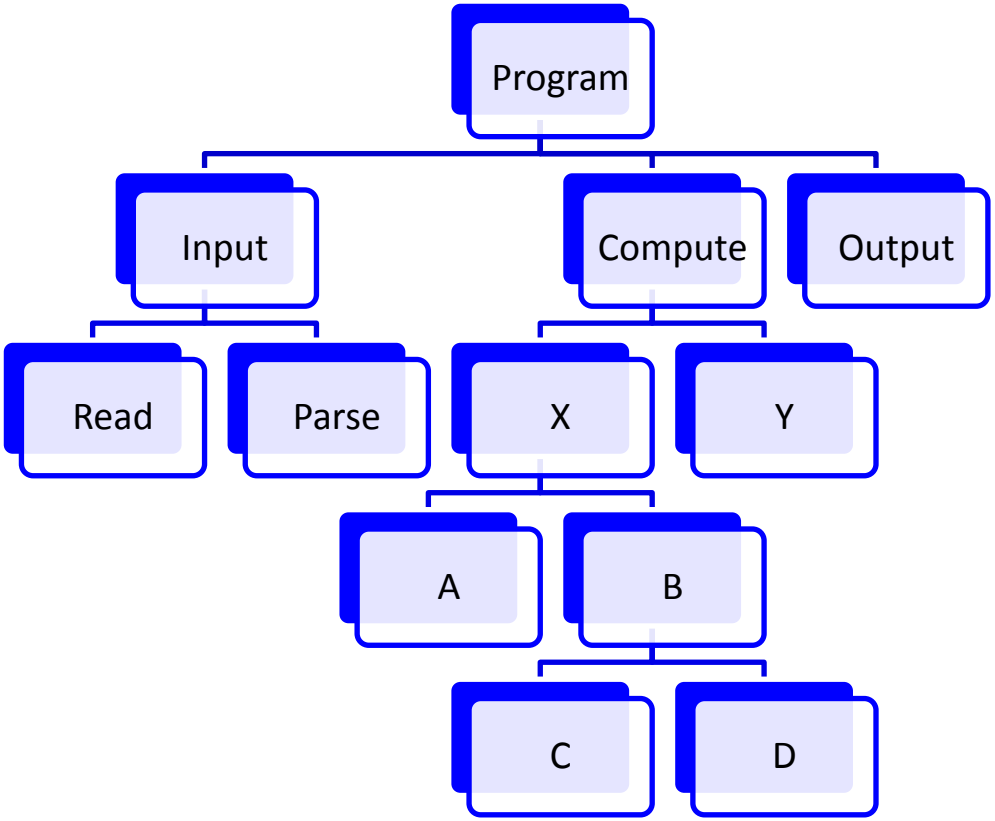


Functional decomposition I

- Divide-and-conquer based on functions
  - input;
  - compute;
  - output
- Then proceed to decompose compute
- This is stepwise refinement [Wirth, Dijkstra, Hoare, …]
- There is an enormous body of work in this area, including many formal calculi to support the approach
  - Closely related to proving programs correct
  - Weakest preconditions, Hoare triples, …

Functional decomposition II

- Heavily focused on creating implementations that satisfy specifications
- More effective in the face of stable specifications
  - Which decisions must be repeated?



Interlude: it's all about you!

- Companies, organizations?
- Job focus (development, testing, etc.)?
- Domain?
- Undergraduate degree in …?
- …

A shift to a focus on change
Parnas CACM 1972

"A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶I0.23]…:

> "A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. …

"Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules."

- The major criterion observed by Parnas is to identify likely changes and design to make them easier to accommodate

Accommodating change

- "…accept the fact of change as a way of life, rather than an untoward and annoying exception."
  —Brooks, 1974
- "Software that does not change becomes useless over time." —Belady and Lehman
- It is generally believed that to accommodate change one must anticipate possible changes
  - Counterpoint: Extreme Programming
- By anticipating changes, one defines additional criteria for guiding the design activity
- But it is not possible to anticipate all changes

Information hiding

- Information hiding is perhaps the most important intellectual tool developed to support software design
  - Makes the anticipation of change a centerpiece in decomposition into modules
- Provides the fundamental motivation for abstract data type (ADT) languages
  - And thus a key idea in the OO world, too
- The conceptual basis is key

Basics of information hiding

- Modularize based on anticipated change
  - Fundamentally different from Brooks' approach in OS/360 (see old and new MMM)
- Separate interfaces from implementations
  - Implementations capture decisions likely to change
  - Interfaces capture decisions unlikely to change
  - Clients know only interface, not implementation
  - Implementations know only interface, not clients
- Modules are work assignments

Outstanding questions

- Can we effectively anticipate changes?
- Is it true that changing an implementation is the best change, since it's isolated?
- What does it mean for the semantics of the module to remain unchanged when implementations are modified?
- To what degree can one implementation satisfy multiple clients?

Information Hiding and OO

- Are these the same? No
  - OO classes are chosen based on the domain of the problem (in most OO analysis approaches)
  - Not necessarily based on change
- But they are obviously related (separating interface from implementation, e.g.)
- Modeling vs. lower-level design confuses the issue somewhat

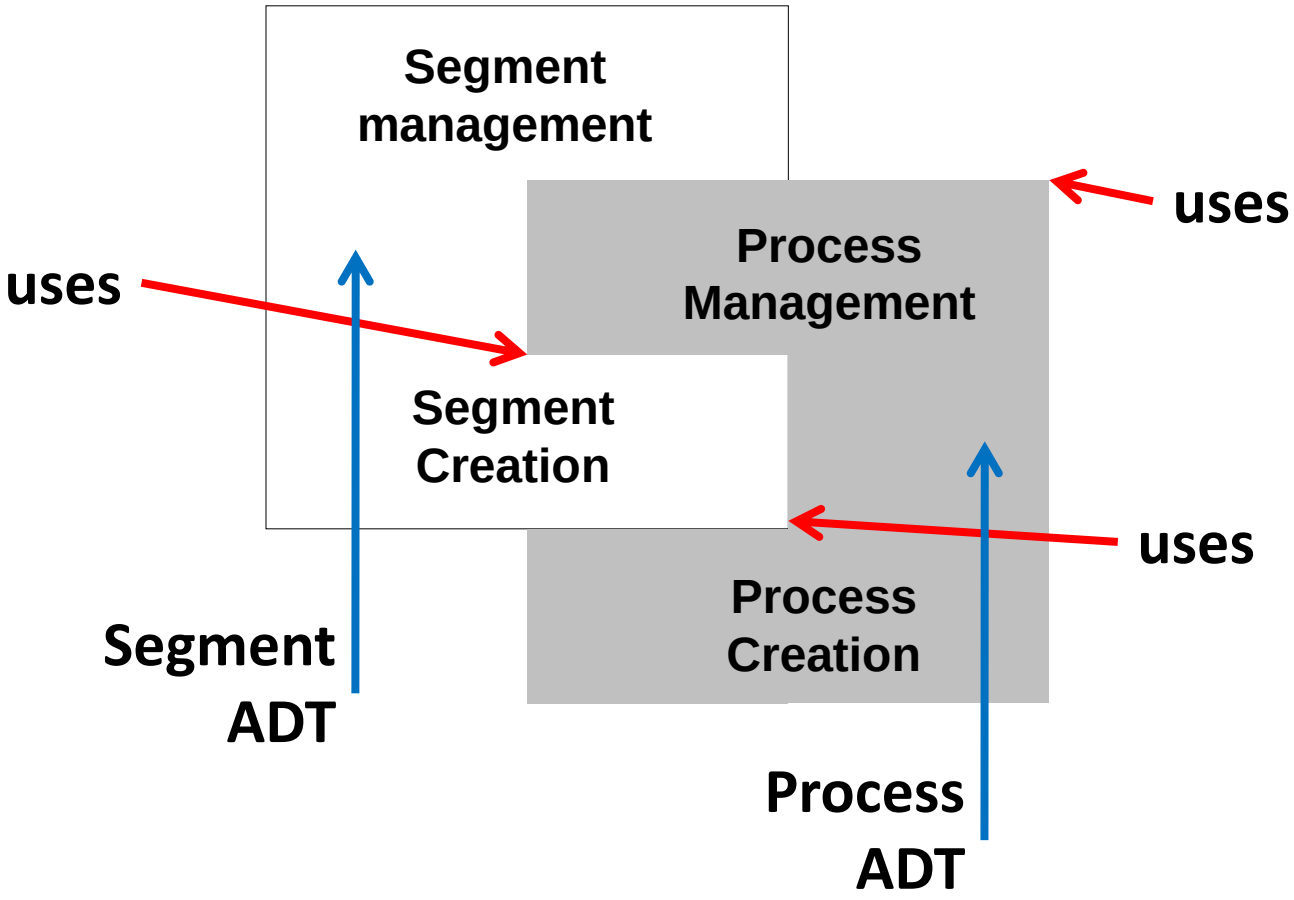
Layering

- Parnas also considered abstract machines (layers) in support of program families [1979]
  - Systems that have “so much in common that it pays to study their common aspects before looking at the aspects that differentiate them”
- Still focused on anticipated change

The uses(A,B) relation

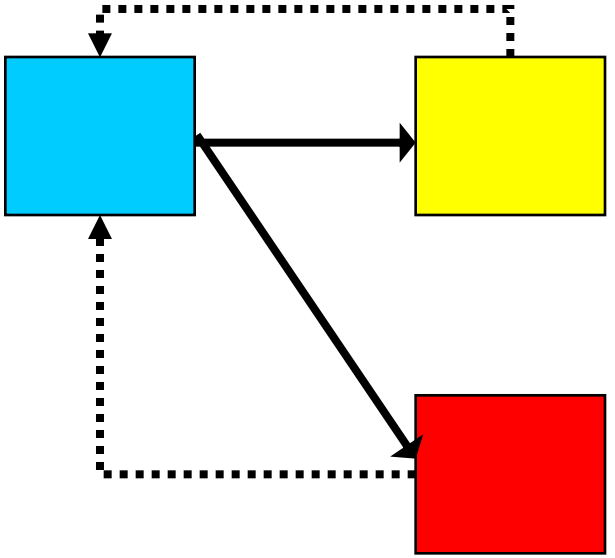
- A program A uses a program B if the correctness of A depends on the presence of a correct version of B
- A non-hierarchical uses relation makes it difficult to produce useful subsets of a system
- So, it is important to design the uses relation
  - A is essentially simpler because it uses B
  - B is not substantially more complex because it does not use A
  - There is a useful subset containing B but not A
  - There is no useful subset containing A but not B

Modules and layers are orthogonal



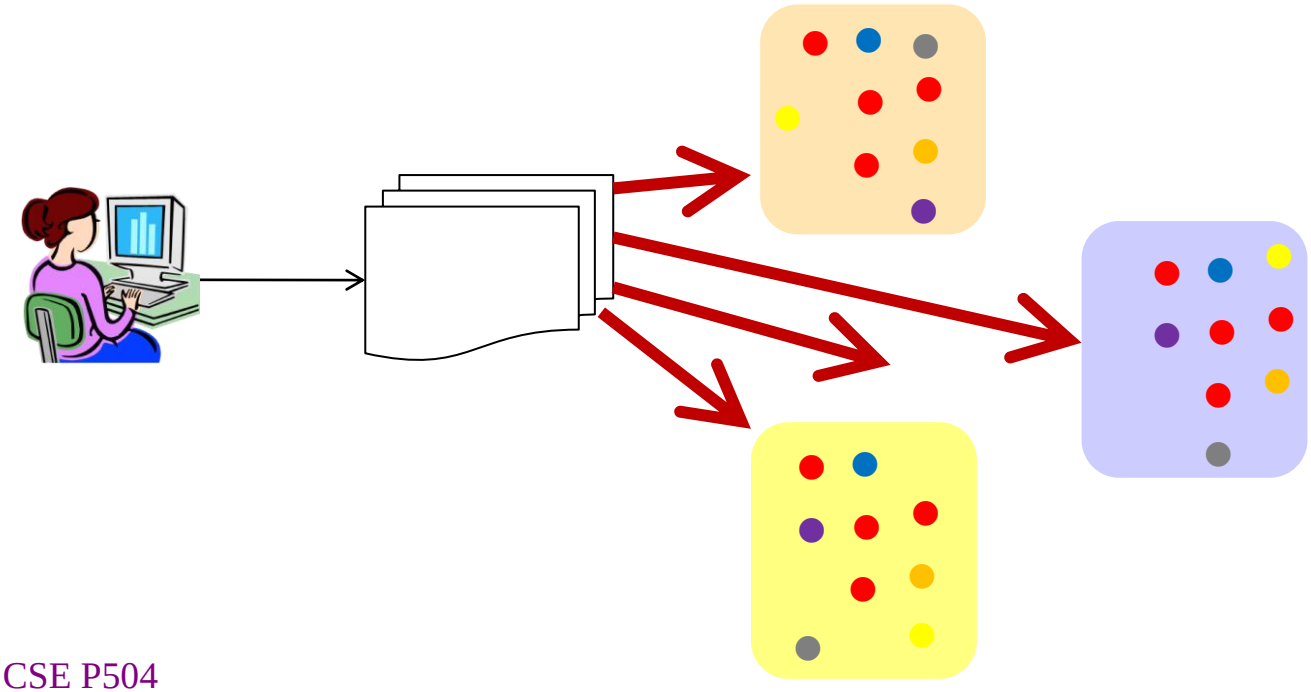
Implicit invocation: event-based design

- Again, a focus on change
- Components announce events that other components can choose to respond to
- Yellow and red register interest in an event from blue
  - When blue announces that event, yellow and red are invoked
- In implicit invocation, the invokes relation is the inverse of the names relation
- Invocation does not require ability to name



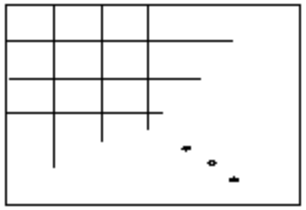
Dijkstra reprise: static ~ dynamic

- The simpler static-dynamic relation Dijkstra advocated is made more complex by information hiding, by layering, by implicit invocation, etc.
- That is, we've collectively decided that the power of these mechanisms in practice dominates the desire for that simplicity



Aspect-oriented design

- Much work on aspect-oriented design came from concerns expressed by Kiczales about conventional information hiding: clients depend on some aspects of the underlying implementations in a broad variety of domains and situations
- What happens when the implementation strategy for a module depends on how it will be used? Aren't we supposed to separate policy from mechanism?
- Example: spreadsheet via many small windows?



```
for i = 1 to 100
  for j = 1 to 100
    mkwindow(100, 100, i*100, j*100);
  end
end
```

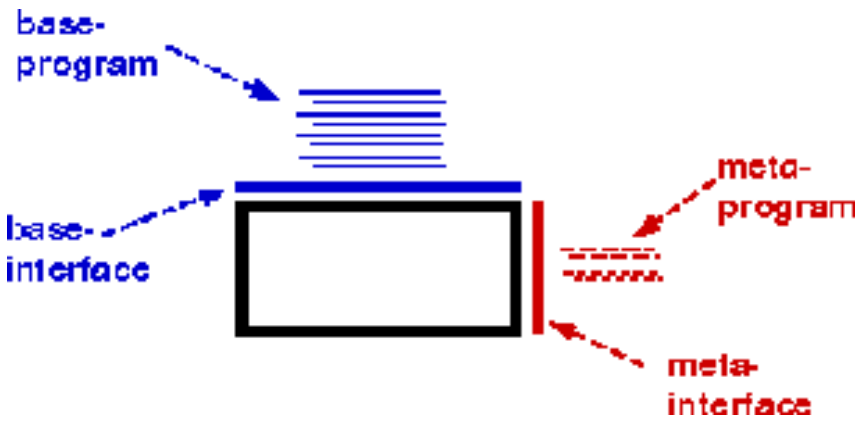
Poor client performance often leads to…

“hematomas of duplication”

“coding between the lines”

Open implementation

- Decompose into base interface (the “real” operations) and the meta interface (the operations that let the client control aspects of the implementation)
- Arose from work in (roughly) reflection in the Meta-Object protocol (MOP) and led to the development of aspect-oriented programming



Leap to aspect-oriented design

Slides modified from Kiczales

Bad modularity

- scattering – code spread around
- tangling – code in one region addresses multiple concerns

Good (better) modularity

separated – implementation of a concern can be treated as relatively separate entity

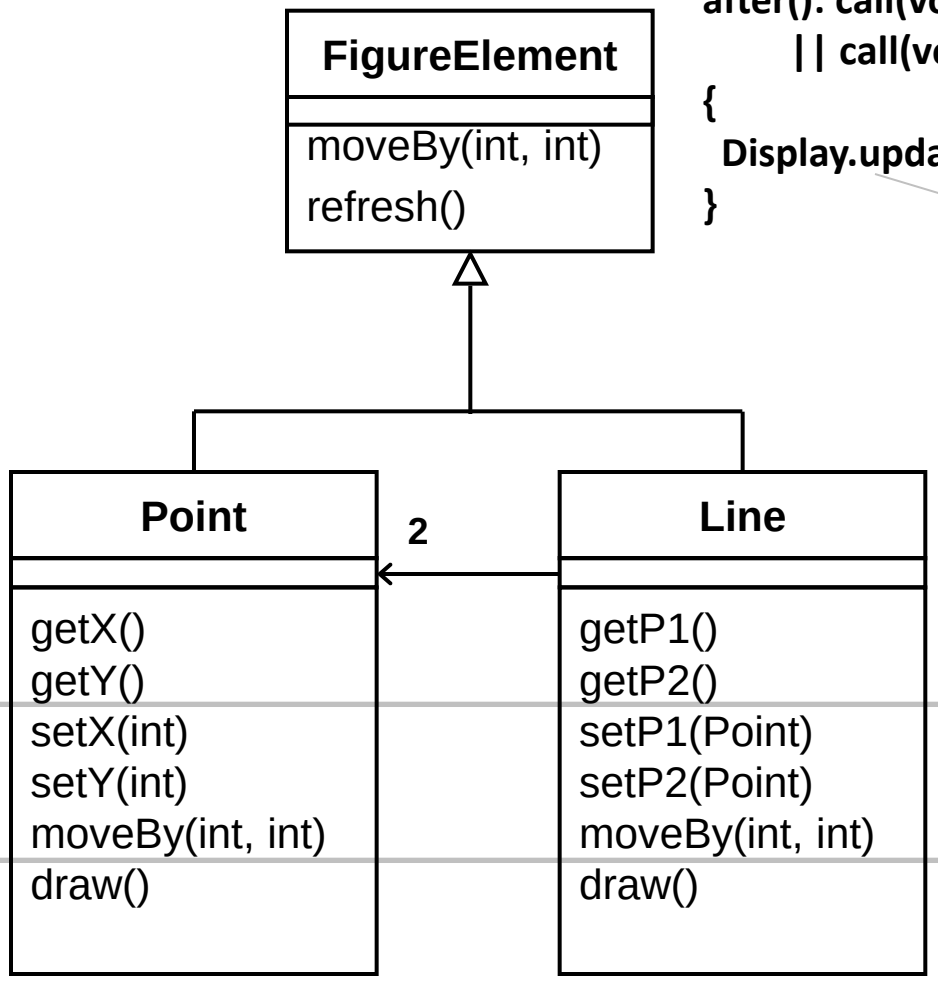
localized – implementation of a concern appears in one part of program

modular – above + has a clear, well defined interface to rest of system

Accounts of aspect-orientation

- problem: tyranny of single decomposition
  - cannot, in a single decomposition, modularize all concerns
  - bad modularity → brittle (non-adaptive), fragile, buggy mess
- principle: aspects …
  - concerns that are local in the alternate decomposition
  - aspect of is different than part of
- structure: crosscutting
  - relationship between decompositions such that
    - some concerns are localized (aspects)
    - and others are potentially spread out
- mechanism: join point mechanisms
  - coordinates effect of programs from different decompositions

AspectJ: sketch of an example




```
after(): call(void FigureElement+.set*(..))
      || call(void FigureElement.moveBy(int, int))
{
  Display.update();
}
```

DisplayUpdating

Administrivia

- You'll need your UW NetID
- Mailing list is set up (automatically using your NetID)
- No official office hours – phone, email, etc. posted on the web – and I try to be around a bit before and after lecture
- I will be gone for two lectures (January 25 and February 22) – Yuriy and Reid will lecture, respectively
- There are two holidays (January 18 and February 15)
- I haven't decided about March 15th – final or final class or …

Expected work

- Structured, 200-400 word reports on your choice of 10 assigned papers during the quarter (20% of grade, 2% each)
- Two state-of-the-research reports (60% of the grade, 30% each). These are secondary research reports
  - on topics that you select – and we approve – within the scope of the three parts of the course,
  - with identification of pertinent papers and materials (perhaps with help from us), perhaps some hands-on experience for some kinds of topics, and
  - a written, scholarly report on the topic and your analysis of it, complete with citations, open questions, etc.
- Class participation during lecture and on online forums (10%)
- Other 10% to be decided

Structured reports I

I. 1-2 sentences: what is the major claim of the paper?
II. 2-3 sentences: what is the form of the evidence supporting the claim?
III. 2-3 sentences: what are the strongest points of the paper?
IV. 2-3 sentences: what are the weakest points of the paper?
V. Remainder: other comments/questions about the work

- The suggested lengths are guidelines: you can use the space differently. But I-IV must be addressed explicitly

Structured reports II

- Due dates – these can come in earlier
  - 1st and 2nd: due 11PM Sunday January 17th
  - 3rd and 4th: due 11PM Sunday January 31th
  - 5th and 6th: due 11PM Sunday February 14th
  - 7th and 8th: due 11PM Sunday February 28th
  - 9th and 10th: due 11PM Sunday March 14th
- Done individually
- Submitted through UW Catalyst dropbox (link also on the 504 web) – you'll need your UW NetID
  - https://catalysttools.washington.edu/collectit/dropbox/notkin/8463

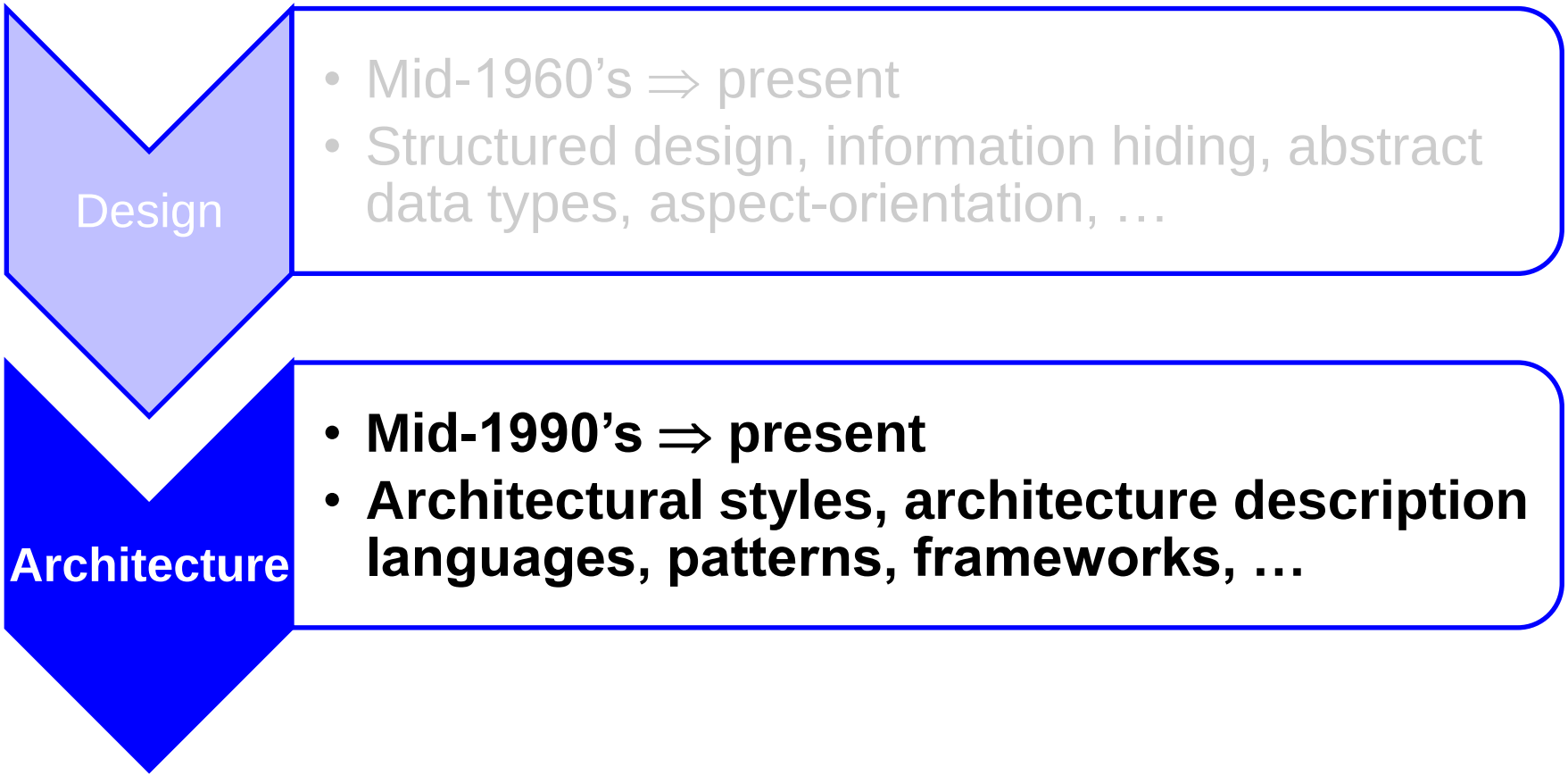
State-of-the-research reports

- Can be done individually or in groups of two or three
  - You can work differently for each report – it's up to you
  - Barring anything really unusual, all participants in a group share the same grade for a report
- The reports will be posted for comment by the staff and other students in the course (details forthcoming)
- The web points at some "roadmap" papers that have a general feel like what we expect
- Due dates are yet to be decided
  - There will be an earlier due date (for each report) for the agreement upon a topic, a first-cut at papers to read, identification of the group (if any), etc.

Class participation

- It's all about me – and I don't learned a darned thing if you don't participate
- Questions and comments in lecture
- Questions and comments outside of lecture
- Comments and discussions on the state-of-the-research reports
- …

From software design to architecture



Software Architecture

Perry and Wolf. Foundations for the Study of Software Architecture. ACM Software Engineering Notes (Oct. 1992)

"…architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and the interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design."

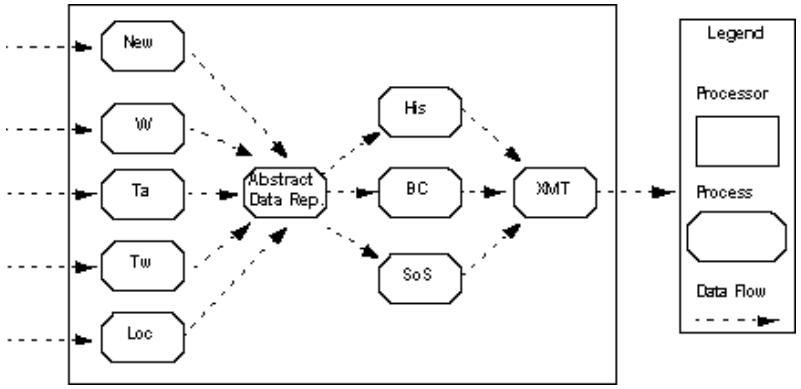
Garlan and Shaw. Software Architecture Perspectives on an Emerging Discipline (1996) [TR 1994]

"...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 gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives."

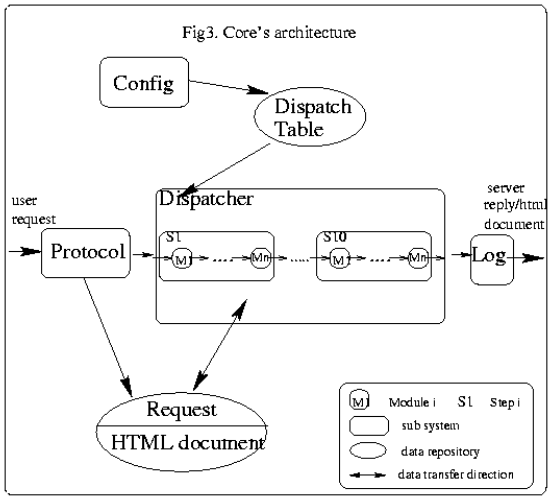
Two primary objectives

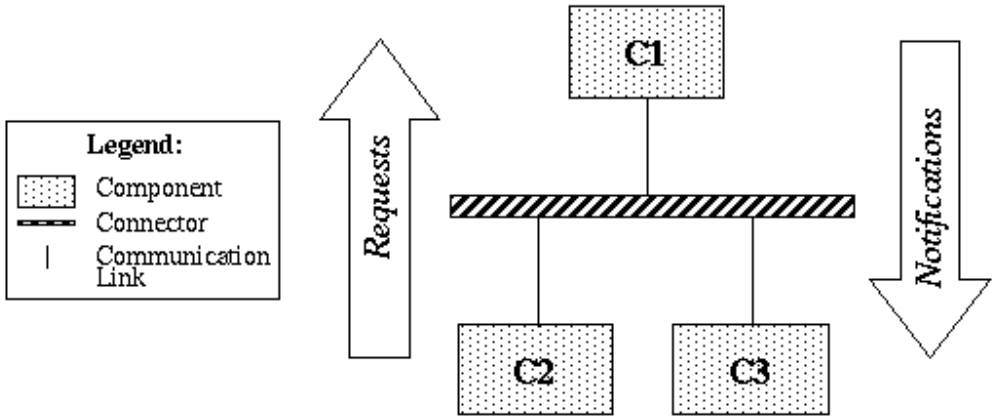
- Capturing, cataloguing and exploiting experience in software designs
- Allowing reasoning about classes of designs

Box-and-arrow diagrams:
taken from the web without attribution



Parallel application		
MPICH		
GM	FM	TCP / IP
Myrinet		Fast Ethernet





These diagrams have value

- You can find them all over the web, in textbooks, in technical documents, in research papers, over whiteboards in your office, on napkins in the cafeteria, etc.
- Another Christopher Alexander quotation: “Drawings help people to work out intricate relationships between parts.”
- At the same time, they are generally ill-defined: what does a box represent? an arrow? a layer? adjacent boxes? etc.
- One view of software architecture research is to determine ways to give these diagrams clearer semantics and thus additional value

Compilers I

- The first compilers had *ad hoc* designs
- Over time, as a number of compilers were built, the designs became more structured
  - Experience yielded benefits
    - Compiler phases, symbol table, etc.
  - Plenty of theoretical advances
    - Finite state machines, parsing, ...

Compilers II

- Compilers are perhaps the best example of shared experience in design
  - Lots of tools that capture common aspects
  - Undergraduate courses build compilers
  - Most compilers look pretty similar in structure
- But we still don't fully generate compilers
  - Despite lots of effort and lots of money
  - In any case, the code in compilers is often less clean than the designs
- Despite this, the perception of a shared design gives leverage
  - Communication among programmers
  - Selected deviations can be explained more concisely and with clearer reasoning

Examples of architectures

- Blackboard
- Client-server
- Database-centered architecture
- Distributed computing
- Event-driven architecture
- Peer-to-peer
- Pipes and filters
- Plugin
- Service-oriented architecture
- Three-tier model
- …

Another motivation: architectural mismatch

- Garlan, Allen, Ockerbloom tried to build a toolset to support software architecture definition from existing components
  - OODB (OBST)
  - graphical user interface toolkit (Interviews)
  - RPC mechanism (MIG/Mach RPC)
  - Event-based tool integration mechanism (Softbench)
- It went to hell in a hand-basket, not because the pieces didn't work, but because they didn't fit together
- Architectural Mismatch: Why Reuse Is So Hard. IEEE Software (Nov. 1995)
  - Reprised in July 2009: Architectural Mismatch: Why Reuse Is Still So Hard.

Mismatches included

- Excessive code size
- Poor performance
- Needed to modify out-of-the-box components (e.g., memory allocation)
- Error-prone construction process
- …

- The claim is that many of the problems were of an architectural nature
  - What assumptions are made, need they be made, etc.?
- With some forethought, many of these mismatches could, in principle, be avoided

Some classic definitions:

http://www.sei.cmu.edu/architecture/definitions.html

- An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization---these elements and their interfaces, their collaborations, and their composition [Booch, Rumbaugh, and Jacobson, 1999]
- The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time [Wolf and Perry].
- ...an abstract system specification consisting primarily of functional components described in terms of their behaviors and interfaces and component-component interconnections [Hayes-Roth].

Components and connectors

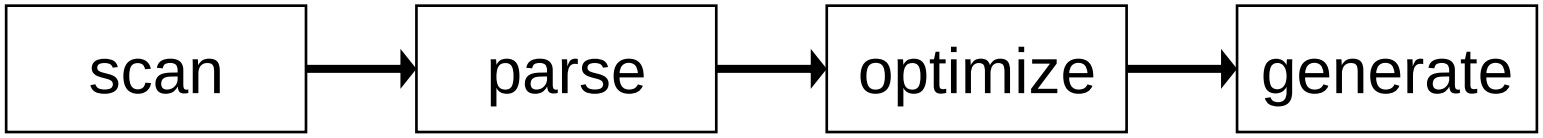
- Software architectures are generally seen as comprising components and connectors
- Components define the basic computations comprising the system
  - Abstract data types, filters, etc.
- Connectors define the interconnections between components
  - Procedure call, event announcement, asynchronous message sends, etc.
- The line between them may be fuzzy at times
  - Ex: A connector might (de)serialize data, but can it perform other, richer computations? How about encryption/decryption?

Architectural style

- Defines the vocabulary of components and connectors for a family (style)
- Constraints on the elements and their combination
  - Topological constraints (no cycles, register/announce relationships, etc.)
  - Execution constraints (timing, etc.)
- By choosing a style, one gets all the known properties of that style
  - For any given architecture in that style
- These properties can be quite broad
  - Ex: performance, lack of deadlock, ease of making particular classes of changes, etc.

Not just boxes and arrows

- Consider pipes & filters, for example (Garlan and Shaw)
  - Pipes must compute local transformations
  - Filters must not share state with other filters
  - There must be no cycles
- If these constraints are not satisfied, it's not a pipe & filter system
  - One can't tell this from a picture
  - One can formalize these constraints (we'll come back to this next week)

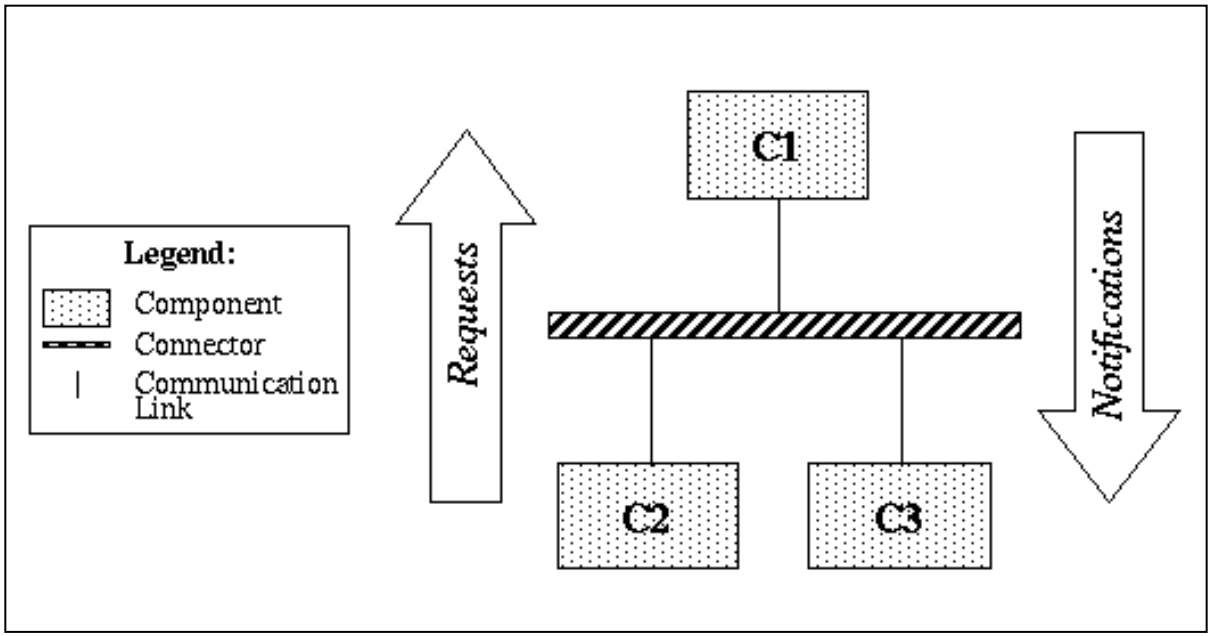


Specializations

- Architectural styles can have specializations
  - A pipeline might further constrain an architecture to a linear sequence of filters connected by pipes
  - A pipeline would have all properties that the pipe & filter style has, plus more

C2 Architecture:
UC Irvine (Taylor et al.)

- Based on generalization of a collection of designs of user interface systems http://www.ics.uci.edu/pub/c2/c2.html
- Informally, a C2 architecture is a network of concurrent components linked together by connectors



C2 Composition

- The top of a component may be connected to the bottom of a single connector
- The bottom of a component may be connected to the top of a single connector
- There is no bound on the number of components or connectors that may be attached to a single connector
- When two connectors are attached to each other, it must be from the bottom of one to the top of the other

C2 Communication

- Solely by exchanging messages
- Each component has a top and bottom domain
  - The top specifies the set of notifications to which a component responds, and the set of requests it emits upwards
  - The bottom specifies the set of notifications that a component emits downwards and the set of requests to which it responds
- Central principle: limited visibility (substrate independence)
  - A component within the hierarchy can only be aware of components “above” it and is completely unaware of the components “beneath” it

What's coming later on?

- Design-based software architecture
  - Formal reasoning (WRIGHT, etc.)
  - Static vs. dynamic architectures
- Property-based software architecture
  - Autonomic systems
  - Relationship to model-based design
  - Examples from automotive or related domains
- Inspiration from biological systems … and an example in some depth

Suggestions for third topic…

- …after architecture and tools?

Questions?