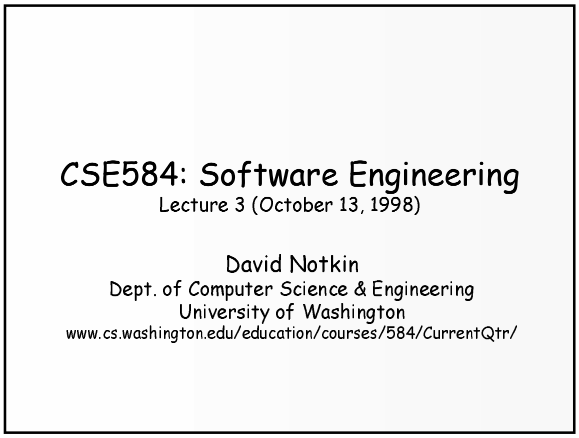
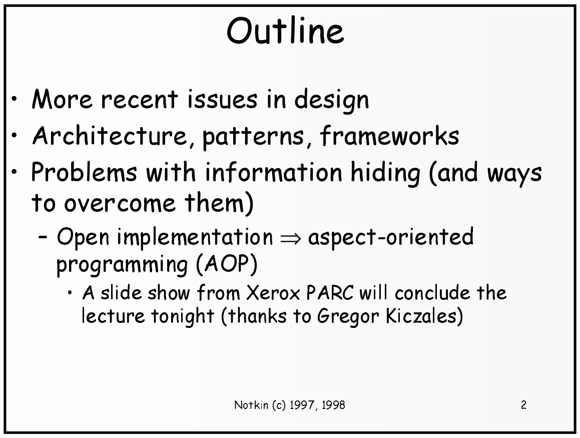


CSE584: Software Engineering

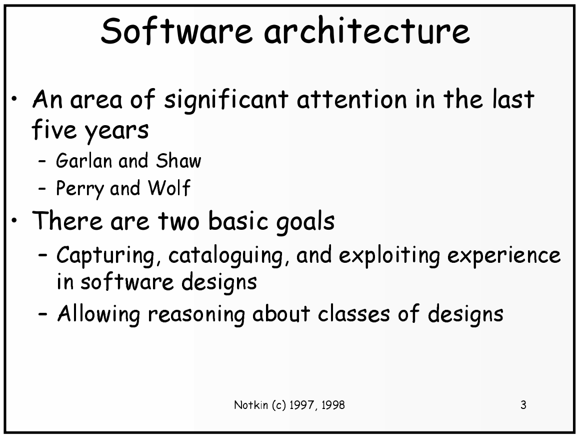
Lecture 3 (October 13, 1998)

David Notkin
Dept. of Computer Science & Engineering
University of Washington
www.cs.washington.edu/education/courses/584/CurrentQtr/



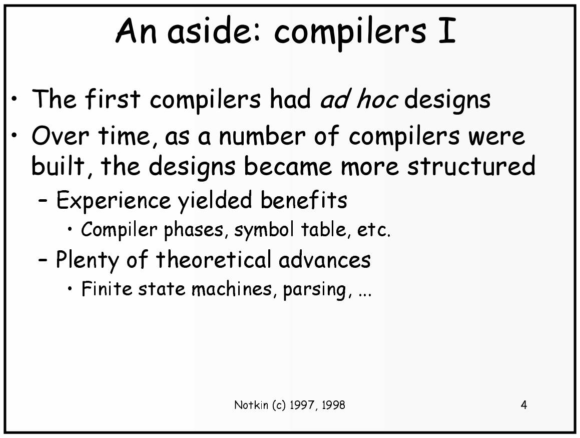
Outline

- More recent issues in design
- Architecture, patterns, frameworks
- Problems with information hiding (and ways to overcome them)
  - Open implementation ⇒ aspect-oriented programming (AOP)
    - A slide show from Xerox PARC will conclude the lecture tonight (thanks to Gregor Kiczales)



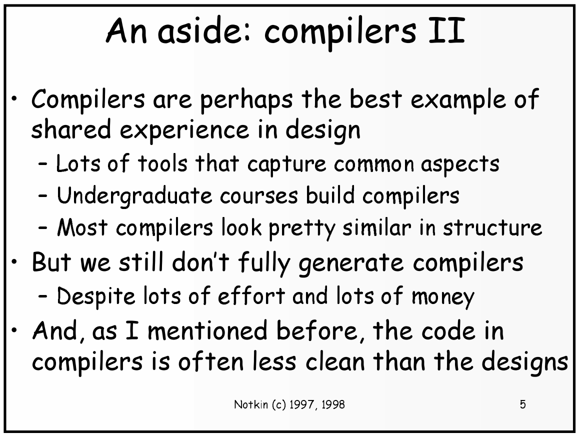
Software architecture

- An area of significant attention in the last five years
  - Garlan and Shaw
  - Perry and Wolf
- There are two basic goals
  - Capturing, cataloguing, and exploiting experience in software designs
  - Allowing reasoning about classes of designs



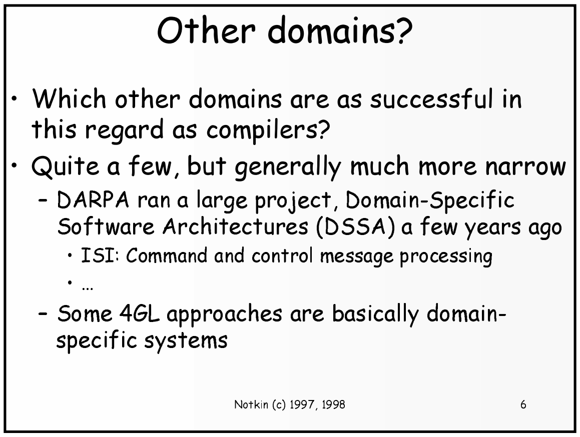
An aside: compilers I

- 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, …



An aside: 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
- And, as I mentioned before, the code in compilers is often less clean than the designs



Other domains?

- Which other domains are as successful in this regard as compilers?
- Quite a few, but generally much more narrow
  - DARPA ran a large project, Domain-Specific Software Architectures (DSSA) a few years ago
    - ISI: Command and control message processing
    - …
  - Some 4GL approaches are basically domain-specific systems

Back to software architecture

- The hope is that by studying our experiences with a variety of systems, we can gain leverage as we did with compilers
- Capture the strengths and weaknesses of various software structures
  - Perhaps enabling designers to select appropriate architectures more effectively
- Benefit from high-level study of software structure

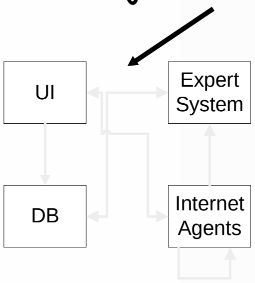
Components and connectors

- Software architectures are composed of *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, etc.
  - The line between them may be fuzzy at times
    - Ex: A connector might (de)serialize data, but can it perform other, richer computations?

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

Not just boxes and arrows



- Consider pipes & filters
  - 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

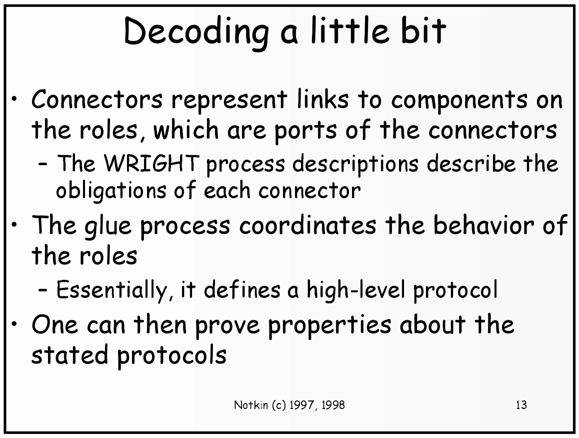
WRIGHT

- WRIGHT provides a formal basis for architectural description (it's an ADL)
  - Language for precisely defining an architectural specification
  - Basis for analyzing the architecture of individual software systems and families of systems
  - Underlying model in CSP, checkable using standard model checking technology
    - Defines a set of standard consistency and completeness checks

Pipe connector in WRIGHT

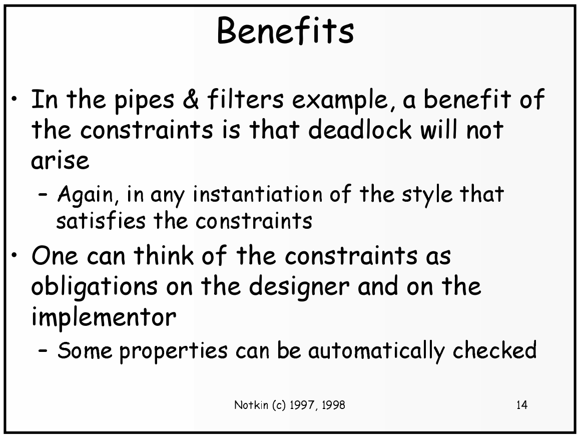
```
Connector Pipe =
  role Write = write → Writer ∇ close → √
  role Reader = let ExitOnly = close → √
                in let DoRead =
                   (read → Reader ◊
                            read-eof → ExitOnly)
                    in DoRead ◊ ExitOnly
  glue = let ReadOnly = Reader.Read → ExitOnly
                        ◊ Reader.read-eof →
                          Reader.close → √
                        ◊ Reader.close → √

  ...
```



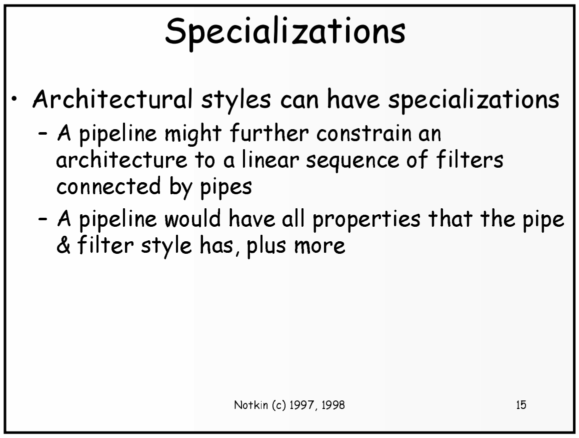
Decoding a little bit

- Connectors represent links to components on the roles, which are ports of the connectors
  - The WRIGHT process descriptions describe the obligations of each connector
- The glue process coordinates the behavior of the roles
  - Essentially, it defines a high-level protocol
- One can then prove properties about the stated protocols



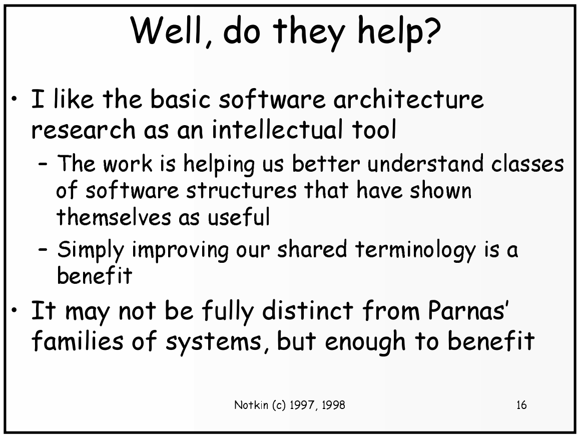
Benefits

- In the pipes & filters example, a benefit of the constraints is that deadlock will not arise
  - Again, in any instantiation of the style that satisfies the constraints
- One can think of the constraints as obligations on the designer and on the implementor
  - Some properties can be automatically checked



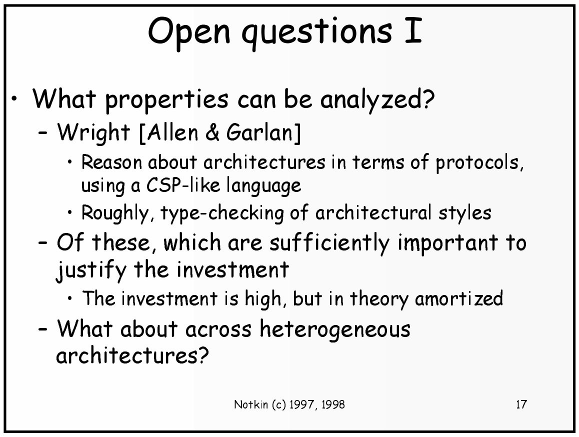
Specializations

- 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



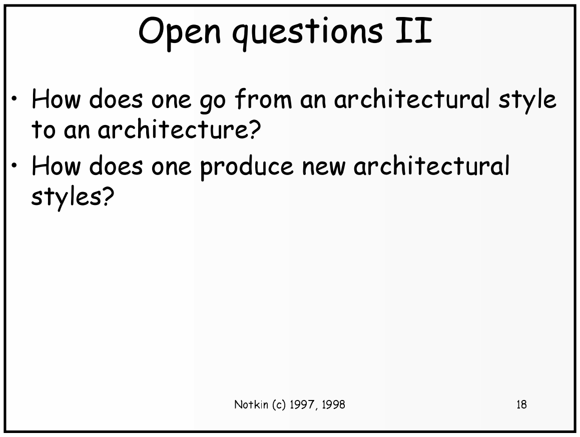
Well, do they help?

- I like the basic software architecture research as an intellectual tool
  - The work is helping us better understand classes of software structures that have shown themselves as useful
  - Simply improving our shared terminology is a benefit
- It may not be fully distinct from Parnas' families of systems, but enough to benefit



Open questions I

- What properties can be analyzed?
  - Wright [Allen & Garlan]
    - Reason about architectures in terms of protocols, using a CSP-like language
    - Roughly, type-checking of architectural styles
  - Of these, which are sufficiently important to justify the investment
    - The investment is high, but in theory amortized
  - What about across heterogeneous architectures?



Open questions II

- How does one go from an architectural style to an architecture?
- How does one produce new architectural styles?

Open questions III

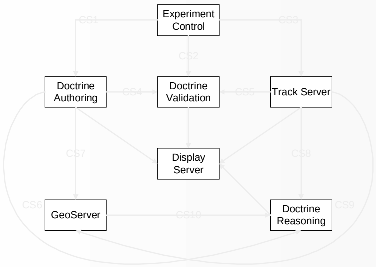
- What is the relationship between architectural and implementation?
  - Does architectural information aid in going from design to implementation?
  - What happens as the implementation evolves in ways inconsistent with the architecture?
    - Which properties still hold, and how do we know this?

Experience

- It's a hot area, with lots of companies paying attention
- Allen & Garlan recently reported on a case study in applying architectural modeling to the AEGIS Weapons System
  - Used formalism to help "expose and resolve some of the architectural problems that arose in implementing the system"
- Similar advantages for the HLA project

AEGIS

- AEGIS Weapons System, control of US Navy ships
  - Model problem for work in software architecture



Example benefits in AEGIS

- Clarifying client-server misconceptions
  - Which party initiated interactions?
  - Re-established after every request?
  - Synchronous or asynchronous?
- WRIGHT used to clarify
  - Avoiding deadlocks
  - Reducing unnecessary synchronization
  - And to simplify instrumentation of the architecture

Forcing discussions

- In some ways, the primary benefit of architecture a la Garlan is that it forces discussions of some critical issues
  - The Xerox PARC Mesa/Cedar group did roughly the equivalent by spending enormous amounts of times in defining and clarifying interfaces, before coding
- I'm unsure the degree to which the formalism per se helps, although there are surely some supporting examples

On-going research

- Environments to support the design of architectural styles and architectures
- Architectural design languages (ADLs)
- Formal models of architectures
- Architectural case studies
- Use of informal architectures
- …

Design patterns

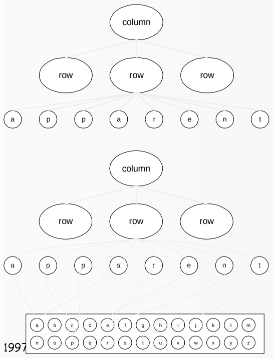
- Design patterns are idioms that are intended to be *"simple and elegant solutions to specific problems in object-oriented software design."*
- They are drawn from actual software systems
- They are intended to be language-independent

A weak analogy

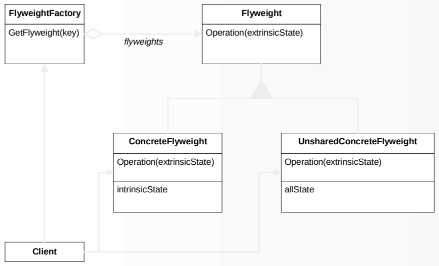
- I view high-level control structures in programming languages as quite the same
  - For example, a while loop is an idiomatic collection of machine instructions
- Knuth's 1974 article ("Structured Programming with go to Statements") shows that this is not a language issue alone
- Patterns are a collection of "mini-architectures" that combine structure and behavior

Example: **flyweight** [Gamma et al.]

- Intent
  - Use sharing to support many fine-grained objects efficiently
  - Can't usually afford to have small elements (like characters) be full-fledged objects
- Separate logical model from physical model



Flyweight structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Categories of patterns

- Creational
- Structural
- Behavioral

An enlightening experience

- At a workshop a year or two ago, I had an experience with two of the Gang of Four
- They sat down with Griswold and me to show how to use design patterns to (re)design a software design we had published
- The rate of communication between these two was unbelievable
  - And much of it was understandable to us without training (good sign for a learning curve)

This is the real thing

- Design patterns are not a silver bullet
- But they are impressive, important and worthy of attention
- I think that (slowly?) some of the patterns will become part and parcel of designers' vocabularies
  - This will improve communication and over time improve the designs we produce
- The relatively disciplined structure of the pattern descriptions may be a plus

The future

- I'm somewhat worried that "second wave" R&D will hurt more than help
  - They may be considered a panacea
  - They are surely going to be misunderstood
    - Everything now is a "pattern", even if it doesn't have the key characteristics
  - Tools and languages for patterns may help, but may also hinder
- How do patterns interact?

Patterns resources

- Patterns Home Page
  - http://st-www.cs.uiuc.edu/users/patterns/patterns.html
- Portland Pattern Repository
  - http://c2.com/ppr/index.html
- FAQ
  - http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html
- Gang of Four book
  - Design Patterns: Elements of Reusable Object-Oriented Software. Gamma et. al. (as of 10/12/98 @ 12:45PM PDT, Amazon sales rank of 173)
- OO journals, OOPSLA, etc.

Do any of you use patterns?

Frameworks

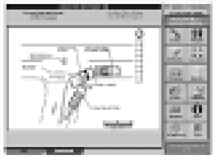
- Frameworks are another design buzzword
- One way to think about them is as upside-down layers
  - That is, layered systems allow us to construct families of systems by sharing lower layers
  - Frameworks allow us to construct families of systems by sharing upper "layers"
- Instantiate and specialize provided classes
  - "More" than patterns

Examples

- DuPont's business model
  - `http://www-cat.ncsa.uiuc.edu/~yoder/Research/catdesc.html`
  - Visual table-based framework for improving financial decisions, etc.
- CHOICES: customizing operating systems
  - `http://choices.cs.uiuc.edu/choices/choices.html`
  - Frameworks for VM, memory management, process management, file storage, exceptions and hardware device drivers, distributed processing and communication

A commercial example

- Visio is in many ways a framework
- It is also a complete application on its own, but it can be specialized (in a number of ways) that is consistent with being a framework



Open implementation

- Last week in discussing information hiding I listed some central premises
- Two important ones are especially questionable
- Kiczales et al. have studied this question carefully, leading to some work generally called Open Implementation
  - http://www.parc.xerox.com/spl/projects/oi/

Central premises III and IV

- The semantics of the module must remain unchanged when implementations are replaced
  - Specifically, the client should not care how the interface is implemented by the module
- One implementation can satisfy multiple clients
  - Different clients of the same interface that need different implementations would be counter to the principle of information hiding
    - Clients should not care about implementations, as long as they satisfy the interface

These are often false

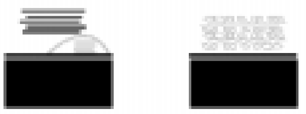
- What defines the semantics of the interface?
  - Much is not (cannot?) be defined, but is inferred by the client
- Once properties are inferred, clients start to assume that they are true
- Multiple clients may infer different properties
  - So changing those properties consistently may be impossible
- Client do, in practice, care about (aspects of) the implementation

Examples

- The flyweight pattern example points out a few of these issues
- Logically, any implementation of the interface is OK
  - But not all implementations are equally adequate for all clients
- The Kiczales spreadsheet example

Two approaches often taken

- Programmers often respond to these problems in one of two ways
  - Write own windowing system
  - Clever coding tricks
    - Paging example

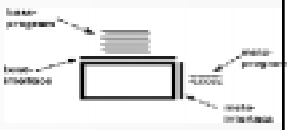


The experts say

- "I found a large number of programs perform poorly because of the language's tendency to hide `what is going on' with the misguided intention of `not bothering the programmer with details'"
  - N. Wirth, 1974
- "An interface should capture the *minimum* essentials of an abstraction.
- "When an interface undertakes to do too much, the result is a large, slow complicated implementation."
  - B. Lampson, 1984

The OI solution

- Define two interfaces
  - The *base interface*, which provides the essential semantics
  - The *meta-interface*, which is used to customize aspects of the implementation of the base
- Based on experience
  - Common Lisp Meta-Object Protocol (CLOS MOP)
  - Reflective computing



Allows the client to

- Use the module's primary functionality alone when the default implementation is adequate
- Control the module's implementation-strategy decisions when necessary
- Deal with functionality and implementation strategy decisions in largely separate ways

Design issues: OI claims

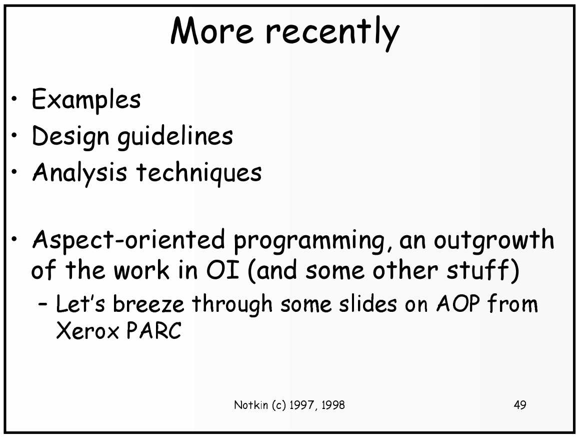
- The base interface design requires similar techniques to current interface design
- The design of the meta-interface and of the coupling of the meta- and base interface is more complicated
  - Requires expertise in the definition and uses of the components

Design issues: meta-interface

- Scope control
  - Are controls over the implementation for instances, classes, other?
- Conceptual separation & incrementality
  - Can the client of the meta-interface understand and use just parts of it?
- Robustness
  - Are bugs in a client's meta-program limited in effect?

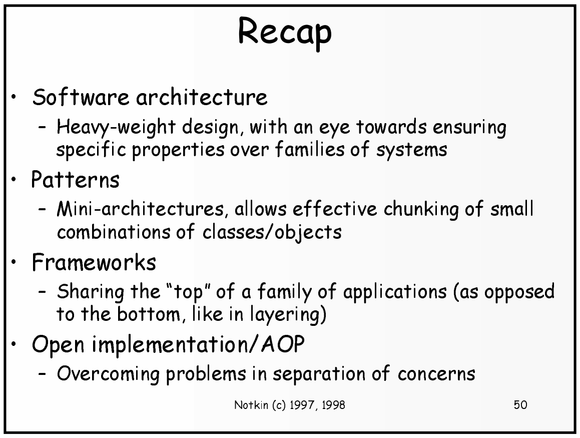
It's not an entirely new idea

- Compiler pragmas
- Multiple implementations of an interface
  - With client choice [Hermes]
- User-directed parallelization
- Unix `madvise`
  - Influence page replacement
- Many more



More recently

- Examples
- Design guidelines
- Analysis techniques
- Aspect-oriented programming, an outgrowth of the work in OI (and some other stuff)
  - Let's breeze through some slides on AOP from Xerox PARC



Recap

- Software architecture
  - Heavy-weight design, with an eye towards ensuring specific properties over families of systems
- Patterns
  - Mini-architectures, allows effective chunking of small combinations of classes/objects
- Frameworks
  - Sharing the "top" of a family of applications (as opposed to the bottom, like in layering)
- Open implementation/AOP
  - Overcoming problems in separation of concerns