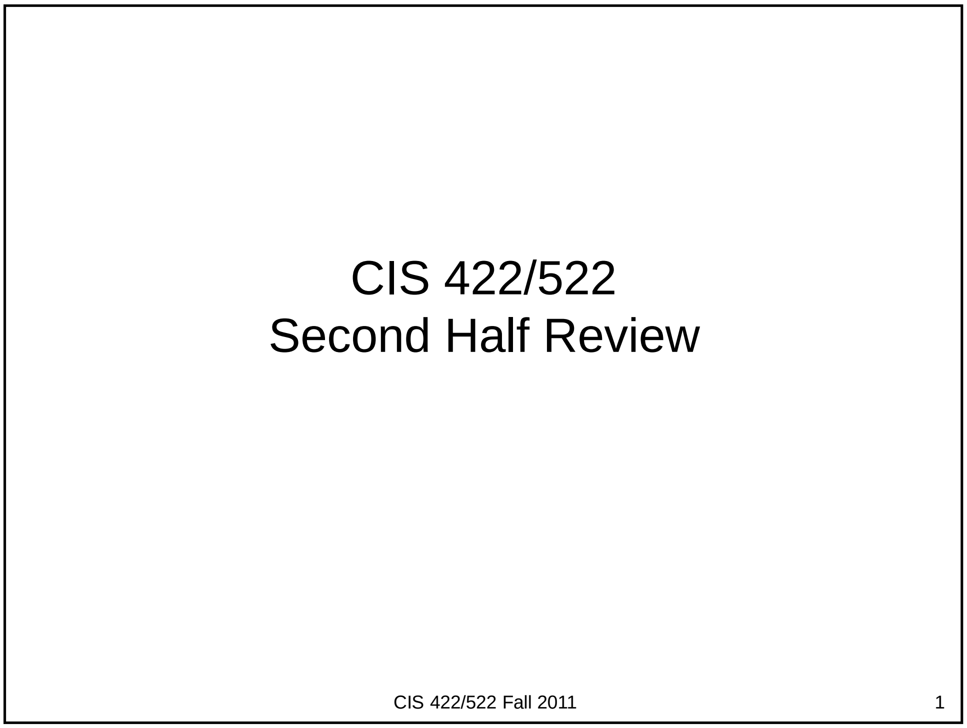
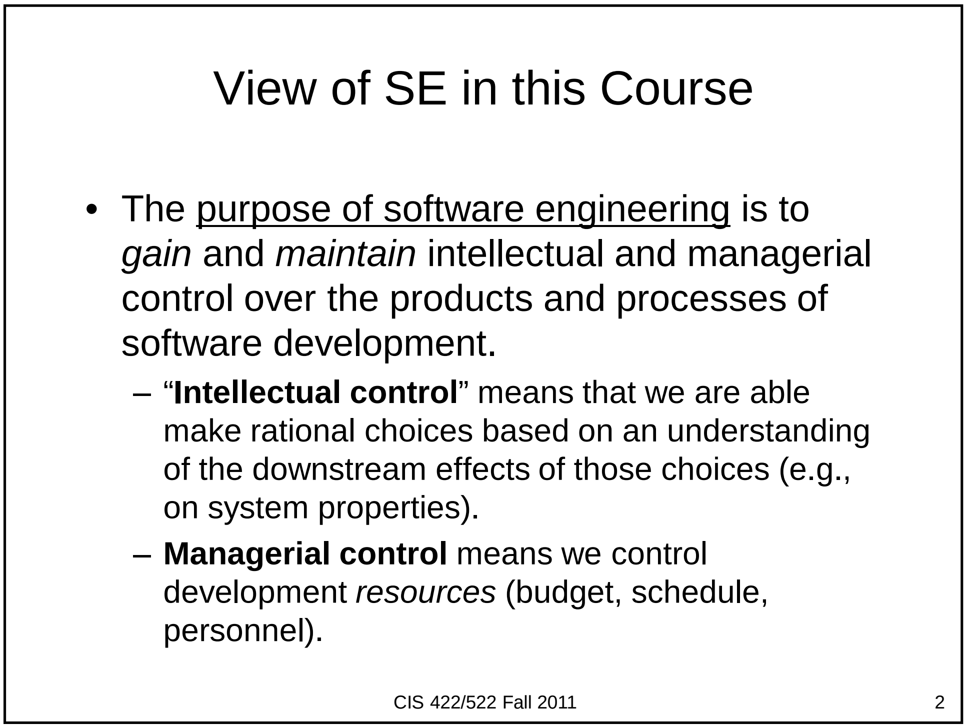
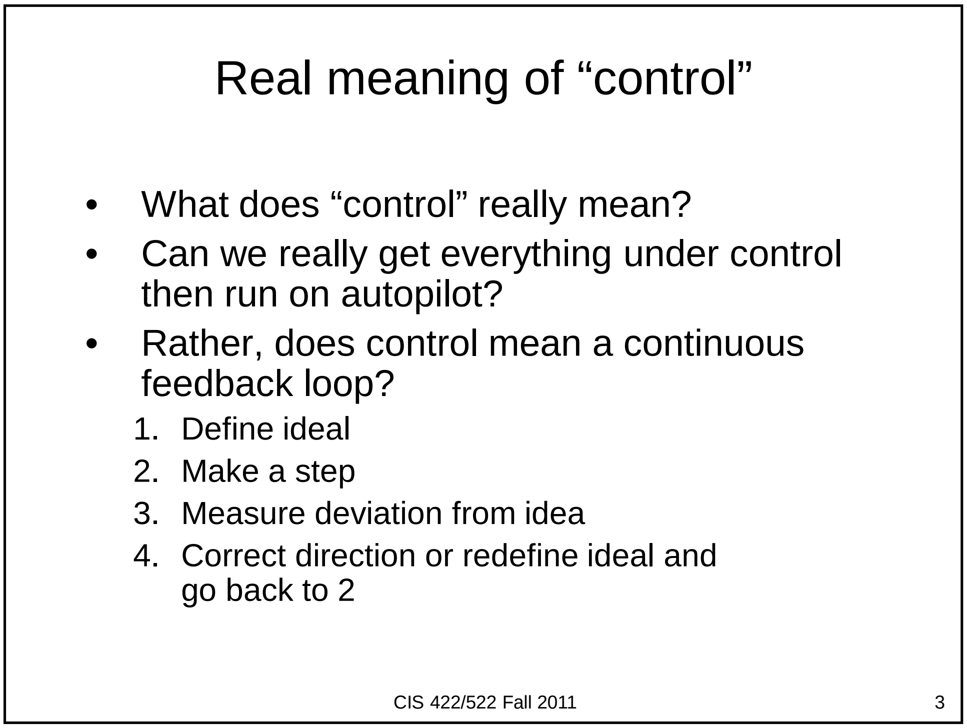


CIS 422/522
Second Half Review



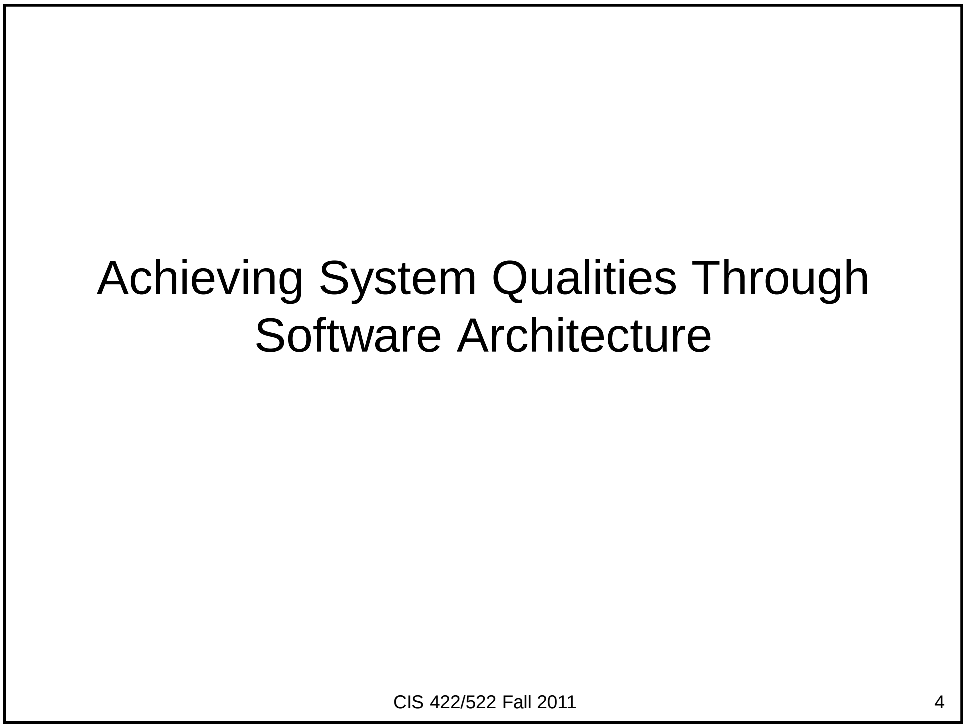
View of SE in this Course

- The <u>purpose of software engineering</u> is to *gain* and *maintain* intellectual and managerial control over the products and processes of software development.
  - “**Intellectual control**” means that we are able make rational choices based on an understanding of the downstream effects of those choices (e.g., on system properties).
  - **Managerial control** means we control development *resources* (budget, schedule, personnel).



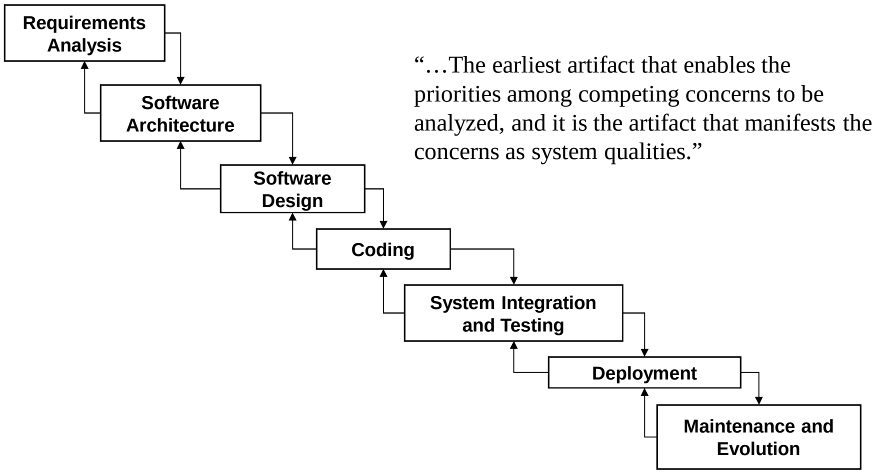
Real meaning of “control”

- What does “control” really mean?
- Can we really get everything under control then run on autopilot?
- Rather, does control mean a continuous feedback loop?
  1. Define ideal
  2. Make a step
  3. Measure deviation from idea
  4. Correct direction or redefine ideal and go back to 2



Achieving System Qualities Through Software Architecture

Fit in the Development Cycle



"…The earliest artifact that enables the priorities among competing concerns to be analyzed, and it is the artifact that manifests the concerns as system qualities."

Definition

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." - Bass, Clements, Kazman

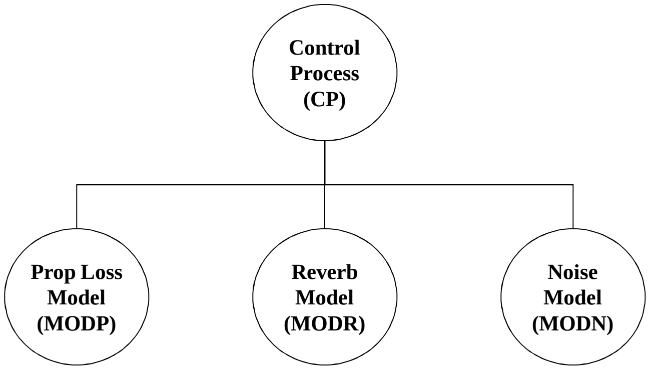
- Systems typically comprise more than one architecture
  - There is more than one useful decomposition into components and relationships
  - Each addresses different system properties or design goals
- It exists whether any thought goes into it or not!
  - Decisions are necessarily made if only implicitly
  - Issue is who makes them and when
- Many "architectural specifications" aren't

Examples: These are architectures

- **An architecture comprises a set of**
  - **Software components**
  - **Component interfaces**
  - **Relationships among them**
- **Examples**

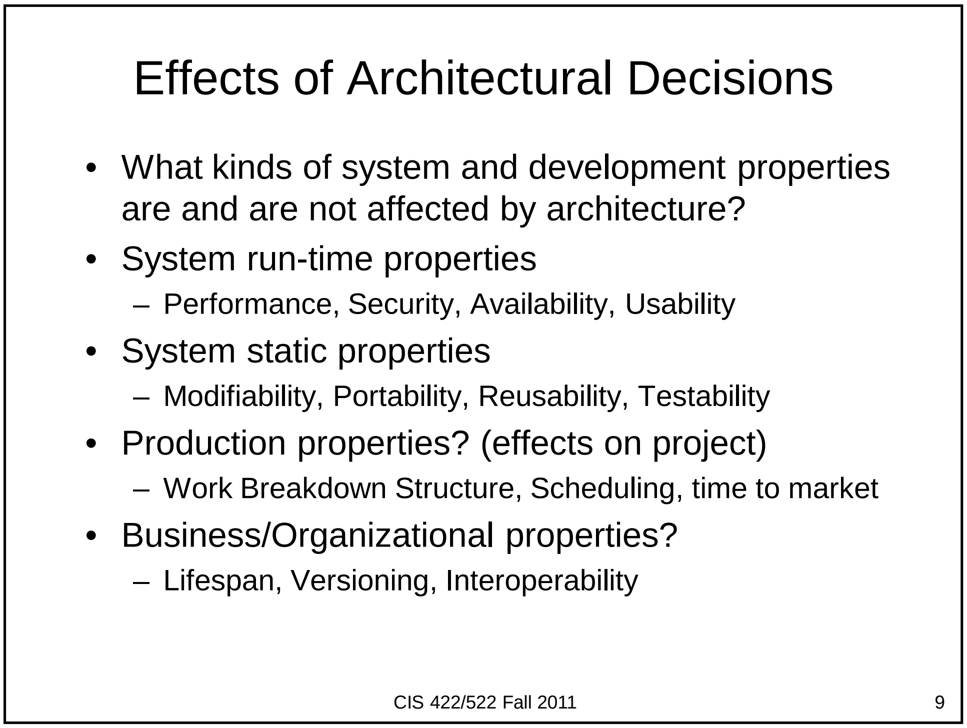
Structure	Components	Interfaces	Relationships
Calls Structure	Programs	Program interface and parameter declarations.	Invokes with parameters (A calls B)
Data Flow	Functional tasks	Data types or structures	Sends-data-to
Process	Sequential program (process, thread, task)	Scheduling and synchronization constraints	Runs-concurrently-with, excludes, precedes

This is not



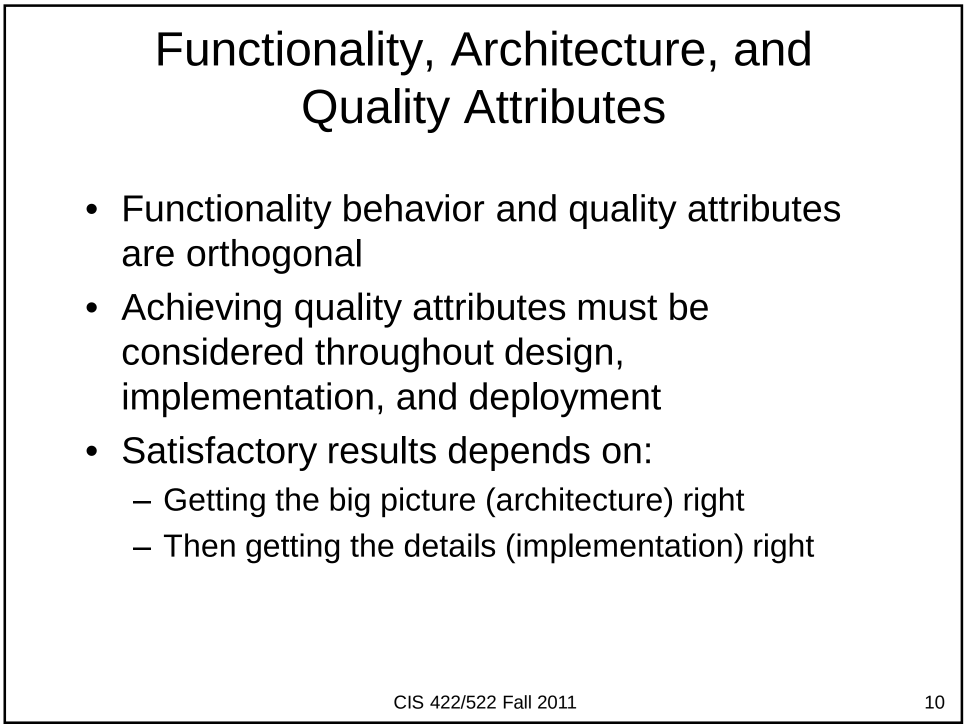
Typical (but uninformative) architectural diagram

- What is the nature of the components?
- What is the significance of the link?
- What is the significance of the layout?



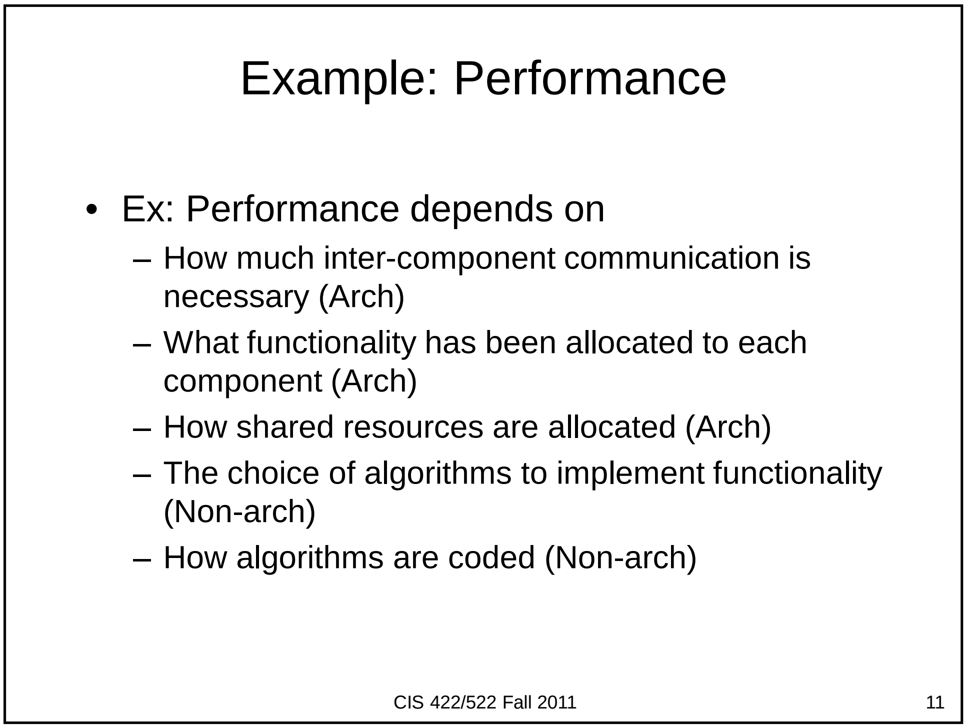
Effects of Architectural Decisions

- What kinds of system and development properties are and are not affected by architecture?
- System run-time properties
  - Performance, Security, Availability, Usability
- System static properties
  - Modifiability, Portability, Reusability, Testability
- Production properties? (effects on project)
  - Work Breakdown Structure, Scheduling, time to market
- Business/Organizational properties?
  - Lifespan, Versioning, Interoperability



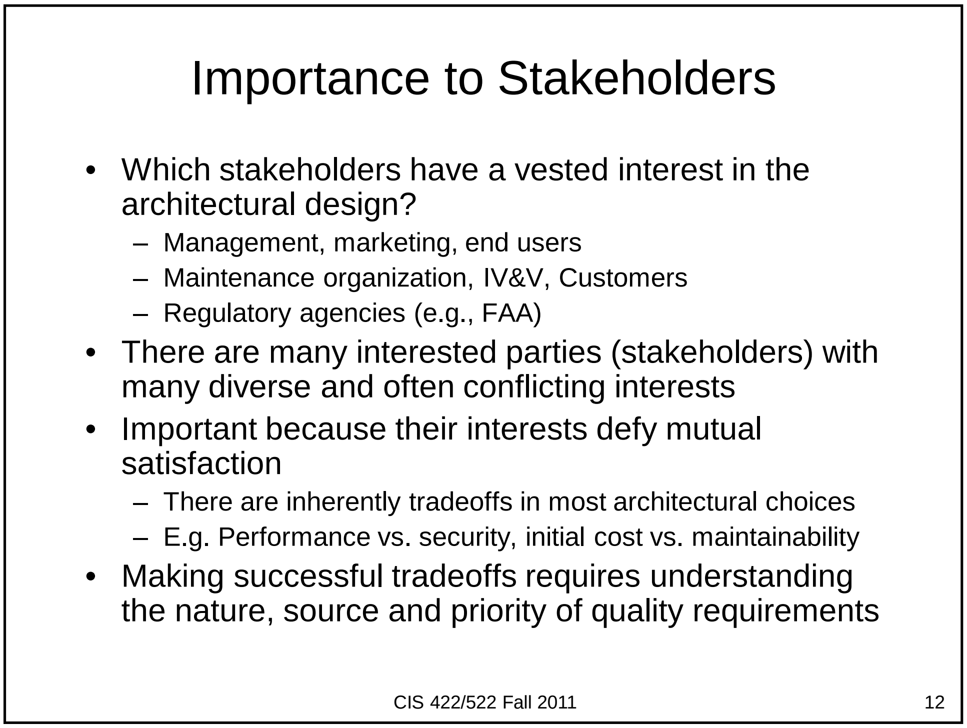
Functionality, Architecture, and Quality Attributes

- Functionality behavior and quality attributes are orthogonal
- Achieving quality attributes must be considered throughout design, implementation, and deployment
- Satisfactory results depends on:
  - Getting the big picture (architecture) right
  - Then getting the details (implementation) right



Example: Performance

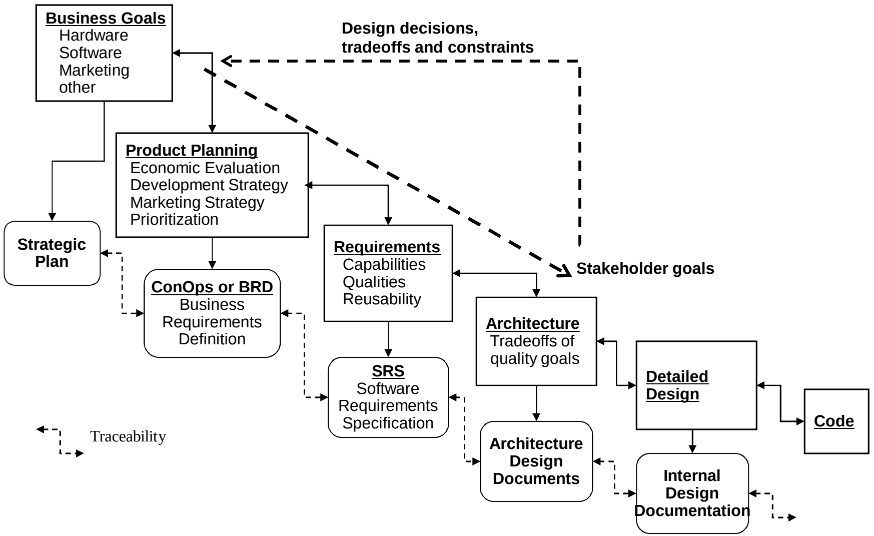
- Ex: Performance depends on
  - How much inter-component communication is necessary (Arch)
  - What functionality has been allocated to each component (Arch)
  - How shared resources are allocated (Arch)
  - The choice of algorithms to implement functionality (Non-arch)
  - How algorithms are coded (Non-arch)



Importance to Stakeholders

- Which stakeholders have a vested interest in the architectural design?
  - Management, marketing, end users
  - Maintenance organization, IV&V, Customers
  - Regulatory agencies (e.g., FAA)
- There are many interested parties (stakeholders) with many diverse and often conflicting interests
- Important because their interests defy mutual satisfaction
  - There are inherently tradeoffs in most architectural choices
  - E.g. Performance vs. security, initial cost vs. maintainability
- Making successful tradeoffs requires understanding the nature, source and priority of quality requirements

The Architectural Business Cycle



Business Goals
Hardware
Software
Marketing
other

Design decisions,
tradeoffs and constraints

Product Planning
Economic Evaluation
Development Strategy
Marketing Strategy
Prioritization

Strategic Plan

Requirements
Capabilities
Qualities
Reusability

Stakeholder goals

ConOps or BRD
Business Requirements Definition

Architecture
Tradeoffs of quality goals

Detailed Design

SRS
Software Requirements Specification

Code

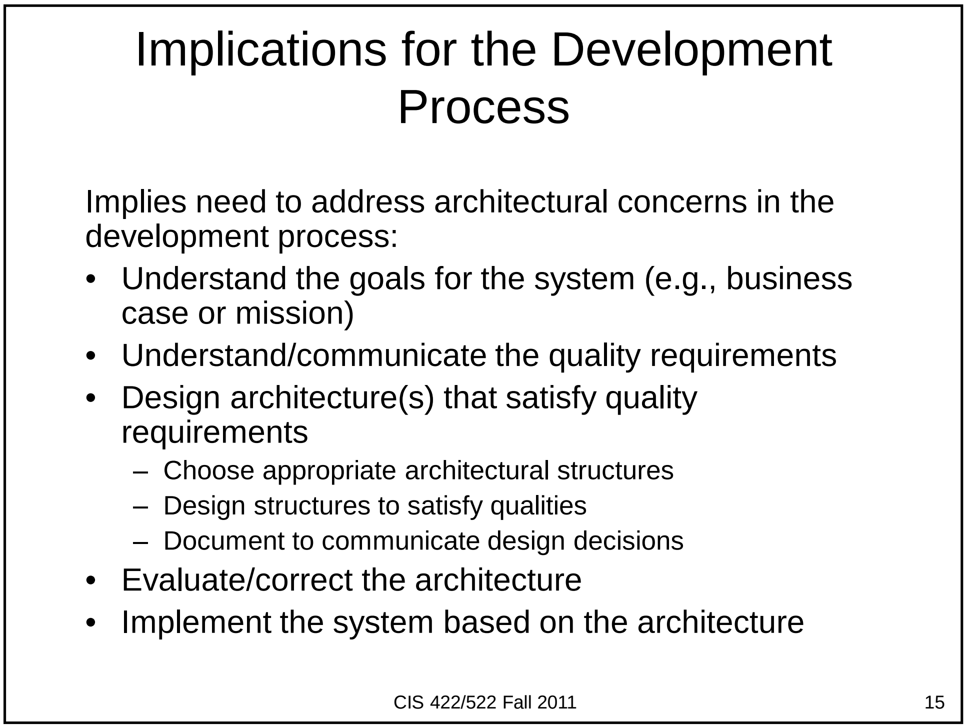
Traceability

Architecture Design Documents

Internal Design Documentation

Engineering Software Architecture

- Goal is to keep developmental goals and architectural capabilities in synch
- Proceed from an understanding of desired qualities to an *acceptable* system design
- For full control, must also consider downstream effects of architectural decisions
  – Maintenance
  – New versions of the system

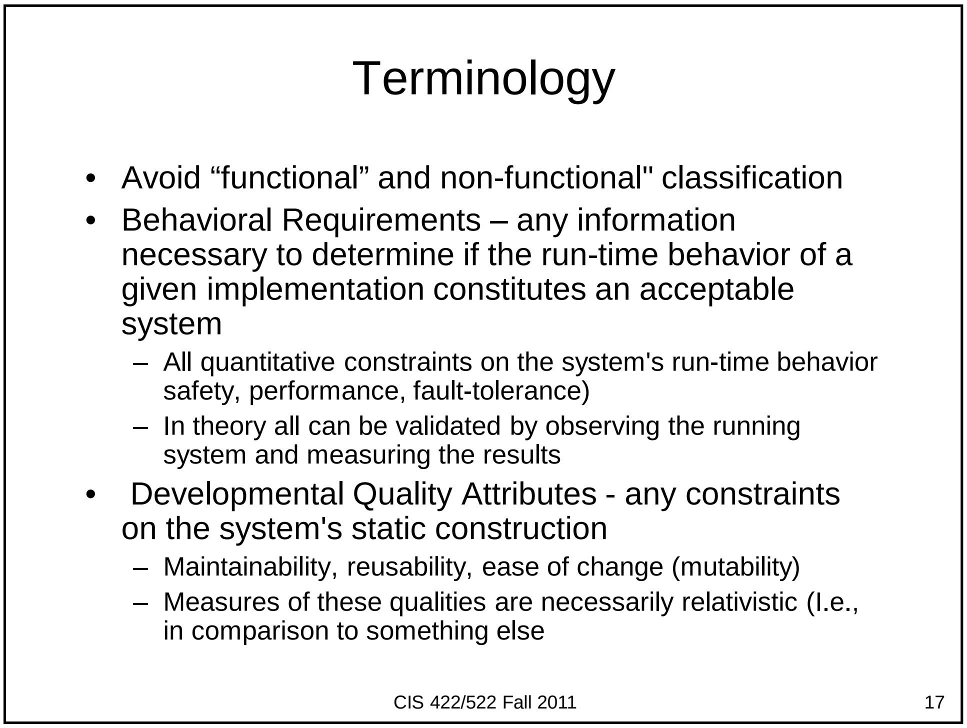


Implications for the Development Process

Implies need to address architectural concerns in the development process:

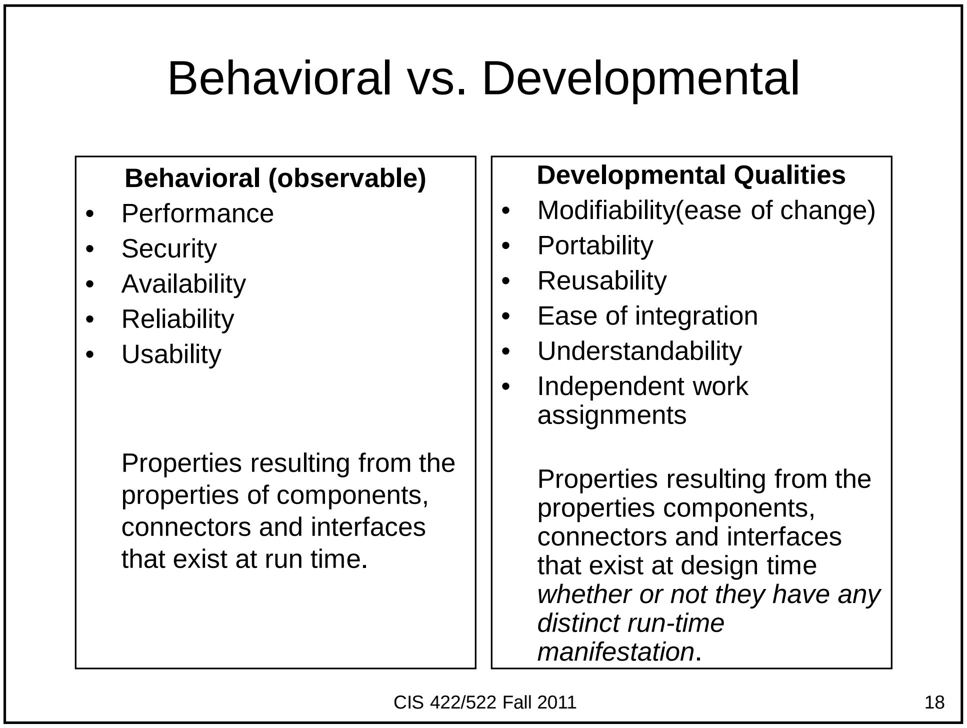
- Understand the goals for the system (e.g., business case or mission)
- Understand/communicate the quality requirements
- Design architecture(s) that satisfy quality requirements
  - Choose appropriate architectural structures
  - Design structures to satisfy qualities
  - Document to communicate design decisions
- Evaluate/correct the architecture
- Implement the system based on the architecture

Quality Requirements



Terminology

- Avoid “functional” and non-functional" classification
- Behavioral Requirements – any information necessary to determine if the run-time behavior of a given implementation constitutes an acceptable system
  - All quantitative constraints on the system's run-time behavior safety, performance, fault-tolerance)
  - In theory all can be validated by observing the running system and measuring the results
- Developmental Quality Attributes - any constraints on the system's static construction
  - Maintainability, reusability, ease of change (mutability)
  - Measures of these qualities are necessarily relativistic (I.e., in comparison to something else



Behavioral vs. Developmental

Behavioral (observable)

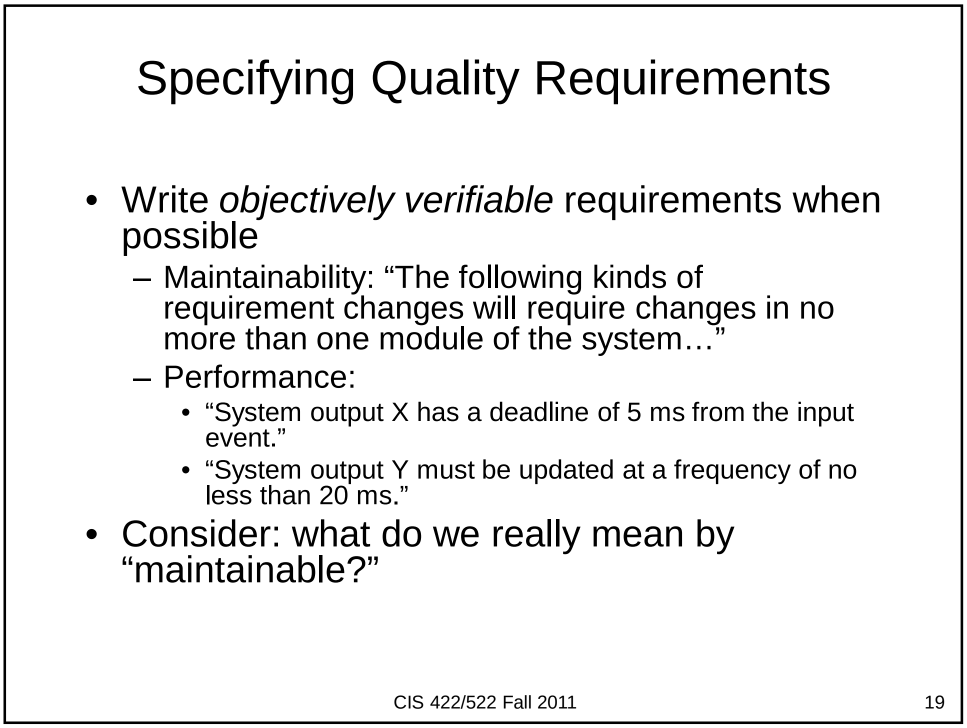
- Performance
- Security
- Availability
- Reliability
- Usability

Properties resulting from the properties of components, connectors and interfaces that exist at run time.

Developmental Qualities

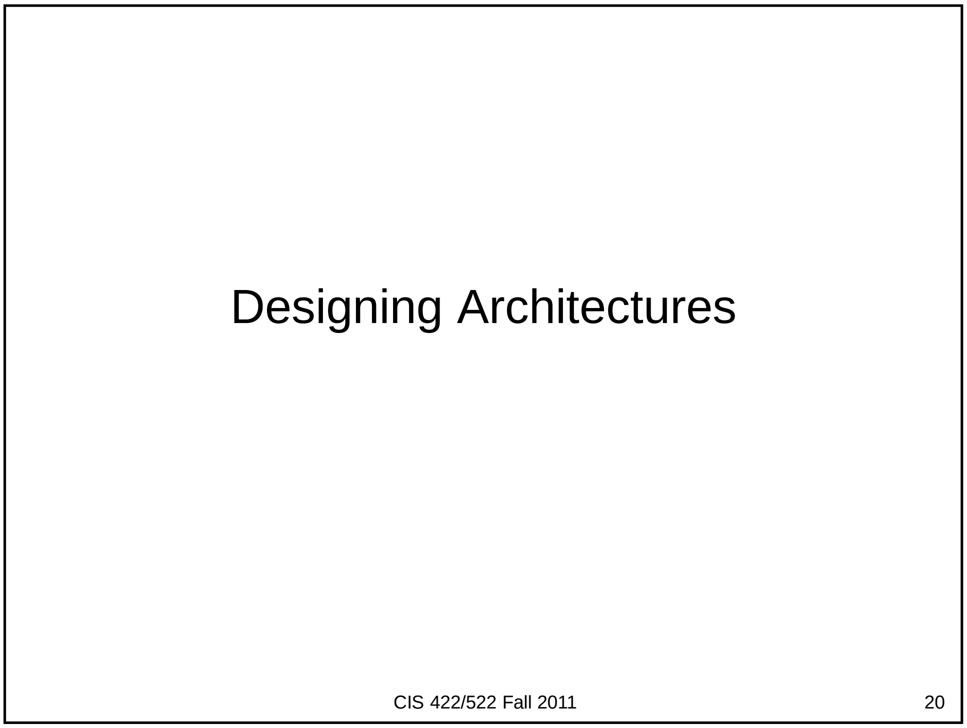
- Modifiability(ease of change)
- Portability
- Reusability
- Ease of integration
- Understandability
- Independent work assignments

Properties resulting from the properties components, connectors and interfaces that exist at design time *whether or not they have any distinct run-time manifestation*.



Specifying Quality Requirements

- Write *objectively verifiable* requirements when possible
  - Maintainability: “The following kinds of requirement changes will require changes in no more than one module of the system…”
  - Performance:
    - “System output X has a deadline of 5 ms from the input event.”
    - “System output Y must be updated at a frequency of no less than 20 ms.”
- Consider: what do we really mean by “maintainable?”



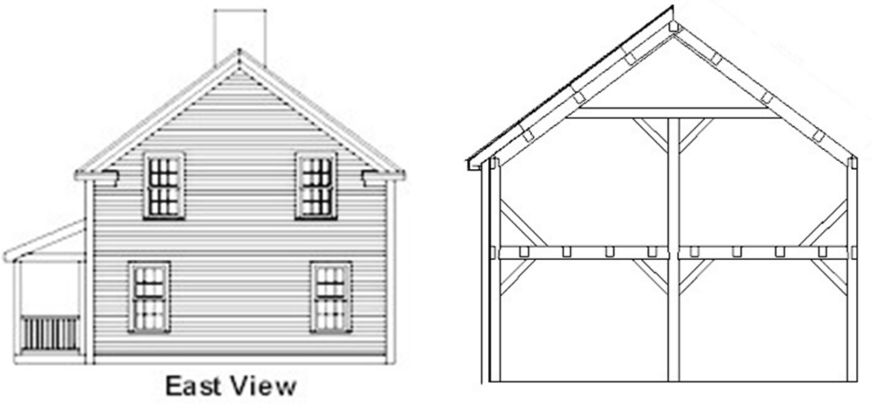
Designing Architectures

Which structures should we use?

Structure	Components	Interfaces	Relationships
Calls Structure	Programs (methods, services)	Program interface and parameter declarations	Invokes with parameters (A calls B)
Data Flow	Functional tasks	Data types or structures	Sends-data-to
Process	Sequential program (process, thread, task)	Scheduling and synchronization constraints	Runs-concurrently-with, excludes, precedes

- Choice of structure depends the *specific design goals*
- Compare to architectural blueprints
  - Different blueprint for load-bearing structures, electrical, mechanical, plumbing

Elevation/Structural



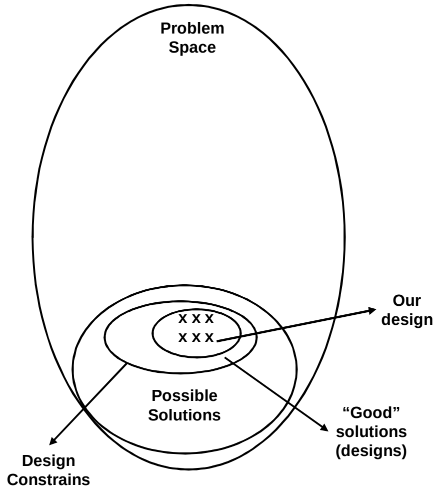
Models/Views

- Each is a view of the same house
- Different views answer different kinds of questions
  - How many electrical outlets are available in the kitchen?
  - What happens if we put a window here?
- Designing for particular software qualities also requires the right architectural model or “view”
  - Any model can present only a subset of system structures and properties
  - Different models allows us to answer different kinds of questions about system properties
  - Need a model that makes the properties of interest and the consequences of design choices visible to the designer, e.g.
    - Process structure for run-time property like performance
    - Module structure for development property like maintainability

Design Means…

- Design Goals: the purpose of design is to solve some problem in a context of assumptions and constraints
  - Assumptions: what must be true of the design
  - Constraints: what should not be true
  - **These define the *design goals***
- Process: design proceeds through a sequence of decisions
  - A *good* decision brings us closer to the design goals
  - An idealized design process systematically makes good decisions
  - Any real design process is chaotic
- Good Design: *by definition* a good design is one that satisfies the design goal

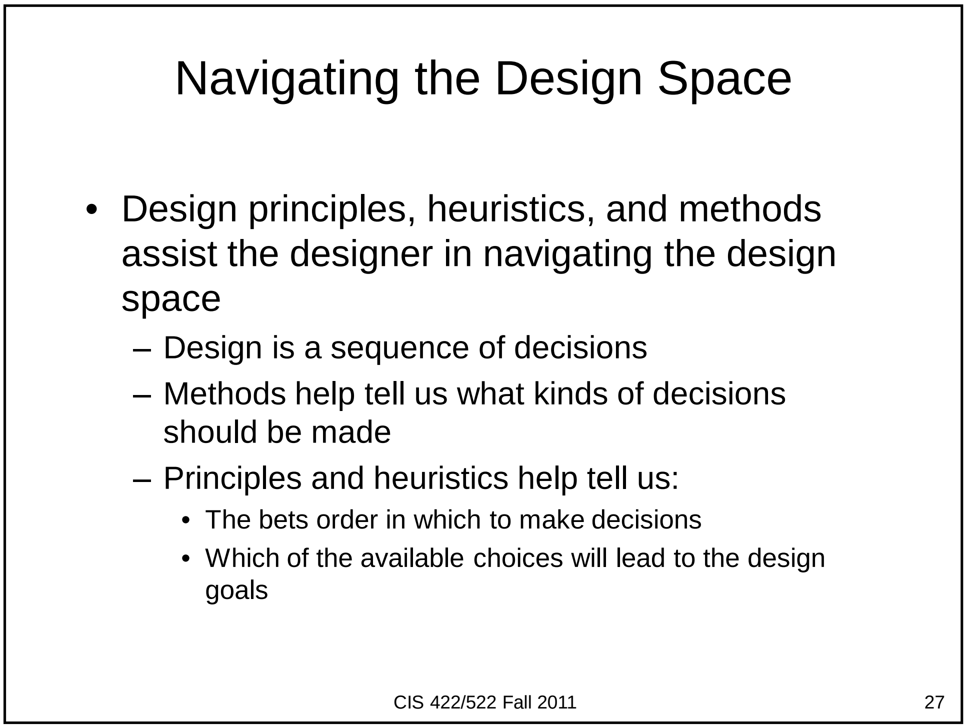
The Design Space



- A Design: is (a representation of) a solution to a problem
  - – Represents a set of choices
    - Typically very large set of possible choices
    - Must navigate through possibilities
    - Invariably requires tradeoffs
  - – Possible choices are limited by *assumptions and constraints*
    - e.g., must be ISO 2000 compliant, legacy compatible, etc.
  - – Some designs are better than others (notion of *good design*)

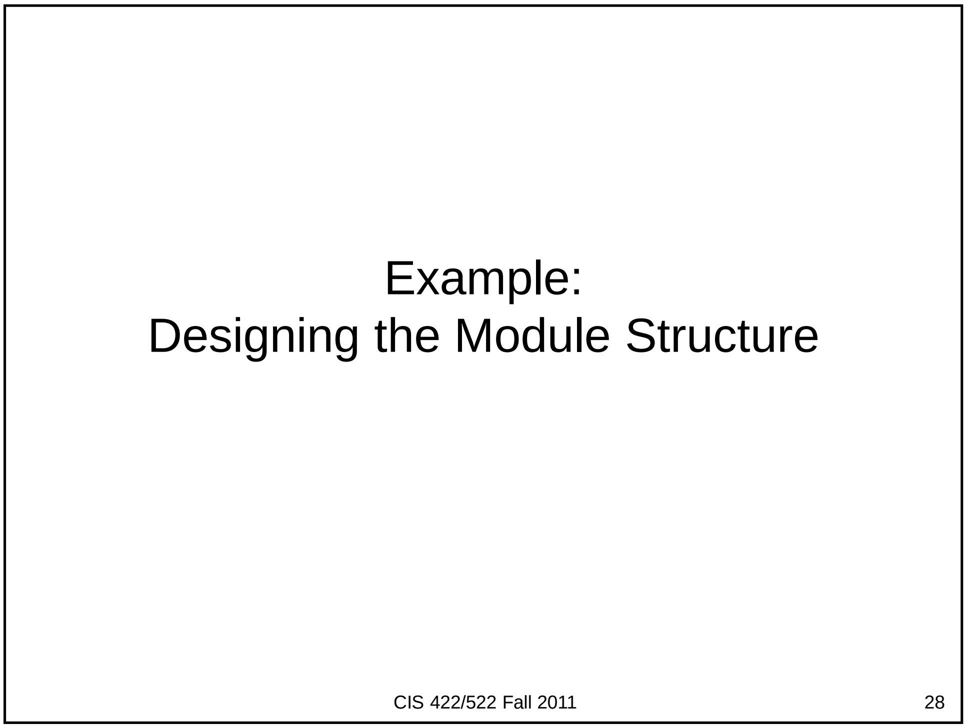
Architectural Design Elements

- Design goals
  - – What are we trying to accomplish in the decomposition?
- Relevant Structure
  - – How to we capture and communicate design decisions?
  - – What are the components, relations, interfaces?
- Decomposition principles
  - – How do we distinguish good design decisions?
  - – What decomposition (design) principles support the objectives?
- Evaluation criteria
  - – How do I tell a good design from a bad one?

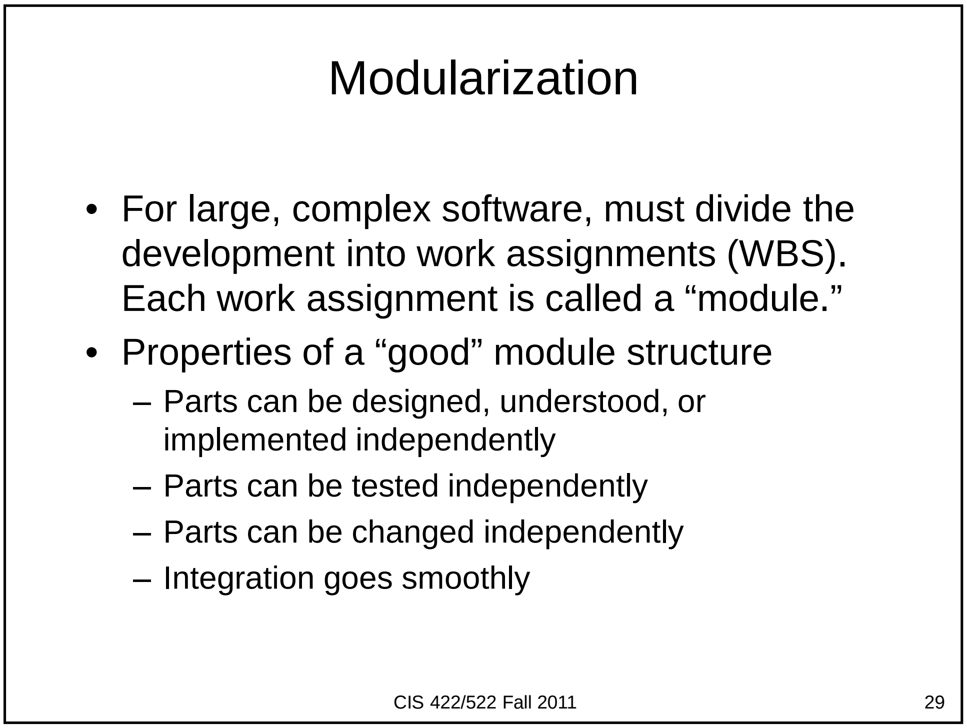


Navigating the Design Space

- Design principles, heuristics, and methods assist the designer in navigating the design space
  - – Design is a sequence of decisions
  - – Methods help tell us what kinds of decisions should be made
  - – Principles and heuristics help tell us:
    - The bets order in which to make decisions
    - Which of the available choices will lead to the design goals

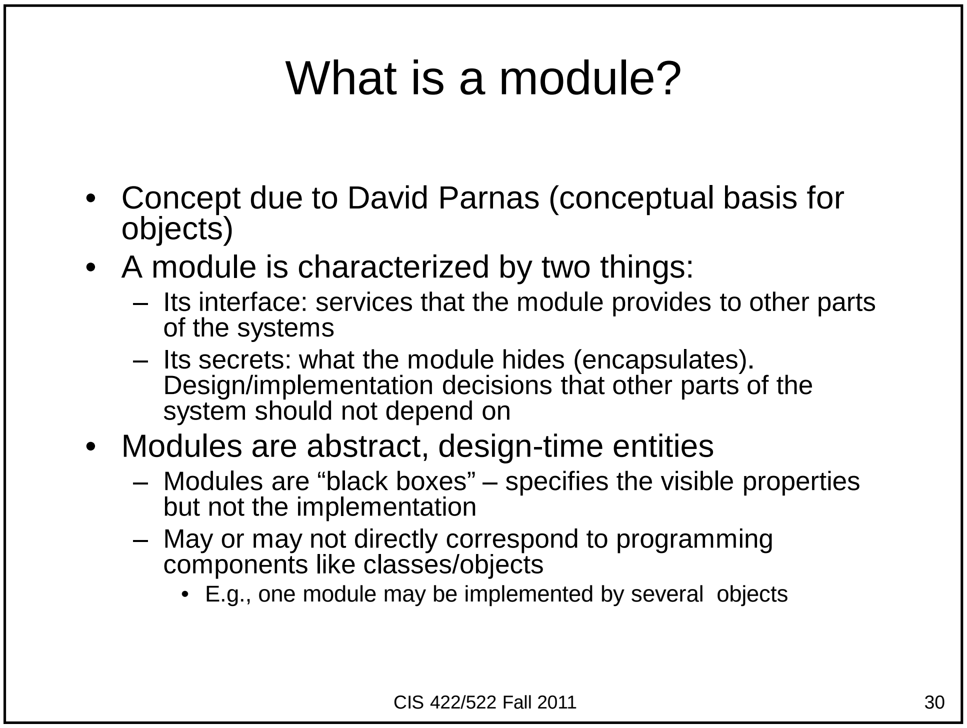


Example:
Designing the Module Structure



Modularization

- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a “module.”
- Properties of a “good” module structure
  - Parts can be designed, understood, or implemented independently
  - Parts can be tested independently
  - Parts can be changed independently
  - Integration goes smoothly

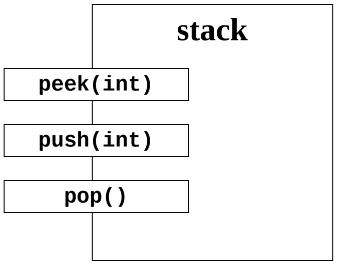


What is a module?

- Concept due to David Parnas (conceptual basis for objects)
- A module is characterized by two things:
  - Its interface: services that the module provides to other parts of the systems
  - Its secrets: what the module hides (encapsulates). Design/implementation decisions that other parts of the system should not depend on
- Modules are abstract, design-time entities
  - Modules are “black boxes” – specifies the visible properties but not the implementation
  - May or may not directly correspond to programming components like classes/objects
    - E.g., one module may be implemented by several objects

A Simple Module

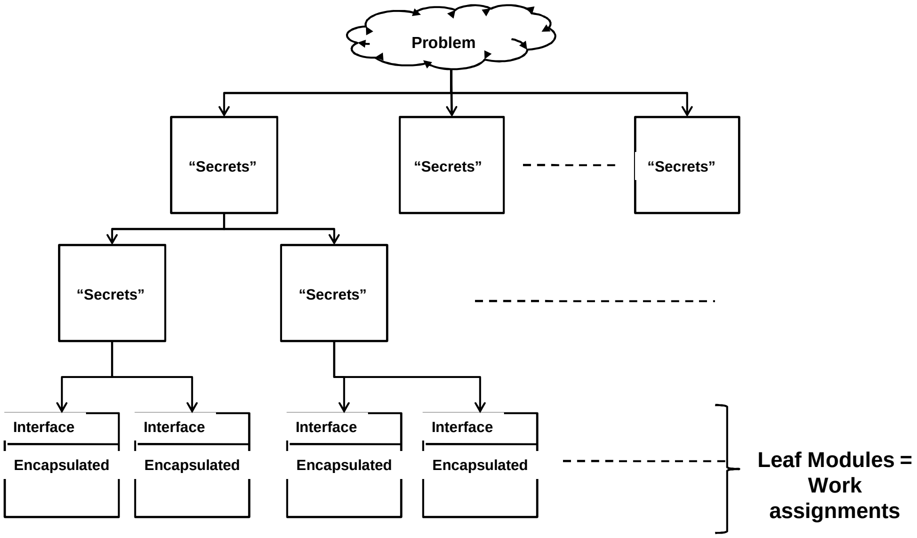
- A simple integer stack
- The *interface* specifies what a programmer needs to know to use the stack correctly, e.g.
  - *push*: push integer on stack top
  - *pop*: remove top element
  - *peek*: get value of top element
- The *secrets* (encapsulated) any details that might change from one implementation to another
  - Data structures, algorithms
  - Details of class/object structure
- A module spec is *abstract*: describes the services provided but allows many possible implementations
- Note: a real spec needs much more than this (discuss later)



Module Hierarchy

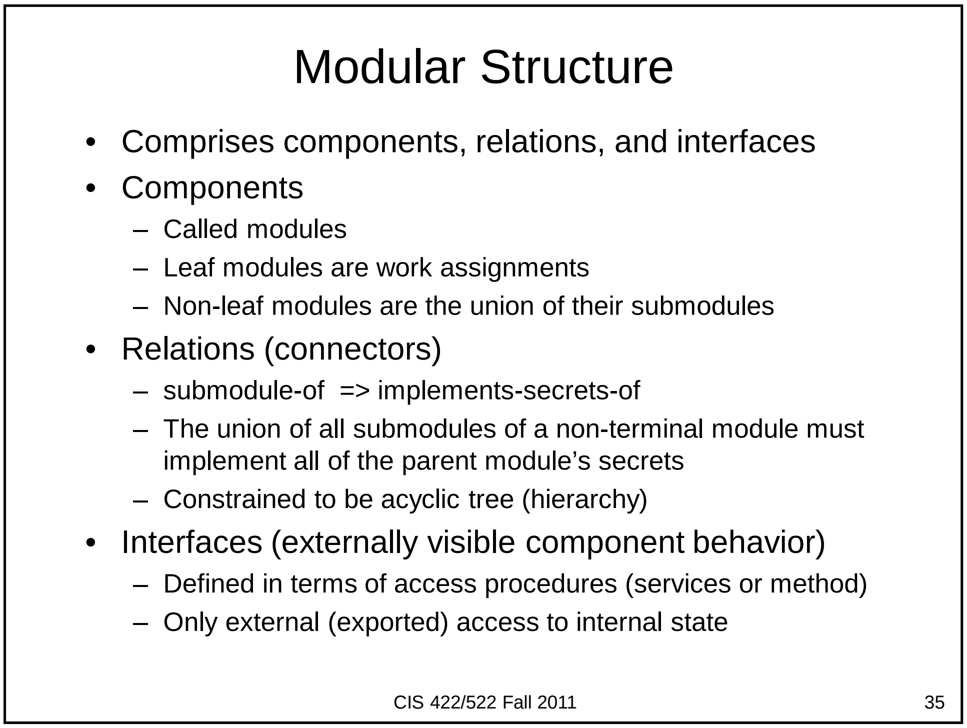
- For large systems, the set of modules need to be organized such that
  - We can check that all of the functional requirements have been allocated to some module of the system
  - Developers can easily find the module that provides any given capability
  - When a change is required, it is easy to determine which modules must be changed
- The module hierarchy defined by the *submodule-of* relation provides this architectural view

Module Hierarchy



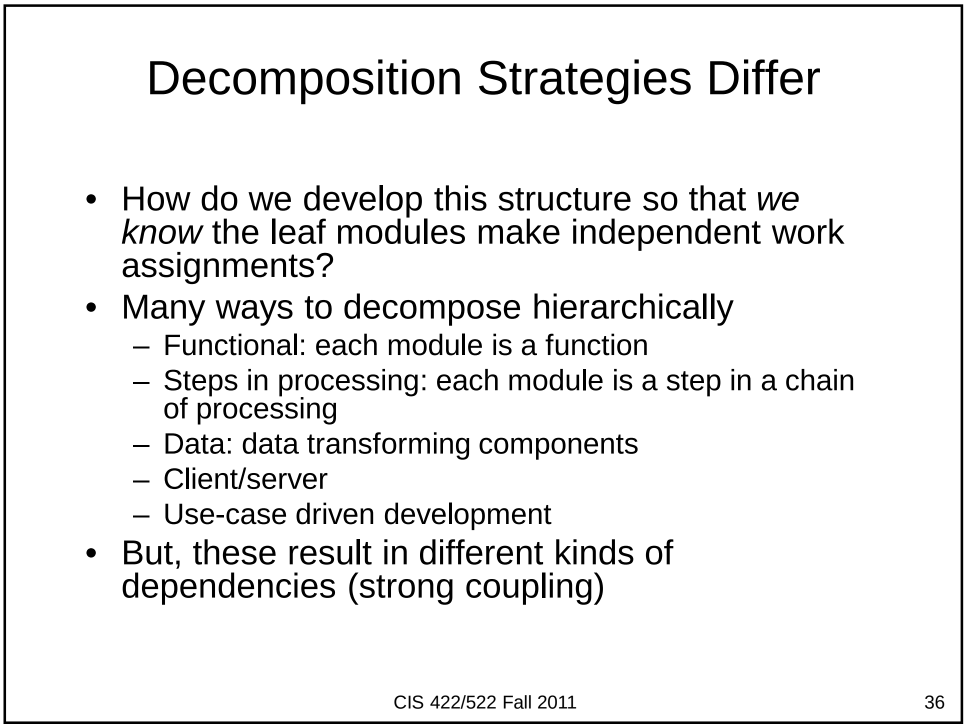
⟶ Submodule-of relation

Decomposition Approach



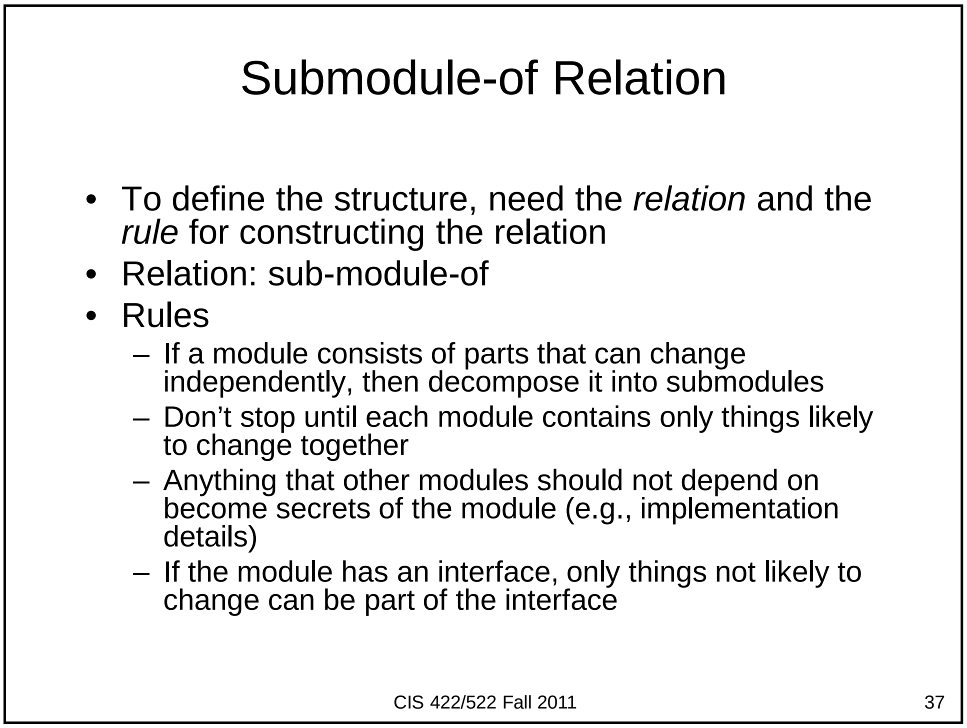
Modular Structure

- Comprises components, relations, and interfaces
- Components
  - Called modules
  - Leaf modules are work assignments
  - Non-leaf modules are the union of their submodules
- Relations (connectors)
  - submodule-of => implements-secrets-of
  - The union of all submodules of a non-terminal module must implement all of the parent module’s secrets
  - Constrained to be acyclic tree (hierarchy)
- Interfaces (externally visible component behavior)
  - Defined in terms of access procedures (services or method)
  - Only external (exported) access to internal state



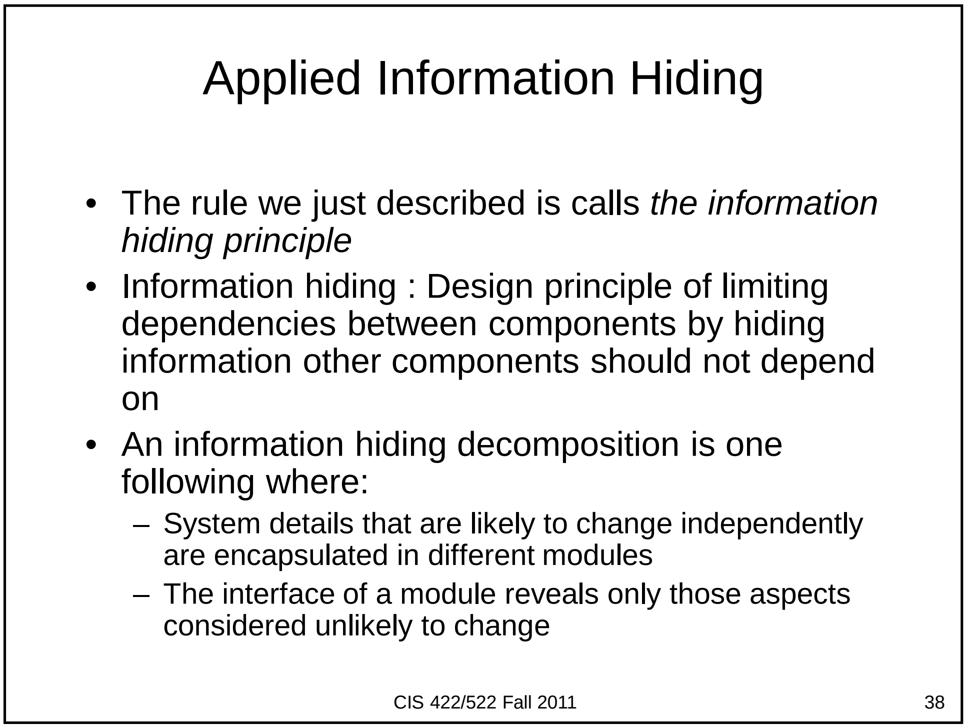
Decomposition Strategies Differ

- How do we develop this structure so that *we know* the leaf modules make independent work assignments?
- Many ways to decompose hierarchically
  - Functional: each module is a function
  - Steps in processing: each module is a step in a chain of processing
  - Data: data transforming components
  - Client/server
  - Use-case driven development
- But, these result in different kinds of dependencies (strong coupling)



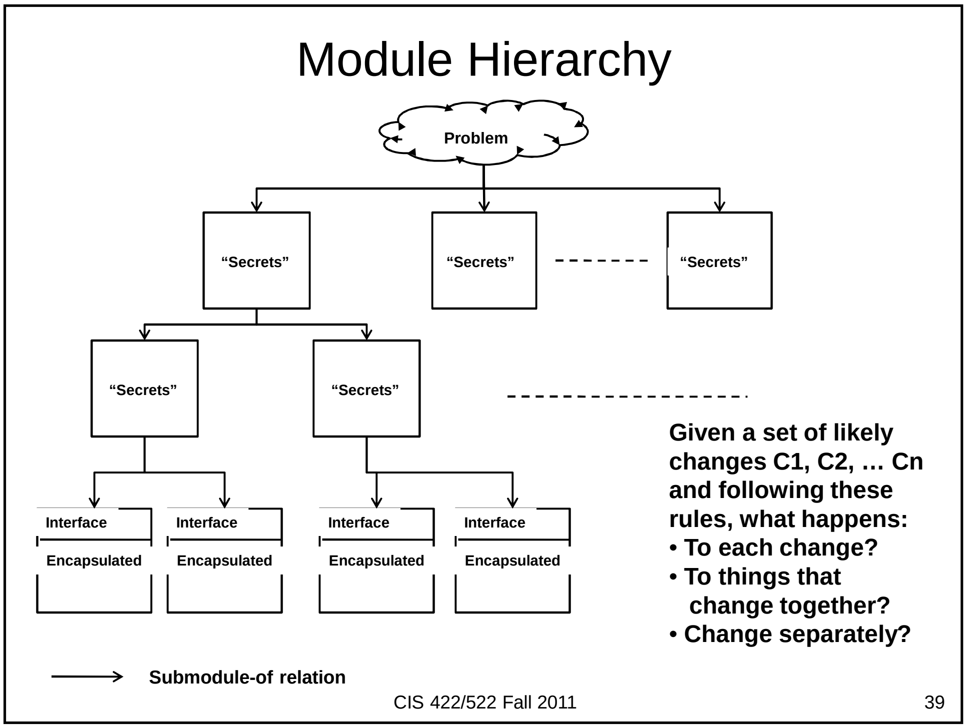
Submodule-of Relation

- To define the structure, need the *relation* and the *rule* for constructing the relation
- Relation: sub-module-of
- Rules
  - If a module consists of parts that can change independently, then decompose it into submodules
  - Don’t stop until each module contains only things likely to change together
  - Anything that other modules should not depend on become secrets of the module (e.g., implementation details)
  - If the module has an interface, only things not likely to change can be part of the interface



Applied Information Hiding

- The rule we just described is calls *the information hiding principle*
- Information hiding : Design principle of limiting dependencies between components by hiding information other components should not depend on
- An information hiding decomposition is one following where:
  - System details that are likely to change independently are encapsulated in different modules
  - The interface of a module reveals only those aspects considered unlikely to change

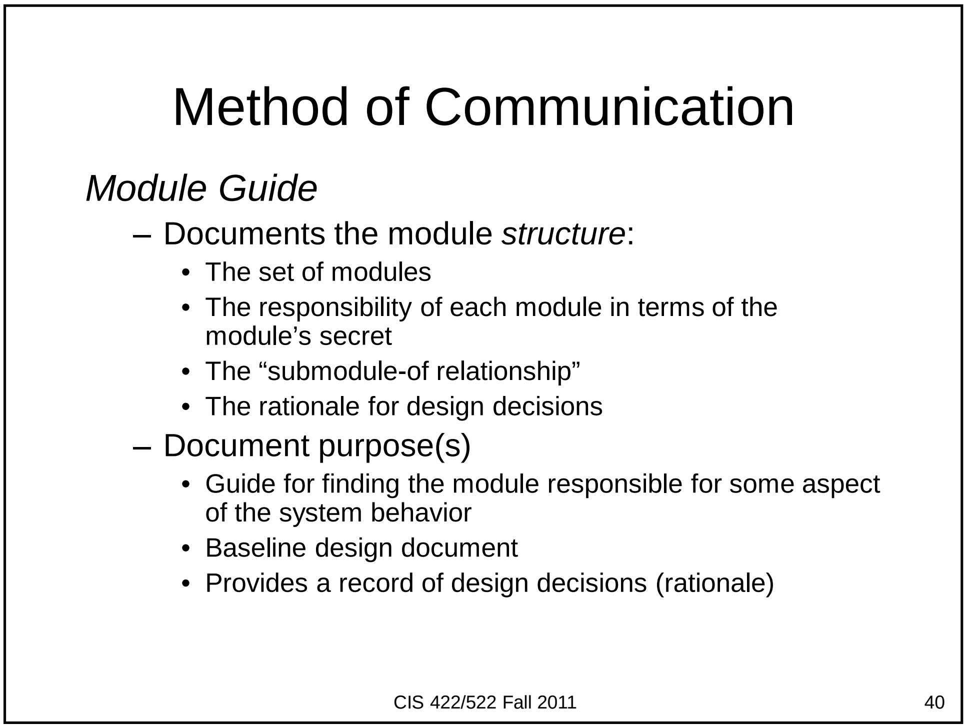


Module Hierarchy

Given a set of likely changes C1, C2, … Cn and following these rules, what happens:

- **To each change?**
- **To things that change together?**
- **Change separately?**

Submodule-of relation



Method of Communication

Module Guide

- Documents the module *structure*:
  - The set of modules
  - The responsibility of each module in terms of the module's secret
  - The "submodule-of relationship"
  - The rationale for design decisions
- Document purpose(s)
  - Guide for finding the module responsible for some aspect of the system behavior
  - Baseline design document
  - Provides a record of design decisions (rationale)

Evaluation Criteria

- Evaluation criteria follow from goals of the model: should be able to answer “yes” to the following review questions?
- Completeness
  - Is every aspect of the system the responsibility of one module?
  - Do the submodules of each module partition its secrets?
- Ease change
  - Is each likely change hidden by some module?
  - Are only aspects of the system that are very unlikely to change embedded in the module structure?
  - For each leaf module, are the module’s secrets revealed by it’s access programs?
- Usability
  - For any given change, can the appropriate module be found using the module guide

Information Hiding Decomposition

- Approach: divide the system into submodules according to the kinds of design decisions they encapsulate (secrets)
  - Design decisions that are closely related (likely to change together , high cohesion) are grouped in the same submodule
  - Design decisions that are weakly related (likely to change independently) are allocated to different modules
  - Characterize each module by its secrets (what it hides)
- Viewed top down, each module is decomposed into submodules such that
  - Each design decision allocated to the parent module is allocated to exactly one child module
  - Together the children implement all of the decisions of the parent
- Stop decomposing when each module is
  - Simple enough to be understood fully
  - Small enough that it makes sense to throw it away rather than re-do
- This is called an *information-hiding decomposition*

Specifying Abstract Interfaces

Method of Communication

Module Interface Specifications

- – Documents all assumptions user’s can make about the module’s externally visible behavior (of leaf modules)
  - Access programs, events, types, undesired events
  - Design issues, assumptions
- – Document purpose(s)
  - Provide all the information needed to write a module’s programs or use the programs on a module’s interface (programmer’s guide, user’s guide)
  - Specify required behavior by fully specifying behavior of the module’s access programs
  - Define any constraints
  - Define any assumptions
  - Record design decisions

Need for Precise Interface Specifications

- But, informal description is not enough to write the software
- To support independent development, need a precise interface specification

Why these properties?

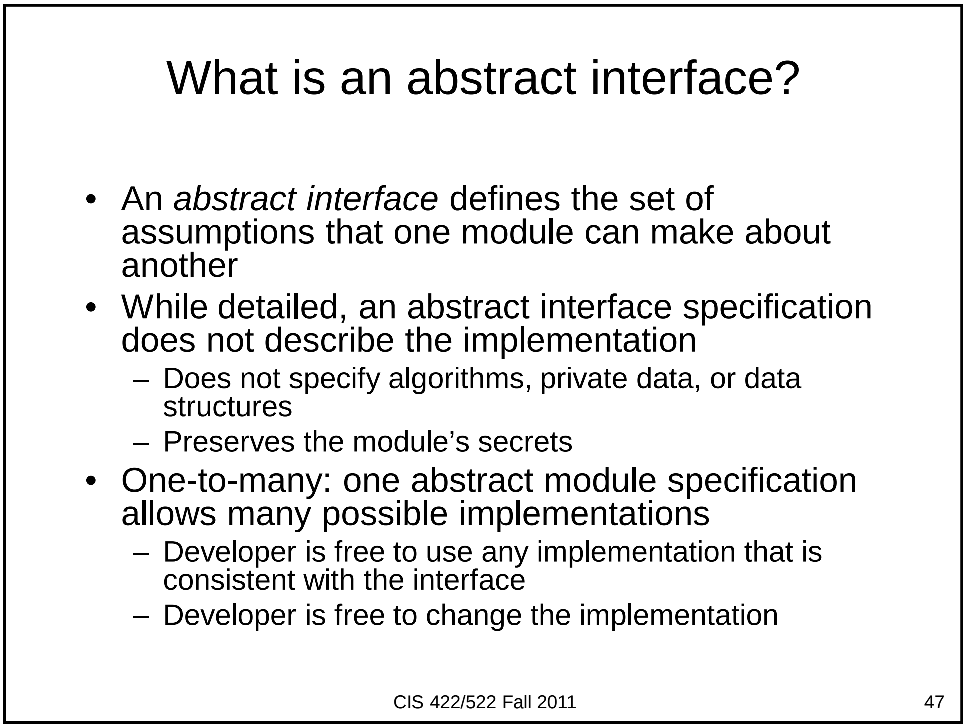
Module Implementer

- The specification tells me exactly what capabilities my module must provide to users
- I am free to implement it any way I want to
- I am free to change the implementation if needed as long as I don't change the interface

Module User

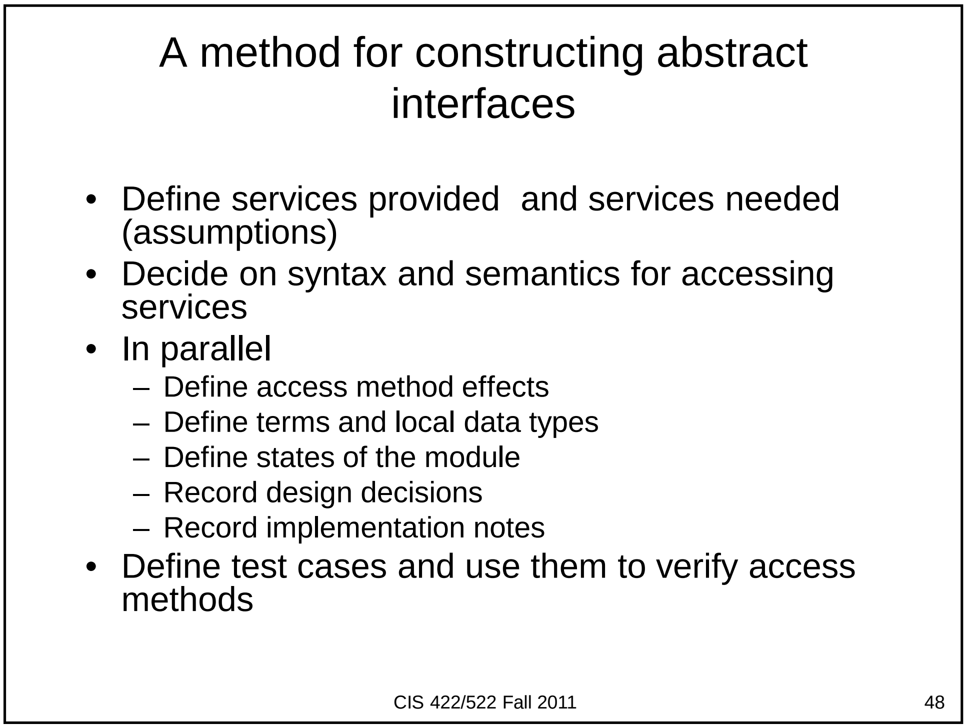
- The specification tells me how to use the module's services correctly
- I do not need to know anything about the implementation details to write my code
- If the implementation changes, my code stays the same

***Key idea*: the abstract interface specification defines a contract between a module's developer and its users that allows each to proceed independently**



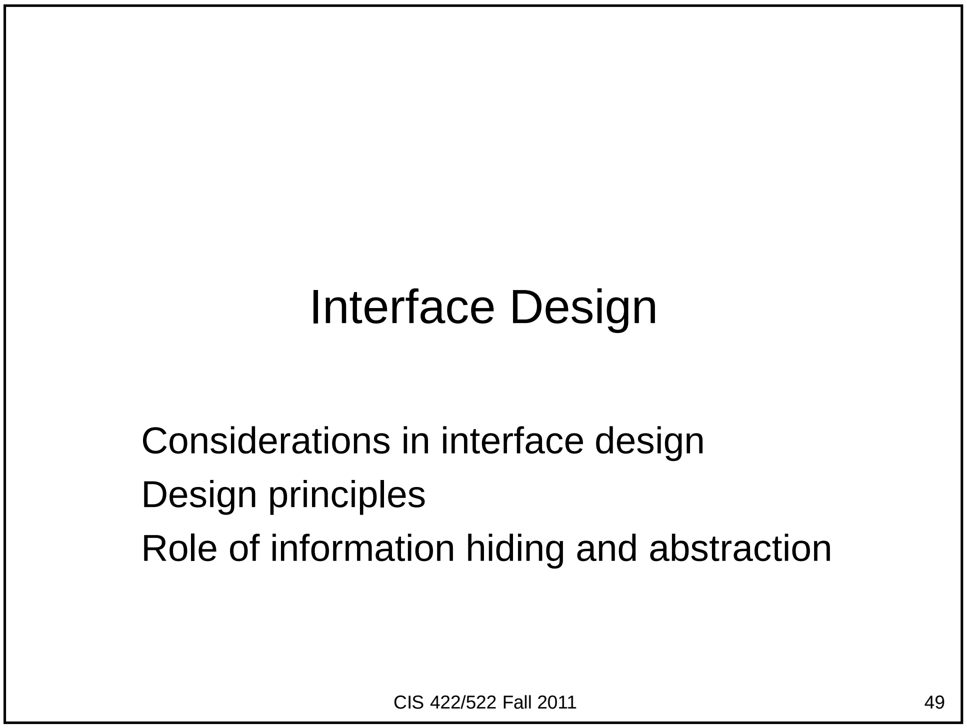
What is an abstract interface?

- An *abstract interface* defines the set of assumptions that one module can make about another
- While detailed, an abstract interface specification does not describe the implementation
  - Does not specify algorithms, private data, or data structures
  - Preserves the module’s secrets
- One-to-many: one abstract module specification allows many possible implementations
  - Developer is free to use any implementation that is consistent with the interface
  - Developer is free to change the implementation



A method for constructing abstract interfaces

- Define services provided and services needed (assumptions)
- Decide on syntax and semantics for accessing services
- In parallel
  - Define access method effects
  - Define terms and local data types
  - Define states of the module
  - Record design decisions
  - Record implementation notes
- Define test cases and use them to verify access methods

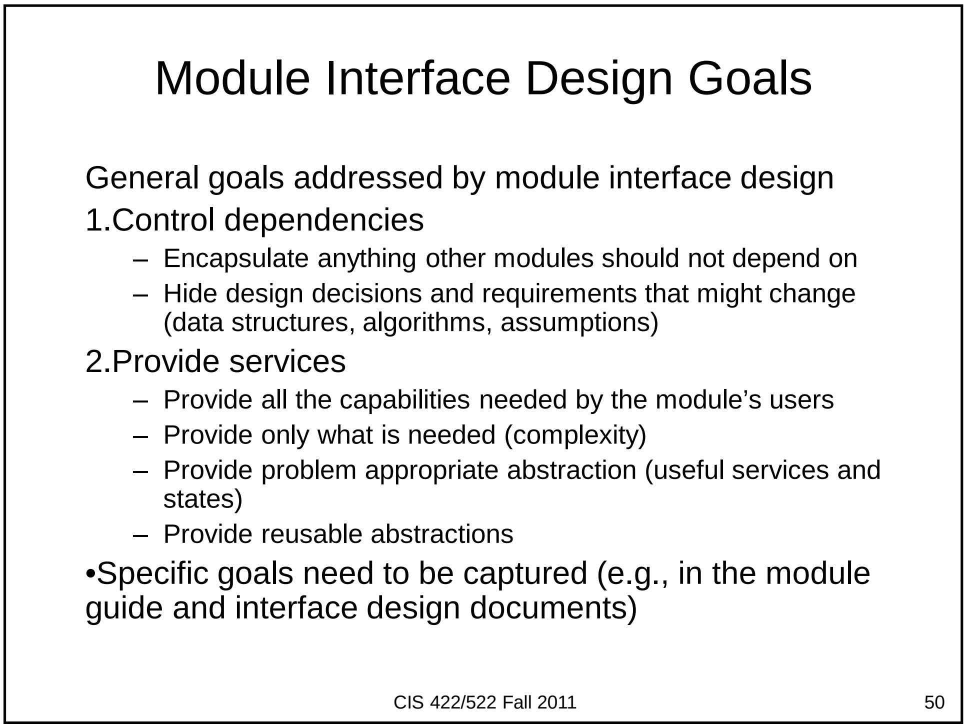


Interface Design

Considerations in interface design

Design principles

Role of information hiding and abstraction



Module Interface Design Goals

General goals addressed by module interface design

1.Control dependencies
  - Encapsulate anything other modules should not depend on
  - Hide design decisions and requirements that might change (data structures, algorithms, assumptions)

2.Provide services
  - Provide all the capabilities needed by the module’s users
  - Provide only what is needed (complexity)
  - Provide problem appropriate abstraction (useful services and states)
  - Provide reusable abstractions

•Specific goals need to be captured (e.g., in the module guide and interface design documents)

1. Controlling Dependencies

- Addressed using the <u>principle of information hiding</u>
- IH: design principle of limiting dependencies between components by hiding information other components should not depend on
- When thinking about what to put on the interface
  - Design the module interface to reveal only those design decisions considered unlikely to change
  - Required functionality allocated to the module and considered likely to change must be encapsulated
  - Each data structure is used in only one module
  - Any other program must access internal data by calling access programs on the interface
- Consistent with good OOD principles

2. Provide Services

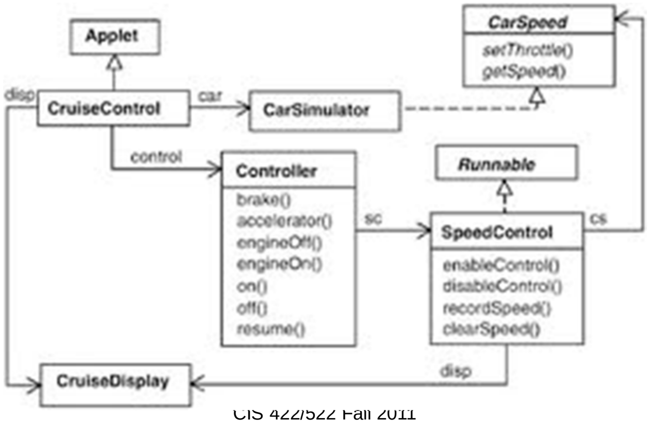
- Interface provides the capabilities of the module to other modules in the system, addressed by:
- *Abstraction*: interface design principle of providing only essential information and suppressing unnecessary detail

Abstraction

- Two primary uses
- Reduce Complexity
  - Goal: manage complexity by reducing the amount of information that must be considered at one time
  - Approach: Separate information important to the problem at hand from that which is not
  - Abstraction suppresses or hides “irrelevant detail”
  - Examples: stacks, queues, abstract device
- Model the problem domain
  - Goal: leverage domain knowledge to simplify understanding, creating, checking designs
  - Approach: Provide components that make it easier to model a class of problems
    - May be quite general (e.g., type real, type float)
    - May be very problem specific (e.g., class automobile, book object)

Example: Car Object

- What are the abstractions?
  - Purpose of each?
- What information is hidden?



Which Principle to Use

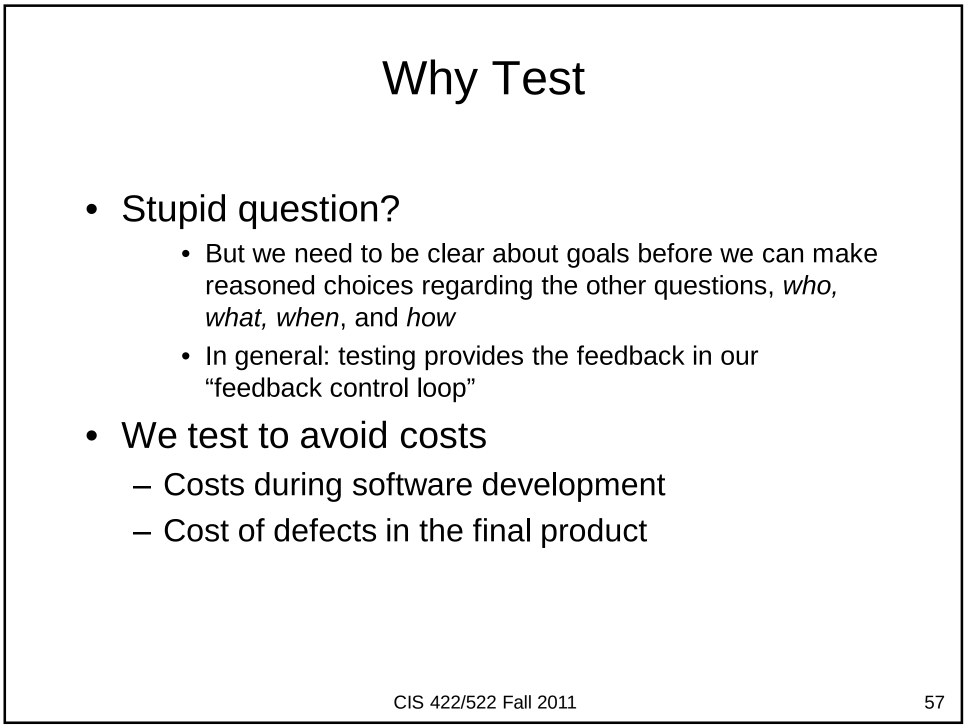
- Use abstraction when the issue is what should be on the interface (form and content)
- Use information hiding when the issue is what information should not be on the interface (visible or accessible)

Quality Assurance

The role of testing*

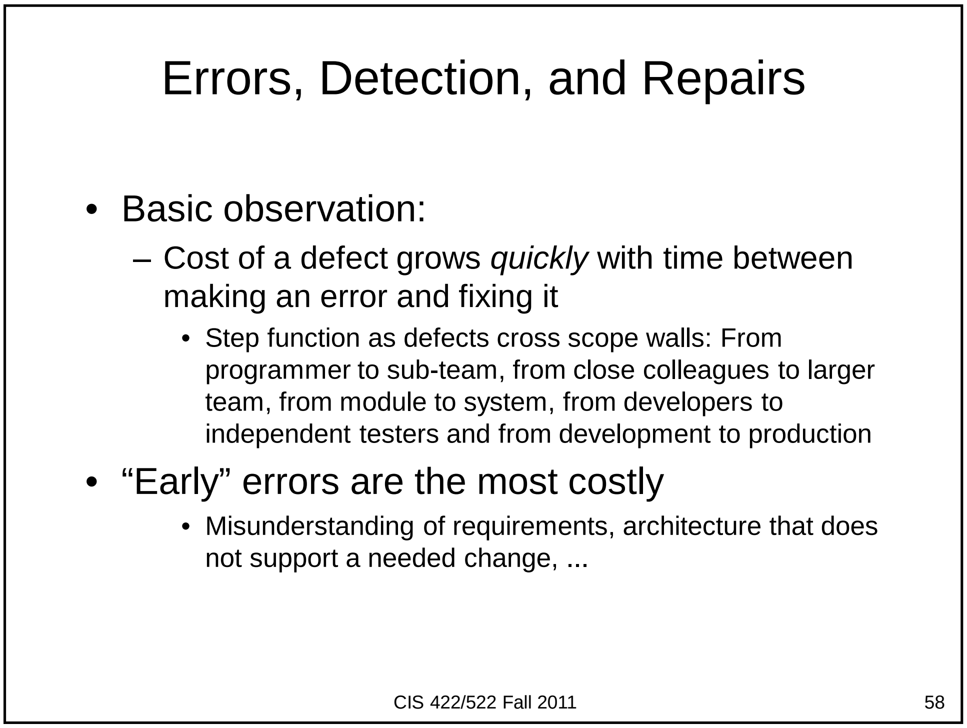
Active reviews

***From Prof. Michal Young**



Why Test

- Stupid question?
    - But we need to be clear about goals before we can make reasoned choices regarding the other questions, *who, what, when*, and *how*
    - In general: testing provides the feedback in our “feedback control loop”
- We test to avoid costs
    - Costs during software development
    - Cost of defects in the final product



Errors, Detection, and Repairs

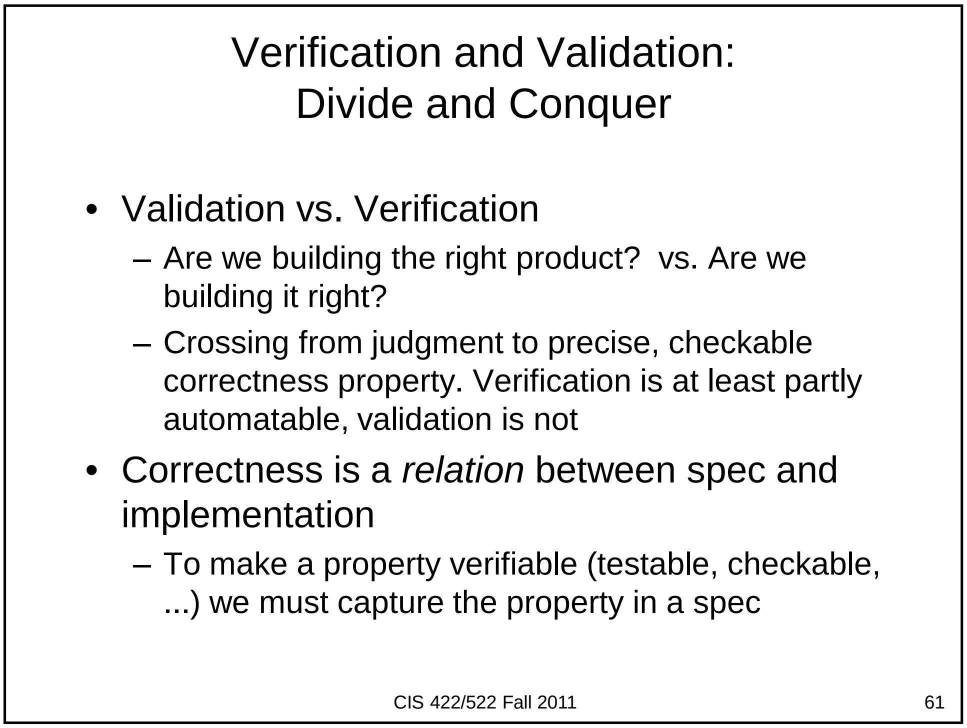
- Basic observation:
    - Cost of a defect grows *quickly* with time between making an error and fixing it
        - Step function as defects cross scope walls: From programmer to sub-team, from close colleagues to larger team, from module to system, from developers to independent testers and from development to production
- “Early” errors are the most costly
    - Misunderstanding of requirements, architecture that does not support a needed change, ...

When

- As early as possible
  - Reduce the gap between making an error and fixing it
    - Ideally to “immediately” ... which we call “prevention” or “syntactic checking”
    - E.g., error detection/correction in Eclipse, other programming environments
- Throughout development
  - People make mistakes in every activity, so every work product must be tested as soon as possible

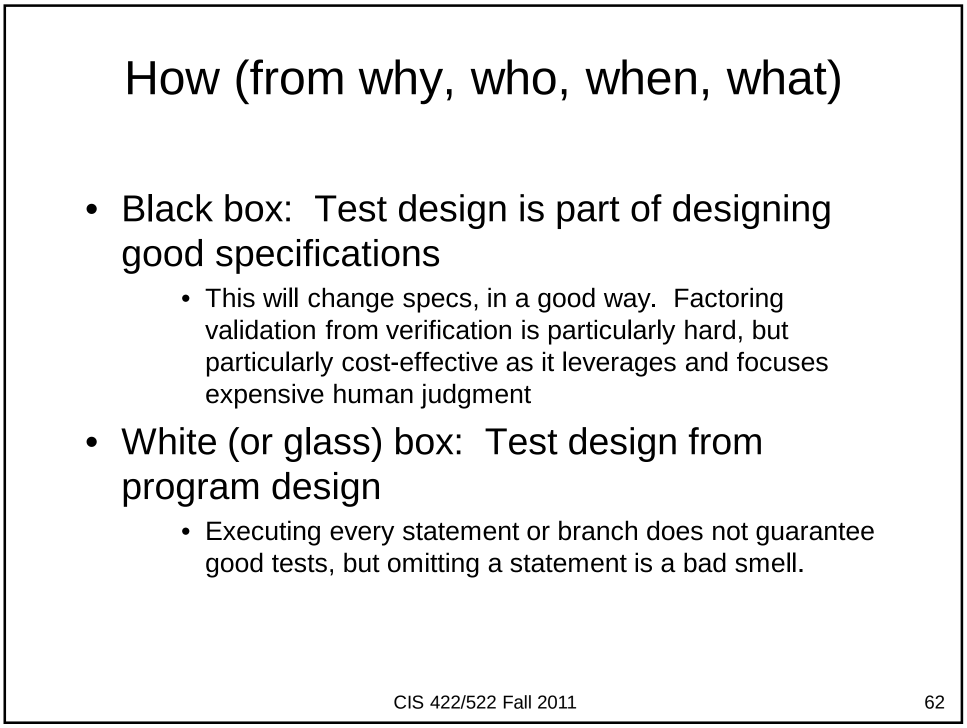
Choosing What

- For every work product, we ask: How can I find defects as early as possible
  - Ex: How can I find defects in software architecture before we’ve designed all the modules? How can I find defects in my module code before it’s integrated into the system?
- Divide and conquer
  - What properties can be checked automatically?
  - What properties can be (effectively) tested dynamically?
  - How can I make reviews cost-effective?



Verification and Validation: Divide and Conquer

- Validation vs. Verification
  - Are we building the right product? vs. Are we building it right?
  - Crossing from judgment to precise, checkable correctness property. Verification is at least partly automatable, validation is not
- Correctness is a *relation* between spec and implementation
  - To make a property verifiable (testable, checkable, ...) we must capture the property in a spec



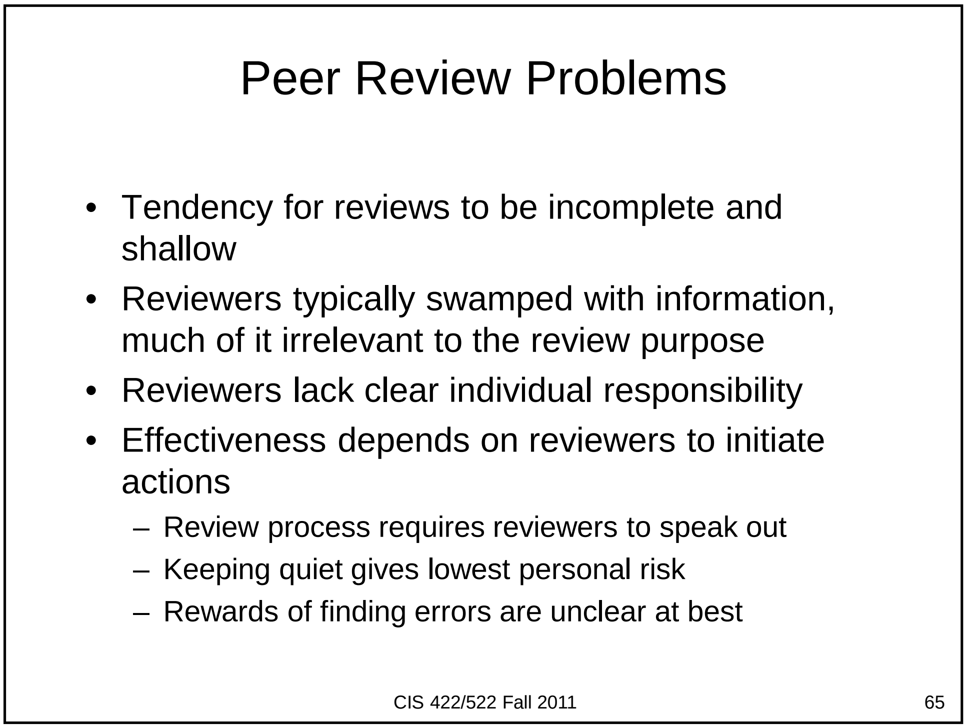
How (from why, who, when, what)

- Black box: Test design is part of designing good specifications
  - This will change specs, in a good way. Factoring validation from verification is particularly hard, but particularly cost-effective as it leverages and focuses expensive human judgment
- White (or glass) box: Test design from program design
  - Executing every statement or branch does not guarantee good tests, but omitting a statement is a bad smell.

Active Reviews

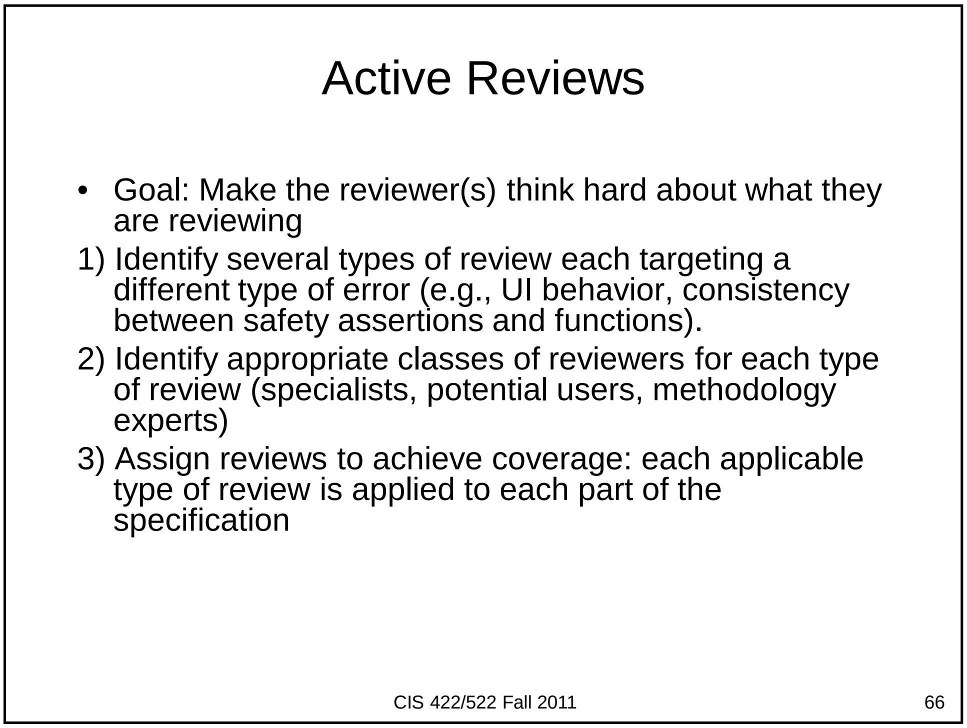
Peer Reviews

- Peer Review: a process by which a *software product is examined by peers of the product's authors with the goal of finding defects*
- Why do we do peer reviews?
  - Review is often the only available verification method before code exists
  - Formal peer reviews (inspections) instill some discipline in the review process
- Particularly important for distributed teams
  - Supports communication and visibility
  - Provides feedback on both *quality and understanding*
    - i.e., makes the communication effectiveness and level of understanding visible
  - A good review shows communication is working!



Peer Review Problems

- Tendency for reviews to be incomplete and shallow
- Reviewers typically swamped with information, much of it irrelevant to the review purpose
- Reviewers lack clear individual responsibility
- Effectiveness depends on reviewers to initiate actions
  - Review process requires reviewers to speak out
  - Keeping quiet gives lowest personal risk
  - Rewards of finding errors are unclear at best



Active Reviews

- Goal: Make the reviewer(s) think hard about what they are reviewing

1) Identify several types of review each targeting a different type of error (e.g., UI behavior, consistency between safety assertions and functions).
2) Identify appropriate classes of reviewers for each type of review (specialists, potential users, methodology experts)
3) Assign reviews to achieve coverage: each applicable type of review is applied to each part of the specification

Active Reviews (2)

4) Design review questionnaires (key difference)
   - Define questions that the review must answer by using the specification
   - Target questions to bring out key issues
   - Phrase questions to require “active” answers (not just “yes”)

5) Review consists of filling out questionnaires defining
   - Section to be reviewed
   - Properties the review should check
   - Questions the reviewer must answer

6) Review process: overview, review, meet
   - One-on-one or small, similar group
   - Focus on discussion of issues identified in review
   - Purpose of discussion is understanding of the issue (not necessarily agreement)

Conventional vs. Active Questions

- **Goal: Make the reviewer(s) think hard about what they are reviewing***
  - **Define questions that the review must answer by using the specification**
  - **Target questions to bring out key issues**
  - **Phrase questions to require “active” answers (not just “yes”)**

Conventional Design Review Questions	**Active Design Review Questions***
Are exceptions defined for every program?	For each access program in the module, what exceptions that can occur?
Are the right exceptions defined for every program?	What is the the range or set of legal values?
Are the data types defined?	For each data type, what are • an expression for a literal value of that data type; • a declaration statement to declare a variable for that type; • the greatest and least values in the range of that data type?
Are the programs sufficient?	Write a short pseudo-code program that uses the design to accomplish {some defined task}.

Why Active Reviews Work

- Focuses reviewer’s skills and energies where they have skills and where those skills are needed
  - Questionnaire allows reviewers to concentrate on one concern at a time
  - No one wastes time on parts of the document where there is little possibility of return.
- Largest part of review process (filling out questionnaires) is conducted independently and in parallel
- Reviewers must participate actively but need not risk speaking out in large meetings
- Downside: much more work for V&V (but can be productively pursued in parallel with document creation)

Real meaning of “control”

- What does “control” really mean?
- Can we really get everything under control then run on autopilot?
- Rather, does control mean a continuous feedback loop?
  1. Define ideal
  2. Make a step
  3. Measure deviation from idea
  4. Correct direction or redefine ideal and go back to 2

Questions?

Schedule

- Thursday (Final Presentations)
  - Meet in Colloquium Room, Deschutes
  - 20 minutes per team
    - Project overview
    - Demo
    - Lessons learned
- Final:
  - Alternate date: Tuesday, Dec. 6th, 12:00, Room 200, Deschutes
  - Scheduled Fri., Dec, 9th, 8:00

分布式软件开发导论

Introduction to Distributed Software Development CIS 423/510

Prof. Stuart Faulk
CIS
University of Oregon

Prof. Lian Yu
School of Software and Microelectronics
Peking University

CIS 423

- CIS 423: Globally Distributed Software Development
- Increasingly companies develop software using globally distributed teams
  - Different countries, languages, time zones, cultures
- Introduces interesting Software Engineering challenges
  - Coordinating the work
  - Communicating effectively

Course Structure

- Gain real experience with problems by simulating an industrial DSD project
  - Similar to way in-class project simulates co-located development
- Collaborate with students at Peking University on a software project
  - Each team will be roughly half UO, half PKU
  - Collaborate over the web to create, review, and present the results of development
- Learn to apply SE principles, methods and tools to support long-distance collaboration

Remote Collaboration

- **Offered Spring 2012**
- **Let me know if you have questions**

Distributed Software Development
Peking University, University of Oregon
Release 1 – June 20, 2010