CIS 422/522
2nd Half Concept Review

WALLY, YOU HAVE ACCOMPLISHED NONE OF YOUR GOALS. I HAVE TO LET YOU GO.
ACTUALLY, I ACCOMPLISHED A LOT.
I SPENT THE PAST TEN YEARS CREATING A TANGLE OF UN-DOCUMENTED PRO-GRAMMING CODE.
EVERY ONE OF OUR MAJOR SYSTEMS IS LINKED TO IT.
IF I DON'T ENTER A PASSWORD EVERY DAY, THE ENTIRE COMPANY WILL GO INTO A TECH-NOLOGY DEATH SPIRAL.
IF YOU VALUE YOUR JOB, YOU'LL GIVE ME A HUGE RAISE AND DANCE ON THIS TABLE LIKE A MONKEY!!!
LET'S CALL IT A TIE.
YEAH, I'M GOOD WITH THAT.
DilbertCartoonist@gmail.com
© 2013 Scott Adams, Inc.
www.dilbert.com

Presentations

- Plan on 12 minutes for your presentation including a couple minutes for questions and setup. Practice so the timing is right
- Presentations should address the following
  - Status against project plan
    - What was planned for this date?
    - What was actually produced?
  - Demo of project software
  - Lessons learned including but not limited to:
    - What were the root causes of any schedule delays and what would have helped?
    - What kinds of control issues caused the most problems?
    - Which software engineering techniques proved most effective in keeping your project under control?
  - Course improvements: suggestions for making the course better

Final Deliverables

- Fill out and return Peer Evaluation by Wednesday evening
- Make sure project deliverables are complete on your assembla site
  - Home page provides a directory to all deliverables
  - It is clear how to install and execute your app.
  - It is clear how to run your app
    - Provide any test cases and results you have
  - The final version of your code is easily downloadable
  - Your presentation slides are available

Final Exam

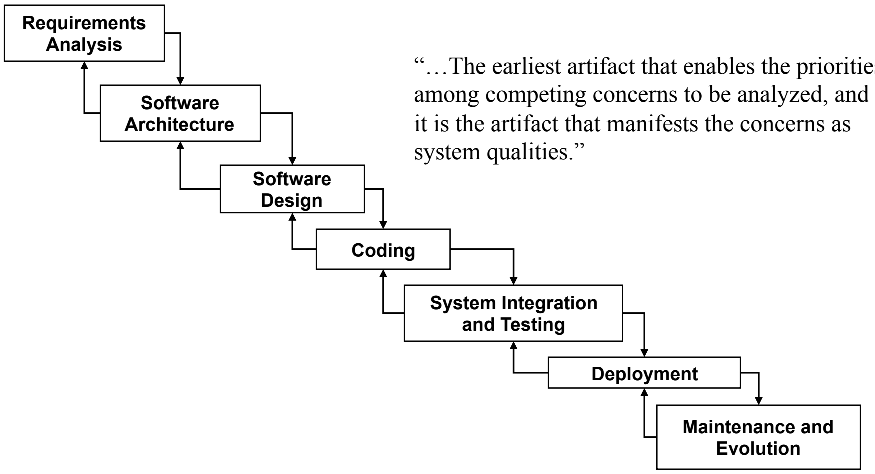
- Final Monday 10:15
- Same format as midterm

Concept Review

View of SE in this Course

- The purpose of software engineering 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).

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

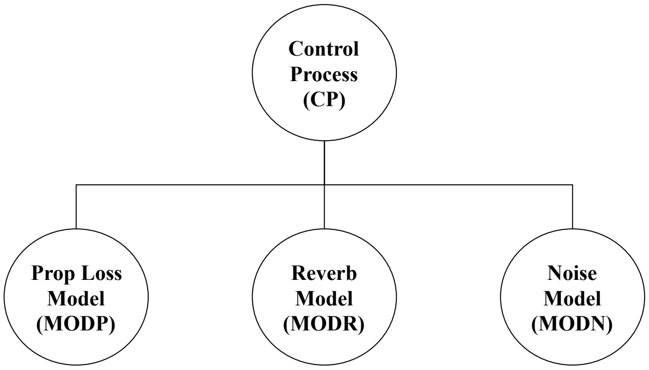
- Systems typically comprise more than one architecture
  - There is more than one useful decomposition into components and relationships
  - Each addresses different system properties or design goals
- It exists whether any thought goes into it or not!
  - Decisions are necessarily made if only implicitly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

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?

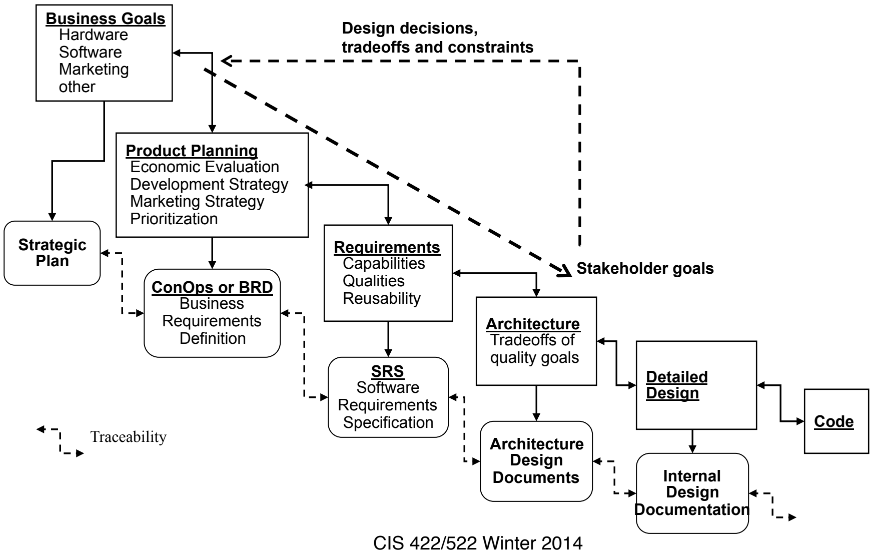
Effects of Architectural Decisions

- 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
- But not functional behavior

Relation to Stakeholders

- Many stakeholders have a vested interest in the architectural design
  - Management, marketing, end users
  - Maintenance organization, IV&V, Customers
  - Regulatory agencies (e.g., FAA)
- Important because their interests are diverse and often 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

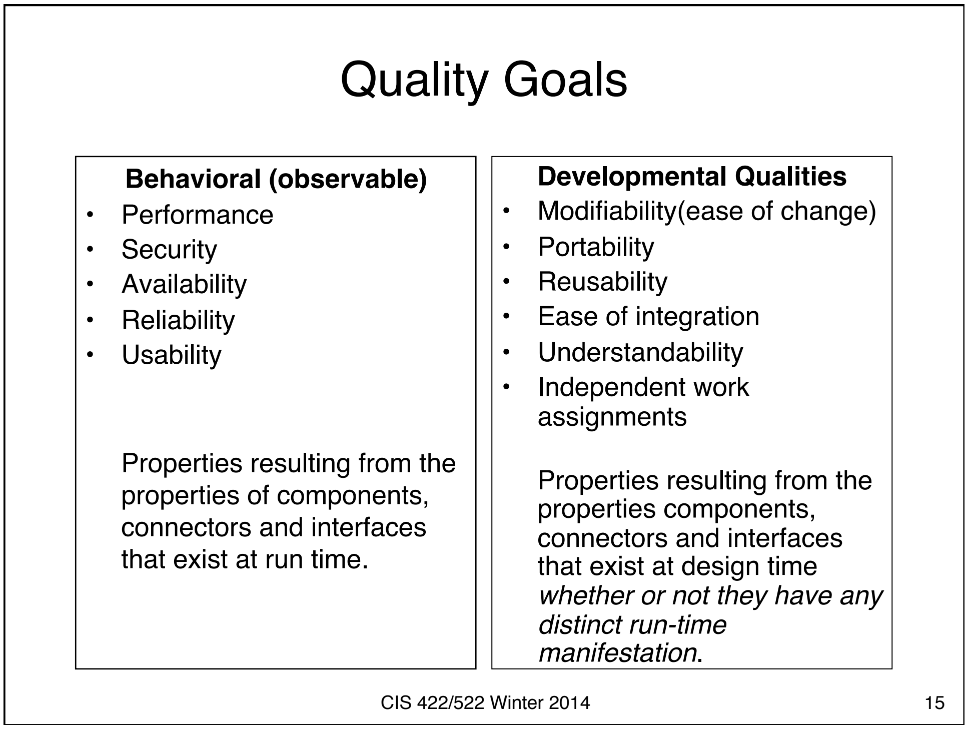
The Architectural Business Cycle



Implications for the Development Process

Goal is to keep developmental goals and architectural capabilities in synch:

- 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 Goals

Behavioral (observable)

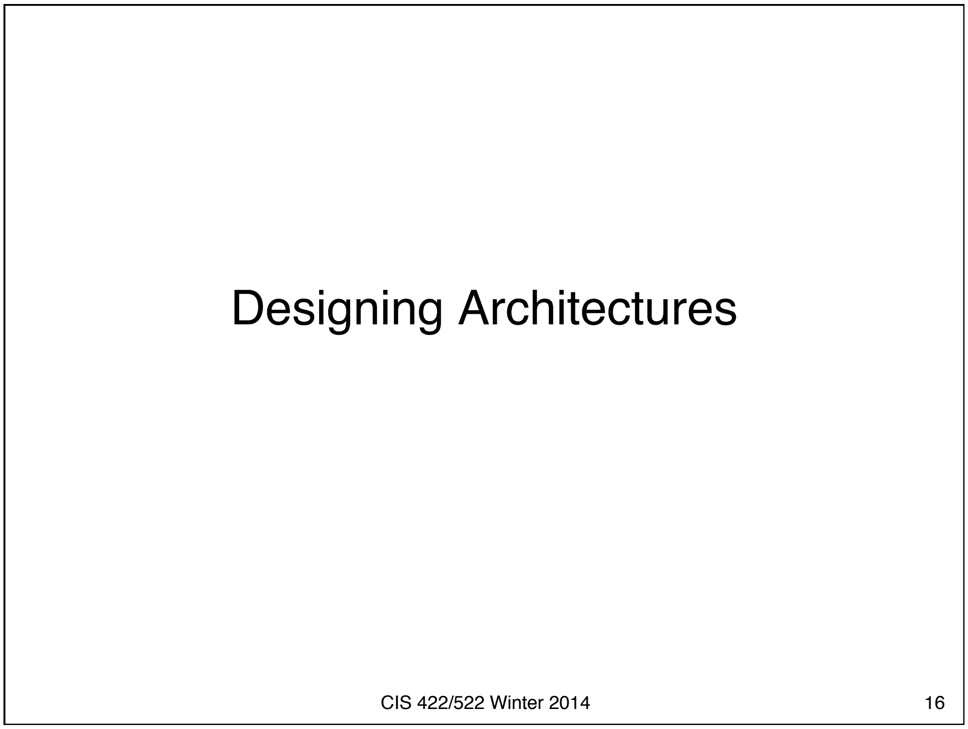
- Performance
- Security
- Availability
- Reliability
- Usability

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

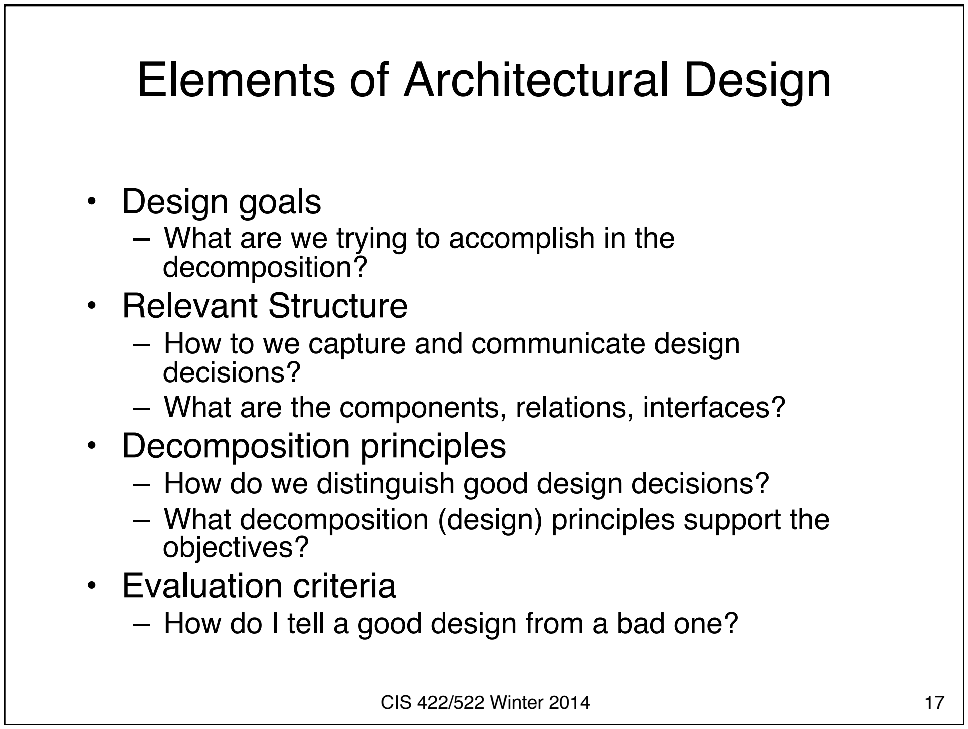
Developmental Qualities

- 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*.

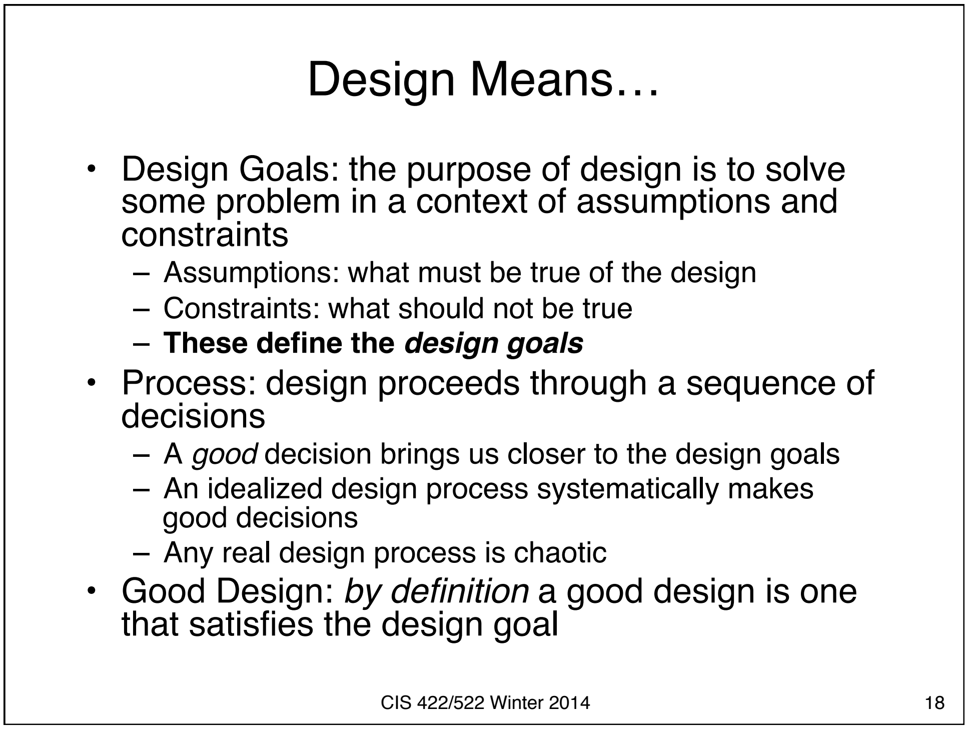


Designing Architectures



Elements of Architectural Design

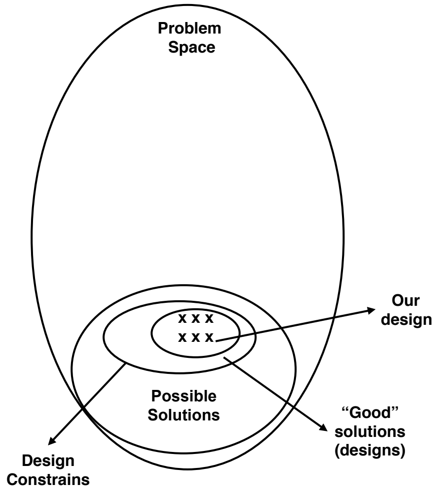
- Design goals
  - What are we trying to accomplish in the decomposition?
- Relevant Structure
  - How to we capture and communicate design decisions?
  - What are the components, relations, interfaces?
- Decomposition principles
  - How do we distinguish good design decisions?
  - What decomposition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



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



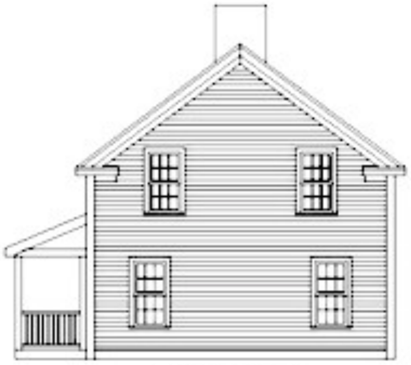
- A Design: is (a representation of) a solution to a problem
  - Represents a set of choices
    - Typically large set of possible choices
    - Must navigate through possibilities
    - Invariably requires tradeoffs
  - Some designs are better than others (notion of *good design*)
  - How do we know:
    - Where we are going?
    - Which choice to make?
    - Whether we have arrived?

Which structures should we use?

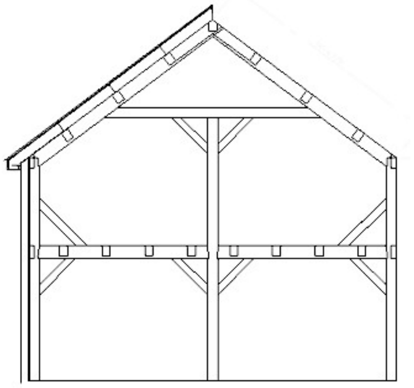
Structure	Components	Interfaces	Relationships
Calls Structure	Programs (methods, services)	Program interface and parameter declarations	Invokes with parameters (A calls B)
Data Flow	Functional tasks	Data types or structures	Sends-data-to
Process	Sequential program (process, thread, task)	Scheduling and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

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

Elevation/Structural



East View



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 and other stakeholders

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 best order in which to make decisions
    - Which of the available choices will lead to the design goals

Example: Designing the Module Structure

Modularization

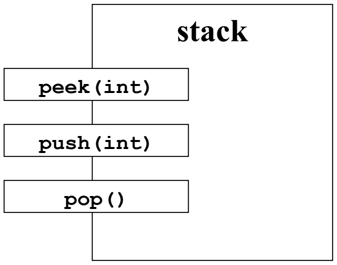
- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a “module.”
- Properties of a “good” module structure
  - Parts can be designed, understood, or implemented independently
  - Parts can be tested independently
  - Parts can be changed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

What is a module?

- 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

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



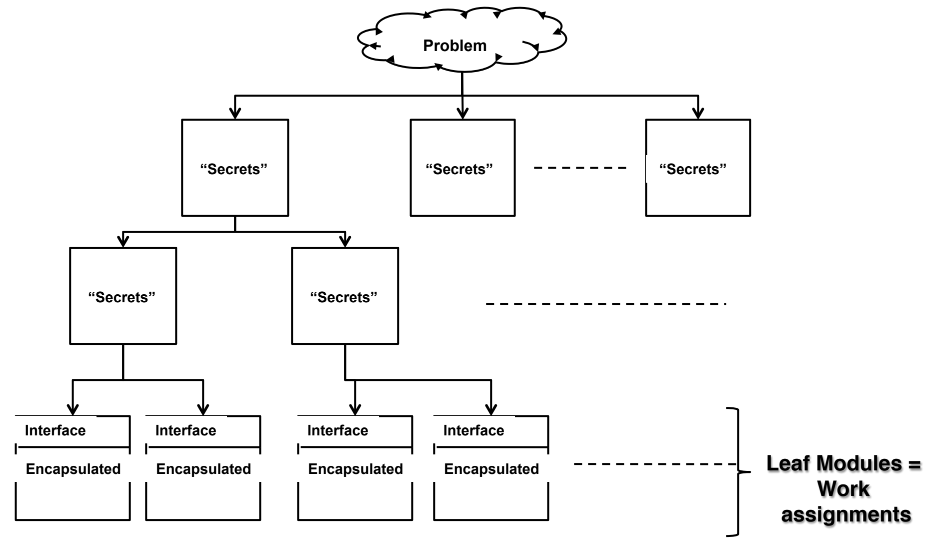
Module Hierarchy

- 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

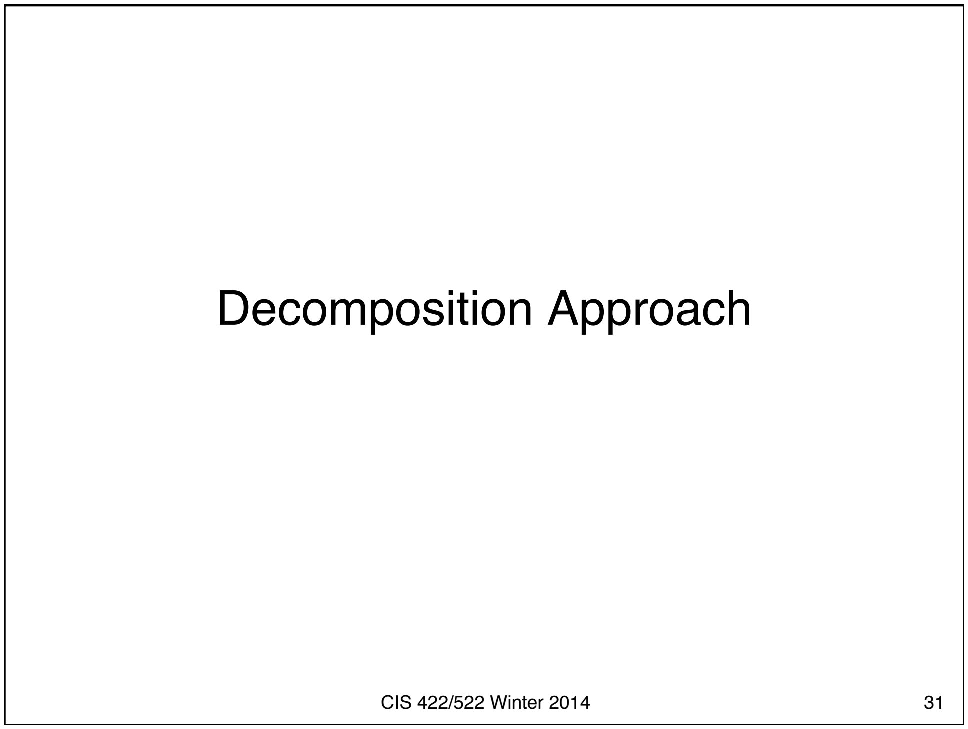
Modular Structure

- 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

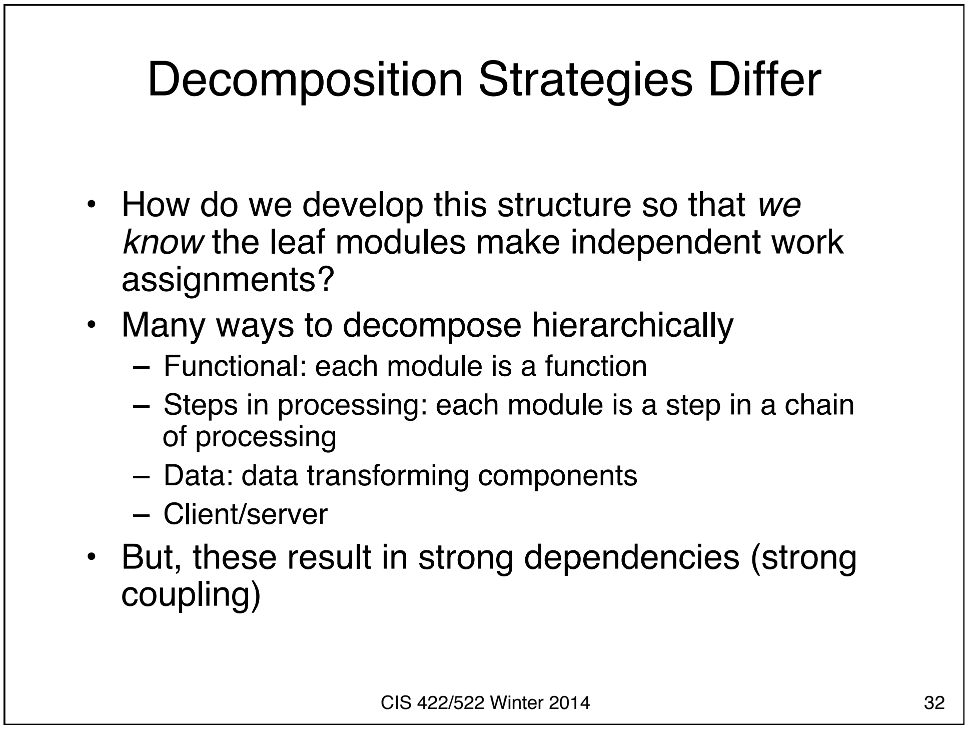
Module Hierarchy



→ Submodule-of relation



Decomposition Approach



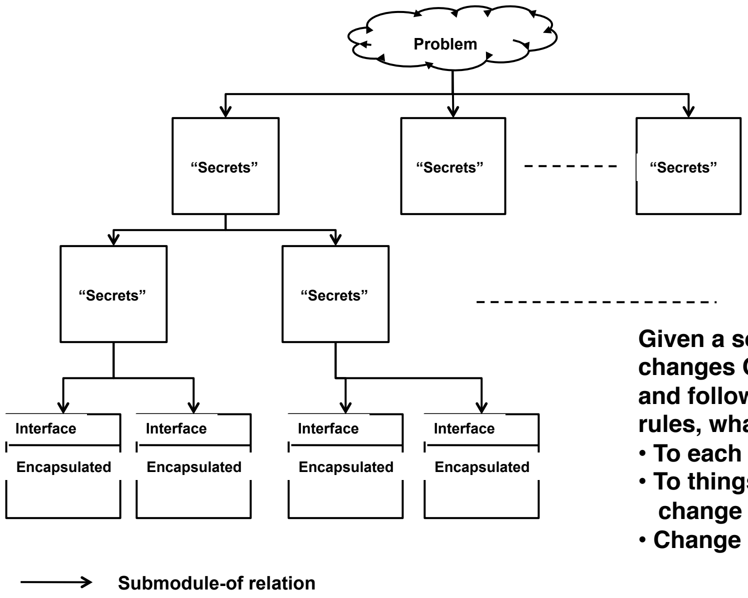
Decomposition Strategies Differ

- 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
- But, these result in strong dependencies (strong coupling)

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) 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*

Module Hierarchy



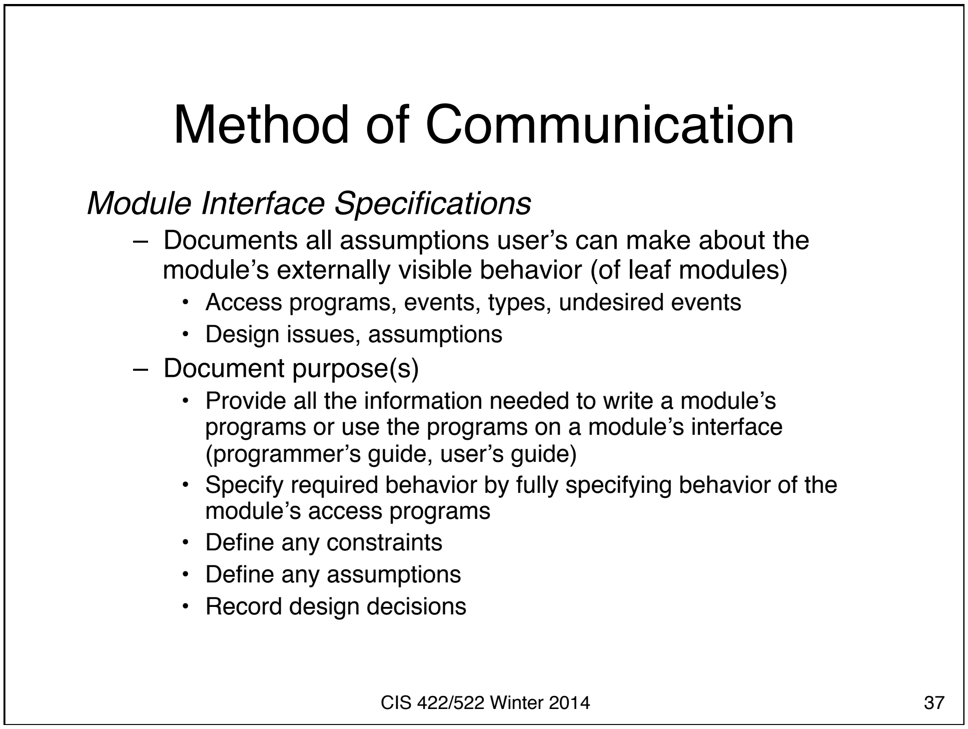
Given a set of likely changes C1, C2, … Cn and following these rules, what happens:

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

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

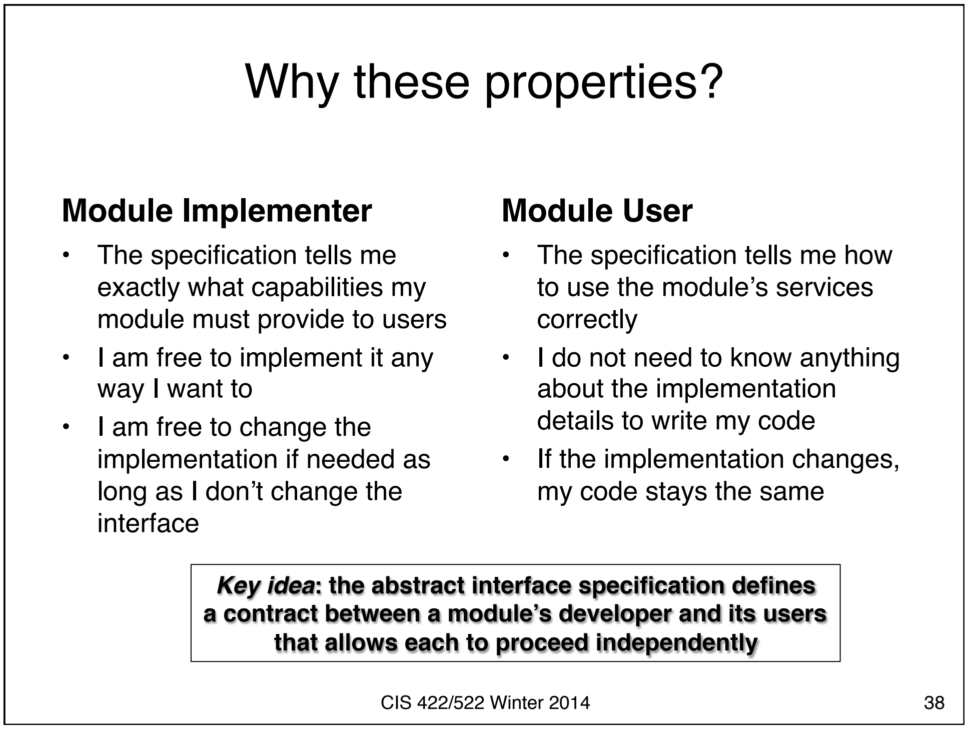
Specifying Abstract Interfaces



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



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

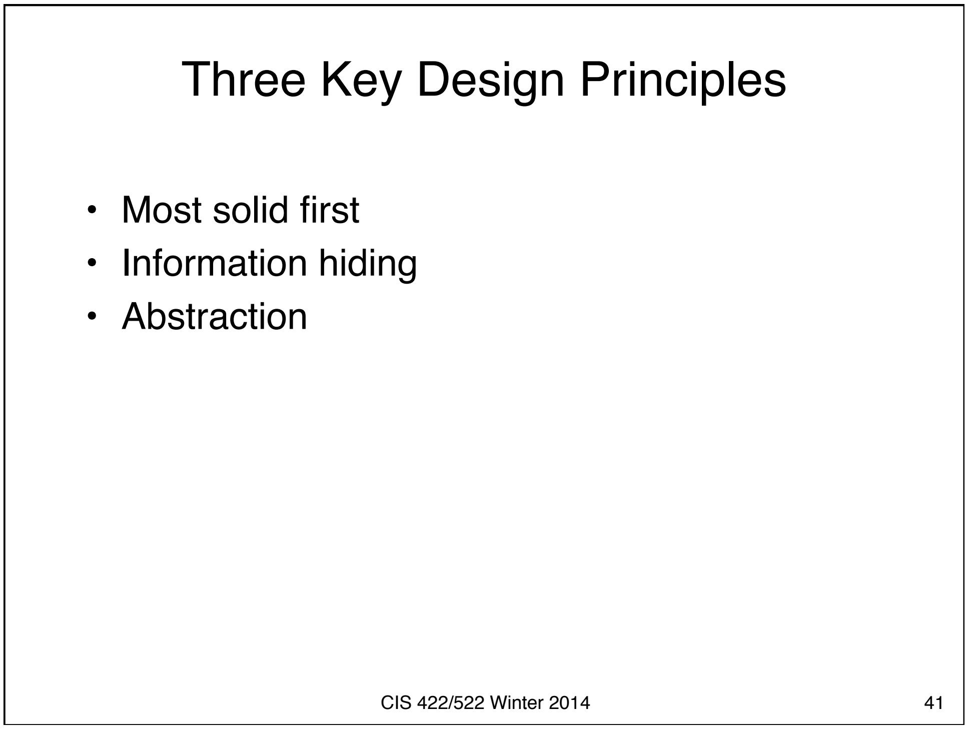
- The specification tells me how to use the module's services correctly
- I do not need to know anything about the implementation details to write my code
- If the implementation changes, my code stays the same

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

Design Principles

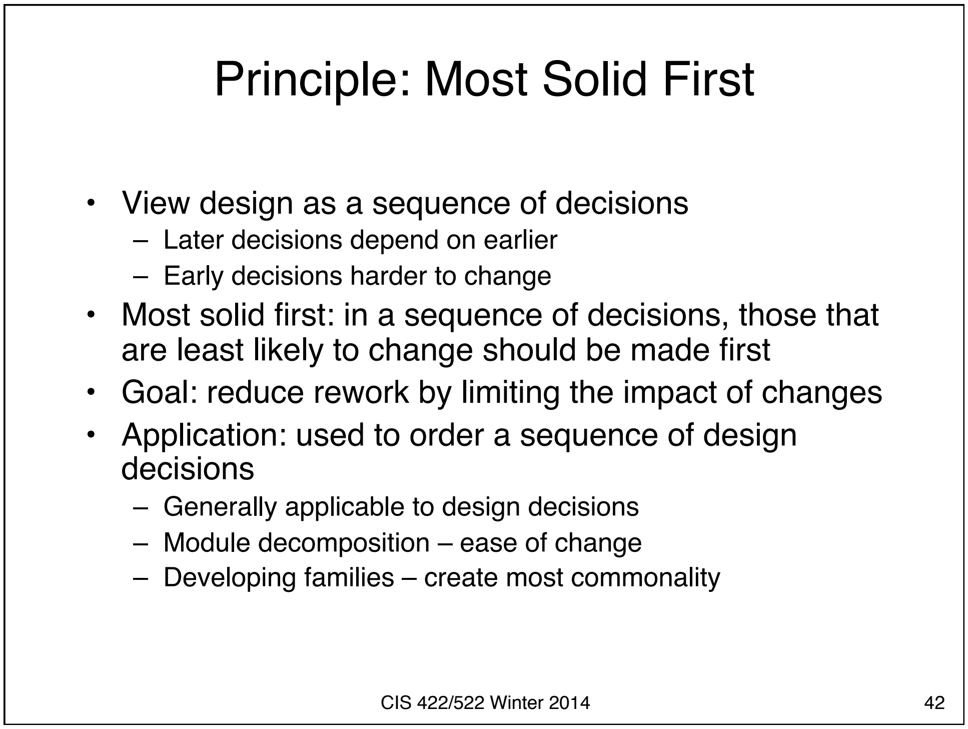
What are Principles?

- Principle (n): a comprehensive and fundamental rule, doctrine, or assumption
- Design Principles – rules that guide developers in making design decisions consistent with overall design goals and constraints
  - Guide the decision making process of design by helping choose between alternatives
  - Embodied in methods and techniques (e.g., for decompositions)



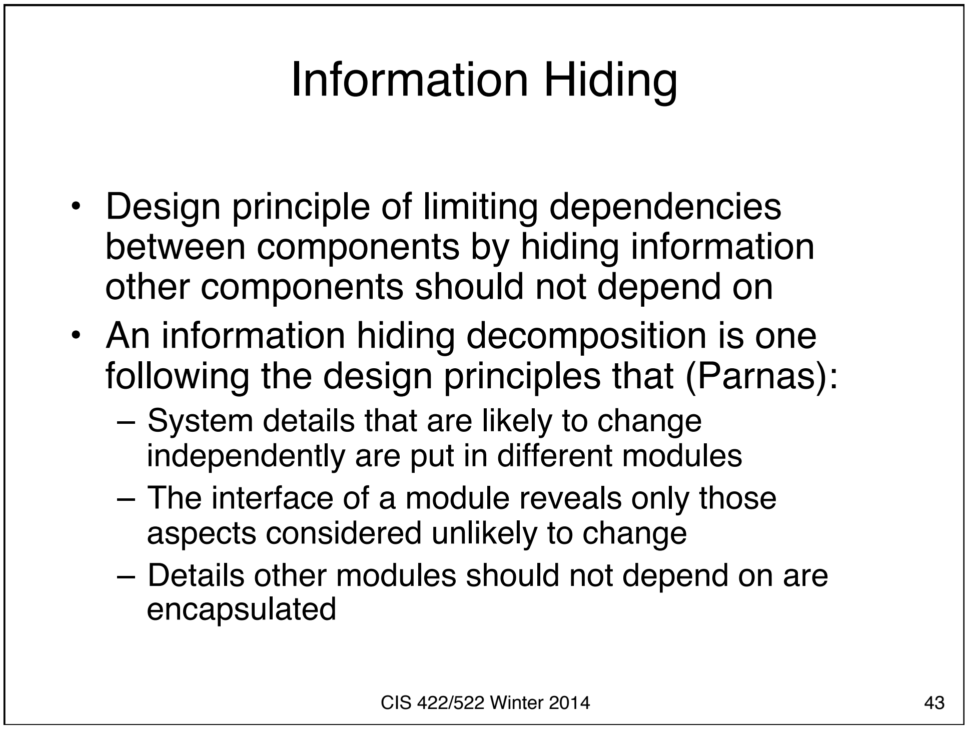
Three Key Design Principles

- Most solid first
- Information hiding
- Abstraction



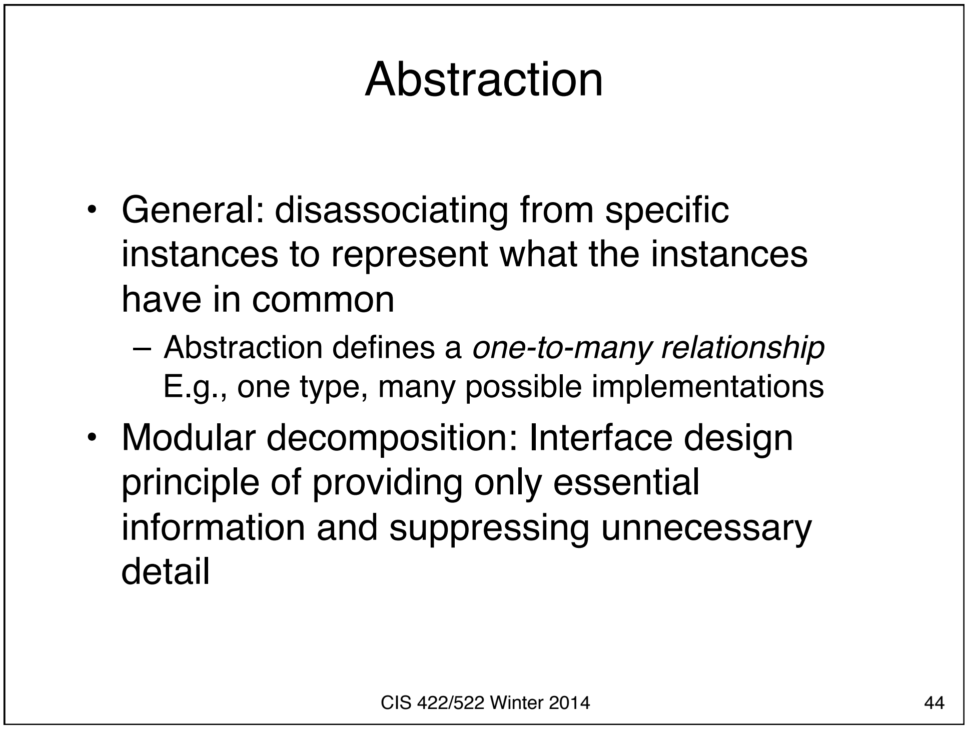
Principle: Most Solid First

- View design as a sequence of decisions
  - Later decisions depend on earlier
  - Early decisions harder to change
- Most solid first: in a sequence of decisions, those that are least likely to change should be made first
- Goal: reduce rework by limiting the impact of changes
- Application: used to order a sequence of design decisions
  - Generally applicable to design decisions
  - Module decomposition – ease of change
  - Developing families – create most commonality



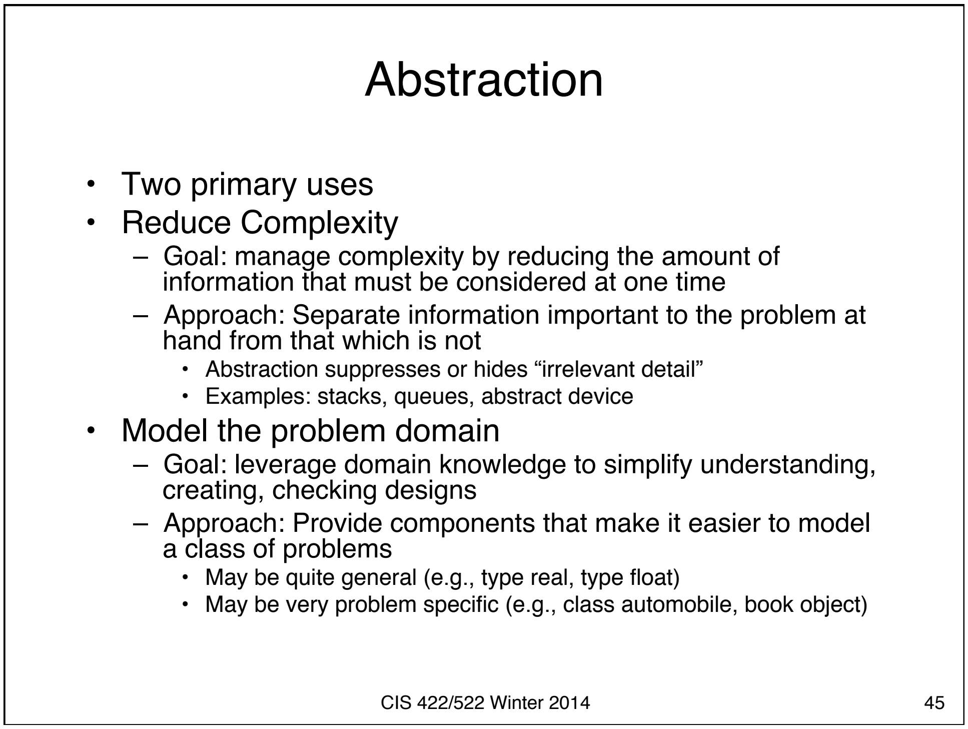
Information Hiding

- Design principle of limiting dependencies between components by hiding information other components should not depend on
- An information hiding decomposition is one following the design principles that (Parnas):
  - System details that are likely to change independently are put in different modules
  - The interface of a module reveals only those aspects considered unlikely to change
  - Details other modules should not depend on are encapsulated



Abstraction

- General: disassociating from specific instances to represent what the instances have in common
  - Abstraction defines a *one-to-many relationship* E.g., one type, many possible implementations
- Modular decomposition: Interface design principle of providing only essential information and suppressing unnecessary detail



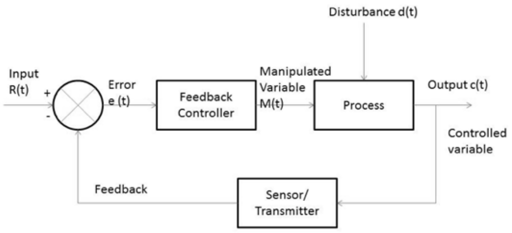
Abstraction

- 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)

Quality Assurance

Requires Feedback-Control

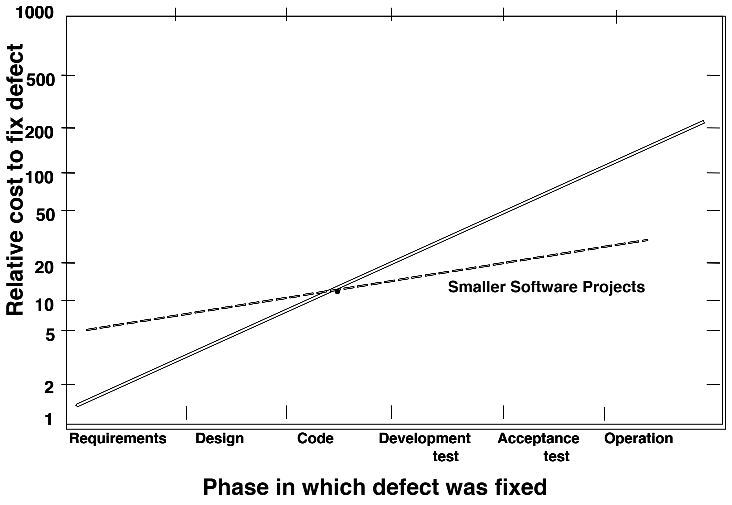
- Uncertainty means we cannot get everything under control then run on autopilot
- Rather control requires continuous feedback
  1. Define ideal
  2. Make a step
  3. Measure deviation from idea
  4. Correct direction or redefine ideal and go back to 2



QA Questions

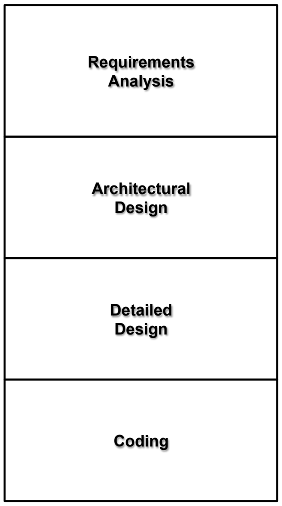
- Do the requirements capture what the stakeholders want?
  – Are they correct?
  – Are they complete relative to stakeholder needs?
    - Do they define functional and quality requirements?
- Are they internally complete and consistent?
- What if they change?
- Is the code consistent with the requirements?
- How do we check for these properties?

Increase in Software Cost-to-fix vs. Phase (1976) *



* Barry Boehm - A View of 20th and 21st Century Software Engineering

Quality is Cumulative


Requirements Analysis

Architectural Design

Detailed Design

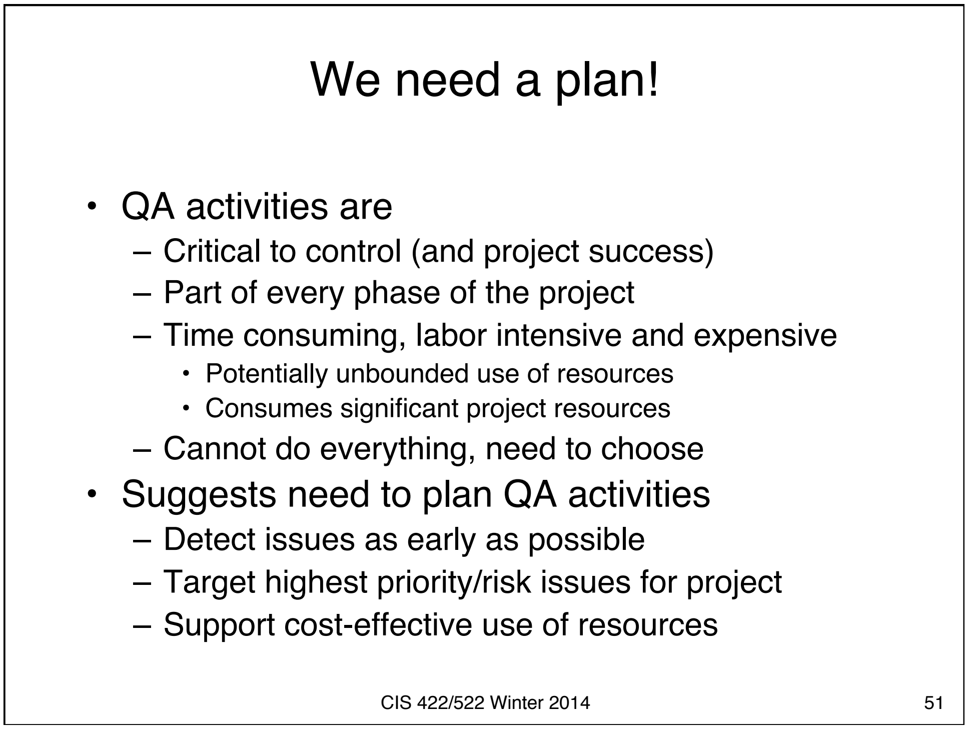
Coding

- Are the requirements valid?
- Complete? Consistent? Implementable?
- Testable?

- Does the design satisfy requirements?
- Are all functional capabilities included?
- Are qualities addressed (performance, maintainability, usability, etc.?

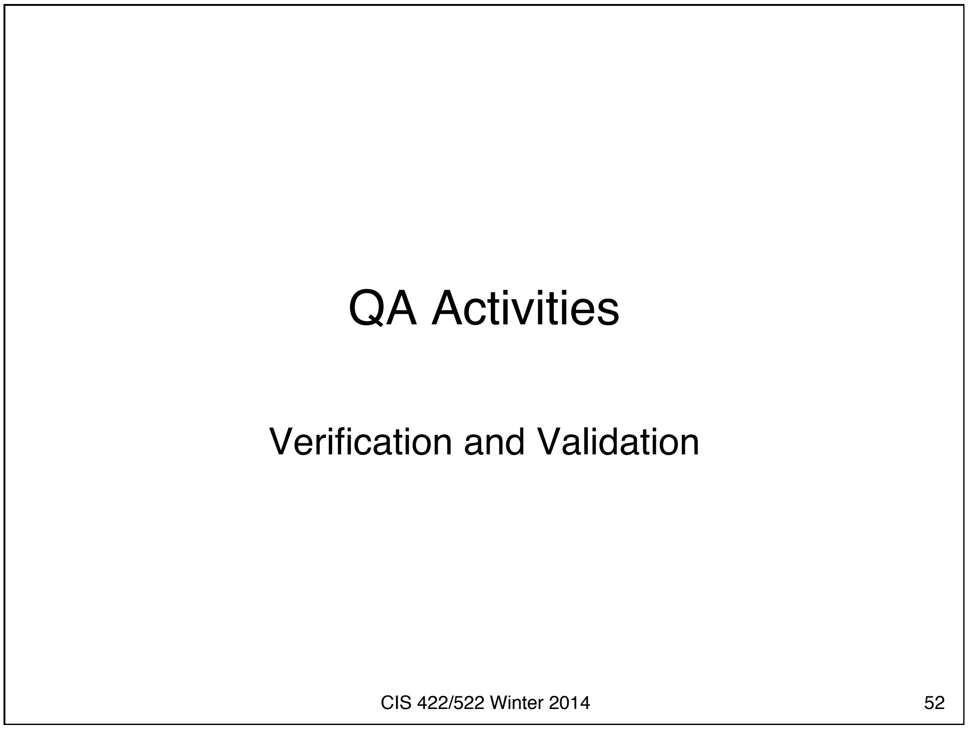
- Do the modules work together to implement all the functionality?
- Are likely changes encapsulated?
- Is every module well defined

- Implement the required functionality?
- Race conditions? Memory leaks? Buffer overflow?



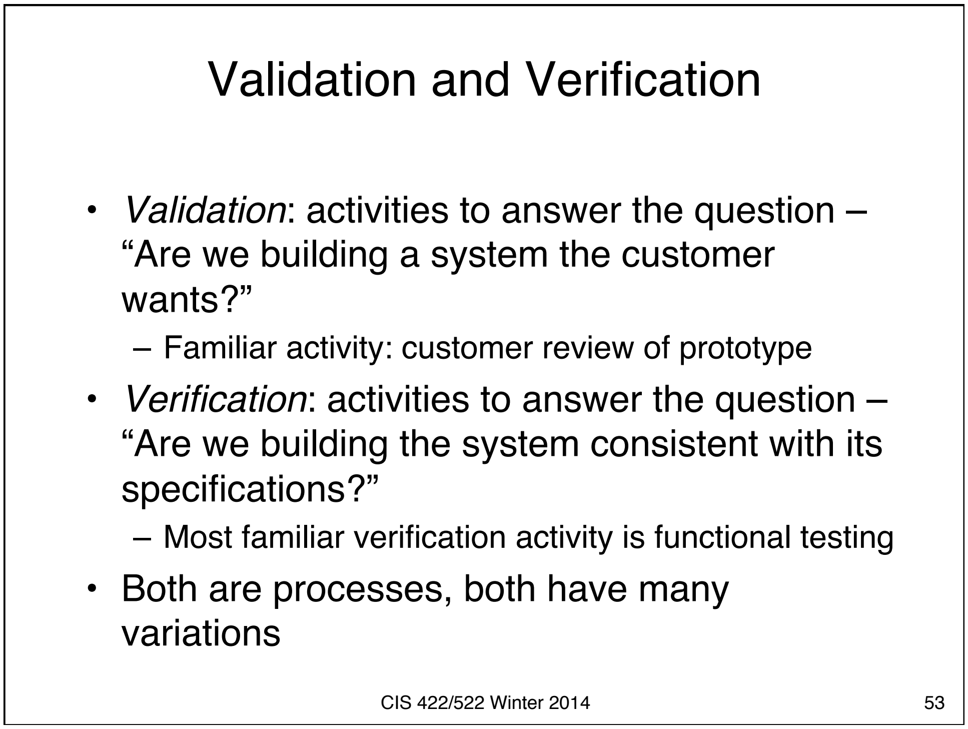
We need a plan!

- QA activities are
  - Critical to control (and project success)
  - Part of every phase of the project
  - Time consuming, labor intensive and expensive
    - Potentially unbounded use of resources
    - Consumes significant project resources
  - Cannot do everything, need to choose
- Suggests need to plan QA activities
  - Detect issues as early as possible
  - Target highest priority/risk issues for project
  - Support cost-effective use of resources



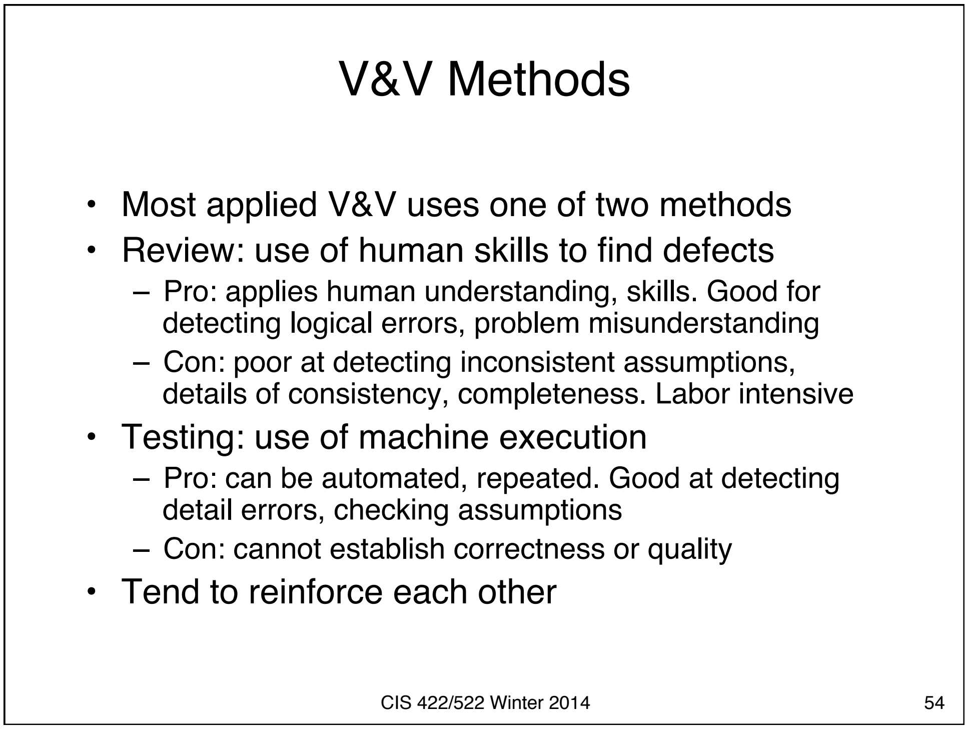
QA Activities

Verification and Validation



Validation and Verification

- *Validation*: activities to answer the question – “Are we building a system the customer wants?”
  - – Familiar activity: customer review of prototype
- *Verification*: activities to answer the question – “Are we building the system consistent with its specifications?”
  - – Most familiar verification activity is functional testing
- Both are processes, both have many variations

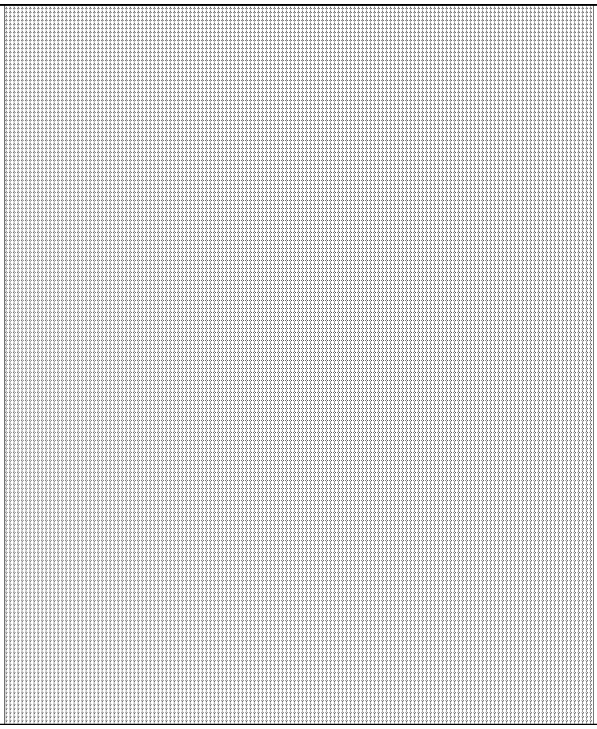


V&V Methods

- Most applied V&V uses one of two methods
- Review: use of human skills to find defects
  - – Pro: applies human understanding, skills. Good for detecting logical errors, problem misunderstanding
  - – Con: poor at detecting inconsistent assumptions, details of consistency, completeness. Labor intensive
- Testing: use of machine execution
  - – Pro: can be automated, repeated. Good at detecting detail errors, checking assumptions
  - – Con: cannot establish correctness or quality
- Tend to reinforce each other

Peer Review Process

- 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
  - Generally the *most effective manual technique for detecting defects*
- Means that you should be doing peer reviews, but…
  - Doesn't mean that manual inspections cannot be improved
  - Doesn't mean that manual inspections are the best way to check for every properties (e.g., completeness)
    - Should be one component of the overall V&V process



Example: IEEE software inspection process
(aka Fagan Inspection)

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
- Large meeting size hampers effectiveness, increases cost
- No way to cross-check unstated assumptions

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
2) Identify appropriate classes of reviewers for each type of review
3) Assign reviews to achieve coverage: each applicable type of review is applied to each part of the specification
4) Design review questionnaires (key difference)
   – Define questions that the reviewer 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
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

Examples

- In practice: an active review asks a qualified reviewer to check a specific part of a work product for specific kinds of defects by answering specific questions, e.g.,
  - Ask a designer to check the functional completeness by showing the calls sequences sufficient to implement a set of use cases
  - Ask a systems analyst to check the ability to create required subsets by showing which modules would use which
  - As a developer to check the data validity of a module's specification by showing what the output would be for in-range and out-of-range values
  - Ask a technical writer to check the SRS for grammatical errors
- Can be applied to any kind of artifact from requirements to code

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)

Development Realities

Developer Realities

- Nothing counts but delivery
  - Software product properties
    - Sufficient desired functionality
    - Acceptable qualities
  - Process properties
    - Timely
    - “low cost” (acceptable ROI)
- But…
  - Delivery must be repeatable, usually building on legacy systems
  - The target moves
  - The process is done largely in the dark

Issues

- Balancing all these factors is difficult
- Easiest to come up with partial, short-term solutions
  - Acceptable solution but late, over cost
  - On time delivery but difficult to change, maintain
  - Deliver but is not what the customer wants
  - Quick fix, difficult to maintain, etc.
- Results from complexity, shortsighted approach
  - Huge pressure to "code first, ask questions later"
  - Overall problem too complex to comprehend at once
  - Focus on parts of the problem, excluding others
  - Fail to look ahead (paint ourselves into a corner)

Software Engineering

- Principles of Software Engineering provide an antidote
- Helps to foresee downstream problems of poor decisions
- Supports doing the right thing rather than only the most "urgent"
- Provides principles and tools to keep a project in control

Real meaning of “control”

- What does “control” really mean?
- Cannot get everything under control then run on autopilot
- Rather, control means a continuous feedback loop
    1. Define ideal or goal
    2. Make a step
    3. Measure deviation from idea
    4. Correct direction or redefine ideal and go back to 2

Questions?