CIS 422/522
Second Half Review

Grading Modifications

- Will drop lowest test score, if that improves your overall grade
  - Means that you can skip 2nd midterm
- Case 2: you skip final (or drop grade):

Proj. 1	Proj. 2	Test	Participation
25	45	20	10

- Case 1: you take the final

Proj. 1	Proj. 2	Midterm	Final	Participation
20	40	15	15	10

  - Compute your grade with and without lowest test grade, take best

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).

Real meaning of "control"

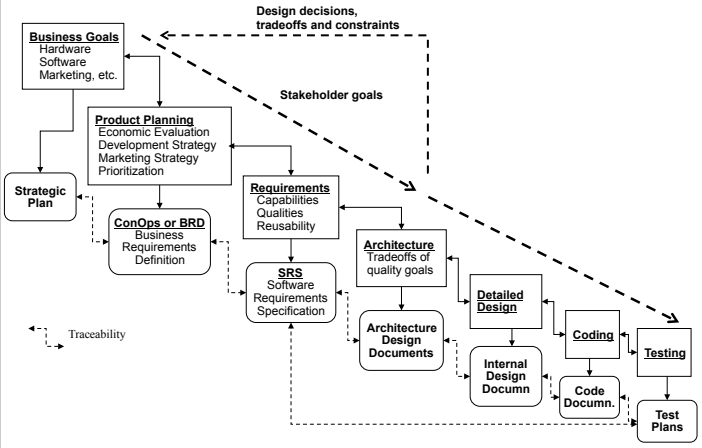
- What does "control" really mean?
- Cannot get everything under control then run on autopilot
- Rather, control means a continuous feedback loop
  1. Define ideal or goal
  2. Make a step
  3. Measure deviation from idea
  4. Correct direction or redefine ideal and go back to 2

Achieving System Qualities Through Software Architecture

Key System Properties

- System qualities stakeholders may require
- 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

Controlling the Product Cycle



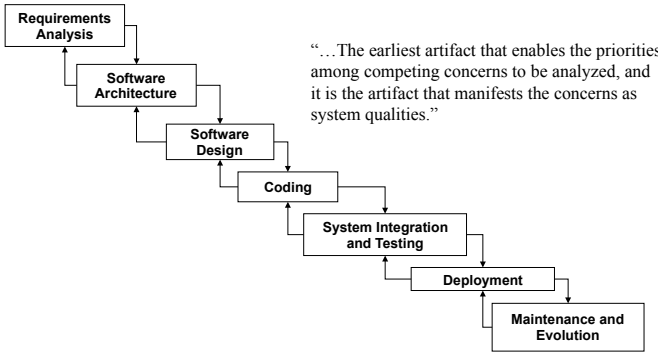
Behavioral vs. Developmental

Behavioral (observable)	**Developmental Qualities**
• Performance • Security • Availability • Reliability • Usability Properties resulting from the properties of components, connectors and interfaces that exist at run time.	• 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*.

Importance to Stakeholders

- There are many stakeholders with many possible quality requirements
- Important because their interests often conflict
  - E.g. Performance vs. security, initial cost vs. maintainability
  - Requires making tradeoffs in the system design
  - Making successful tradeoffs requires understanding the nature, source and priority of the quality requirements
- This is this is the job of the system architect

Fit in the Development Cycle



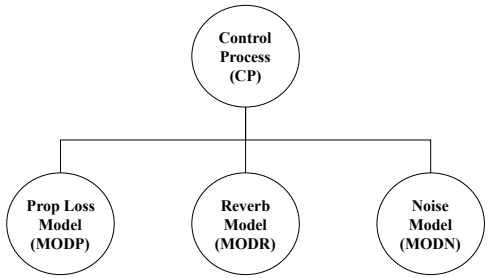
"…The earliest artifact that enables the priorities among competing concerns to be analyzed, and it is the artifact that manifests the concerns as system qualities."

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 things called "architecture" are not

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?

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

Implications for the Development Process

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

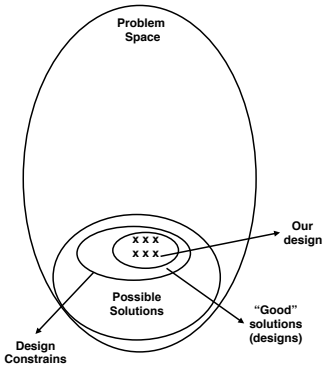
- Understand the goals for the system (e.g., business case or mission)
- Understand/communicate the quality requirements
- Design architecture(s) that satisfy quality requirements
  - Choose appropriate architectural structures
  - Design structures to satisfy qualities
  - Document to communicate design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

Designing Architectures

Design Means…

- Design Goals: the purpose of design is to solve some problem in a context of assumptions and constraints
  - Requirements: behavioral and developmental
  - 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*)

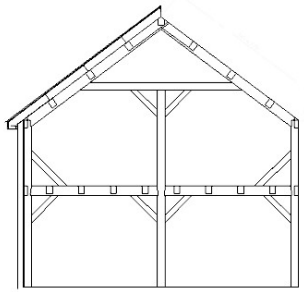
Which structures should we use?

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

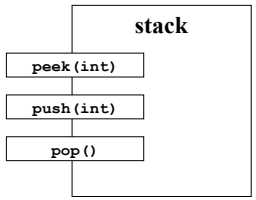
- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a “module.”
- Properties of a “good” module structure
  - Parts can be designed, understood, or implemented independently
  - Parts can be tested independently
  - Parts can be changed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

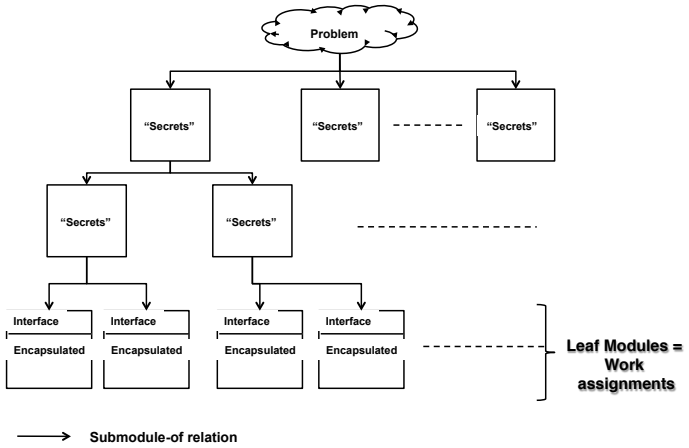
- A simple integer stack
- The *interface* specifies what a programmer needs to know to use the stack correctly, e.g.
  - *push*: push integer on stack top
  - *pop*: remove top element
  - *peek*: get value of top element
- The *secrets* (encapsulated) any details that might change from one implementation 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

Module Hierarchy



Modular Structure

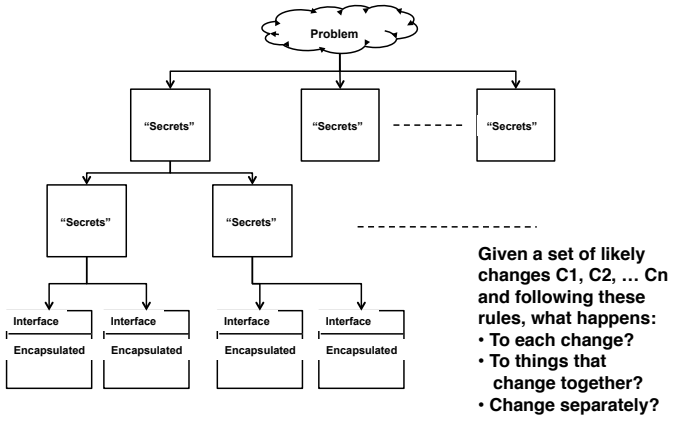
- Comprises components, relations, and interfaces
- Components
  - Called modules
  - Leaf modules are work assignments
  - Non-leaf modules are the union of their submodules
- Relations (connectors)
  - submodule-of => implements-secrets-of
  - The union of all submodules of a non-terminal module must 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 Approach

Module 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?**

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)

Specify Module Interfaces

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

Define as *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

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**

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

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

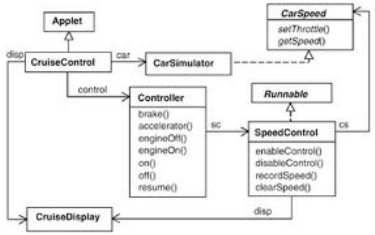
- Addressed using the *principle of information hiding*
- IH: design principle of limiting dependencies between components by hiding information other components should not depend on
- When thinking about what to put on the interface
  - Design the module interface to reveal only those design decisions 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

2. Provide Services

- Addressed by principle of *Abstraction: a*bstract interface provide only the needed services
- Helps reduce complexity
  - Approach: Separate information important to the problem at hand from that which is not
  - Reduces the amount of information that must be considered at one time
  - Abstraction suppresses or hides "irrelevant detail"
- Improves understanding
  - Approach: Provide abstractions (e.g., types) 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)
  - Leverage domain knowledge to simplify understanding, creating, checking designs

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*

*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
  - "Early" errors are the most costly
    - Misunderstanding of requirements, architecture that does not support a needed change, ...
- Goal is to reduce the gap between making an error and fixing it
  - Continue 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.

Questions?