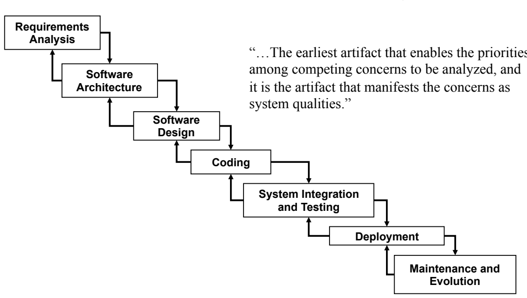
CIS 422/522
2nd Half Concept Review

Stuart Faulk

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)

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."

Implications of the 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

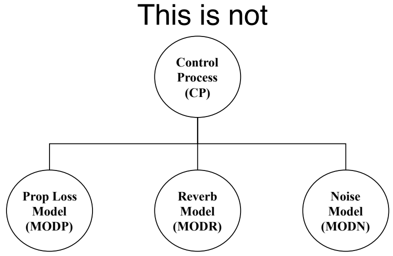
- Systems typically comprise more than one architecture
  - There is more than one useful decomposition into components and relationships
  - Each addresses different system properties or design goals
- It exists whether any thought goes into it or not!
  - Decisions are necessarily made if only implicitly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

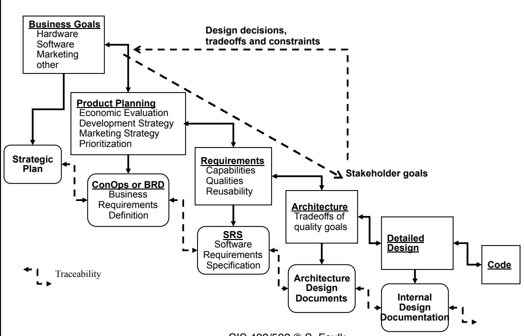
Effects of Architectural Decisions

- 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
- *But not functional behavior*

Relation to Stakeholders

- Many stakeholders have a vested interest in the architectural design
  - Management, marketing, end users, maintenance, IV&V, Customers, etc
- Important because their interests are diverse and often defy mutual satisfaction
  - There are inherently tradeoffs in most architectural design 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: 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
- Evaluate/correct the architecture
- Implement the system based on the architecture

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…

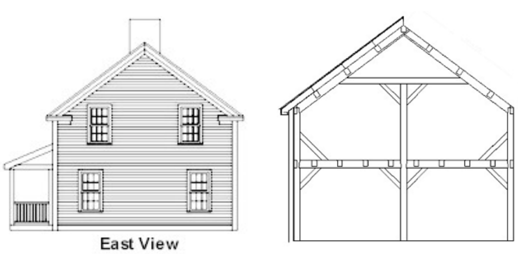
- Design Goals: the purpose of design is to solve some problem in a context of assumptions and constraints
  - Assumptions: what must be true of the design
  - Constraints: what should not be true
  - **These define the *design goals***
- Process: design proceeds through a sequence of decisions
  - A *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

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



Models/Views

- Different views answer different kinds of questions
- Designing for particular software qualities also requires the right architectural model or “view”
  - Any model presents a subset of system structures and properties
  - Different models answer different kinds of questions about system properties
- Goal is choose a set of views where
  - Structures determine key required qualities
  - Consequences of related design choices are made visible

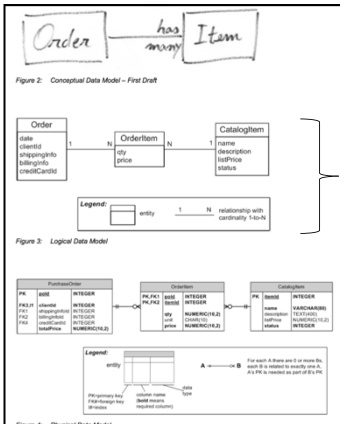


Figure 2: Conceptual Data Model – First Draft

Figure 3: Logical Data Model

Figure 4: Physical Data Model

Example: Data Model View

- Data Model Architecture
  - Entities: data structures
  - Relations: cardinality, aggregation, generalization/ specialization
  - Interface: attributes
- Model/communicate structure of complex data
  - What data is kept?
  - How is it related?
  - How is it structured and accessed in the system?

Example: Designing the Module Structure

Modularization

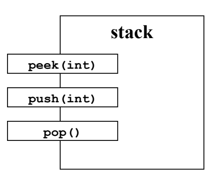
- For large, complex software, must divide the development into work assignments (WBS). Each work assignment is called a “module.”
- Properties of a “good” module structure
  - Parts can be designed, understood, or implemented independently
  - Parts can be tested independently
  - Parts can be changed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

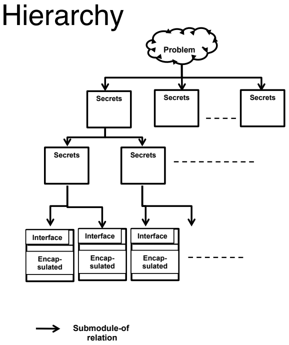
A Simple Module

- A simple integer stack
- The *interface* specifies what a programmer needs to know to use the stack correctly, e.g.
- 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, organize modules such that
  - Every requirement is allocated to some module
  - Can easily find the module providing a 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



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

Design Approach

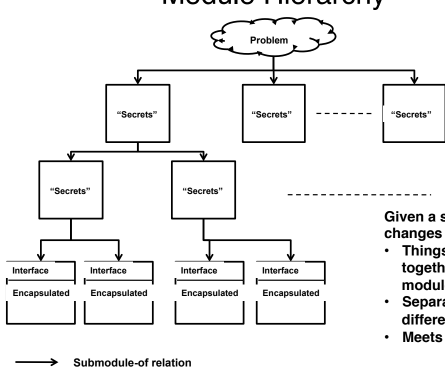
Decomposition Strategies Differ

- 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)
  - Group design decisions that are likely to change together in the same module
  - Allocate design decisions that are likely to change independently in 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
- **Things that change together in same module**
- **Separately in different modules**
- **Meets design goals**

Takeaway

- Understand the module structure design goals
- Understand how the IH principle is used to decompose modules
- Understand how its application results in a structure that satisfies those goals

Specifying Abstract Interfaces

Module Interface Specs

- Documents all assumptions user's can make about the module's externally visible behavior
  - 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)

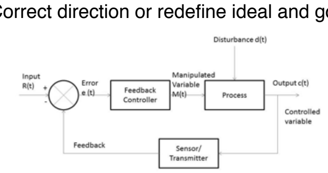
Key Design Principles

- Three principles covered
  - Most solid first
  - Information hiding
  - Abstraction
- Should understand
  - Design guidance provided by each principle
  - The result of applying the principle (e.g., from examples covered in class)
  - Why principles are more effective than heuristics

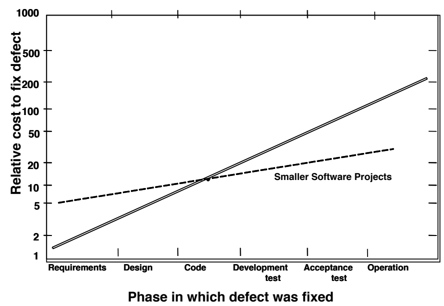
Quality Assurance

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



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

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

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

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

We need a plan!

- QA activities are
  - Critical to control
  - Part of every phase of the project
  - Time consuming, labor intensive and expensive
    - 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?"
  - E.g. customer review of prototype
- *Verification*: activities to answer the question – "Are we building the system consistent with its specifications?"
  - E.g., functional testing

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 there are issues

Issues with Standard Reviews

- 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 Review Method

Key idea: Works by forcing the reviewer to actually use the artifact to answer specific questions

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
4. Design review questionnaires
5. Review consists of filling out questionnaires defining
6. Review process: overview, review, meet

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
  - Ask a technical writer to check the SRS for grammatical errors
- Can be applied to any kind of artifact from requirements to code

Takeaway

- Understand when and why reviews should be used
- Understand how active reviews work
- Understand why they are better at detecting defects

Testing

Testing Fundamentals

- Coding produces errors
  - Data show 30-85 errors are made per 1000 SLOC
- Testing: processes of executing the code to detect errors
- In practice, it is impossible to check for all possible errors by testing
- Even checking a useful subset is expensive
  - 40%-80% of development cost
  - Must be re-done when software changes
  - Potentially unbounded effort

Testing Fundamentals (2)

- Reality: must settle for testing a subset of possible inputs
  - Even extensively tested software contains 0.5-3 errors per 1000 SLOC
    - Pesticide Paradox: *every method used to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual [Beizer]*
  - Always a tradeoff of cost vs. errors found
- Fundamental cost/benefit questions
  - Which subsets of possible test cases will find the most errors?
  - Which will find the most important errors?
  - How much testing is enough?

Ideal Testing Goal

- Goal: choose a sufficiently small but adequate set of test cases (input domain)
  - Small enough to economically run the complete set and re-run when software changes
  - "Adequate" much harder to define, generally means some combination of:
    - Acceptably close to required functional behavior
    - Contains no catastrophic faults
    - Reliable to an acceptable level (mean time to failure)
    - Within tolerance levels for qualities like performance, security, etc.

Number of Approaches

- Fault detection vs. Confidence building
- White-box vs. Black Box
- Different methods for choosing “adequate” test set
  - Coverage, fault-detection, operational profiles

Experimental Results

- There is no uniformly best technique
- Different techniques tend to reveal different types of faults
- Multiple techniques reveal more faults (at a cost)
- Cost-effectiveness of run-time testing is low, particularly compared to inspections (vast majority of tests find no errors)
  - Design review: 8.44
  - Code review: 1.38
  - Testing: 0.17

Interpretation

- A combination of manual and automated techniques is most cost effective
  - People are better at detecting many kinds of errors than machines
  - Machines are better at repetitive checks and minute details (comparing values)
- Testing works best in a supporting role (checking assumptions)
  - Activity of producing test cases and results double-checks other artifacts
    - Is it well enough defined to write a good test case?
    - Are edge cases defined? Etc.
  - Gives feedback on assumptions and expectations: does the system do what we expect?

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

End