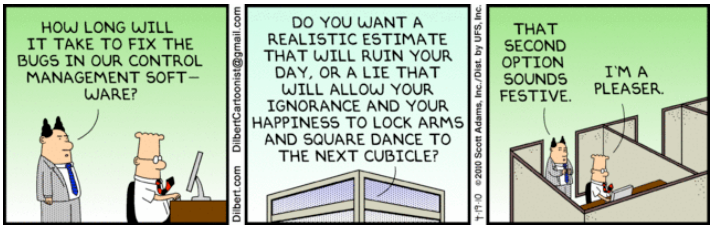
Midterm Review

CIS 422/522

Stuart Faulk

(with slight modifications by A. Hornof, 2018)



The “Software Crisis”

- Have been in “crisis” since the advent of big software (roughly 1965)
- What we want for software development
  - Low risk, predictability
  - Lower costs and proportionate costs
  - Faster turnaround
- What we have:
  - High risk, high failure rate
  - Poor delivered quality
  - Unpredictable schedule, cost, effort
  - Examples: Ariane 5, Therac 25, Mars Lander, DFW Airport, FAA ATC, Cover Oregon
- Characterized by **lack of *control***

Large System Context

- Discuss issues in terms of large, complex systems
  - Multi-person: many developers, many stakeholders
  - Multi-version: intentional and unintentional evolution
- Quantitatively distinct from small developments
  - Complexity of software rises exponentially with size
  - Complexity of communication rises exponentially
- Qualitatively distinct from small developments
  - Multi-person introduces need for organizational functions, policies, oversight, etc.
  - More stakeholders and more kinds of stakeholders
- We can only approximate this in our projects

Implications: the Large System Difference

- Small system development is driven by technical issues (I.e., programming)
- Large system development is dominated by organizational issues
  - Managing complexity, communication, coordination, etc.
  - Projects fail when these issues are inadequately addressed
- Lesson #1: **programming ≠ software engineering**
  - Techniques that work for small systems often fail utterly when scaled up
  - Programming alone won't get you through real developments or even this course

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**: able to make rational development decisions based on an understanding of the downstream effects of those choices.
  - **Managerial control** means we likewise control development *resources* (budget, schedule, personnel).

Course Approach

- Learn methods for acquiring and maintaining control of software projects (two threads)
- Managerial control
  - Team organization and people management
  - Organizing people and tasks
  - Planning and guiding development
- Intellectual control
  - Establishing and communicating exactly what should be built
  - Making effective decisions about system properties (behavioral and developmental)
  - Choosing appropriate order for decisions and ensuring feedback/correction

The Software Lifecycle

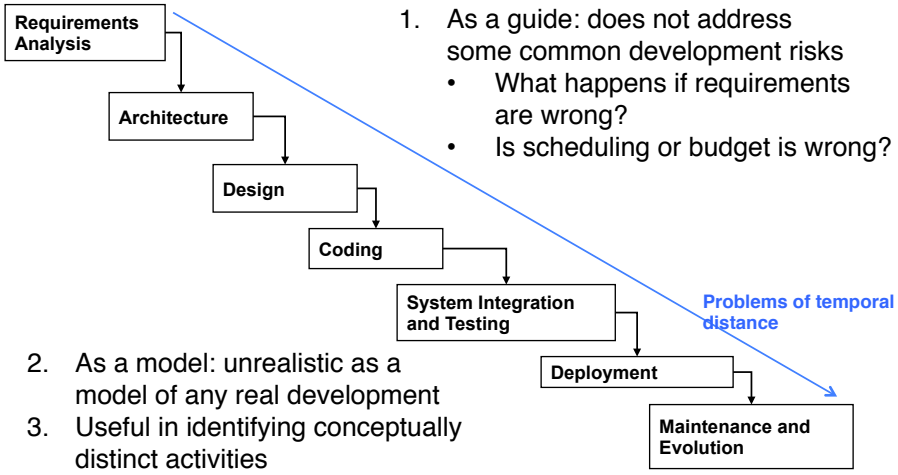
Need to Organize the Work

- Nature of a software project
  - Software development produces a set of interlocking, interdependent work products
    - E.g. Requirements -> Design -> Code
  - Implies dependencies between tasks
  - Implies dependencies between people
- Must organize the work such that:
  - Every task gets done
  - Tasks get done in the right order
  - Tasks are done by the right people
  - The product has the desired qualities
  - The end product is produced on time

Usefulness of Life Cycle Models

- Application of "divide-and-conquer" to software processes and products
  - Identify distinct process objectives
  - Can then address each somewhat separately
- Intended use
  - Provide guidance to developers in what to produce and when to produce it
  - Provide a basis for planning and assessing development progress
- Never an accurate representation of what really goes on

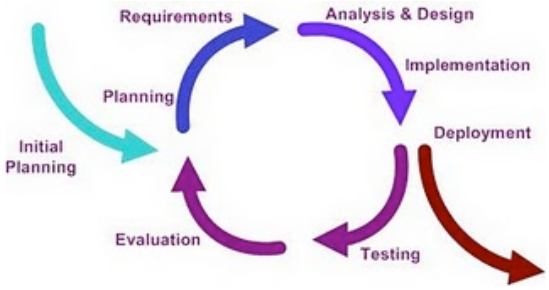
A "Waterfall" Model



1. As a guide: does not address some common development risks
   - What happens if requirements are wrong?
   - Is scheduling or budget is wrong?
2. As a model: unrealistic as a model of any real development
3. Useful in identifying conceptually distinct activities

Characteristic Processes: An Iterative Model...

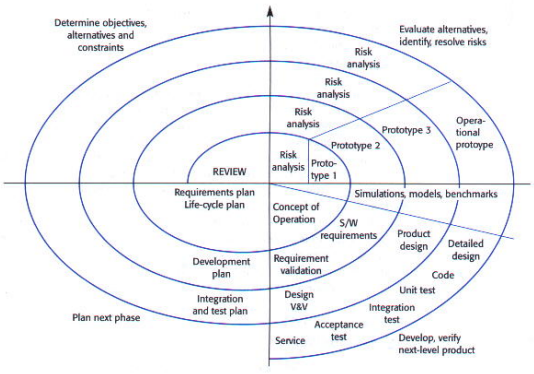
- Process is viewed as a sequence of iterations
  - Essentially, a *series of waterfalls*
- Addresses some common waterfall risks
  - Risk that software cannot be completed – build incremental subsets
  - Risk of building the wrong system – stakeholder have opportunities to see the software each increment
  - Also, can double check feasibility, schedule, budget and others issues



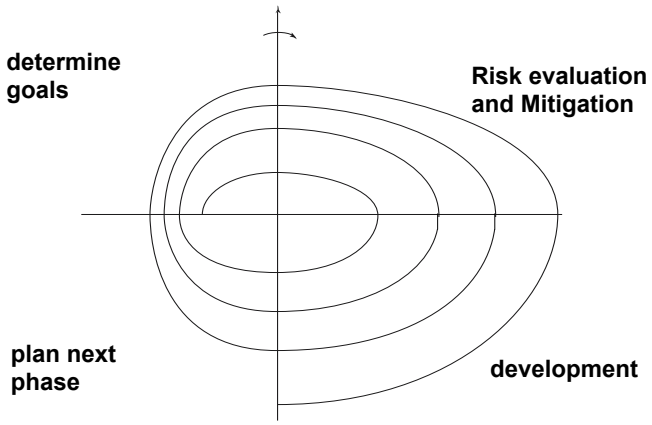
11

Characteristic Processes: The Spiral Model

- Process viewed as repeating cycles of increasing scale
- Identify risks and determine (next set of) requirements
- Each cycle builds next version by extension, increasing scale each time
- Explicit Go/No-Go decision points in process

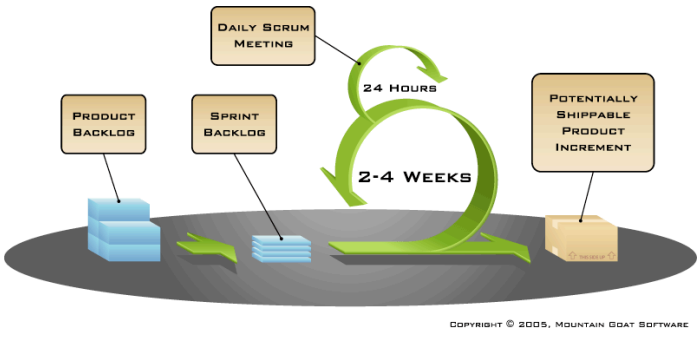


Spiral Model



Characteristic Processes: Agile (e.g. scrum)

- Process viewed as nested sequence of builds (sprints)
  - Each build adds very small feature set (one or two)
  - Nightly build/test, frequent customer validation
  - Focus on delivering code, little or no time spent on documentation



Process Objectives

- Objective: proceed in a controlled manner from stakeholder needs to a design that demonstrably meets those needs, within design and resource constraints
  - Understand that any process description is an abstraction
  - Always must compensate for deviation from the ideal
  - Still important to have a well-defined process to follow and measure against
- Choose process to provide an appropriate level of control for the given product and context
  - Sufficient control to achieve results, address risks
  - No more than necessary to contain cost and effort
- Question of control vs. cost: processes introduce overhead

Example

- Project 1 requirements and constraints
  1. Deadline and resources (time, personnel) are fixed
  2. Delivered functionality and quality can vary (though they affect the grade)
  3. Risks:
     1. Missing the deadline
     2. Technology problems
     3. Inadequate requirements
     4. Learning while doing
- Process model
  - All of these risks can be addressed to some extent by building some version of the product, then improving on it as time allows (software and docs.)
  - Technology risk requires building/finding software and trying it (prototyping)
  - Most forms of incremental development will address these

Project Planning and Management

Document Types and Purposes

- Management documents
  - Basis for **managerial control** of resources
    - Calendar time, skilled man-hours, budget
    - Other organizational resources
  - Project plan, WBS, Development schedule
  - Utility: supports resource allocation to meet time and budget constraints
    - allows managers to track actual against expected use of resources
- **Development documents**
  - **Basis for intellectual control of product content and quality**
  - **ConOps, Requirements (SRS), Architecture, Detail design, code, User's Guide, etc.**
  - **Utility: vehicles for making and recording development decisions**
    - **Allows developers to track decisions from stakeholder needs to implementation**

Project Plan

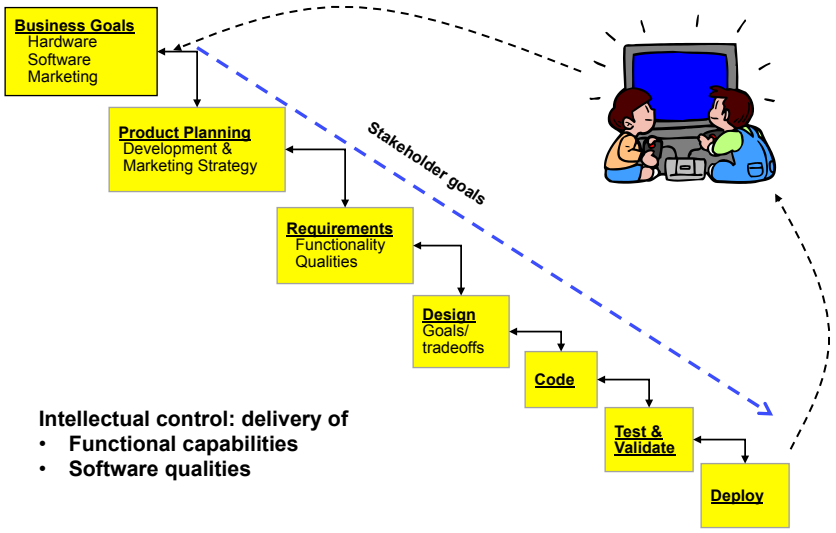
- Purpose: specifies how project resources will be organized to:
  - Create each deliverable
  - Meet quality goals
  - Address developmental goals (e.g., mitigate risk)
- Audience: should answer specific kinds of questions for different types of users, e.g.:
  - Customers: When will the product be delivered?
  - Stakeholders: What is the development approach? How does it address project risks?
  - Managers: When will tasks be completed? What is the current progress against the plan?
  - Developers: What should I be working on and when?

Planning Tools

- Work Breakdown Structure: decompose tasks and allocate responsibilities
  - If incomplete, some tasks may not be done
  - If imprecise, people do not know exactly what to do
  - Without a complete set of tasks, schedules are unrealistic
- PERT charts: identify where ordering of tasks may cause problems
  - Represent precedence or resource constraints
  - Identify critical path
- Gantt Charts: method for visualizing project schedule (tasks, dependencies, timing, persons)
- Note that these help address problems our projects have encountered

Intellectual Control

Product Development Cycle



Intellectual control: delivery of
- **Functional capabilities**
- **Software qualities**

Document Types and Purposes

- Management documents
  - Basis for project management (managerial control of resources)
    - Calendar time, skilled man-hours budget
    - Other organizational resources
  - Project plan, WBS, Development schedule
  - Use: allows managers to track actual against expected consumption of resources
- Development documents
  - Basis for intellectual control
    - Used for making and communicating engineering decisions (requirements, design, implementation, verification, etc.)
    - Allows developers to track decisions from stakeholder needs to implementation
  - Basis for communicating decisions
  - Requirements, Architecture, Detail design, Reviews, Tests

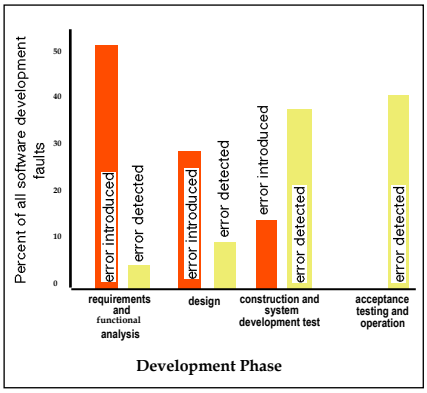
What is a “software requirement?”

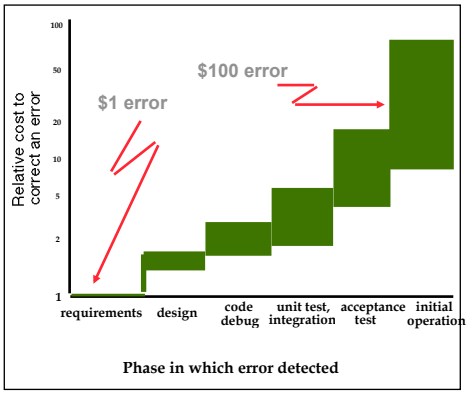
- A description of something the software must do or property it must have
- The set of system requirements denote the problem to be solved and any constraints on the solution
  - Specifies “what” not “how”
  - Bounds the set of acceptable implementations

Importance of Getting Requirements Right

1. The majority of software errors are introduced early in software development

2. The later that software errors are detected, the more costly they are to correct





Requirements Phase Goals

- What does “getting the requirements right” mean in the systems development context?
- Only three goals
  1. Understand precisely what is required of the software
  2. Communicate that understanding to all of the parties involved in the development (stakeholders)
  3. Control production to ensure the final system satisfies the requirements
- Sounds easy but hard to do in practice, observed this and the resulting problems in projects
- Understanding what makes these goals difficult helps us understand how to mitigate the risks

What makes requirements difficult?

- Comprehension (understanding)
  - People don't (really) know what they want (…until they see it)
  - Superficial grasp is insufficient to build correct software
- Communication
  - People work best with regular structures, coherence, and visualization
  - Software's conceptual structures are complex, arbitrary, and difficult to visualize
- Control (predictability, manageability)
  - Difficult to predict which requirements will be hard to meet
  - Requirements change all the time
  - Together make planning unreliable, cost and schedule unpredictable
- Inseparable Concerns
  - Many requirements issues cannot be cleanly separated
  - Difficult to apply "divide and conquer," must make tradeoffs
- Implication: all the requirements goals are difficult to achieve, must be managed as a risks!

Requirements Communication (Specification)

- Many potential stakeholders using requirements for different purposes
  - Customers: the requirements document what should be delivered
  - Managers: provides a basis for scheduling and a yardstick for measuring progress
  - Software Designers: provides the "design-to" specification
  - Coders: defines the range of acceptable implementations
  - Quality Assurance: basis for validation, test planning, and verification
  - Also: potentially Marketing, regulatory agencies, etc.

Documentation Approaches

- ConOps: informal requirements to describe the system's capabilities from the customer/user point of view
  - Answer the questions, "What is the system for?" and "How will the user use it?"
  - Tells a story: "What does this system do for me?"
  - Helps to use a standard template
- SRS: formal, technical requirements for development team
  - Purpose is to answer specific technical questions about the requirements quickly and precisely
  - Precise, unambiguous, complete, and consistent as practical

Scenario Analysis and Use Cases

- Common user-centered analysis method
- Requirements Elicitation
  - Identify stakeholders who interact with the system
  - Collect "user stories" - how people would interact with the system to perform specific tasks
- Requirements Specification
  - Record as use-cases with standard format
  - Use templates to standardize, drive elicitation
- Requirements verification and validation
  - Review use-cases for consistency, completeness, user acceptance
  - Apply to support prototyping
  - Verify against code (e.g., use-case based testing)

1 Brief Description

This use case describes how the Bank Customer uses the ATM to withdraw money to his/her bank account.

2 Actors

2.1 Bank Customer
2.2 Bank

3 Preconditions

There is an active network connection to the Bank.
The ATM has cash available.

4 Basic Flow of Events

1. The use case begins when Bank Customer inserts their Bank Card.
2. Use Case: Validate User is performed.
3. The ATM displays the different alternatives that are available on this unit. [See Supporting Requirement SR-xxx for list of alternatives]. In this case the Bank Customer always selects "Withdraw Cash".
4. The ATM prompts for an account. See Supporting Requirement SR-yyy for account types that shall be supported.
5. The Bank Customer selects an account.
6. The ATM prompts for an amount.
7. The Bank Customer enters an amount.
8. Card ID, PIN, amount and account is sent to Bank as a transaction. The Bank Consortium replies with a go/no go reply telling if the transaction is ok.
9. Then money is dispensed.
10. The Bank Card is returned.
11. The receipt is printed.

5 Alternative Flows

5.2 Wrong account

If in step 8 of the basic flow the account selected by the Bank Customer is not associated with this bank card, then

1. The ATM shall display the message "Invalid Account – please try again".

2. The use case resumes at step 4.

Example Use Case

- Avoids design decisions
- References other use cases
- References more precise definitions where necessary
- Some terms need further definition (e.g. PIN)

31

Benefits and Drawbacks

- Use cases can be an effective tool for:
  - Eliciting user-group's functional requirements
  - Communicating to non-technical stakeholders
  - Creating initial test cases
  - Verifying expected behavior
- Generally inadequate for detailed technical requirements
  - Difficult to find specific requirements
  - Inherently ambiguous and imprecise
  - Cannot establish completeness or consistency
- True of all informal specification methods

Technical Specification

The SRS

The role of rigorous specification

Requirements Documentation

- Is a detailed requirements specification necessary?
- How do we know what "correct" means?
  - How do we decide exactly what capabilities the modules should provide?
  - How do we know which test cases to write and how to interpret the results?
  - How do we know when we are done implementing?
  - How do we know if we've built what the customer asked for (may be distinct from "want" or "need")?
  - Etc…
- Correctness is a *relation* between a spec and an implementation (M. Young)
  - Implication: until you have a spec, you have no standard for "correctness"

Technical Requirements

- Focus on developing a technical specification
  - Should be straight-forward to determine acceptable inputs and outputs
  - Can systematically check completeness consistency
- Provides
  - Detailed specification of precisely what to build
  - Design-to specification
  - Build-to specification for coders
  - Characterizes expected outputs for testers
- Little application in Project 1

Quality Requirements

Quality Requirement Types

- Avoid “functional” and non-functional" classification
- Behavioral Requirements – any requirements or constraints on the system's run-time behavior
  - Measurable qualities (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 and Developmental Requirements

Behavioral (observable)	**Developmental Qualities**
• Performance • Security • Availability • Reliability • Usability	• Modifiability(ease of change) • Portability • Reusability • Ease of integration • Understandability • Support concurrent development
Properties resulting from the behavior of components, connectors and interfaces that exist at run time.	Properties resulting from the structure of components, connectors and interfaces that exist at design time *whether or not they have any distinct run-time manifestation*.

Importance

- Quality requirements are as or more important to user acceptance than functional
  - Every system has critical quality requirements
  - The most frequent reason for user dissatisfaction
- Quality requirements are often implicit or assumed
  - E.g., response time, data integrity
- Must be explicit to be controlled
  - Implicit requirements cannot be communicated, tracked, verified, etc.
  - Left out at crunch time

Specifying Quality Requirements

- When using natural language, write *objectively verifiable* requirements when possible
  - Load handling: The system will support a minimum of 15 concurrent users while staying with required performance bounds.
  - 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.”

Requirements Validation and Verification

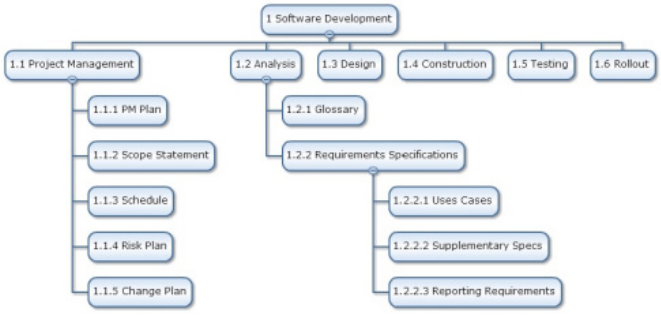
- Feedback-control for requirements
- Should answer two distinct questions:
  - Validation: “Are we building to the right requirements?”
  - Verification: “Are we building what we specified?”
- Validation requires going back to the stakeholders: can use many techniques
  - Review of specifications
  - Prototyping, software review
  - Use case walkthroughs
- Verification requires checking work products against specifications
  - Review
  - Testing
  - Formal modeling and analysis

Real meaning of “control”

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

End

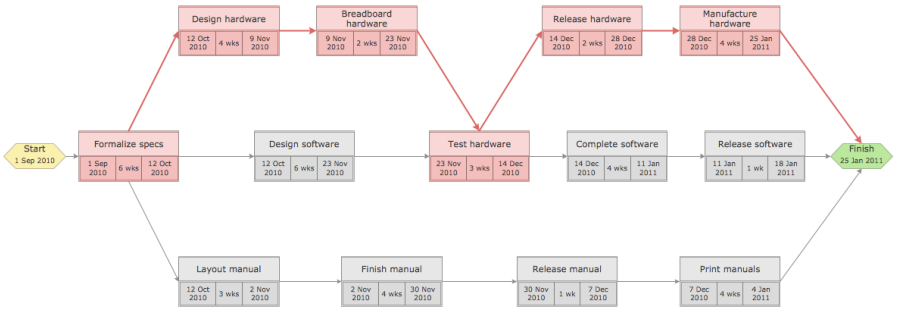
Work Breakdown Structure



1. Software Development
    1. Project Management
    2. Analysis
        1. Glossary
        2. Requirements Specification
            1. Use Cases
            2. Supplementary Specs…

Equivalent list format

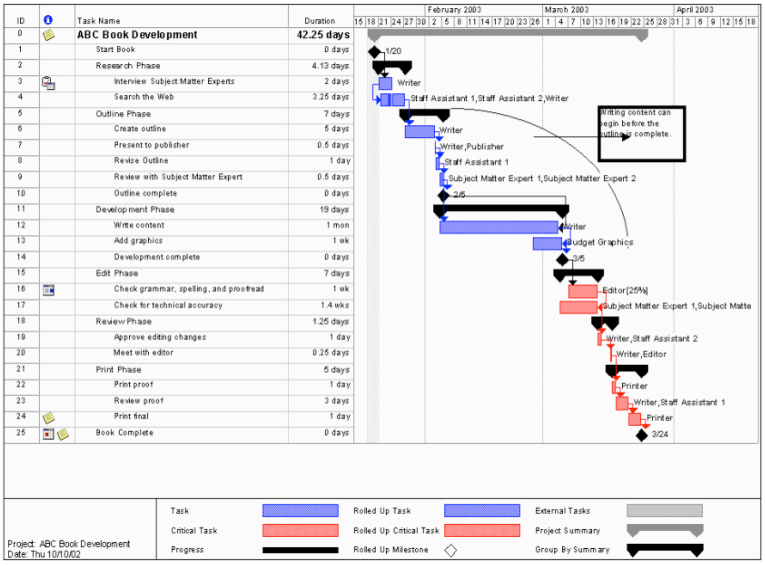
PERT Chart Example



- Which tasks can we start on?
- Which tasks can be done concurrently?
- Which tasks depend on which other tasks?
- *Critical Path*: which path has the longest duration?
  - Gives minimum length of project

http://www.conceptdraw.com/samples/project-chart

Example Gantt Chart



Teamwork and Group Dynamics

What do software developers do?

- Most time is not spent coding
- So how do they spend their time?
- IBM study (McCue, 1978):
  - 50% team interactions
  - 30% working alone (coding & related)
  - 20% not directly productive

-Technical excellence is not enough

Must understand how to work effectively in teams

Being a Good Team Member

- Attributes most valued by other team members
  - Dependability
    - When you say you'll do something, you do it
    - Correctly
    - On time
  - Carrying your own weight (doing a fair share of the work)
- People will overlook almost everything else if you do these

Consensus decision making

- Consensus is not counting votes
  - Democracy is 51% agreement
  - Unanimity is 100% agreement
- Consensus is neither
  - Everyone has their say
  - Everyone accepts the decision, even if they don't prefer it
  - It is "buying in" by group as a whole, including those who disagree
- Usually best approach for peer groups

Consensus takes time and work, but is worthwhile

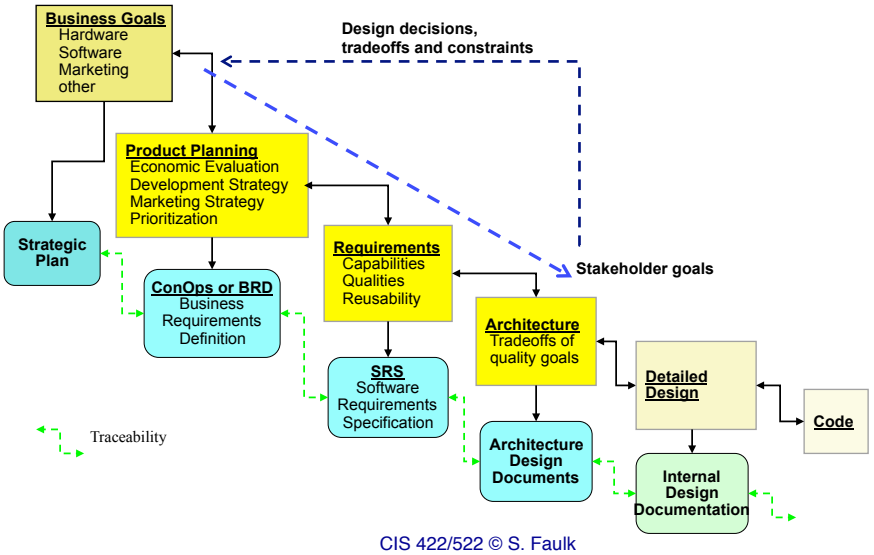
CIS 422/522
2nd Half Concept Review

Stuart Faulk

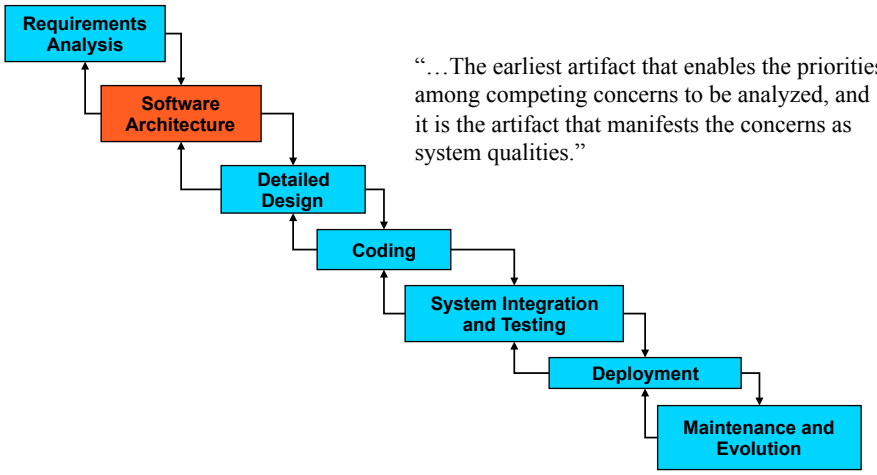
View of SE in this Course

- 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)

The Architectural Business Cycle



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

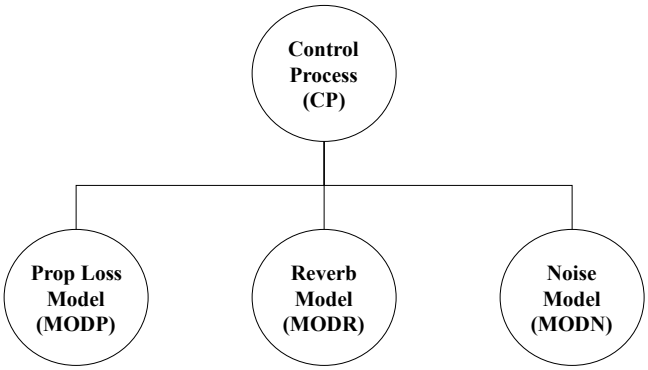
- Systems typically comprise more than one architecture
  - There is more than one useful decomposition into components and relationships
  - Each addresses different system properties or design goals
- It exists whether any thought goes into it or not!
  - Decisions are necessarily made if only implicitly
  - Control issue is who makes them and when
  - Being in control implies having the right person make each decision at the appropriate time

Examples: These are architectures

- **An architecture comprises a set of**
  - **Software components**
  - **Component interfaces**
  - **Relationships among them**
- **Partial 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
- Their interests 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

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?
- Architectural Structures
  - 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…

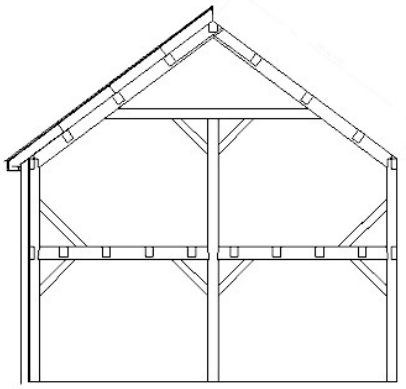
- Design Goals: the purpose of design is to solve some problem in a context of assumptions and constraints
  - Assumptions: what must be true of the design
  - Constraints: what should not be true
- Process: design proceeds through a sequence of decisions
  - A *good* decision brings us closer to the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

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

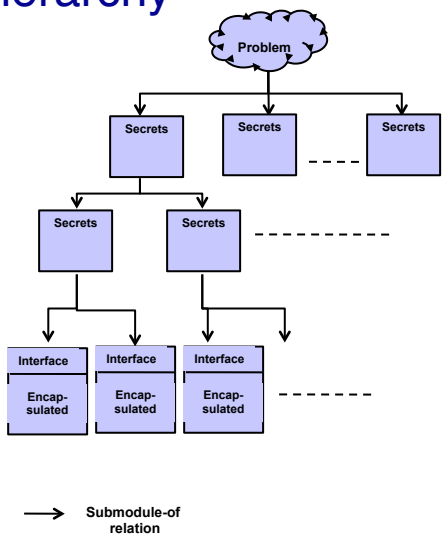
Example: Designing the Module Structure

Modularization

- For large, complex software, must divide the development into work assignments (WBS). Work assignments can focus on “modules.”
- 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

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

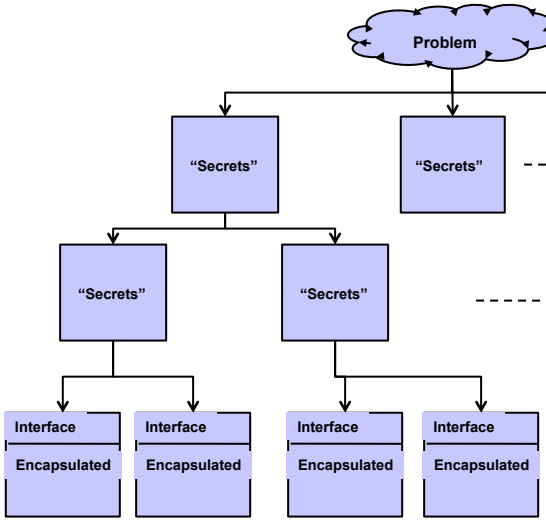
Decomposition Strategies Differ

- 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)
  - Put design decisions likely to change together in the same module
  - Put design decisions likely to change independently in different modules
- 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 to re-write easily
- 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**

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)

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

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