Midterm Review

Overview

- The “software crisis”
- What is “software engineering” - course view
- The software life cycle – controlling the process
  - Faking it
  - Common process models addressing different risks
- Project planning and management – controlling the resources
- Documentation – communicating decisions
  - Product specification
  - SRS

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
- Characterized by **lack of *control*** (inability plan the work, work the plan)

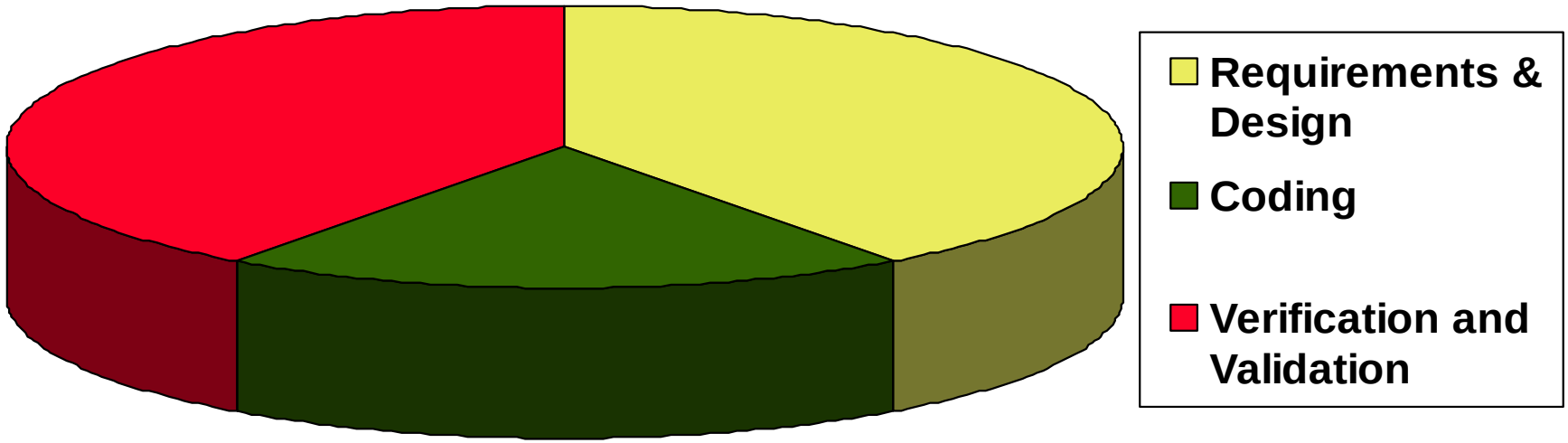
Symptoms of the Crisis

- Two of every eight large software project is cancelled
- Average projects overshoot schedule by 50%, large project do much worse
- 75% of large systems are failures in the sense that they do not operate as intended
- 60% of them fail to deliver a single working line of code

Large is Different

- Large system development *quantitatively* and *qualitatively* distinct from small development
- 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 fail utterly when scaled up
  - Programming alone won't get you through real developments or even this course

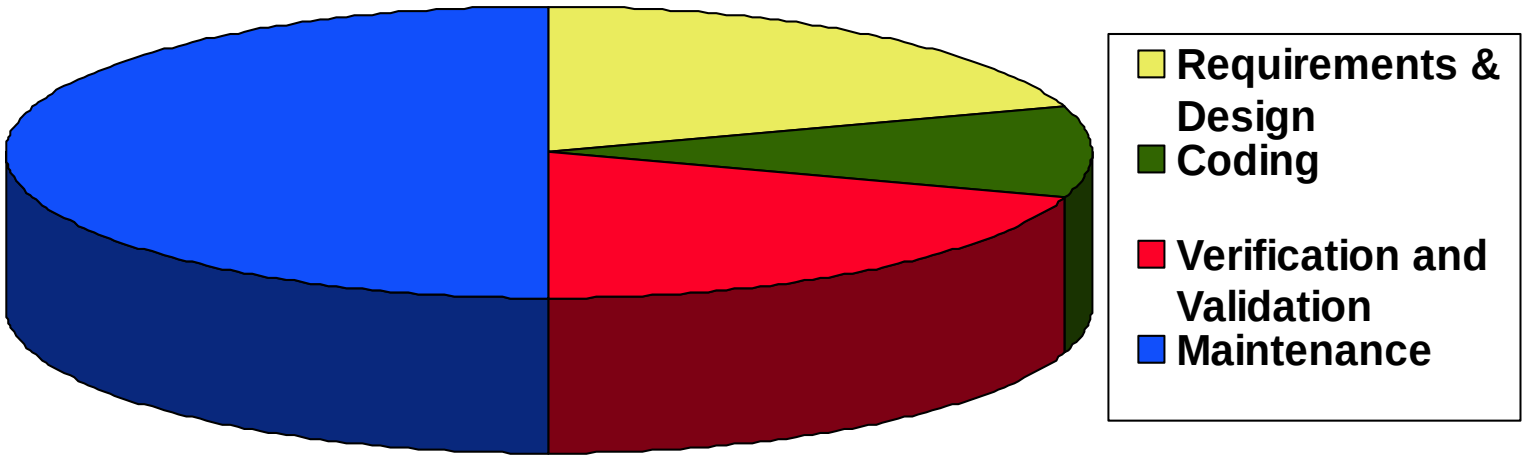
40-20-40 Rule



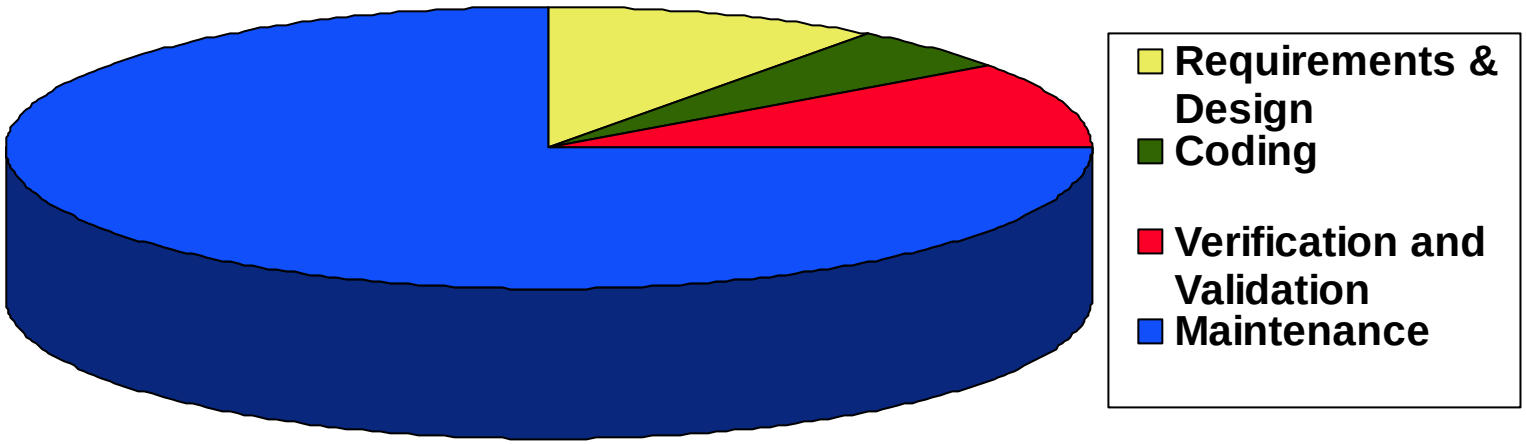
Global distribution of effort

- Rule of thumb: 40-20-40 distribution of effort in development
- But: development is less than half the story
- For real systems maintenance alone consumes 50-75% of total effort
  - Coding < 10%

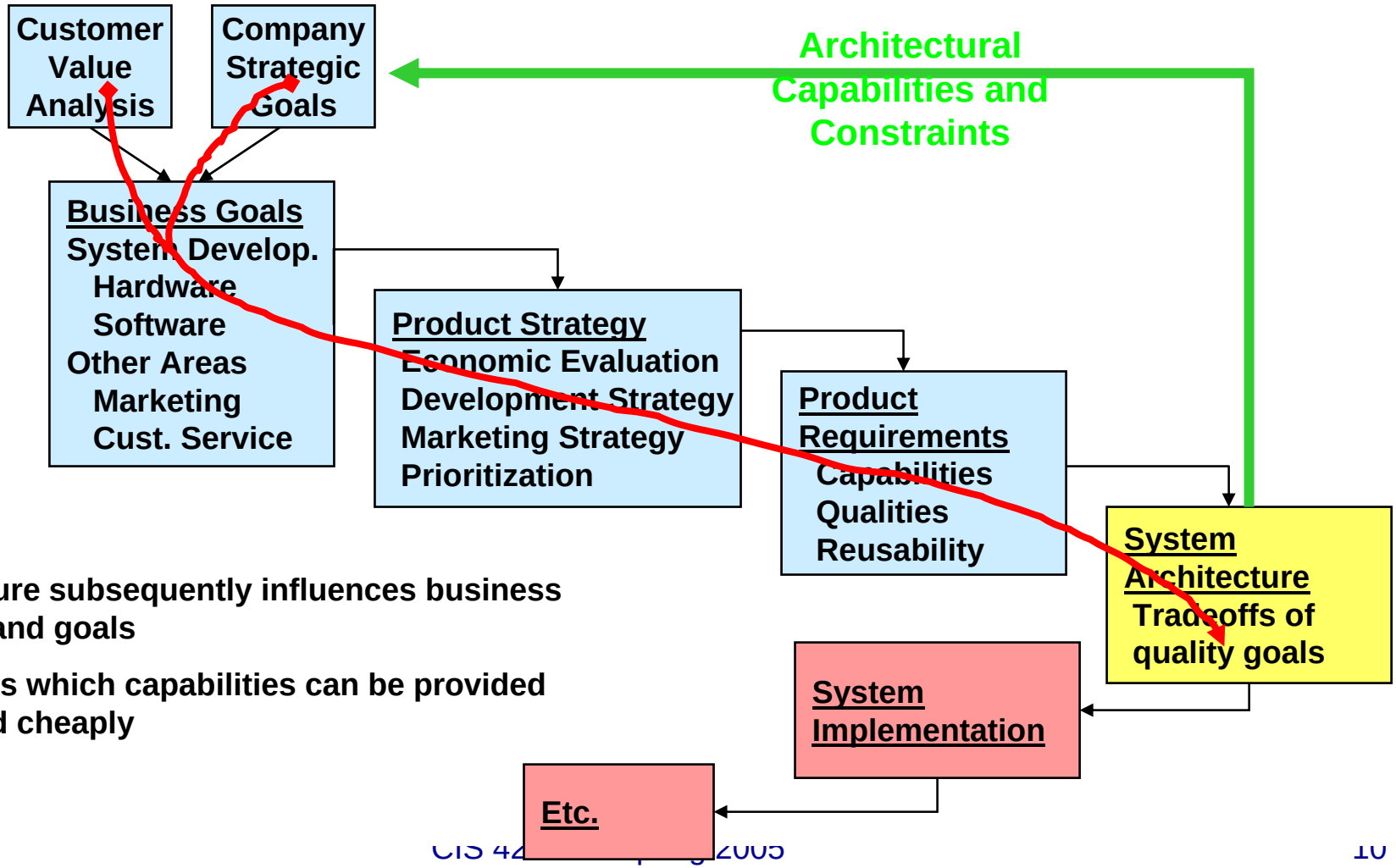
Life-Cycle View



Life-Cycle View



Must Consider Whole Cycle



- Architecture subsequently influences business decisions and goals
- Determines which capabilities can be provided quickly and cheaply

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).

Meaning of “Control”

- Both are necessary for success!
- Intellectual control implies
  - We understand what we are trying to achieve
  - Can distinguish good choices from bad
  - We can reliably and predictably achieve what we want
- Managerial control implies
  - We make accurate estimations
  - We deliver on schedule and within budget
- Assertion: Managerial control is not really possible without intellectual control

Course Approach

- Will learn methods for acquiring and maintaining control of software projects
- Managerial control (most of focus to date)
  - Planning and controlling development
  - Process models addressing development issues (e.g. risk, time to market)
  - People management and team organization
- Intellectual control
  - Methods for software requirements, architecture, design, test
  - Notations, verification & validation

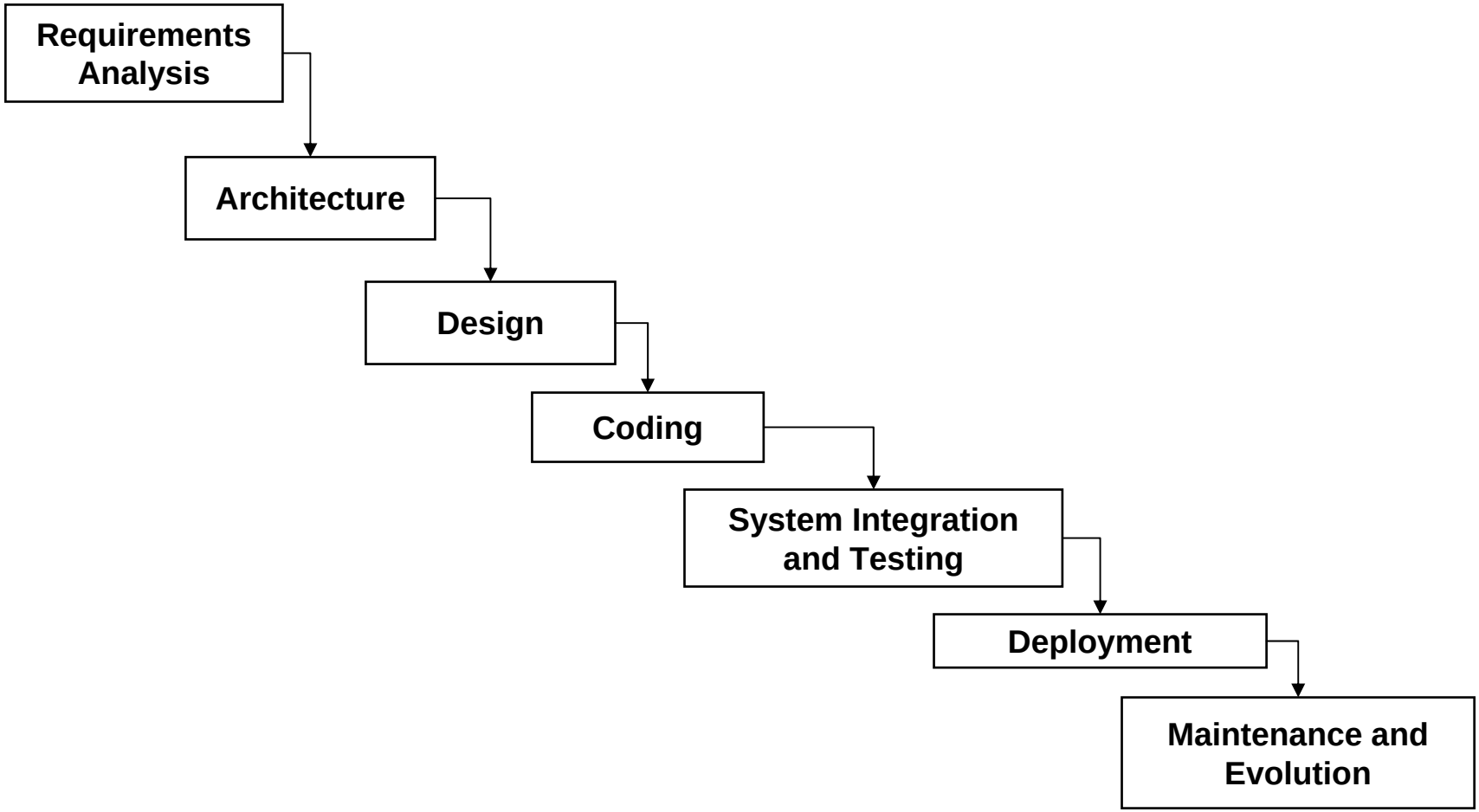
The Software Lifecycle

Introduction

Definition

- Software Life Cycle: evolution of a software development effort from concept to retirement
- *Life Cycle Model*: Abstract representation of a software life cycle as a sequence of 1) activities or phases and 2) products (usually graphic)
- *Software Process* (process model): institutionalized version of a life cycle model. Usually intended to provide guidance to developers.

A “Waterfall” Model



Issues with Life Cycle Models

- Application of “divide-and-conquer” to software processes and products
  - Goal: identify distinct and relatively independent phases and products
  - Can then address each 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*.

It Pays to “Fake it”

- Assertion: Design is an inherently “irrational” process
- Thesis: It is nonetheless useful to “fake” a rational design process
  - Follow the ideal process as closely as possible
  - Write the documentation and other work products as is we had followed the ideal
- Rationale
  - Idealized process can provide guidance
  - Helps come closer to the ideal (emulation)
  - Helps standardize the process (provide a common view of how to proceed and what to produce)
  - Provides a yardstick for assessing progress
  - Provides better products (e.g. final draft not first)

Contents of a Process Specification

- Details depend on the purpose of the specification
- In general terms [Parnas &Clements]
  - What product we should work on next
    - Equivalently – what decision(s) must we make next
  - What kind of person should do the work
  - What information is needed to do the work
  - When is the work finished?
  - What criteria the work product must satisfy

Process Specification (2)

- One of the project lessons-learned is that a vague description is inadequate
  - “Build a module to do XXX.” vs. “Build a module that correctly passes the following test set.”
  - Unless answers to these questions are specific these questions cannot be answered satisfactorily
  - E.g., carefully defined interfaces, clear user documents, testable requirements, test cases

Required Document Types

- Need two types of documents
- Management documents
  - Project plan, schedule, WBS, etc.
  - Tools for managerial control: i.e., *control of resources*
- Development documents
  - ConOps, Requirements (SRS), Architecture, Detail design, etc.
  - Tools for intellectual control: i.e., *control of product produced (functionality, properties)*

Common Process Models

Prototyping

Iterative

RAD or Xtreme

Spiral

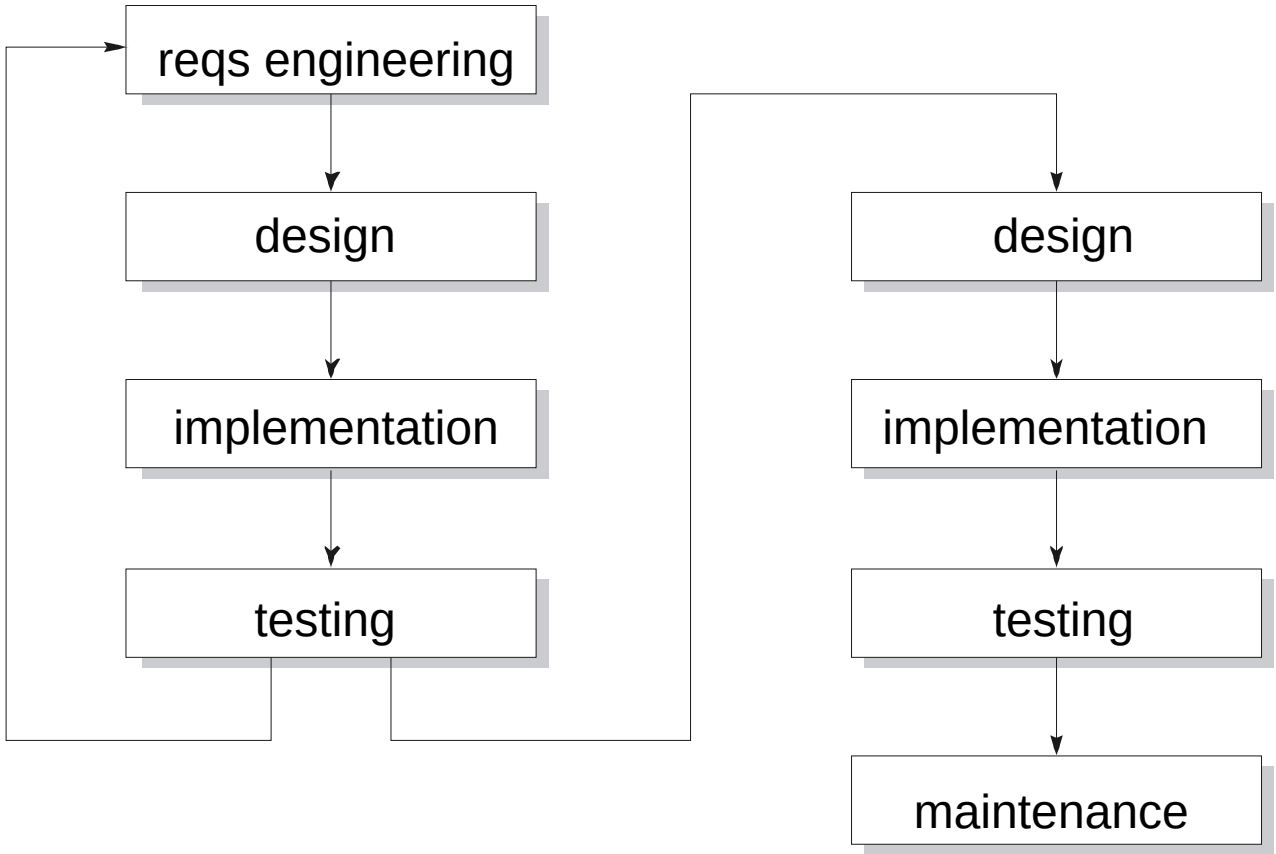
“Appropriate” Control

- Goal: control appropriate to the product and development context
- What constitutes “appropriate” control will be *vastly* different for different types of developments
  - Large vs. small
  - New problems vs. old
  - Time to market vs. quality
  - These are neither independent nor exclusive
- Development approaches vary in their assumptions about these issues
  - Useful to view in terms of which risk area they address
  - E.g., RAD vs. Spiral vs. Prototyping

I. Prototyping

- Traditionally used to address two distinct risk issues
  - *Requirements*: problem that the user's don't know what they want until they see it
  - *Technical feasibility*: technical unknowns or technical risk in development
- Two types of prototypes
  - Demonstration: a concrete (visible) realization of some user need. May or may not provide real functionality (e.g., a mock-up of user interface)
    - Answers the question: "Is this what we should build?"
  - Engineering: a part of a working system sufficient to demonstrate the feasibility of meeting some requirement
    - Answers the question: "Can we build it using technology T?"

Prototyping as a tool for requirements understanding



Adapted from van Vliet © 2001 with permission

Prototyping Recommendations

- Use prototyping when the requirements are unclear or there are major technical risk areas
- Prototyping needs to be planned and controlled as well
  - Tendency to become the system
  - Explicit definition of system qualities
  - Explicit control of how they will be achieved
  - Prototype never defaults to the delivered system

II. Incremental Development

- A software system is delivered in small increments of increasing capability
  - Avoids the Big Bang effect (nothing works until system integration at the end)
  - There's always a working system
- The steps of the waterfall model may be employed in each phase (or variations)
- The customer is closely involved in directing the next steps

Incremental Development

- Requires careful attention to architectural design (I.e., how the system is decomposed into components)
  - The sequence of increments (useful subsets) must be planned in advance
  - Dependencies between components must be understood and mapped out
    - Avoid circular dependencies
    - Make sure capabilities are present when needed for the next increment

III. RAD: Rapid Application Development

- Incremental development with **time boxes**: fixed time frames within which activities are done
  - Time frame is decided upon first, then one tries to realize as much as possible within that time frame
- Close customer collaboration
  - Joint Requirements Planning (JRD) and
  - Joint Application Design (JAD),
- Requirements prioritization through a *triage*;
- “Xtreme Programming” is a variation on this theme

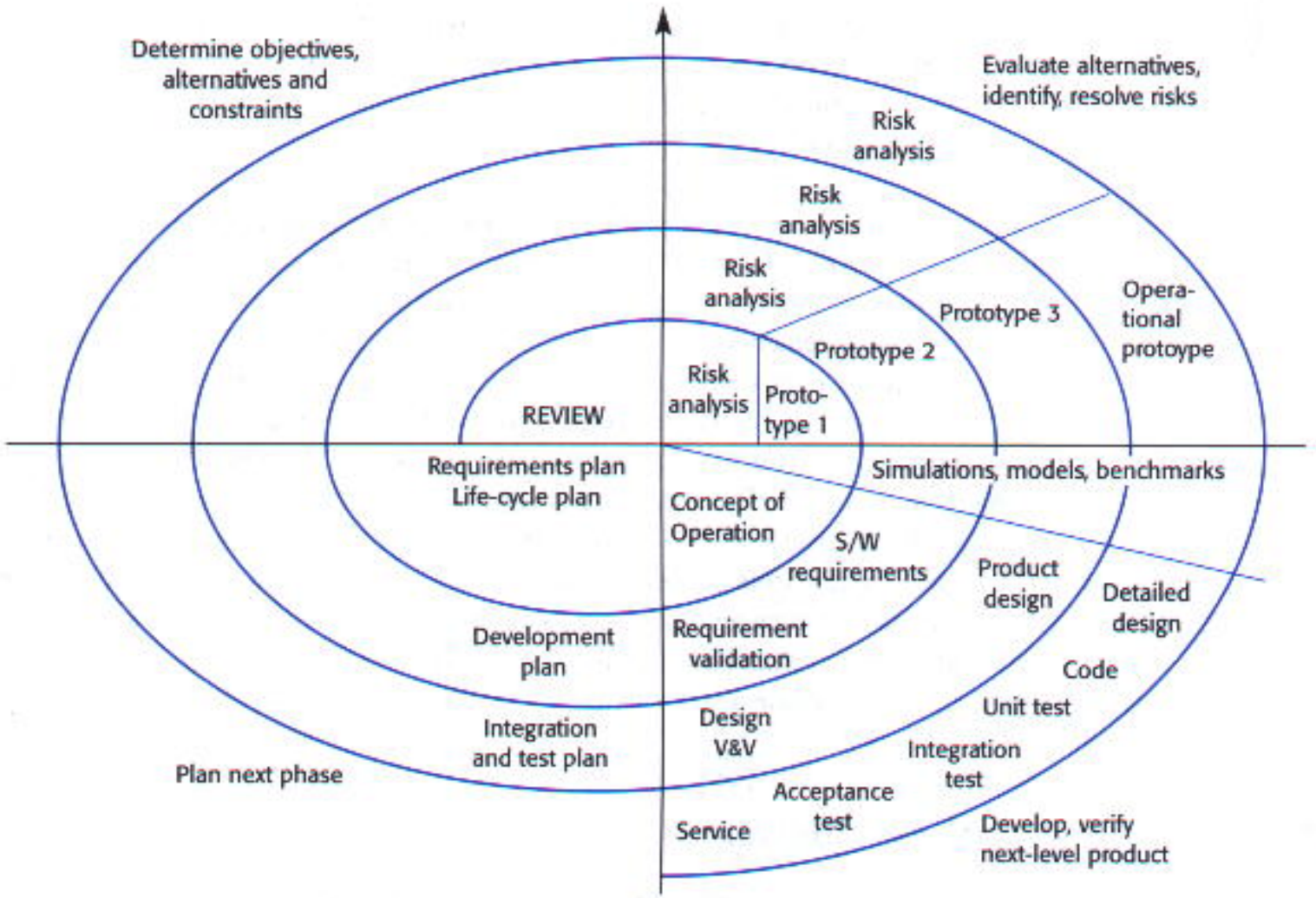
RAD: Rapid Application Development

- Must be able to sacrifice functionality for schedule
- Requires, close, rapid communication cycles between developers and with stakeholders
- Best suited for small team development and modestly sized projects

IV. Spiral Model

- All development models have something in common: reducing the risks
  - in prototyping, getting the right requirements is a major risk
  - in the waterfall model, the schedule is seen as a risk
- The **spiral model** subsumes these different models
  - I.e., *the model can be used to address any or all of the risks by continually revisiting any perceived risk issues.*

Spiral Process Model (Boehm)



Spiral Model Goals

- Response lack of risk analysis and risk mitigation in “waterfall” process
  - Make risk analysis standard part of process
  - Address risk issues early and often
- Explicit risk analysis at each phase
- Framework for explicit risk-mitigation strategies
  - E.g., prototyping (what risk/difficulty is addressed?)
- Explicit Go/No-Go decision points in process

How do we Choose a Development Process?

E.g., for your projects

Goals vs. Risks

- Goal: proceed as rationally and systematically as possible (I.e., in a controlled manner) from a statement of goals to a design that demonstrably meets those goals with design and management constraints
  - Understand that any process description is an abstraction
  - Always must compensate for deviation from the ideal (e.g., by iteration)
- Risk: Anything that might lead to a loss of control is a project **risk**
  - E.g., won't meet the schedule, will overspend budget, will fail to deliver the proper functionality

A Software Engineering Perspective

- Choose processes, methods, notations, etc. to provide *an appropriate level of control* for the given *product* and *context*
  - Sufficient control to achieve results
  - No more than necessary to contain cost and effort
- Provides a basis for choosing or evaluating processes, methods, etc.
  - Does it achieve our objectives at reasonable cost?

Project Relevance

- Need to agree on kind of control you need to address most serious risks, and how you will accomplish that control
- Process model (description) will then help keep everyone on track
  - Basis for planning and scheduling
  - Each person knows what to do next
  - Basis for tracking progress against schedule

Example

- Project 1 assumptions
  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
- 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 & docs.)
  - Technology risk requires building/finding software and trying it (prototyping)
  - Most forms of incremental development will address these

Project Planning and Management

Methods and Tools for Resource Control

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.
  1. "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).
  2. Managerial control means we control development *resources* (budget, schedule, personnel).

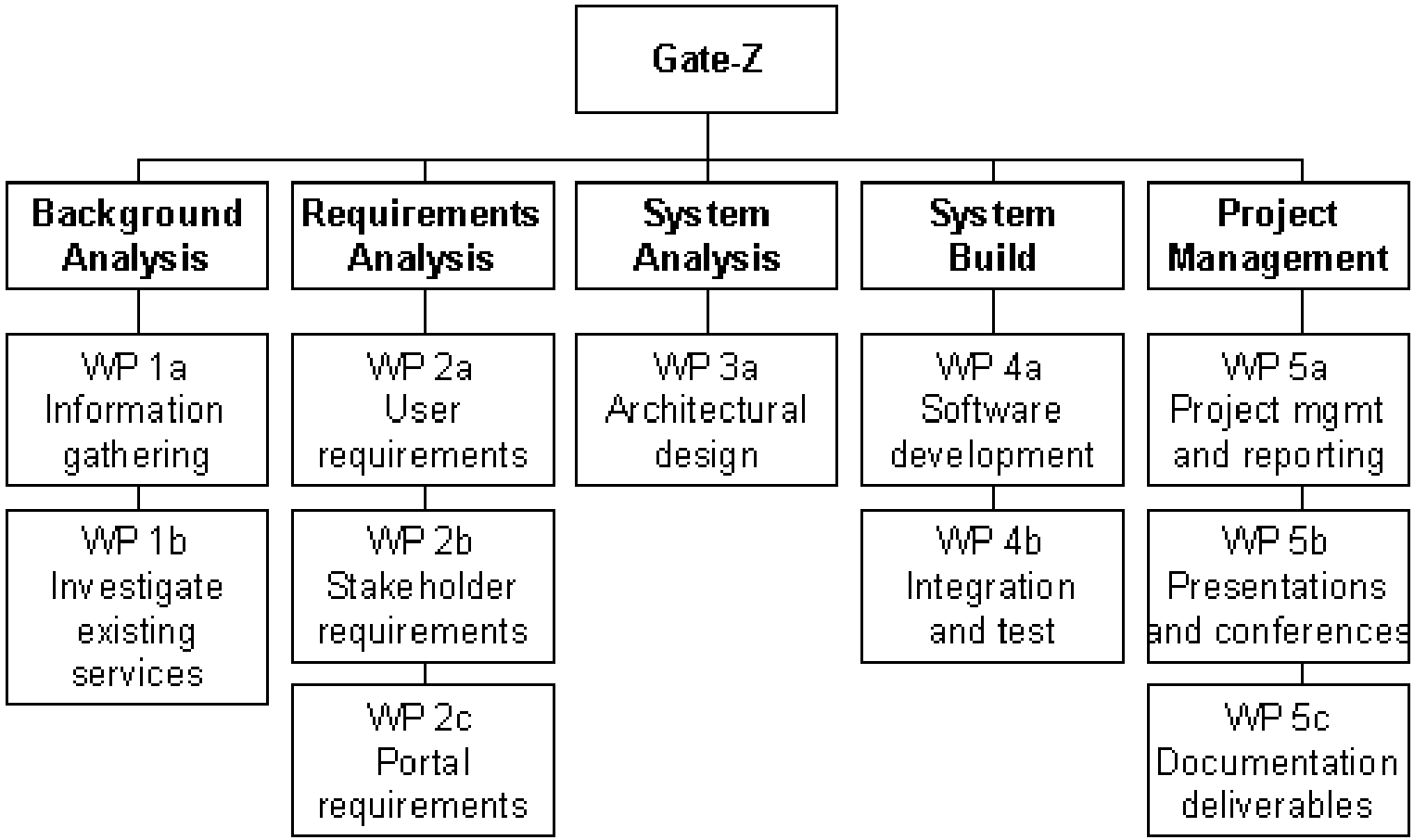
Lessons Learned

- Observation 1: Most projects underestimated technical difficulty hence did not have full “intellectual control” (the consequences of technical decisions were not always clear)
- Observation 2: Lack of intellectual control sometimes disrupted managerial control
  - E.g. Failure to fully understand requirements, inadequate understanding of technology, vague definition of interfaces
  - End up affecting schedule, level of effort, delivered functionality

Work Breakdown Structure

- This is a technique to analyze the content of work and cost by breaking it down into its component parts. It is produced by :-
  - Identifying the key elements
  - Breaking each element down into component parts
  - Continuing to breakdown until manageable work packages have been identified and allocated to the appropriate person
- The WBS helps identify distinct tasks for allocation and scheduling

Work Breakdown Structure



Milestone Planning

- Milestone planning is used to show the major steps that are needed to reach the goal on time
- Milestones typically mark completion of key deliverables (e.g., completion of WBS task)
- Often associated with management review points
- E.g., Requirements baseline, project plan complete, code ready to test

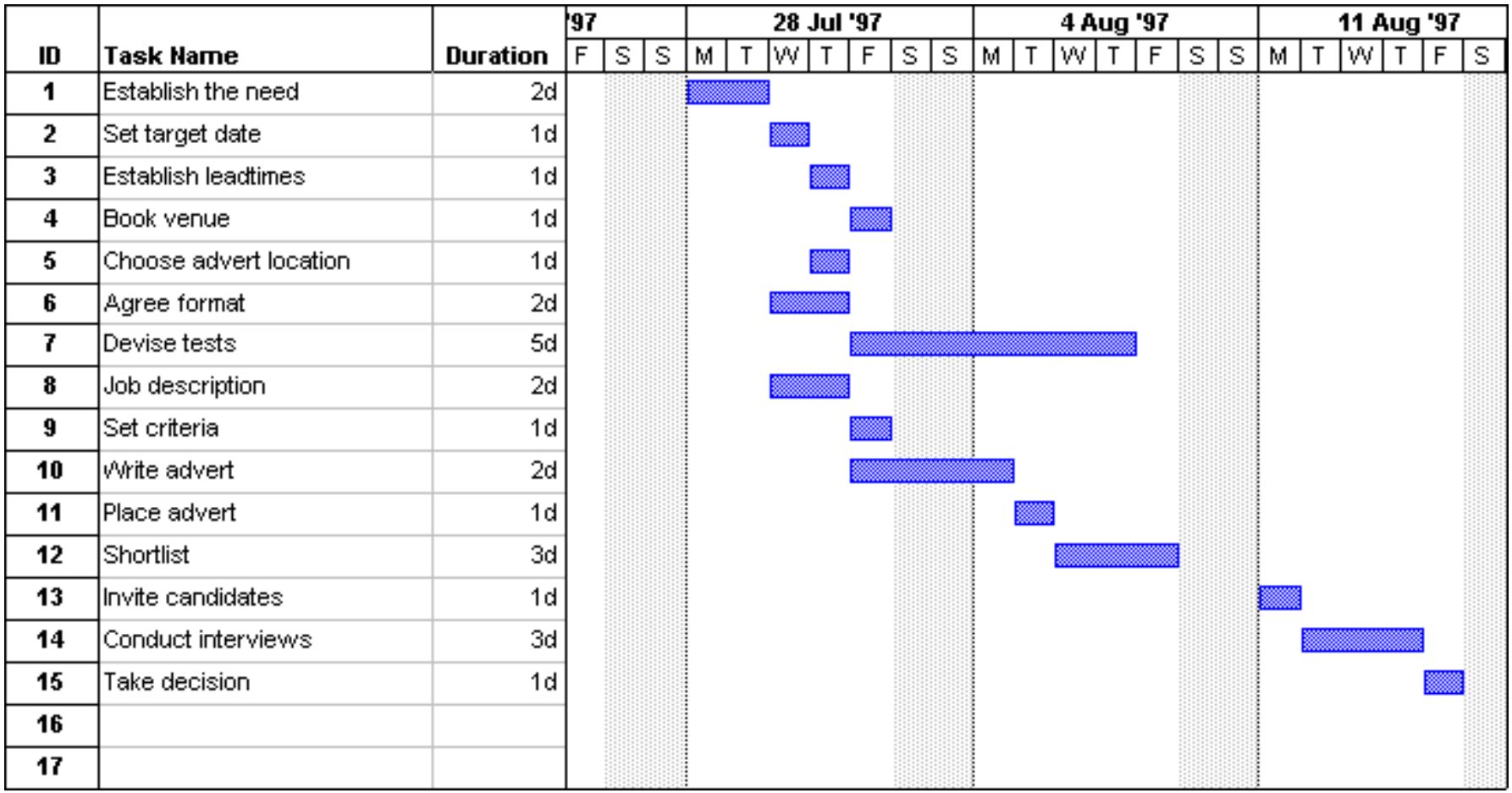
Pert Chart

- Network analysis or PERT is used to analyze the inter-relationships between the tasks identified by the work breakdown structure and to define the dependencies of each task
- Helps identify where ordering of tasks may cause problems because of precedence or resource constraints
  - Where one person cannot do two tasks at the same time
  - Where adding a person can allow tasks to be done in parallel, shortening the project
- Helps control allocation of resources over time

Gantt Charts

- Method for visualizing a project schedule showing
  - The set of tasks
  - Start and completion times
  - Task dependencies
  - May include responsibilities
- PERT charts can be reformatted as Gantt charts
- Control of people and time against tasks

Example Gantt Chart



ID	Task Name	Duration
1	Establish the need	2d
2	Set target date	1d
3	Establish leadtimes	1d
4	Book venue	1d
5	Choose advert location	1d
6	Agree format	2d
7	Devise tests	5d
8	Job description	2d
9	Set criteria	1d
10	Write advert	2d
11	Place advert	1d
12	Shortlist	3d
13	Invite candidates	1d
14	Conduct interviews	3d
15	Take decision	1d
16		
17		

http://www.spottydog.u-net.com/guides/faq/faq.html

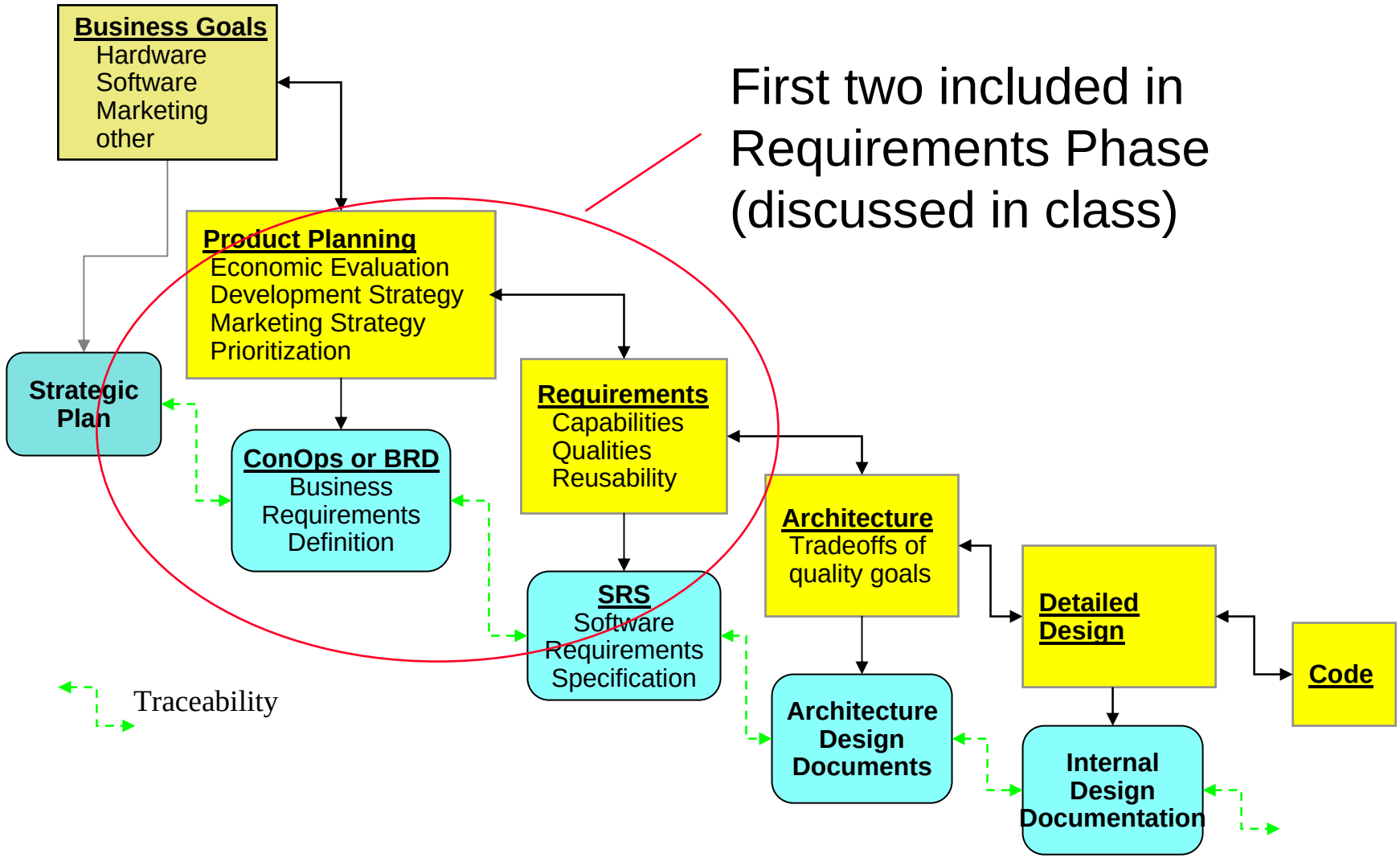
Lessons Learned

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

Development Documents

Fallout from Faking It

Product Development Cycle



Requirements Phase Goals

What are the goals of the requirements phase?

- Standard definition: “Establish and specify precisely what the software must do without describing how to do it.”
- *Establish* what to build be for starting to build it.
  - Make the “what decisions” explicitly before starting design … not implicitly during design.
  - Make sure you build the system that’s actually wanted/needed.
  - Allow the users a chance to comment before it’s built.
- *Specify* in terms of an organized reference document - the Software Requirements Specification (SRS)
  - Communicate the results of analysis
  - Provide a baseline reference document for developers and QA

Parts of the Requirements Phase

- *Problem Analysis*
  - The (“establish” part) also called “problem understanding, requirements exploration, etc
  - Goal is to understand precisely what problem is to be solved
  - Includes: Identify exact system purpose, who will use it, the constraints on acceptable solutions, and possible tradeoffs among conflicting constraints.
- *Requirements Specification*
  - Goal is to develop a technical specification - the Software Requirements Specification (SRS)
  - SRS specifies exactly what is to be built, captures results of problem analysis, and characterizes the set of acceptable solutions to the problem.

Two Kinds of Software Requirements

- Concept of Operations: documents user needs and customer expectations
  - Link to organizational goals
  - Stated in terms the the users' / customer's can understand
- Technical Specifications: a concise and unambiguous statement of technical parameters for a system that will satisfy the operational requirements
  - Stated in the developers' terminology
  - Covers issues such as performance, interfaces, safety, security, and reliability

For Your Project

- Apply Use Cases (scenarios) to describe the system's mission from the user's point of view
  - Answers the questions, "What is the system for?" and "How will the user use it?"
  - Tells a story
- For the "Functional Requirements" be as rigorous as possible
  - Use tables, bullets, or case-by-case behavior description
  - Purpose is to answer questions about the requirements quickly and precisely
    - Answers, "What should the system output in this circumstance?"
    - Reference, not a narrative, does not "tell a story"

Use Case Contents (Generic)

- Use case identifier
- Summary – summary of use case
- Actors – roles enacting use case
- Scenarios
  - Basic scenario – the normal case
  - Alternative scenarios – other ways to reach goal
  - Exceptions – problem scenarios
- Trigger – what causes the use case to start
- Assumptions
- Preconditions – what must be true before the interaction can occur?
- Post conditions – what must be true after the interaction occurs

Requirements Difficulties

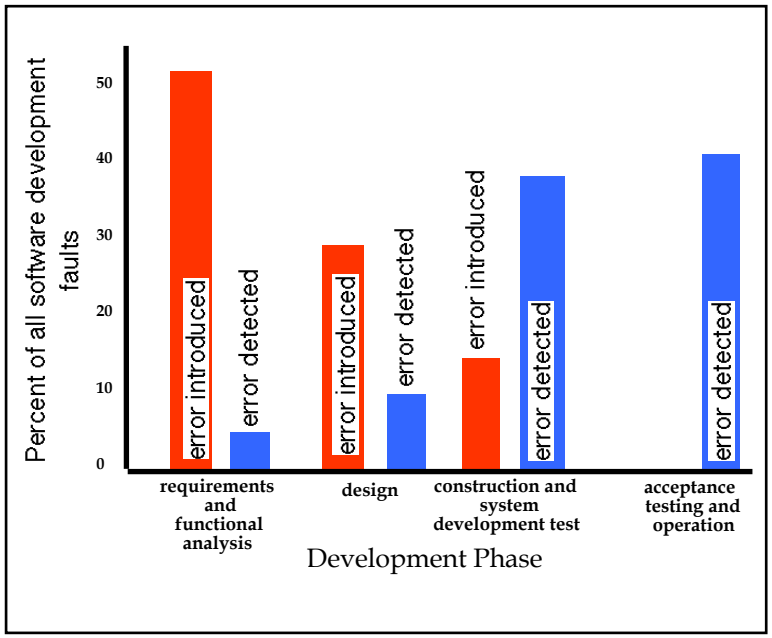
The Software Requirements Specification (SRS)

“The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements...No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later.”

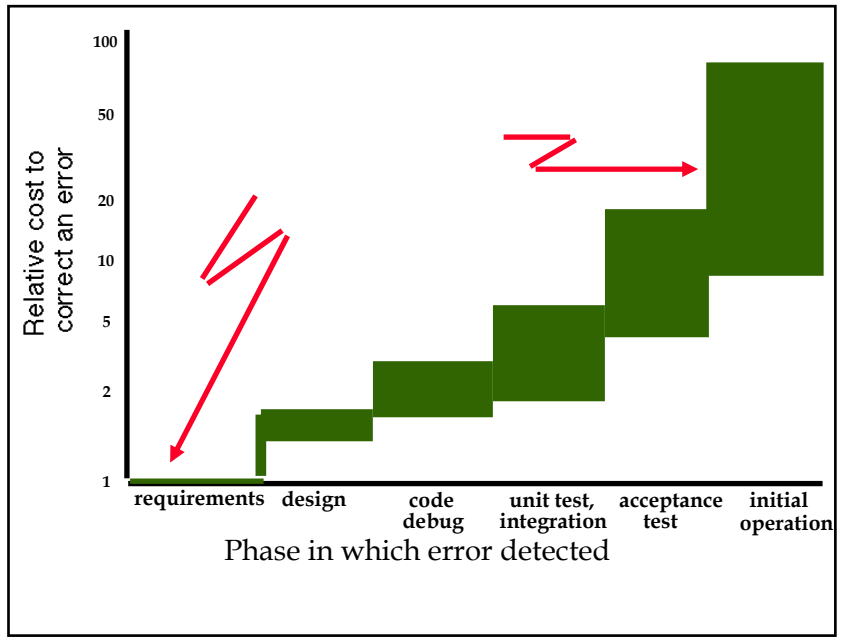
F.P. Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering”

Distribution and Effects of Requirements Errors

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 Goal

- Standard definition:
  "Identify and specify precisely what the software must do without describing how to do it."
  - Idiomatically: Specify "what" without specifying "how."

What's so hard about that?

Overall Goals of Requirements

- Only three goals (given the large development context)
  - 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
- Problems in requirements result from the failure to adequately accomplish one of these goals
- In practice, difficult to accomplish

Essential vs. Accidental

- Thesis:

  Meeting the requirements phase goals is
  essentially difficult
  - but not as difficult as we make it!

- Essential difficulties - part of the essence of the problem
- Accidental difficulties - difficulties introduced or added by the way we do things

*(not “essential” as in “needed” nor “accidental” as in “unintended”)

Essential Difficulties

- Comprehension (understanding)
  - People don't (really) know what they want (… until they see it)
  - Superficial grasp is insufficient
- Communication
  - People work (think) best with regular structures, conceptual coherence, and visualization
  - Software's conceptual structures are complex, arbitrary, and difficult to visualize
  - Requirements specification has many purposes and audiences

Essential Difficulties (2)

- Control (decideability, predictability, manageability)
  - Control => ability to plan the work, work to the plan
  - Changes, lack of visibility make control difficult
- Inseparable Concerns
  - Separation of concerns - ability to divide a problem into distinct and relatively independent parts
  - Many issues in requirements cannot be cleanly separated
  - Difficult to apply “divide and conquer”
  - Must make tradeoffs

Accidental Difficulties

- Written as an afterthought: common practice to write requirements after the code is done (if at all)
  - Inevitably a specification of the code as written
  - Designers and coders end up defining requirements
- Confused in purpose
  - Authors fail to decide precisely what purposes the SRS will serve and in what way it will serve them
  - Requirements end up mixed with other things
  - Fails to do anything well

Accidental Difficulties

- Not designed to be useful
  - Often little effort is expended on the SRS - and it shows
  - Resulting document of limited usefulness
- Lacking essential properties
  - Must have certain properties: Completeness,Consistency, Ease of change, Precision...
  - Accidental difficulties lead to document that is redundant, inconsistent, unreadable, ambiguous, imprecise, inaccurate.
- End up not being useful useful, not used or maintained

Role of a Disciplined Approach

- Meaning of “disciplined”
  - Objectives are known in advance
    - What’s the document’s purpose?
    - Who are in its audience?
  - Product is well defined and designed to purpose
  - Process is defined, followed, tracked and managed
- Disciplined approach will address the accidental difficulties
- Provides a basis for attacking essential difficulties

Address Accidental Difficulties

- Written as an afterthought
  - Plan for and budget for the requirements phase and its products
  - Specification is carefully written, checked, and maintained
- Confused in purpose
  - Document purpose is defined in advance (who will use it and what for)
  - Plan is developed around the objectives
- Not designed to be useful
  - Specification document itself is designed to facilitate key activities
    - document users (e.g., answer specific questions quickly and easily)
    - document producers (e.g., ability to check for properties like consistency)
  - Designed to best satisfy its purpose given schedule and budget constraints
- Lacking essential properties
  - Properties are planned for then built in
  - Properties are verified by the best means possible (review, automated checking, etc.)

Questions?