Managing Software Requirements 2nd ed.

A Use Case Approach

Dr. Arun K. Datta
National University

Text by

Dean Leffingwell & Don Widrig

Introduction

Chapter 1: The Requirements Problem
Chapter 2: Introduction to Requirements Management
Chapter 3: Requirements and the Software Lifecycle
Chapter 4: The Software Team

Chapter 1

The Requirements Problem

The Requirements Problem

The Goal is to develop quality Software, on time and on budget, that meets customers' real needs.

The Data

- IT Application Development
  - $250 billion spent annually
  - 175,000 projects annually
- Average Development Costs
  - Large Company $2,322,000
  - Medium Company $1,331,000
  - Small Company $434,000

The Standish Group Research

- 31% of all projects cancelled prior to completion
- 52.7% of all projects costs 189% of the original estimates
- Companies and Government Agencies will
  - Spend $81 billion on cancelled software projects
  - Spend $59 billion in additional costs on completed projects

Root Causes of Project Success and Failure

- Standish Group Study (1994)
  - "Challenged" Projects
    - Lack of User Input: 13%
    - Incomplete Requirements: 12%
    - Changing Requirements: 12%
  - 1/3 failures directly related to requirements problems
  - "Successful" Projects – 9% Large Companies & 16% Small Companies
    - User Involvement: 16%
    - Executive Management Support: 14%
    - Clear statement of Requirements: 12%

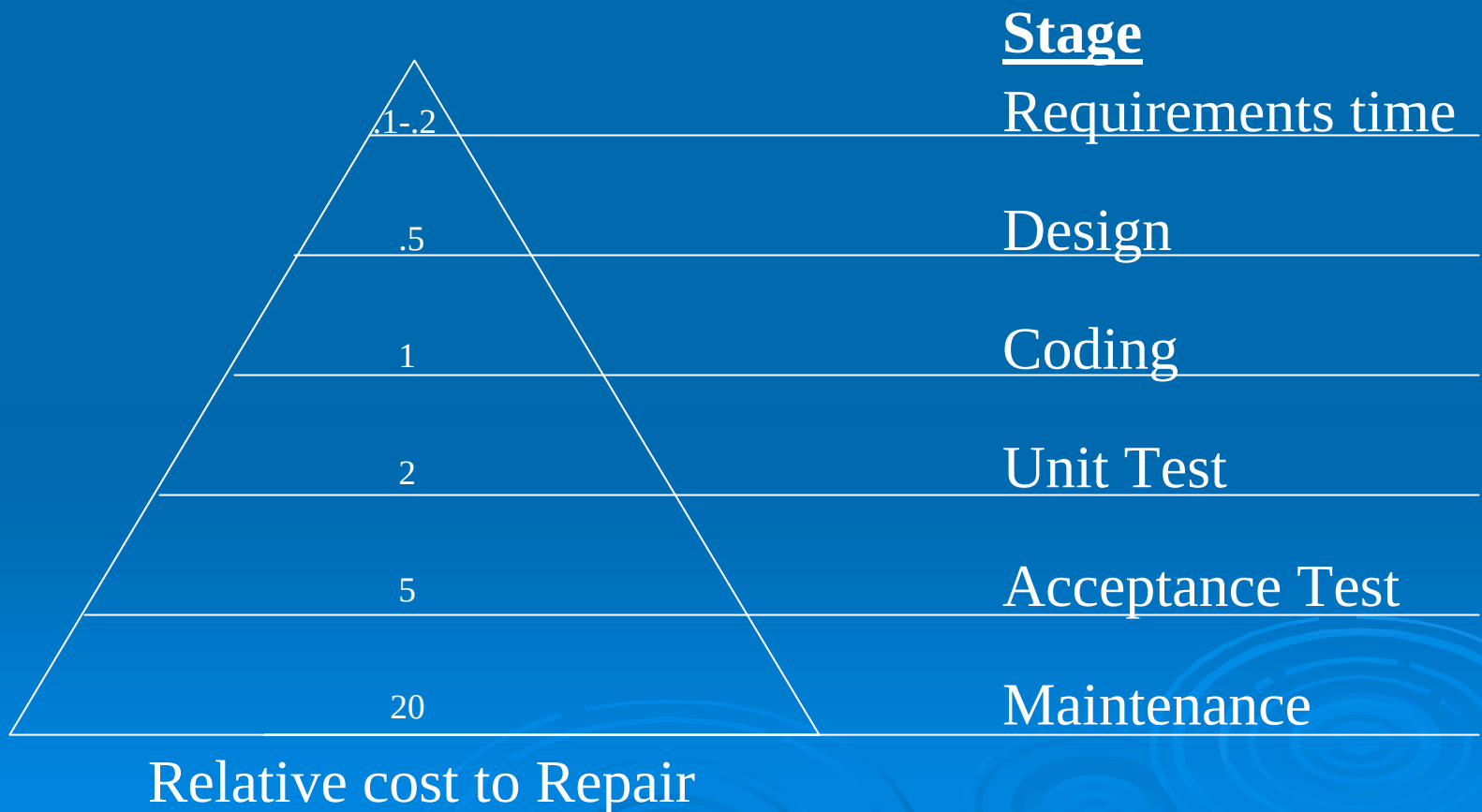
ESPITI Survey

- European Software Process Improvement Training Initiative (ESPITI) Survey (1995): Over 50% of the responses stated the two largest problems as
  - Requirements Specification
  - Managing Customer Requirements
- Requirements are the leading root cause of software problems

The Frequency of Requirements Errors

- Study by Caper Jones (1994)
  - Of the possible defect origins (Requirements Design, Coding, Documentation, Bad Fixes) Requirements errors account for 1/3 of the defects.

The High Cost of Requirements Errors (Davis, 1993)



Conclusion

- Requirements errors are likely to be the most common class of error.
- Requirements errors are likely to be the most expensive errors to fix.

Chapter 2

Introduction to Requirements Management

Introduction to Requirements Management: Definitions

- What Is a Requirement (Dorfman and Thayer)?
  - A software capability needed by the user to solve a problem to achieve an objective
  - A software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation

Introduction to Requirements Management: Definitions

- What Is Requirements Management?
  - A systematic approach to eliciting, organizing, and documenting the requirements of the system
  - A process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system.

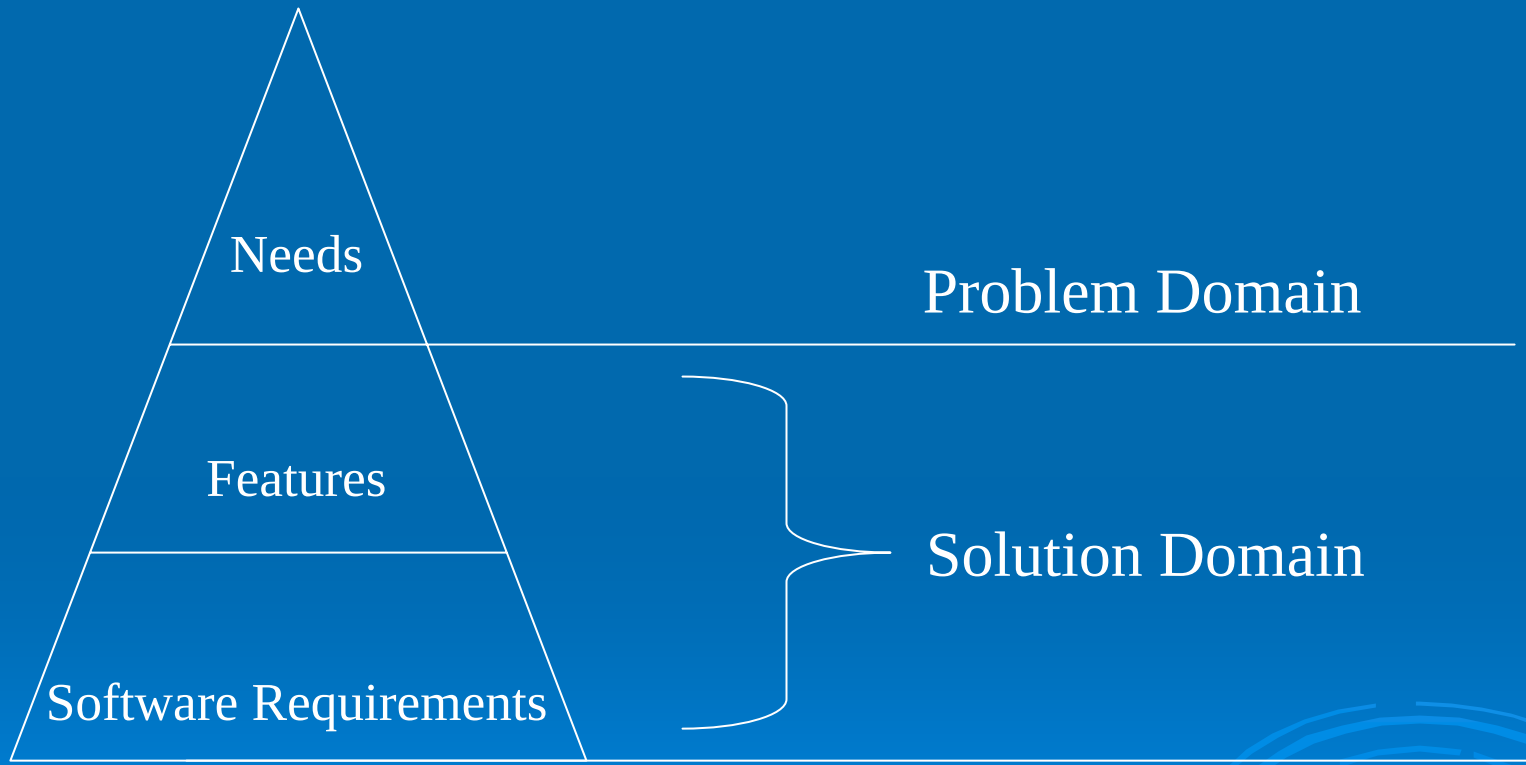
Application of Requirements Management Techniques

- Types of Software Applications
  - Information Systems (IS/IT)
    - Software developed for use within a company
  - Independent Software Vendors (ISD)
    - Software developed for sale as commercial products
  - Embedded Applications
    - Software developed to run on computers embedded in other devices, machines, or complex systems
- Systems Applications
  - Applying requirements management to systems

The Road Map

- The Problem Domain: Understanding the problem to be solved
- Stakeholder Needs: Understanding the user and stakeholder needs
- Solution Domain: Begin to define the solution
- Features of the System: A service that the system provides to fulfill one or more stakeholder needs
- Software Requirements: Develop the specific requirements of the system
- Use Cases: A sequence of actions, performed by a system, that yields a result of value to the user.

Summary


Needs
Features
Software Requirements
Problem Domain
Solution Domain

Chapter 3

Requirements and the Software Lifecycle

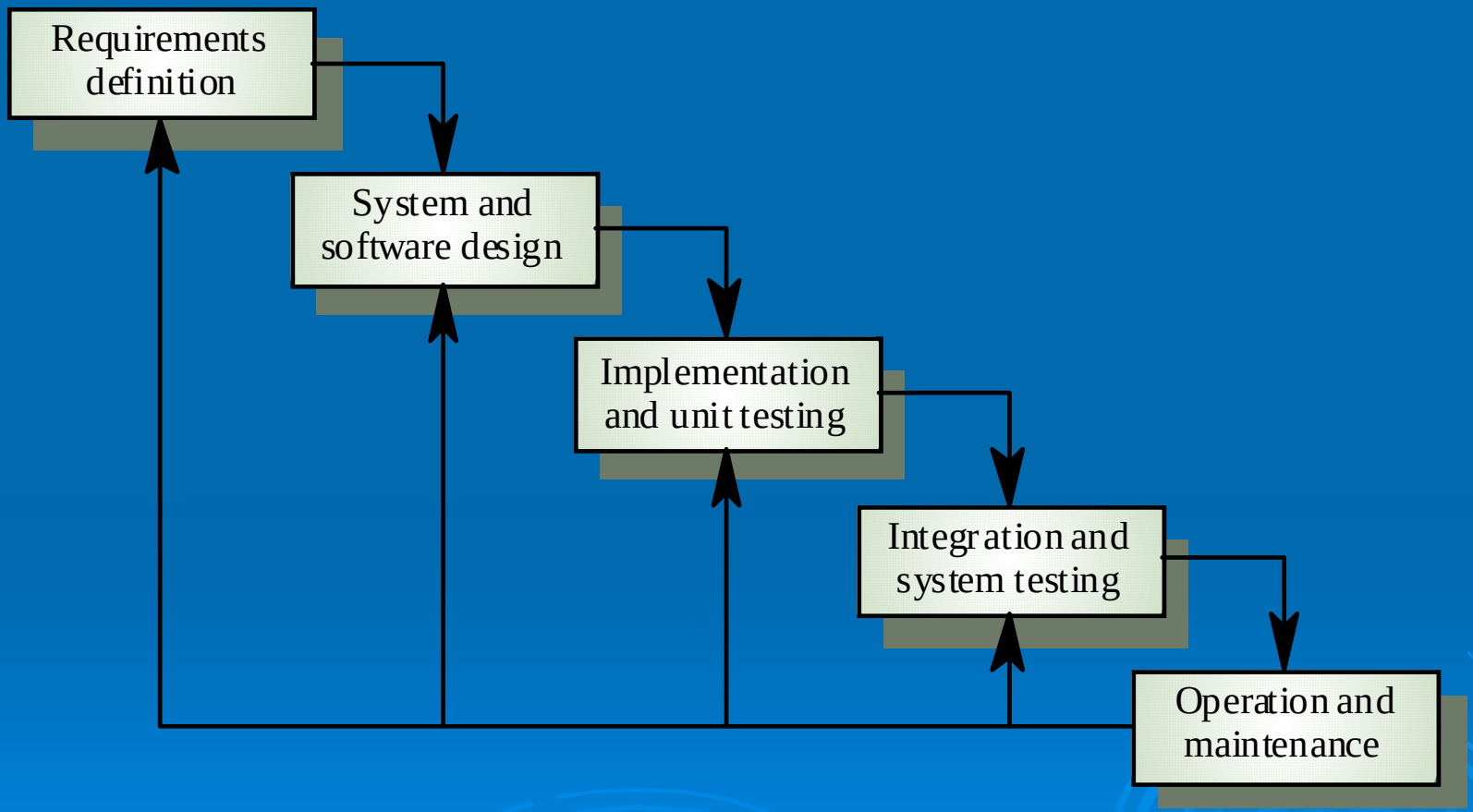
Requirements and the Software Lifecycle

- The Traditional Software Process Models
  - The Waterfall Model
    - Winston Royce, 1970
  - The Spiral Model
    - Barry Boehm, 1988
  - The Iterative Approach (RUP)
    - Kruchten, 1995
    - Lifecycle Phases
    - Iterative and Incremental
    - Disciplines

The Waterfall Model

- Improvement on the stepwise model
  - Useful when requirements are known and unchanging
  - Has feedback loops between stages
  - Prototype system can be developed in parallel with requirements analysis and design activities
- Deficiencies
  - Fixed, ridged approach
  - Requirements frozen
  - Change is anathema
  - Can cause team to disengage from reality

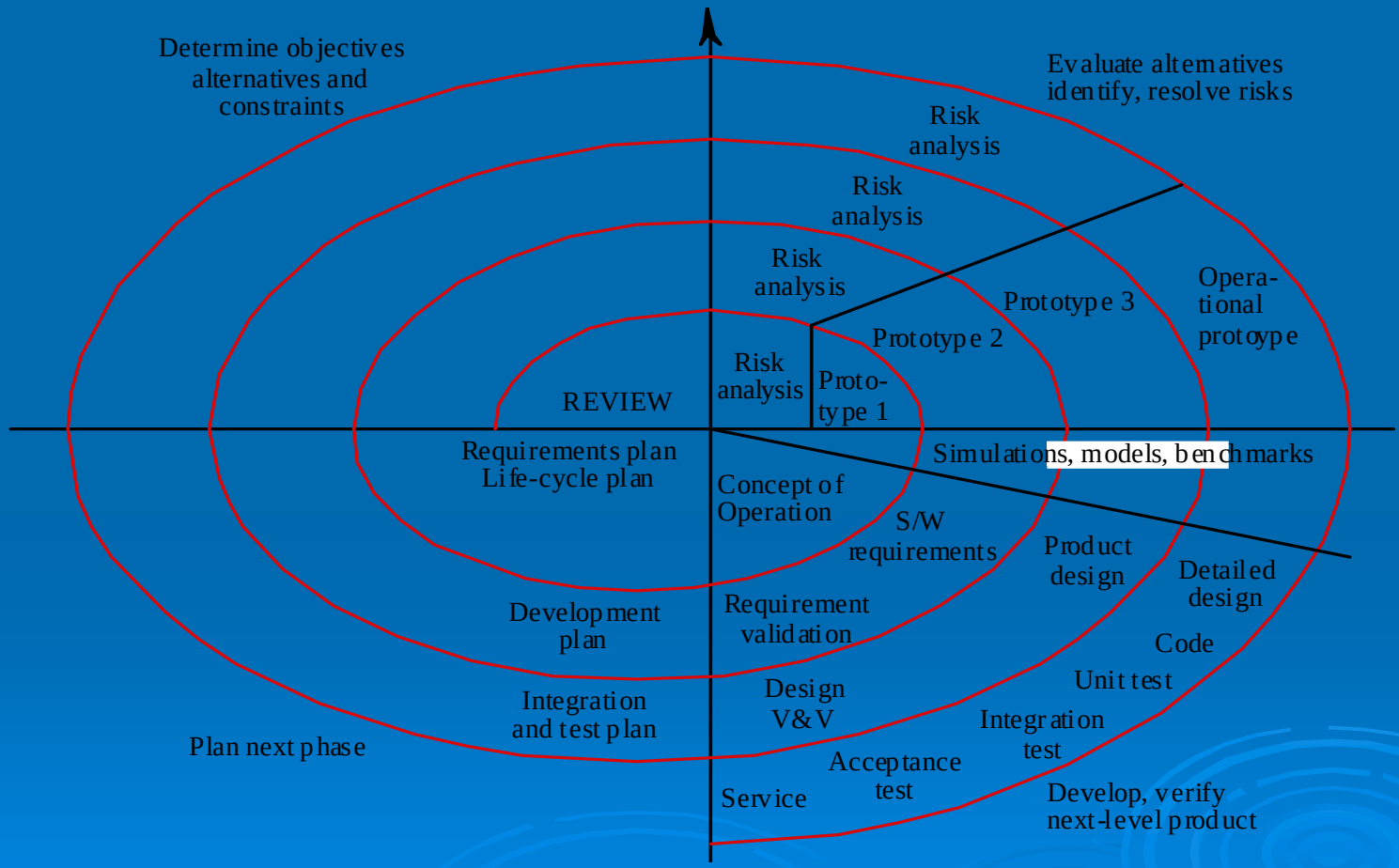
The Waterfall Model



The Spiral Model

- Risk driven approach
  - Starts with requirements planning and concept validation
  - One or more prototypes developed to confirm requirements
  - Rigorous waterfall methodology for final system
- Advantages
  - Multiple feedback opportunities
  - Removes the “Yes, Buts” early
- Deficiencies
  - Assumes time is available
  - Projects can fall into a “cut and try” approach
  - Can create instant legacy code

The Spiral Model



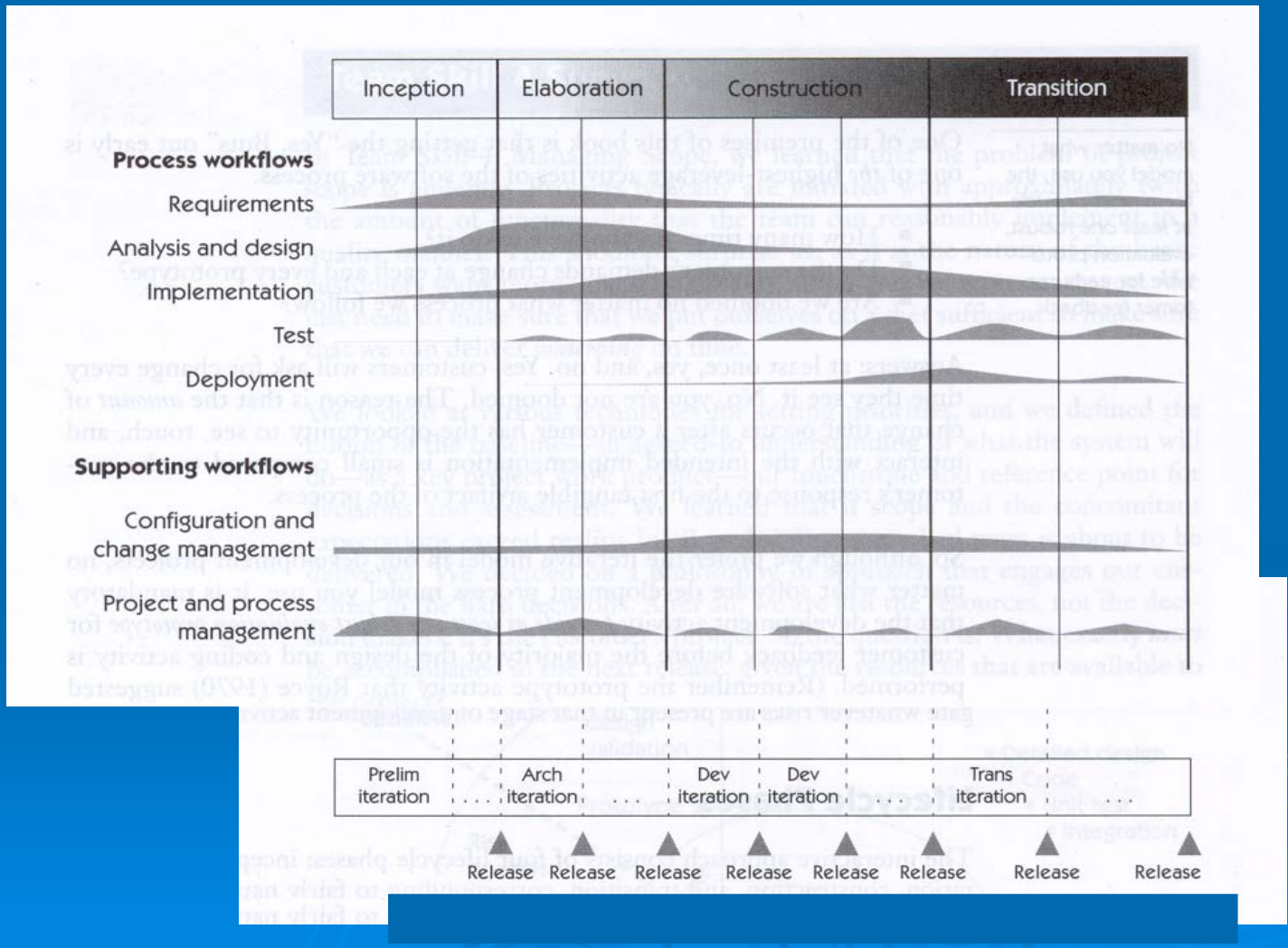
The Iterative Approach

- Hybrid of Waterfall and Spiral
- Life Cycle Phases
  - Inception, Elaboration, Construction, Transition
- Iterations
  - Each phase has multiple iterations
  - Each iteration has a plan, evaluation criteria, and an executable
  - Each iteration builds on previous iterations
  - An iteration can be considered a mini-waterfall that is “tuned” to specific needs of the iteration

The Iterative Approach

- Work Flows
  - Development activities are organized into workflows
  - Activities include
    - Requirements
    - Analysis and Design
    - Implementation
    - Test
    - Configuration and Change Management
    - Project and Process Management
  - Advantages: Get “Yes, Buts” out early and Better scope management

The Iterative Model



Chapter 4

The Software Team

The Software Team

- Software Development as a Team Activity
  - “Software Development has become a team sport” (Booch, 1998)
  - “… Today, large projects typically require the coordinated work of many teams.” (Humphrey, 1989)
- Requisite Team Skills for Effective Requirements Management
  - Team Skill 1: Analyzing the Problem
  - Team Skill 2: Understanding User Needs
  - Team Skill 3: Defining the System
  - Team Skill 4: Managing Scope
  - Team Skill 5: Refining the System Definition
  - Team Skill 6: Building the Right System

The Software Team

- Team Members Have Different Skills
  - Organization and planning
  - Programming Abilities
  - Marketing and customer relations
  - Testing and validation
  - Documentation
- The Organization of Software Teams
  - Example in test is modeled after a real world team

Team Skill One

ANALYZING THE PROBLEM

Chapter 5: The Five Steps in Problem Analysis
Chapter 6: Business Modeling
Chapter 7: Systems Engineering of Software Intensive Systems

Chapter 5

The Five Steps in Problem Analysis

The Five Steps in Problem Analysis

- Step 1: Gain Agreement on the Problem Definition
  - The Problem Statement
- Step 2: Understand the Root Causes--The Problem Behind the Problem
  - Fishbone Diagram
  - Addressing the Root Cause
    - Pareto Charts

The Five Steps in Problem Analysis

- Step 3: Identify the Stakeholders and the Users
- Step 4: Define the Solution System Boundary
  - Our system
  - Things that interact with our system
- Step 5: Identify the Constraints to Be Imposed on the Solution (Table 5-4 & 5-5, Pg 56)

Chapter 6

Business Modeling

Business Modeling

- Purpose of Business Modeling
  - Understand the Structure and Dynamics of the Organization
  - Ensure that Customers, End Users, and Developers have a common understanding of the organization
  - Understand how to deploy new systems to facilitate productivity and which existing systems may be affected by that new system

Business Modeling

- Using Software Engineering Techniques for Business Modeling
  - Rapid proliferation of methods in the 80s & 90s
  - None were the same (Method Wars)
- The Unified Modeling Language (UML)
  - Industry Standard for Visualizing, Specifying, Constructing, and Documenting Software Intensive Systems.

Business Modeling Using UML Concepts

- Business Use Case
  - Model of Intended Functions
  - Identify Organizational Roles and deliverables
- Business Object Model
  - Describes Entities and how they interact to realize the functionality in the Business Use Case

From the Business Model to the Systems Model

- Business workers become actors on the system we are developing
- Behaviors described for business workers are things that can be automated, so they help us find system use cases and define needed functionality
- Business entities are things we may want the system to help us maintain, so they help us find entity classes in the analysis model of the system

Chapter 7

Systems Engineering of Software Intensive Systems

Systems Engineering of Software Intensive Systems

- Systems Engineering (INCOSC 1993)
  - Used to help understand the requirements imposed on the software that runs in the system
- Pragmatic Principles of Systems Engineering
  - Know the Problem, Customer, Consumer
  - Use Effectiveness criteria based on needs to make systems decisions
  - Identify and Assess Alternatives so as to converge on a Solution
  - Verify and Validate Requirements and Solution Performance
  - Maintain the Integrity of the System
  - Use an articulated and Documented Process
  - Manage Against a Plan

Composition/Decomposition of Complex Systems

- Goals of the decomposition process
  - Optimized distribution and partitioning of functionality to achieve overall functionality of the system with minimum costs and maximum flexibility
  - Each Subsystem can be:
    - Defined, Designed, and built by a small team
    - Manufactured within the physical constraints and technologies of the available manufacturing process
    - Reliably tested as a subsystem, subject to availability of fixtures/harnesses that simulate the interface to the other system.
  - Appropriate deference is given to the physical domain (size, weight, location, and distribution of the subsystems) that has been optimized in the overall system context

Requirements Allocation in Systems Engineering

- Derived Requirements are requirements created through the requirements allocation process
  - Subsystem Requirements
    - Must be imposed on the subsystems themselves, but do not necessarily directly benefit end user
  - Interface Requirements
    - Between subsystems

Intelligence Moved from Hardware to Software

- Software, not Hardware:
  - Determines the ultimate functionality and success of the system
  - Consumes the majority of the costs of research and development
  - As the critical path, makes the determination when the system goes to market
  - Absorbs most of the system changes and evolves to meet the system needs
  - Development and maintenance costs are the significant factors in total manufacturing costs

Avoiding the Stovepipe System Problem

- Subsystem requirements may far outweigh external requirements
- Subsystem requirements degrading flexibility
- Political considerations when different companies build different subsystems
  - When a change in requirements means a contract renegotiation
- In the end, we get tied down and spend more money on subsystem requirements than user requirements

1 - Making It Work

- Develop, understand, and maintain the high-level requirements and use cases that span the subsystems and that describe the overall system functionality.
  - These use cases will provide context for how the system is supposed to work and will make sure that you "don't miss the forest for the trees."
  - They will also help ensure that the systems architecture is designed to support the most likely usage scenarios.

2 - Making It Work

- Do the best possible job of partitioning and isolating functionality within subsystems.
  - Use object technology principles
    - Encapsulation and information hiding
    - Interface by contract
    - Messaging rather than data sharing-in your systems engineering work.

3 - Making It Work

- If possible, develop software as a whole, not as several individual pieces, one for each physical subsystem.
- One of the characteristics of stovepipe systems is that on both sides of the interface (well or badly defined):
  - The software needs to reconstruct the state of key elements (objects) that are needed for making decisions on both sides
  - Unlike hardware, the allocation of requirements to both sides does not represent a clear partition.

4 - Making It Work

- When coding the interfaces, use common code on both sides of the interface.
  - Otherwise, there will likely be subtle variations, often blamed on "optimizations," that will make this synchronization of states very difficult.
  - Then, if the boundary between the two physical subsystems later disappears-that is, systems engineering finds out that processors are good enough to support both subsystems A and B-software engineers will have a hard time "merging" the two bodies of software.

5 - Making It Work

- Define interface specifications that can do more than would be necessary to simply meet the known conditions.
- Invest in:
  - A little extra bandwidth
  - An extra I/O port
  - Some IC real estate to provide room for expansion.

Team Skill 1 Summary

- The Five Steps in Problem Analysis
- Business Modeling
- Systems Engineering of Software Intensive Systems
- Composition/Decomposition of Complex Systems
- Avoiding the Stovepipe System Problem

Team Skill Two

UNDERSTANDING USER AND STAKEHOLDER NEEDS

Chapter 8: The Challenge of Requirements Elicitation
Chapter 9: The Features of a Product or System
Chapter 10: Interviewing
Chapter 11: The Requirements Workshop
Chapter 12: Brainstorming and Idea Reduction
Chapter 13: Storyboarding

Chapter 8

The Challenge of Requirements Elicitation

The Challenge of Requirements Elicitation

- Barriers to Elicitation
  - The “Yes, But” Syndrome
    - Wow, this is really great…
    - Yes, but now that I see it, what about…
  - The "Undiscovered Ruins" Syndrome
    - The more requirements found, the more that remain
  - The "User and the Developer" Syndrome
    - User does not know or cannot articulate what they want
    - Users think they know what they want until they get what they said they wanted
    - Analyst think they understand user problems better that the user
    - Everyone believes that everyone else is politically motivated

Techniques for Requirements Elicitation

- Interviewing and Questionnaires
- Requirements Workshops
- Brainstorming and Idea Reduction
- Storyboards

Chapter 9

The Features of a Product or System

The Features of a Product or System

- Stakeholder and User Needs
  - Definition of a Stakeholder Need:
    - A reflection of the business, personal, or operational problem (or opportunity) that must be addressed in order to justify consideration, purchase, or use of a new system
- Features
  - A service the system provides to fulfill one or more stakeholder needs
- Developer must understand the real need behind the feature

Managing Complexity

- Picking the Level of Abstraction
  - New system or increment of an existing system, abstract capabilities to a high enough level so that a maximum of 25 – 99 features result with 50 as a preference

Attributes of Product Features

- Page 100, Table 9-2
- Attributes are data elements that provide additional information about a feature
  - Status
  - Priority/Benefit
  - Effort
  - Risk
  - Stability
  - Target Release
  - Assigned to
  - Reason

Chapter 10

Interviewing

Interviewing

- Context-free Question is a question about the user's problem without context of a solution (Gause & Weinburg, 1989)
  - Forces us to listen
  - Gain a real understanding of the problem
  - Prevent solution bias
- Some examples of context-free Questions
  - Who is the user?
  - Who is the Customer?
  - Are their needs different?
  - Where else can a solution to this problem be found?

Interviewing

- Solutions – Context Questions
  - Discussing some possible solutions
- The generic, almost context-free interview
  - Page 104-5, Figure 10-1
- Customer launching into a dialogue relating the horrors of the current system

Interviewing

- The Moment of Truth: The Interview
  - Prepare appropriate context-free questions
  - Research background
    - Avoid asking questions you can answer in advance
    - Verify answers
  - Take notes during interview
  - Refer to template (Figure 10-1, Pg. 104-5)

Interviewing

- The Analyst's Summary:
  - Last section of the interview form
  - 10+10+10 <> 30
  - Record the three most important needs
    - After several interviews, some will repeat
  - Ten interviews may yield 10-15 different needs
    - Beginning of the requirements repository

Questionnaires

- Some fundamental problems
  - Relevant questions can not be decided in advance
  - The assumptions behind the questions bias the answers
    - Did this class meet your expectations? Assumption: You had expectations, so the question is meaningless
  - It is difficult to explore new domains and there is no interaction to explore domains that need to be explored.
  - It is difficult to follow up on unclear responses

Chapter 11

The Requirements Workshop

Requirements Workshops

- Accelerating the decision process
  - The authors consider the Requirements Workshops the most powerful technique in the book.
- Benefits of the Requirements Workshops
  - Assists in team building toward one goal – a successful project
  - All stakeholders get they say, no one is left out
  - It forges an agreement between the development team and the stakeholders as to what the application must do
  - It can expose and resolve political issues that are interfering with the project success
  - The output, a preliminary system definition at the features level, is available immediately

Requirements Workshops

- Preparing for the Workshop
  - Selling the Concept (not another meeting!!)
  - Ensuring the Participation of the Right Stakeholders Logistics (room, equipt., etc.)
  - "Warm-Up Materials"
    - Project-specific materials
    - Out-of-the-box thinking preparation
      - Sample memo template Pg. 113, Fig. 11-1

Requirements Workshops

- Role of the Facilitator
  - Someone from outside the organization
  - Must be a non-stakeholder
  - Trained in the process
  - Personable and well respected by both internal and external team members
  - Strong enough to chair a challenging meeting
  - Consensus-building or Team-building skills

Requirements Workshops

- Facilitator Responsibilities
  - Establish a professional and objective tone for the meeting
  - Start and stop on time
  - Establish and enforce the “rules” for the meeting
  - Introduce the goals and agenda for the meeting
  - Manage the meeting and keep the team “on track”
  - Facilitate a process of decision and consensus-making, but avoid participating in the content
  - Manage any facilities and logistics issues to ensure that the focus remains on the agenda
  - Make certain that all stakeholders participate and have their input heard
  - Control disruptive or unproductive behavior

Requirements Workshops

- Setting the Agenda
  - Sample agenda on page 114, Table 11-1
- Running the Workshop
  - Workshop tickets
    - Pg. 116, Fig. 11-2
  - Problems that can happen and their solutions
    - Pg. 117, Table 11-2
  - Brainstorming and Idea Reduction (Ch 12)
  - Production and Follow-Up (Ch 12)

Chapter 12

Brainstorming and Idea Reduction

Brainstorming and Idea Reduction

- Live Brainstorming
  - Sticky notes
  - Rules
    - No Criticism or debate
    - Use your imagination
    - Capture all ideas
    - Change and combine ideas

Brainstorming and Idea Reduction

- Idea Reduction
  - Pruning
  - Grouping Ideas
  - Feature Definition (write a short description)
  - Prioritization
    - Cumulative Voting - $100 Test: idea money
    - “Critical, Important, Useful” Categorization
      - 1/3 limit per category
- Web-Based Brainstorming

Chapter 13

Storyboarding

Storyboarding

- Types of Storyboards
  - Passive storyboards tell a story
    - Screen shots, business rules, or output reports
  - Active storyboards present an automatic description of system behavior
    - Slideshow, animation, or simulation
  - Interactive storyboards let the user experience the system
    - Live Demo or interactive presentation

Storyboarding

- What Storyboards Do
  - Three elements of any activity
    - Who the players are
    - What happens to them
    - How it happens
- Tools and Techniques for Storyboarding
  - Post-In Notes
  - PowerPoint
  - Macromedia Director

Storyboarding

- Tips for Storyboarding
  - Do not invest to much in a storyboard
  - If you do not change anything, you do not learn anything
  - Do not make the storyboard too good
  - Whenever possible, make the storyboard interactive
- Summary
  - Storyboard early
  - Storyboard often
  - Storyboard on every project that has new or innovative content

Team Skill 2 Summary

- Interview and Questionnaires
- Requirements Workshop
- Brainstorming and Idea Reduction
- Storyboarding

Team Skill Three

DEFINING THE SYSTEM

Chapter 14: A Use Case Primer
Chapter 15: Organizing Requirements Information
Chapter 16: The Vision Document
Chapter 17: Product Management

Chapter 14

A Use Case Primer

A Use Case Primer

- A use case describes a sequence of actions a system performs that yields a result of value to a particular actor
- The elements of a use case:
  - Sequences of actions
  - System performs
  - An observable result of value
  - A particular actor

A Use Case Primer

- An actor is someone or something that interacts with the system
  - Users
  - Other systems or applications
  - An input or output device

A Use Case Primer

- Use Case Structure
  - See Text Appendix C for specification template
  - Mandatory elements
    - Name
    - Brief Description
    - Actors
    - Flow of events
  - Optional elements
    - Pre-conditions – must be present for use case to start
    - Post-conditions – system state after use case is finished
    - System or Sub-system level use case
    - Other Stakeholders
    - Special requirements

A Use Case Primer

- Building the Use Case Model
  - Step 1 – Identify and describe the actors
    - Who uses the system?
    - Who gets information from the system?
    - Who provides information to the system?
    - Where in the company is the system used?
    - Who supports and maintains the system?
    - What other systems use this system?

A Use Case Primer

- Building the Use Case Model
  - Step 2 – Identify the use cases and write a brief description
    - What will the actor use the system for?
    - Will the actor create, store, change, remove, or read data in the system?
    - Will the actor need to inform the system about external events or changes?
    - Will the actor need to be informed about certain occurrences in the system?

A Use Case Primer

- Building the Use Case Model
  - Step 3 – Identify the actor and use case relationships
    - Analyze each use case to see what actors interact with it
    - Review each actor's anticipated behavior
      - Verify each actor participates in all necessary use cases

A Use Case Primer

- Building the Use Case Model
  - Step 4 – Outline the individual use cases
    - Outline basic and alternate flows
    - Basic Flow
      - Most common path from start to finish
      - No problems or exceptions
      - Questions to discover the basic flow
        - What event started the use case?
        - How does the use case end?
        - How does the use case repeat some behavior?

A Use Case Primer

- Building the Use Case Model
  - Step 4 – Outline the individual use cases
    - Alternate flow
      - Regular circumstances and exceptional events
      - Questions to discover the alternate flow
        - Are there optional situations in this use case?
        - What odd cases might happen?
        - What variants might happen?
        - What may go wrong?
        - What may not happen?
        - What kind of resources can be blocked?

A Use Case Primer

- Building the Use Case Model
  - Step 5 – Refine the use cases
    - All alternate flows, including exception conditions
      - Primary alternate flows are determined by explicit user choices – fairly easy to identity
      - The “What ifs” are a concern
    - Pre and post conditions
      - Identification of state information that controls the behavior of the system
      - Pre-condition states that are required for the use case to start
      - Post-condition states are a persistent state left behind by the use case

A Use Case Primer

- Use Cases and User Interfaces
  - How can I express a use case if I haven't designed the GUI yet?
  - How can I possibly design a set of GUIs to implement a use case that is not yet elaborated?
- They are done in parallel
  - See Pgs. 159 – 160, Fig. 14-3 thru 14-6

Chapter 15

Organizing Requirements Information

Organizing Requirements Information

- Requirements can rarely be documented in a single monolithic document because
  - It may be a complex system with voluminous documentation requiring organizational and interactive access techniques
  - System may be a member of a family of related products and no one document can contain all the specifications
  - System may be a sub-system and only satisfies a subset of the overall requirements
  - Marketing and business goals need to be separate from product requirements
  - Other requirements, perhaps regulatory or legal, may also be imposed upon the system, and these requirements may be documented elsewhere.

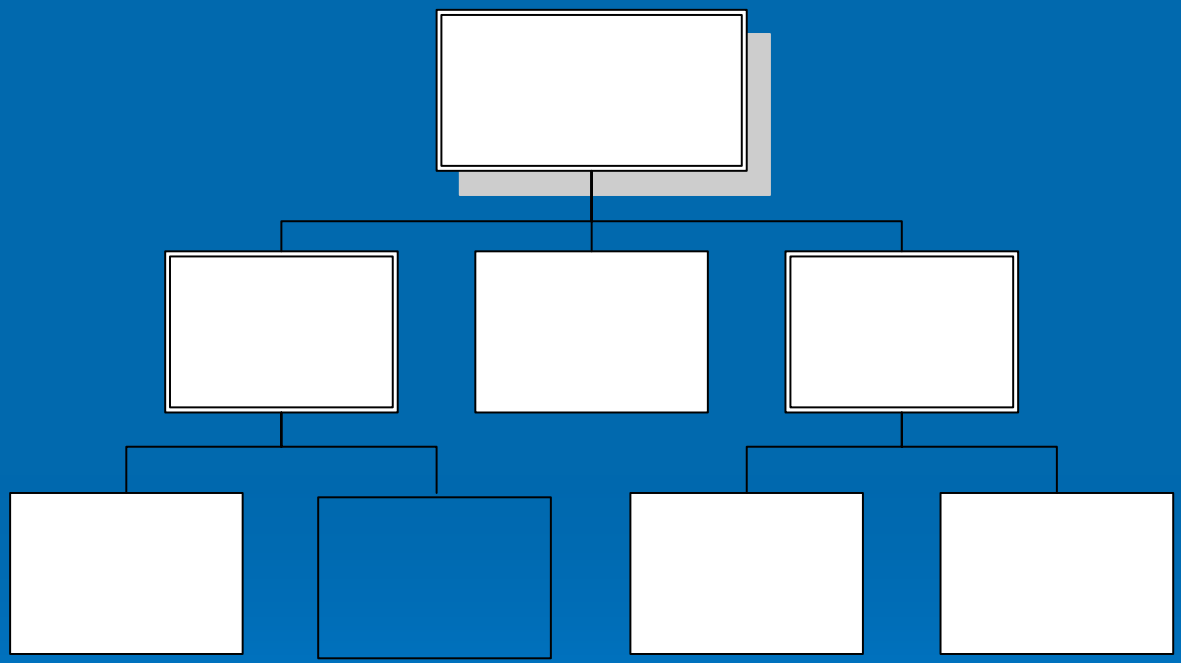
Organizing Requirements Information

- Maintain requirements organized into multiple requirements sets
  - Vision Document
    - A set that defines features in general terms
  - System Use Case Model & associate supplementary requirements
    - A set that defines features in specific terms
  - Parent requirements set
    - System Requirements Specification
  - Software Sub-system
    - Software Requirements Specification
  - A full requirements set for a family of products
  - A requirements specification for a single application and release

Organizing Requirements Information

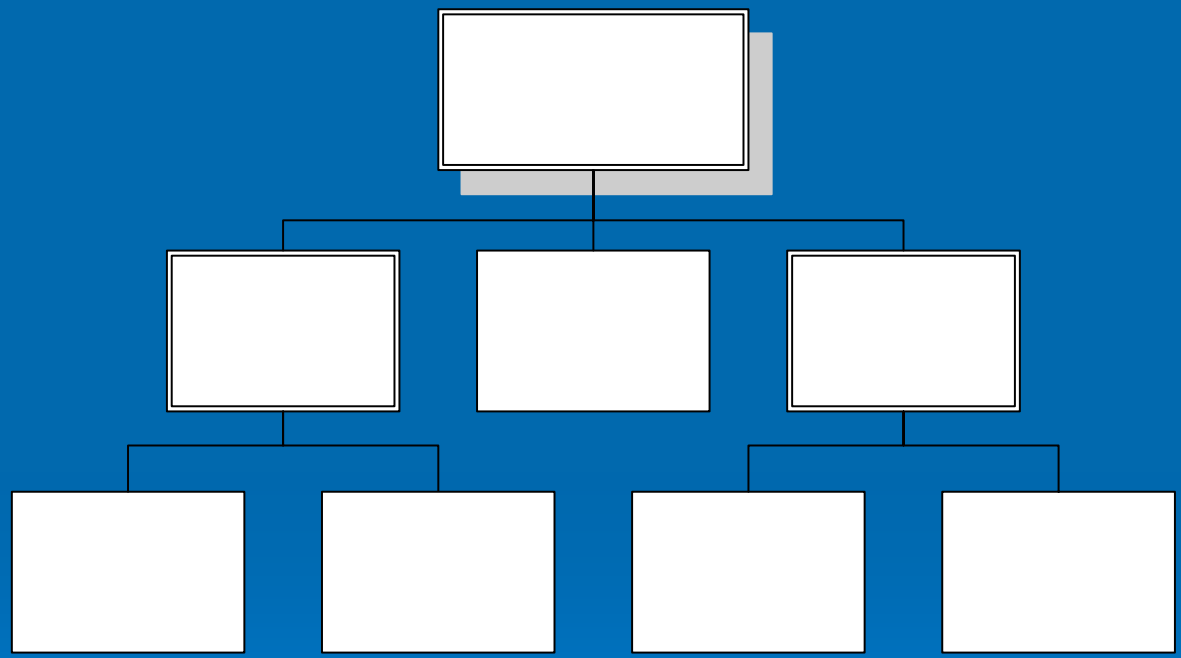
- Organizing Requirements of Complex Hardware and Software Systems
  - Systems Engineering refines system into subsystems
  - System Architecture describes the partitioning and interfaces among system and subsystems
  - Requirements are allocated to each system and subsystem in a hierarchal fashion

Organizing Requirements Information



A System of Subsystems

Organizing Requirements Information



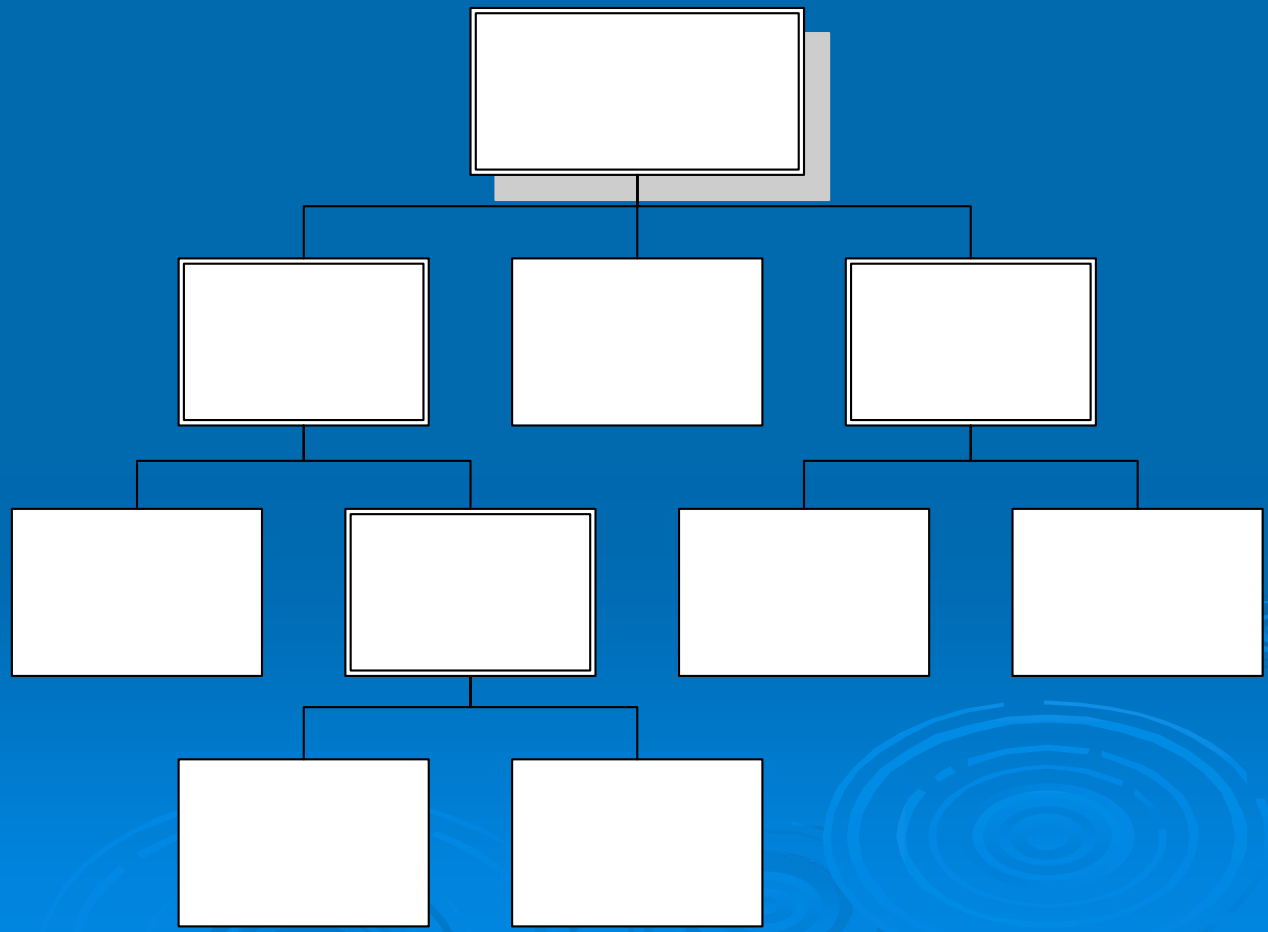
Hierarchy of requirements resulting from system design

Organizing Requirements Information

- Organizing Requirements for Product Families
  - Vision Document, SRS and Family Use Cases for common functionality
  - Vision Document, SRS and Family Use Cases for each specific functionality

Organizing Requirements Information

Hierarchy of resulting requirements, including software and hardware



Organizing Requirements Information

- On "Future" Requirements
  - Document all requirements, including future
  - Clearly identify requirements planned for the current release
- Business and Marketing Requirements
  - Facilitate communication among marketing, management, and developers
  - Helps to establish agreement on product scope

Chapter 16

The Vision Document

The Vision Document

- Components of the Vision Document
  - Defines the problem and solution at a high level of abstraction
  - Basis for agreement between marketing, management, and product developers
  - Template on Page 175-6

The Vision Document

- Vision Document for Release 1.0
  - All requirements, including those not implemented in 1.0
- Vision Document for Version 2.0
  - Requirements not implemented in 1.0 now implemented in 2.0 and new requirements not implemented in 2.0
  - Feature changes for 1.0 requirements that did not deliver what was expected
  - Massive growth over time

The Vision Document

- The "Delta Vision" Document as 2.0
  - Includes only what has changed from 1.0 and any other information needed for context
  - Focuses on what is new and what is different about the new release
- The Delta Vision Document in a Legacy System Environment
  - Define features and use cases around what is new and what is different about the new release

Chapter 17

Product Management

The Role of the Product Champion

- Manage the elicitation process
  - Determine when enough requirements are discovered
- Manage conflicting inputs from stakeholders
- Make required trade-offs
  - Select features that deliver highest value to the greatest number of stakeholders
- Owner of product vision
- Product advocate
- Negotiate with management, users, and developers
- Defend against feature creep

The Role of the Product Champion

- Maintain “healthy tension” between
  - Customer desires
  - Developer capabilities for a given time frame
- Be representative of official channel between customer and development team
- Manage the expectations of customers, executive management, and the marketing and engineering teams

The Role of the Product Champion

- Communicate features of new release to all stakeholders
- Review the use cases and requirements to ensure that they conform to the true vision represented by the features
- Manage changing priorities, and the addition and deletion of features
- Never give up, never surrender

The Product Champion in a Software Product Company

- Who makes the tough decisions to help guide the teams to commercial success?
- Who takes the time to understand what supporting services and other user conveniences might be necessary to ensure customer success?
- It is the product manager who is empowered to make these decisions

Role of the Product Manager

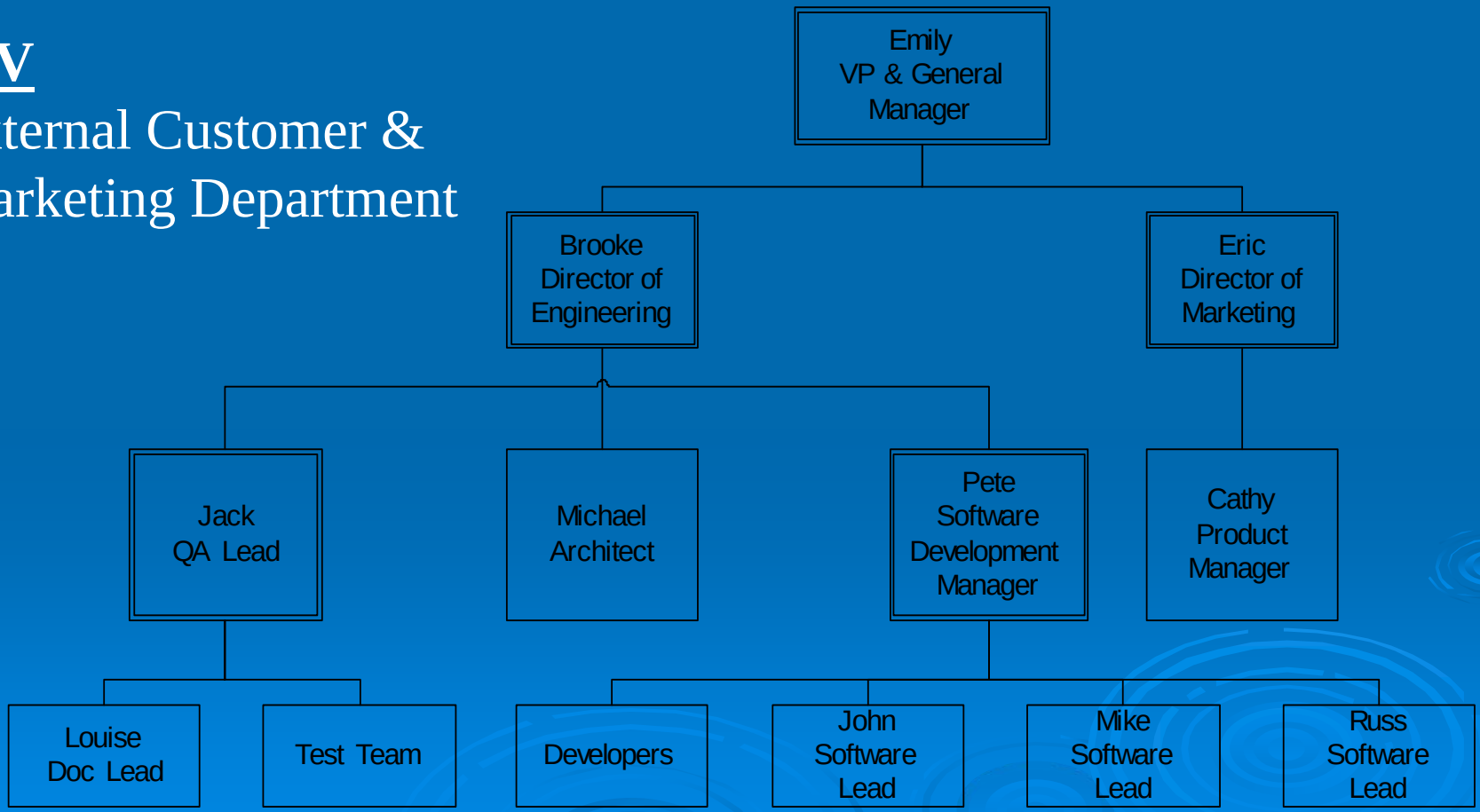
- The product manager's job is to help the software teams build products that customers want to buy. A Project Manager Recipe:
  - 2 parts requirements management
  - 1 part development experience
  - 1 part commercial practices
  - 1 part marketing (measure dose carefully!)

  Mix with plain common sense

  Bake in user's over until done

The Product Champion in a Software Product Environment

ISV
External Customer & Marketing Department



Primary Activities for a Product Manager

- Driving the vision
  - Gaining agreement on what should and can be done
  - Finding the optimum path to market

Primary Activities for a Product Manager

- Defining the Whole Product Plan
  - Template for plan on Pg. 192, Fig. 17-6
  - Covers the four dimensions of a successful customer solution
    - The product itself
    - Accompanying services and support
    - The commercial terms that define the business relationship between the customer and the developer
    - The documentation provided to help assure the customers' success

Primary Activities for a Product Manager

- Sponsoring the use case model and supplementary requirements
- Testing the product concept with customer and user (Page 194, Table 17-1)
- Completing the user experience (user docs, help, tool tips, copyright notes, logos, etc.) Page 194, Table 17-2.

Primary Activities for a Product Manager

- Defining commercial terms (Page 195, Table 17-3)
- Positioning and messaging (marketing activities)
- Supporting activities
  - Branding and product labeling
  - End user training materials
  - Product Demonstration
  - Sales and marketing collateral

Product Champion in an IS/IT Shop

- No External Customer
- No Marketing Department
- Champion should be someone with whom the team can identify
- Tough decisions should be left to the Change Control Board (CCB)

Team Skill 3 Summary

- A Use Case Primer
- Organizing Product Requirements
- The Vision Document
- The Champion

Team Skill Four

MANAGING SCOPE

Chapter 18: Establishing Project Scope

Chapter 19: Managing Your Customer

Chapter 18

Establishing Project Scope

The Problem of Project Scope

- Components of Project Scope
  - The functionality that must be delivered to meet the user's needs
  - The resources available to the project
  - The time available in which to achieve the implementation
- The Hard Question
  - Reducing the scope, yet keeping the customer happy

Establishing Project Scope

- The Requirements Baseline
  - The itemized set of features, or requirements, intended to be delivered in a specific version of the application
    - Baseline must be acceptable to the customer
    - Baseline must have a reasonable probability of success, in the team's view

Setting Priorities and Assessing Effort

- Priorities are set by customers, users, product managers – NOT the development team
  - Establish priority for each feature
  - Critical, Important, Useful
- Effort is set by the development team NOT – customers, users, product managers
  - Assess effort for each feature
  - High, Medium, Low

Adding the Risk Element

- Established by the development team
- Reducing Scope – do critical items first
  - Priority: Critical; Effort: High; Risk: High
    - Alarm! Establish immediate risk-mitigation strategy; resource immediately; focus on feasibility with architecture
  - Priority: Critical; Effort: High; Risk: Low
    - A likely resource-constrained item; resource immediately
  - Priority: Critical; Effort: Low; Risk: Low
    - Resource as a safety factor, or defer until later

Adding the Risk Element

- Layout prioritized features list
- Draw the baseline

Feature	Priority	Effort
Feature 1: External relationship database support	Critical	Medium
Feature 4: Port to new OS release	Critical	High
Feature 6: Import of external data by style	Critical	Low
Feature 3: Ability to clone a project	Important	Medium
Baseline (features above this line are commited features)		
Feature 2: Multiuse	Important	Low
Feature 5: New project wizard	Important	Low
Feature 7: Implement tool tips	Useful	Low
Feature 8: Integrate with version-manager subsystem	Useful	High

Chapter 19

Managing Your Customer

Managing Your Customer

- Some Important Insights…
  - It is in the best financial interest of the customer to meet its external commitments
  - Delivering high quality, on time and on budget, even if scope reduced, is the highest benefit the development team can provide
  - The application belongs to the customer, not the development team

Engaging customers to manage *their* project scope

- We can actively engage our customers in managing ***their*** requirements and ***their*** project scope to ensure both the quality and the timeliness of the software outcomes
- It's ***THEIR*** Project!

Managing Your Customer

- Communicating the Result
  - Customer must be a direct participant in the scope reduction decision to “own” the result
  - Customers not part of the process are unhappy with the result and blame the developer

Managing Your Customer

- Negotiating with the Customer – Underpromise and Overdeliver (Fisher, Ury, & Patton, 1983)
  - Start high, but not unreasonable
  - Separate the people from the process
  - Focus on interests, not positions
  - Understand your walk-away position
  - Invent options for mutual gain
  - Apply objective criteria

Managing Your Customer

- Managing the Baseline
  - Create margins for error
  - Resist features creep
    - Can cause 50% - 100% increase in scope (Jerry Weinberg, 1995)
  - Enough functionality at the right time to meet the customer's real need
- Baseline used as part of an effective change management system

Change Management

- Official Change
  - Customer request for new system capability
  - Any change to the baseline must effect the resources, schedule, or features set
  - If resources and schedule are fixed, the team and customer must prioritize change
- Unofficial Change
  - Change from other sources (review later)

Team Skill 4 Summary

- Establishing Project Scope
- Managing Your Customer

Team Skill Five

REFINING THE SYSTEM DEFINITION

Chapter 20: Software Requirements – A More Rigorous Look
Chapter 21: Refining the Use Cases
Chapter 22: Developing the Supplementary Specification
Chapter 23: On Ambiguity and Specificity
Chapter 24:Technical Methods for Specifying Requirements

Chapter 20

Software Requirements – A More Rigorous Look

Software Requirements – A More Rigorous Look

- Looking Deeper into Software Requirements – A Definition of Software Requirements Reviewed (Dorfman and Thayer)
  - A software capability needed by the user to solve a problem to achieve an objective
  - A software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation

Software Requirements – A More Rigorous Look

- Up until now, we have purposely left things at a high level of abstraction
  - Better understand the main characteristics of the system
    - Focus on features and key use cases
    - How features and use cases fulfill user needs
  - Can assess the system for completeness, consistency, and fit within its environment
  - Use information to determine feasibility and manage scope before making significant resource investments

Software Requirements – A More Rigorous Look

- Five major classes of things needed to fully describe system behavior (Davis)
  - Inputs to the system
    - Content, details of input devices, and protocol (form, look and feel) of the input
  - Outputs from the system
    - Description of output devices to be supported, and the protocols and formats of the generated information
  - Functions of the system
    - Mapping of the inputs to outputs, various combinations
  - Attributes of the system
    - Typical non-behavioral requirements (reliability, maintainability, availability, throughput) to take into account
  - Attributes of the system environment
    - Additional non-behavioral requirements (systems ability to operate with other applications, loads and operating systems

Software Requirements – A More Rigorous Look

- Relationship Between Software Requirements and Use Cases
  - Use cases are just one way to express software requirements
- Relationship Between Features and Software Requirements
  - Features are too abstract to fully describe and to use for writing code
  - Software requirements are *testable* and allow for *traceability*

Software Requirements – A More Rigorous Look

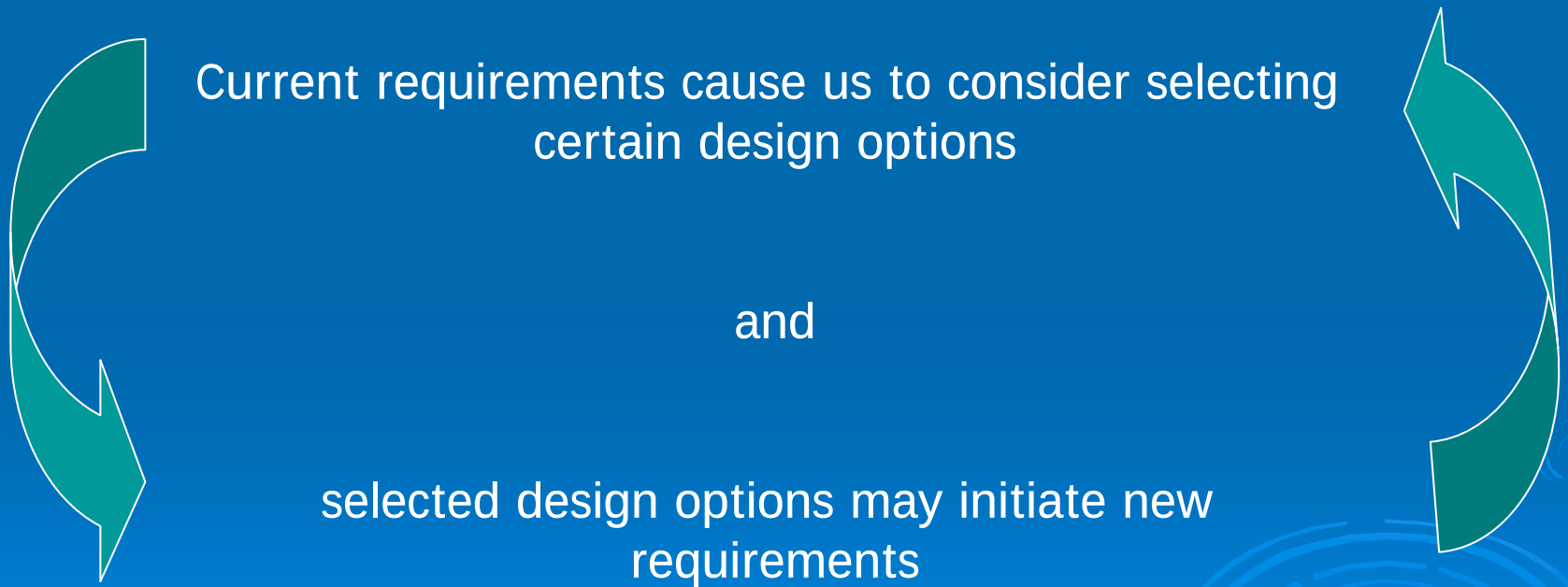
- The Requirements Dilemma: What versus How
  - Avoid stipulating design or implementation details
  - Concentrate on what the system is supposed to do
- Exclude Project Information
  - Plans, schedules, budgets, staffing, configuration management, IV&V plans
- Exclude Design Information
  - Leave system design and architecture for the design document
  - Design constraints – special class of customer driven requirement

Software Requirements – A More Rigorous Look

- More on Requirements versus Design
  - Requirements (mostly) precedes design
  - Users and customers make requirements decisions
  - Technologists make design decisions

Software Requirements – A More Rigorous Look

- Iterating Requirements and Design



Software Requirements – A More Rigorous Look

- A Further Characterization of Requirements
  - Functional Software Requirements
    - How the system behaves
  - Nonfunctional Software Requirements
    - Attributes of the system
    - Usability, Reliability, Performance, Supportability
  - Design constraints
    - Restrictions on the design of a system, or the process by which a system is developed, that do not affect the external behavior of the system but that must be fulfilled to meet technical, business, or contractual obligations

Software Requirements – A More Rigorous Look

- Are Design Constraints True Requirements?
- Using Parent-Child Requirements to Increase Specificity
- Organizing Parent-child Requirements

Chapter 21

Refining the Use Cases

Refining the Use Cases

- Use cases made to date are:
  - Insufficiently detailed to drive design and implementation
  - Not enough use cases defined for all conditions
    - Exception conditions
    - State conditions
    - Other special conditions

Refining the Use Cases

- How Use Cases Evolve
  - The *test* for *enough* use cases
    - Enough to describe all possible ways that the system can be used
    - Level of specificity to drive design, implementation, and testing
  - Use case elaboration is *not* system decomposition
    - Looking for more detailed interactions of the actors with the system
    - Seen as refining a series of actions rather than hierarchically dividing actions into subactions

Refining the Use Cases

- The Scope of a Use Case
  - Is a set of user interactions one or several use cases?
  - It must make sense to the customer
  - The complete dialogue or instance of use is the use case
    - A use case on *Pushing a Button* is not a use case
    - Small actions must be considered within the larger picture of what they are trying to achieve

Refining the Use Cases

- The Case Study: Anatomy of a Simple Use Case
  - Reviewing the Actor(s)
    - Look for key decisions that may add actors
  - Reviewing the Name
    - Use case names must be unique and relevant to the use of the use case
    - Look for changes that require a change in the name
  - Refining the Description
    - Update the description to reflect changes in the use case functionality

Refining the Use Cases

- The Case Study: Anatomy of a Simple Use Case
  - Defining and Refining the Flow of Events
    - Flow has changed and the use case needs to adapt
    - New alternate flows?

Refining the Use Cases

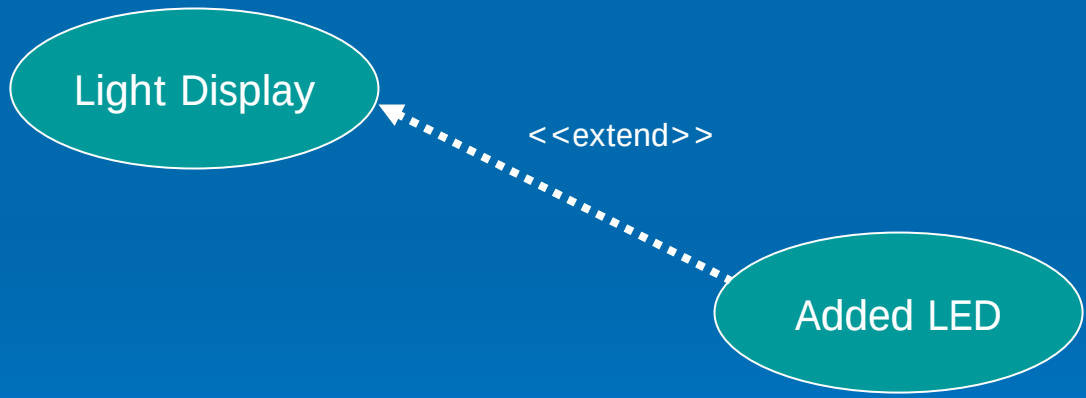
- The Case Study: Anatomy of a Simple Use Case
  - Identify Pre- and Post-conditions
    - Must be user observable states
    - Pre-condition is a constraint of when a use case can start
    - Pre- and Post-conditions are for all flows, but can describe an individual flow if necessary
    - Post-condition should be true for any flow
    - Post condition describes what the use case is supposed to achieve – good tool for helping to describe the use case

Refining the Use Cases

- The Case Study: Anatomy of a Simple Use Case
  - Identifying special requirements
    - Performance – action must respond within 50 milliseconds

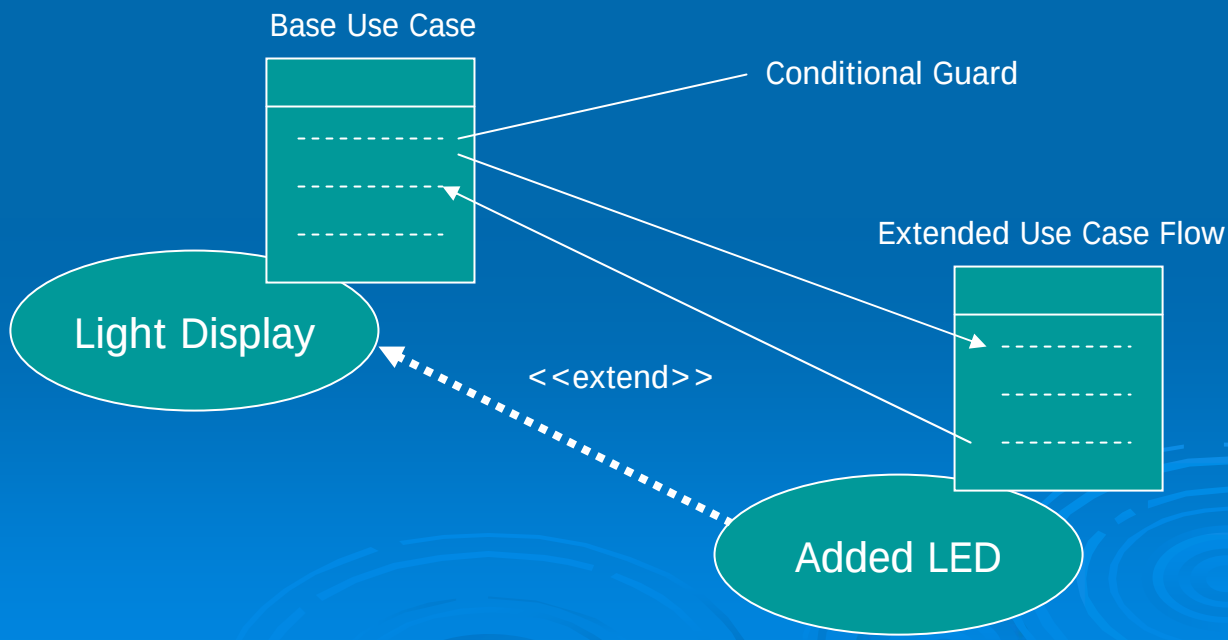
Refining the Use Cases

- Extending Use Cases



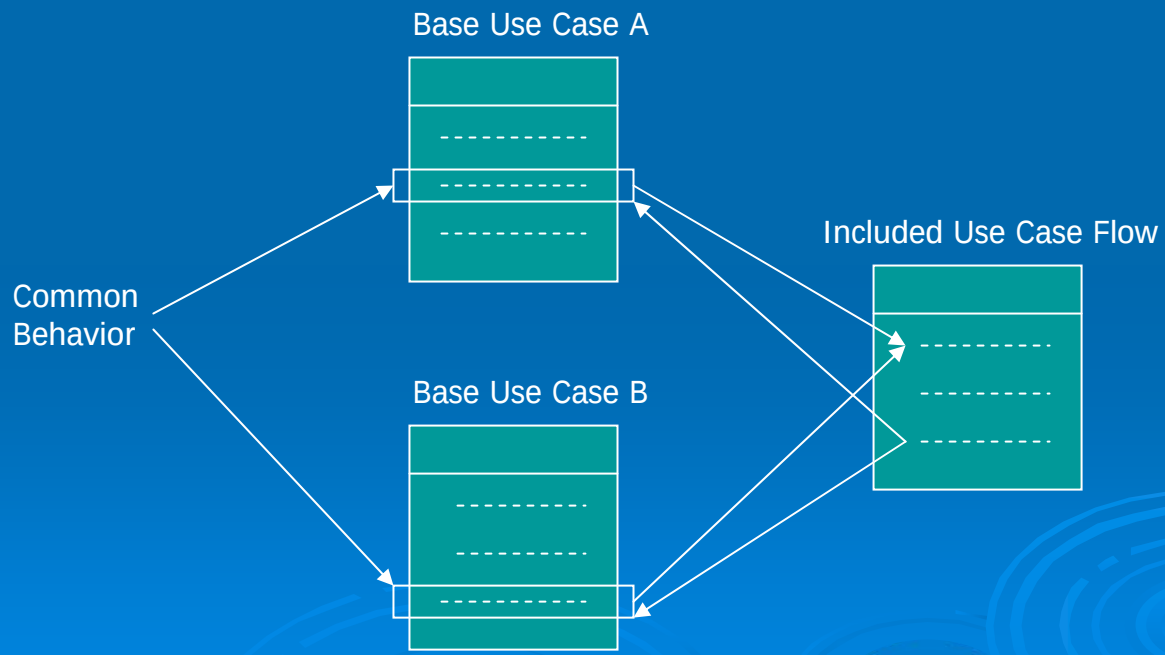
Refining the Use Cases

- Base use case with extended flow



Refining the Use Cases

- Including Use Cases in other use cases



Chapter 22

Developing the Supplementary Specification

Developing the Supplementary Specification

- The role of the Supplementary Specification
  - For requirements that are inappropriate for a use case
    - Application shall run on Windows XP
  - 80/20 rule:
    - 80% use case
    - 20% Supplementary

Developing the Supplementary Specification

- Expressing functional requirements in the Supplementary Specification
  - Traditional *declarative* requirements
    - Simple text-based sentences
  - Systems that are primarily algorithmic and computational in nature
    - Use mathematical expressions or statistical algorithms to represent requirements
  - Applications that work behind the scene such as industrial automation and robotics
    - Augment use case models with state machines and sequential logic expressions
  - Applications that parse input strings, compile code or translate languages should state requirements in other ways.

Developing the Supplementary Specification

- Exploring Non-functional Requirements
  - Usability
  - Reliability
  - Performance
  - Supportability

User's Bill of Rights (Karat, 1998)

1. The user is always right. If there is a problem with the use of the system, the system is the problem, not the user.
2. The user has the right to easily install software and hardware systems.
3. The user has the right to a system that performs exactly as promised.
4. The user has the right to easy-to-use instructions for understanding and utilizing a system to achieve desired goals.
5. The user has the right to be in control of the system and to be able to get the system to respond to a request for attention.

User's Bill of Rights (Karat, 1998)

6. The user has the right to a system that provides clear, understandable, and accurate information regarding the task it is performing and the progress toward completion.
7. The user has the right to be clearly informed about all system requirements for successfully using software or hardware.
8. The user has the right to know the limits of the system's capabilities.
9. The user has the right to communicate with the technology provider and receive a thoughtful and helpful response when raising concerns.
10. The user should be the master of software and hardware technology, not vice-versa. Products should be natural and intuitive to use.

Reliability

- Availability - As defined by the user
- Mean Time Between Failures (MTBF)
- Mean Time To Repair MTTR)
- Accuracy
- Defect Rate
- Bugs per type

Performance

- Response time
- Throughput
- Capacity
- Degradation Modes

Supportability

- The ability of software to be easily modified to accommodate enhancements or repairs

Developing the Supplementary Specification

- Understanding Design Constraints
  - Definition: Restrictions on the design of a system, or the process by which a system is developed, that do not affect the external behavior of the system but that must be fulfilled to meet technical, business, or contractual obligations

Understanding Design Constraints

- Sources of design constraints
  - Restriction of design options
    - Try and leave the choice to the designers rather that specifying it in the requirements
    - Example: Shall use Oracle DBMS
  - Conditions imposed in the development process
    - Compatibility with existing systems
    - Application Standards
    - Corporate best practices and standards
  - Regulations and imposed standards
    - FDA, FCC, DOD, ISO, UL

Developing the Supplementary Specification

- Handling design constraints
  - Distinguish them from other requirements
  - Include design constraints in a special section
  - Identify the source for each design constraint
  - Document the rationale for each design constraint
- Are design constraints true requirements?
  - Do not represent one of the five system elements in the elaborated definition, except:
    - When elevated to the level of a legitimate business, political, or technical concern – read “A Cautionary Tale”

Developing the Supplementary Specification

- Identifying other requirements
  - Physical artifacts (CDs, etc.)
  - Target system configuration and preparation documents
  - Support or training requirements
  - Internationalization and localization requirements

Developing the Supplementary Specification

- Linking the Supplementary Specification to the use cases
  - Define classes of nonfunctional requirements
  - Associate the classes to special requirements recorded in the use case
- Template for the Supplementary Specification
  - Figure 22-1, Page 268-9

Chapter 23

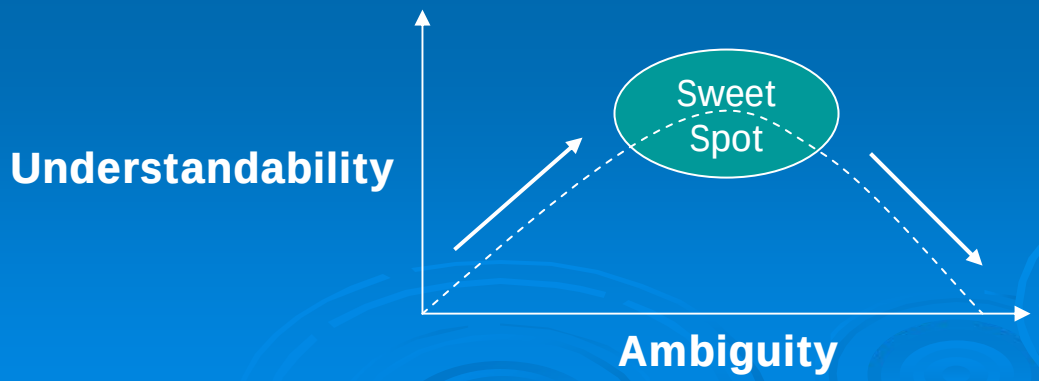
On Ambiguity and Specificity

On Ambiguity and Specificity

- Finding the Sweet Spot
  - To what level of specificity must I state the requirements in order to avoid any chance of being misunderstood? It depends!
  - What does it mean to flash the bulb every one second? See Fig. 23-2, Pg. 273

On Ambiguity and Specificity

- Finding the Sweet Spot
  - What level of specificity must I provide?
  - It depends on the context of your application and on how well whose doing the implementation can make the right decisions or at least ask questions when there is ambiguity



On Ambiguity and Specificity

- Mary Had A Little Lamb
  - See definitions and Table 23-1
- Techniques for Disambiguation
  - Memorization Heuristic
    - Stakeholders and users try to remember the customer's real requirements.
    - Focus on the parts not easily remembered
  - Keyword Technique
    - List definitions of key operative words
    - Mix and match
  - Emphasis Technique
    - Read the requirement out loud putting emphasis on different words to discover different interpretations
  - Other techniques
    - Pictures, graphics, formal methods

Chapter 24

Technical Methods for Specifying Requirements

Technical Methods for Specifying Requirements

- Pseudocode
- Finite State Machines
- Decision Tables and Decision Trees
- Activity Diagrams (Flowcharts)
- Entity-relationship Diagrams

Technical Methods for Specifying Requirements

- Pseudocode
  - “Quasi” programming language
  - Imperative sentences with a single verb and a single object
  - A limited set, typically not more than 40-50, of “action-oriented” verbs from which the sentences must be constructed
  - Decisions represented with a formal IF-ELSE-ENDIF structure
  - Iterative activities represented with DO-WHILE or FOR-NEXT structures

Technical Methods for Specifying Requirements

- Finite State Machines
  - Output and next state determined solely by transition event and current state
  - System's behavior is said to be deterministic; we can mathematically determine every possible state
  - State Transition Diagram

Technical Methods for Specifying Requirements

- Decision Tables and Decision Trees
  - Method for evaluating systems with a combination of inputs
  - Different combinations of inputs lead to different behaviors
- Activity Diagrams (Flowcharts)
- Entity-relationship Diagrams

Team Skill 5 Summary

- Software Requirements – A More Rigorous Look
- Refining the Use Cases
- Developing the Supplementary Specification
- On Ambiguity and Specificity
- Technical Methods for Specifying Requirements

Team Skill Six

BUILDING THE RIGHT SYSTEM

Chapter 25: From Use Cases to Implementation
Chapter 26: From Use Cases to Test Cases
Chapter 27: Tracing Requirements
Chapter 28: Managing Change
Chapter 29: Assessing Requirements Quality in Iterative Development

Chapter 25

From Use Cases to Implementation

From Use Cases to Implementation

- The effort to show that a requirement is fulfilled in the code is not trivial
- Mapping Requirements Directly to Design and Code
  - In some cases, it is not too difficult
  - “Indicate the compilation process to the user”

Mapping Requirements Directly to Design and Code

- The Orthogonality Problem
  - The form of the requirement and design are different
  - “The system shall handle up to 100,000 trades per hour”
  - The language of the real world and code are different
  - You can not physically map a performance requirement to the logical structure of the code
  - Code often needs to interact to achieve functionality. Implementation may be distributed through the code
  - Good design is not determined by the degree that requirements can be traced to design, but by other things.
    - Reuse code, architecture patterns, etc.

Mapping Requirements Directly to Design and Code

- Object Orientation
  - Orthogonality problem is substantially improved
    - Some degree of orthogonality with requirements remain
  - Still not a one-to-one mapping with requirements
    - Collaborative classes yielding behavior that is greater than the sum of its parts

Mapping Requirements Directly to Design and Code

- The Use Case as Requirement
  - Substantially improves the orthogonality problem
  - Does a better job of describing how the system performs its intended function
- Managing the Transition
  - OO methods and use cases help us deal with the orthogonality problem, but it is not solved
    - Digress into modeling and software architecture

Modeling Software Systems

- Software systems are extraordinarily complex undertakings
- Abstraction is used to view the system as a simplified model
  - Still must accurately represent the system
- Model selection is an important issue
  - Help us to understand the system
  - Do not mislead us with errors or abstraction
- The model is not the reality
  - Models can lead us astray – Sun-centered example

The Architecture of Software Systems

- Description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns (Shaw & Garlan)

The Architecture of Software Systems

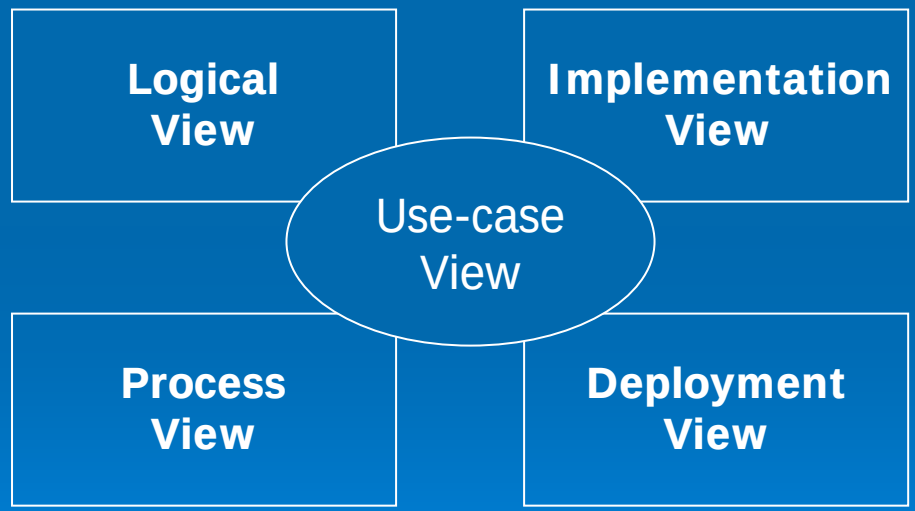
- Kruchten (1999) believes we use Architecture to:
  - Understand what the system does
  - Understand how the system works
  - Think and work on pieces of the system
  - Extend the system
  - Rescue part(s) of the system to build another one

The Architecture of Software Systems

- The 4+1 View of Architecture

End User
Functionality

Programmer
Software Management



Designers / Testers
Behavior

System Integrators
Performance
Scalability
Throughput

System Engineering
System topology
Delivery, installation
Communication

The Architecture of Software Systems

- The 4+1 View of Architecture
  - The Logical View addresses Functionality
  - The Implementation View addresses bits and pieces relevant to implementation
    - Source code, Libraries, Object classes
  - Process Views describe
    - Operations of the systems
    - Systems with parallel tasks
    - Interfaces with other systems
    - Execution interactions
    - Throughput and performance issues
  - Deployment View addresses implementation elements to the supporting infrastructure
    - Operating systems and computing platforms

Role of the Use-Case Model in Architecture

- Returning to the orthogonality problem:
  - Use case view plays a special role by
    - Presenting key use cases
    - Driving the design
    - Tying the architecture views together
  - Allows all stakeholders to examine system implementation plan against actual use cases and requirements of the system

Realizing Use Cases in the Design Model

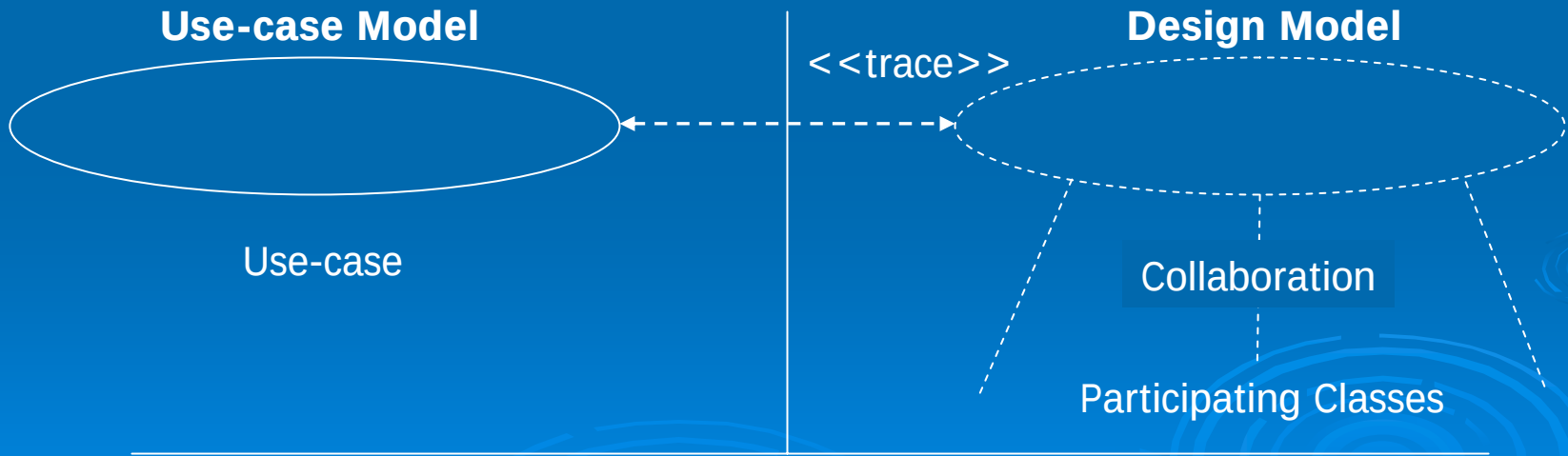
- Use cases are realized using collaborations
  - Societies of classes, interfaces, subsystems
  - Elements that cooperate to achieve some behavior
- Use-case realization
  - Common UML stereotype
  - Special form of collaboration
  - Shows how functionality of a specific use case is achieved in the design model

Realizing Use Cases in the Design Model

- Collaboration
  - Key modeling construct
  - Shows the systematic and aggregate behavioral aspects of the system
  - Deliver the “bigger-than-the-sum-of-its-parts” behavior
  - Can now trace the requirements expressed in the use case model into the design

Collaboration

Symbolic representation of a Collaboration



A Use-case realization in the design model

Structural and Behavioral Aspects of Collaborations

- Collaborations have two aspects
  - Structural part
    - Classes, elements, interfaces, and subsystems on which the implementation is structured
  - Behavioral part
    - Specifies the dynamics of how the elements interact to accomplish the result
- Collaborations are not a physical thing
  - Just a description of how cooperating elements work together

Structural and Behavioral Aspects of Collaborations

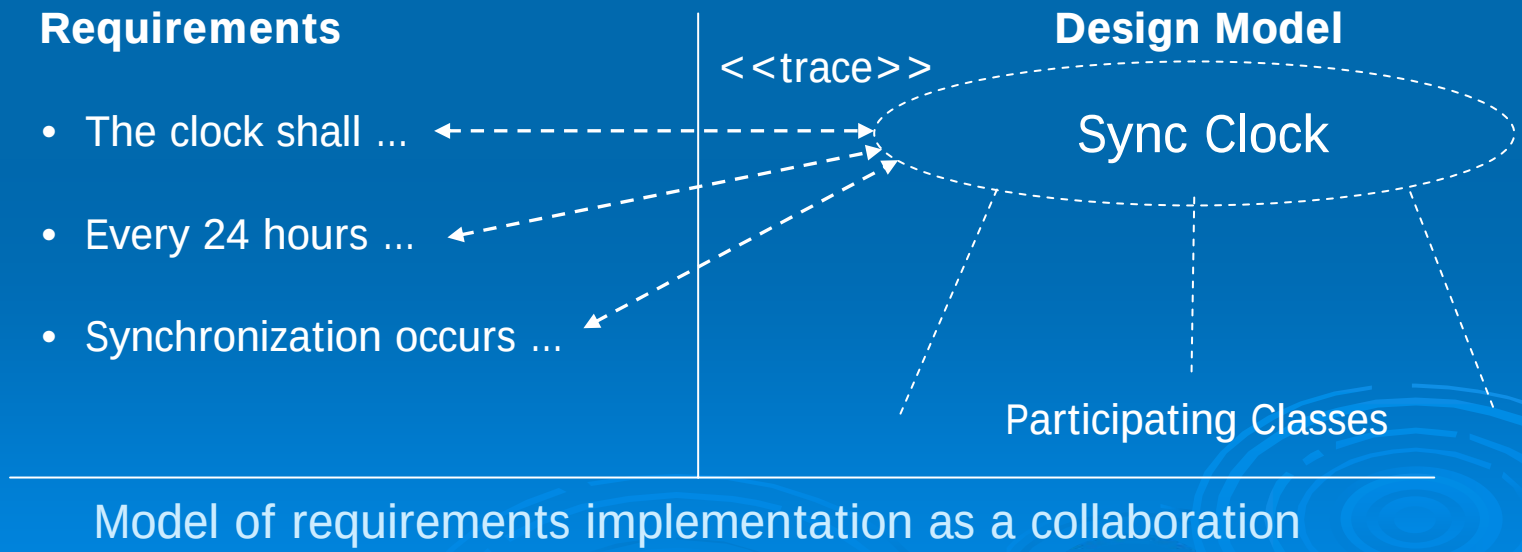
- Inside the collaboration
  - Class diagrams to represent the static structural aspects
  - Interaction or sequence diagram to represent the dynamic or behavioral aspects

Using Collaborations to Realize Sets of Individual Requirements

- Use-case and the use-case realization
  - Fills the gap between
    - Statement of requirements and
    - Design of the system that fulfills those requirements
- Can be used to model the implementation of any requirement, or set of requirements.
- Achieves design to requirements traceability

Collaboration

- Grouping of itemized requirements to accomplish the sequencing of events as in a use case
- Can also be used to model non-functional requirements



Model of requirements implementation as a collaboration

From Design to Implementation

- Ensures that significant requirements and use cases are properly realized in the design model
- Helps ensure software design conforms to the requirements
- A major step in the process of design verification
- Next, the classes and objects in the design model are further refined in the iterative design process
- Finally, they are implemented in terms of the physical software components

Chapter 26

From Use Cases to Test Cases

From Use Cases to Test Cases

- A Tester's Perspective: Musings on the Big Black Box
  - The Use-case technique builds a set of assets that can directly drive the testing process
  - Can achieve a vastly improved testing process over what was previously used

From Use Cases to Test Cases

- Questions regarding the testing task might include the following:
  - What is the system supposed to do and in what order?
  - What are all the things that can go wrong with the system and how does the system behave when this happens?
  - How can I create and record a set of testing scenarios which will put the system through its paces?
  - How will I know when I've tested the system completely and thoroughly?
  - Is there anything else this system is supposed to do, or not do, that I need to know about?
  - Given that the system is coming out of development late, and given that the shipment date has not been moved, is there any way the testing team can start earlier on the next project and avoid this "late discovery" process?

From Use Cases to Test Cases

- Assets available for developing test cases with the use-case process
  - A comprehensive set of use cases that document an ordered set of events describing how the system interacts with the user and how it delivers its results to that user
  - A use case model that documents all the use cases for the system, as well as how they interact, and what actors drive them
  - Within each use case, both a basic flow of eventsand a series of alternate flows that defines what the system does in various “what if” scenarios
  - Descriptions of pre conditions and post conditions
  - A supplementary specification that defines nonfunctional requirements of the system, including the usability, reliability, performance, and supportability of the system

Is a Use Case a Test Case?

- Almost; additional test design work is needed
- Common testing terms
  - A test plan – purpose and goals
  - A test case – test inputs, execution conditions, expected results for a particular objective
  - A test procedure – set of detailed instructions for the setup, execution, and evaluation of the results for a given test case
  - A test script – software script that automates the execution of a test procedure
  - Test coverage – the degree to a specific test or set of tests addresses all specific test cases for a given system or component
  - A test item – is a build that it an object of testing
  - Test results – a repository of data captured during the execution of a test used in calculating the different key measures of testing

Relationships of Test Artifacts

- There are a fair number of test artifacts to define, implement, and manage
- Test plan is the foundation
  - Testing strategy and test cases
- Use cases are a source for the test cases
- Each test case has one or more test procedures that define how to execute the test case
- Test cases are executed manually of automatically using a script
- Results are recorded in the test results
- See Figure 26-1, Pg. 308

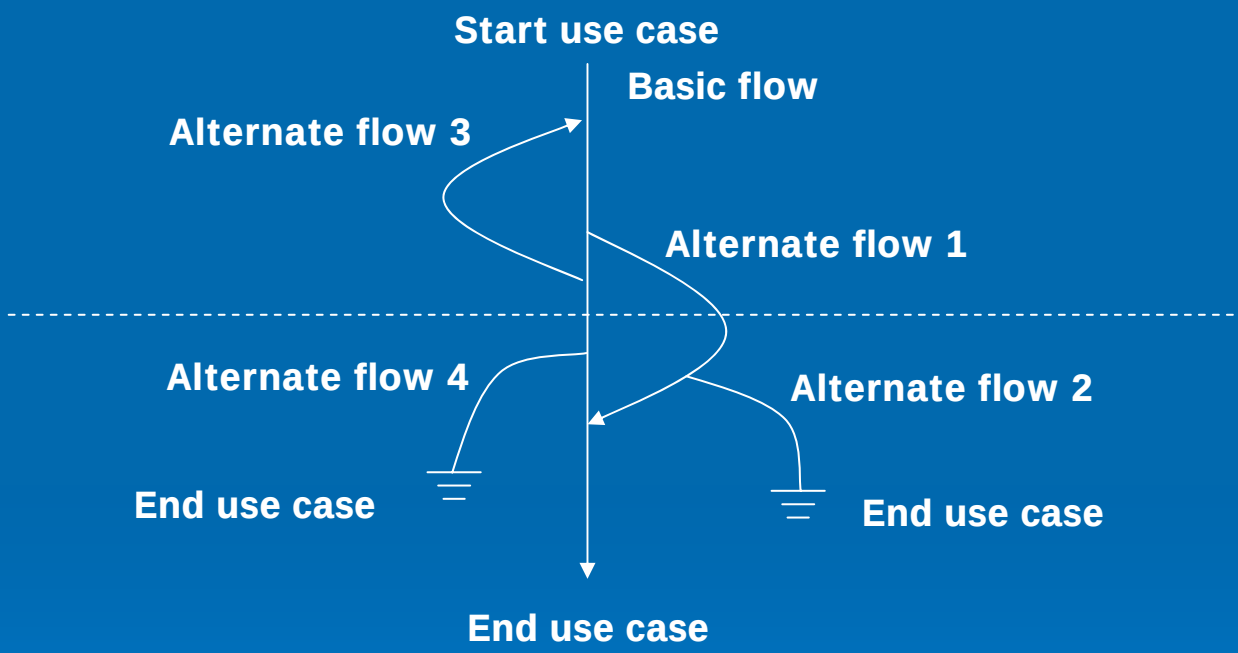
The Role of the Test Cases

- Creation and execution of the test cases are the main activity of testing
  - Test cases form the foundation on which to design and develop test cases
  - The “depth” of the testing is proportional to the number of test cases
  - The scale of the test effort is proportional to the number of test cases
  - Test design and development, and the resources needed, are largely governed by the required rest cases
- Use cases are the input to this process

Use-case scenarios

- A scenario, or an instance of a use case, is a use case execution wherein a specific user executes the use case in a specific way

Use-case scenarios



- Each of these paths is a scenario, or use-case instance, that can be both executed and tested
- Even with a limited number of use cases, a large number of specific scenarios must be tested to assure that the system behaves in accordance with its requirements

Deriving Test Cases from Use Cases: A Four-Step Process

- 1. Identify the use case scenarios
- 2. For each scenario, identify one or more test cases
- 3. For each test case, identify the conditions that will cause it to execute
- 4. Complete the test case by adding values

1. Identify the use case scenarios

Scenario Number	Originating Flow	Alternate Flow	Next Alternate	Next Alternate
1	Basic Flow			
2	Basic Flow	Alternate Flow 1		
3	Basic Flow	Alternate Flow 1	Alternate Flow 2	
4	Basic Flow	Alternate Flow 3		
5	Basic Flow	Alternate Flow 3	Alternate Flow 1	
6	Basic Flow	Alternate Flow 3	Alternate Flow 1	Alternate Flow 2
7	Basic Flow	Alternate Flow 4		
8	Basic Flow	Alternate Flow 3	Alternate Flow 4	

1. Identify the use case scenarios

- Eight possible scenarios for the sample use case
- Can be used to identify all possible use case scenarios
- Need a testing strategy that recognizes that it is impractical to test all possible scenarios
- Not all scenarios are described in the original use case and the discovery process may need to be conducted interactively with the development team
  - Use cases developed are not 100% exhaustive and are written at a level of detail that may be insufficient for testing
  - The test team's review process will create new discoveries and additional scenarios that may result from executing the use case
- The iterative model allows us to plan for and effectively manage this process

2. For each scenario, identify one or more test cases

Test Case ID	Scenario Condition	Data Value 1	Data Value 2	Data Value n	Expected Result	Actual Result
1	Scenario 1					
2	Scenario 2					
3	Scenario 2					

Matrix for testing specific scenarios

Test Case ID	Scenario Condition	Description	Expected Result
1	Scenario 6	Less than seven sequences entered	Sequences saved system beeps
2	Scenario 6	Attempt to enter an eighth sequence	Error?

Two test cases for one scenario

3. For each test case, identify the conditions that will cause it to execute

- Tester searches use case steps for data conditions, branches, etc., that would cause a specific test case to occur
- For each such condition identified, tester enters a new column in the matrix
- Initial pass: Identify condition exists, create column entry, indicate state
  - Valid – indicates a condition must be true for the basic flow to execute
  - Invalid – indicates a condition that will invoke the alternate flow, causing a specific scenario to occur
  - Not Applicable – indicates that an identified condition is not applicable to that specific test caseID
- Example
  - Use case table 26-4, Pg. 314
  - Matrix w/conditions, Table 26-5, Pg. 315

4. Complete the test case by adding values

- May need to use supplementary specification to find
  - Performance specifications
  - Valid data ranges for input forms and interface protocols, etc.
- Make sure test cases address special requirements defined for the use case
  - Min/Max Performance, Min/Max Loads, Data volumes expected
- See Table 26-6, Pg. 316 for final matrix

Managing Test Coverage

- Select the most appropriate or critical use cases for the most thorough testing
  - Primary user interfaces
  - Balance between cost, risk, and necessity of verifying the use case
  - Determine relative importance of use cases by prioritizing

Black Box versus White Box Testing with Use Cases

- Black box testing: input and output testing
  - See only the result of the implementation against the limited test criteria we can establish
- White box testing
  - Looking inside the system to see how it does what it does
  - Internal inspection, or design assurance
- Use cases do help in white box testing
  - Use case -> use case realization
  - We can use the use case realization static and dynamic parts to see how things are happening

Chapter 27

Tracing Requirements

Using Traceability to Support Verification

- The Role of Traceability in Systems Development
  - “Tracing requirements artifacts through the stages of specification, architecture, design, implementation, and testing is a significant factor in assuring a quality software implementation” (Ieffingwell & Widrig, 2003)

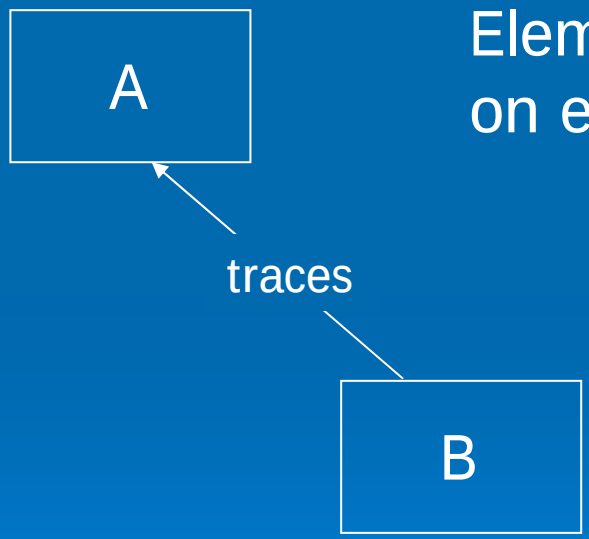
IEEE Definition of Traceability

- “The degree to which a relationship can be established between two or more products of the development process, especially products that have a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given component match.” (IEEE 610.12-1990 #3)
- “The degree to which each element in a software development project establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement it satisfies.” (IEEE 610.12-1990 #3)

The Traceability Relationship

- Definition: A relationship between two project elements
  - Examples: “is fulfilled by”, “is part of”, “is derived from”, “is tested by”
- In UML, we look in one direction or the other at a type of *dependence relationship*
  - Semantic relationship between two or more model elements

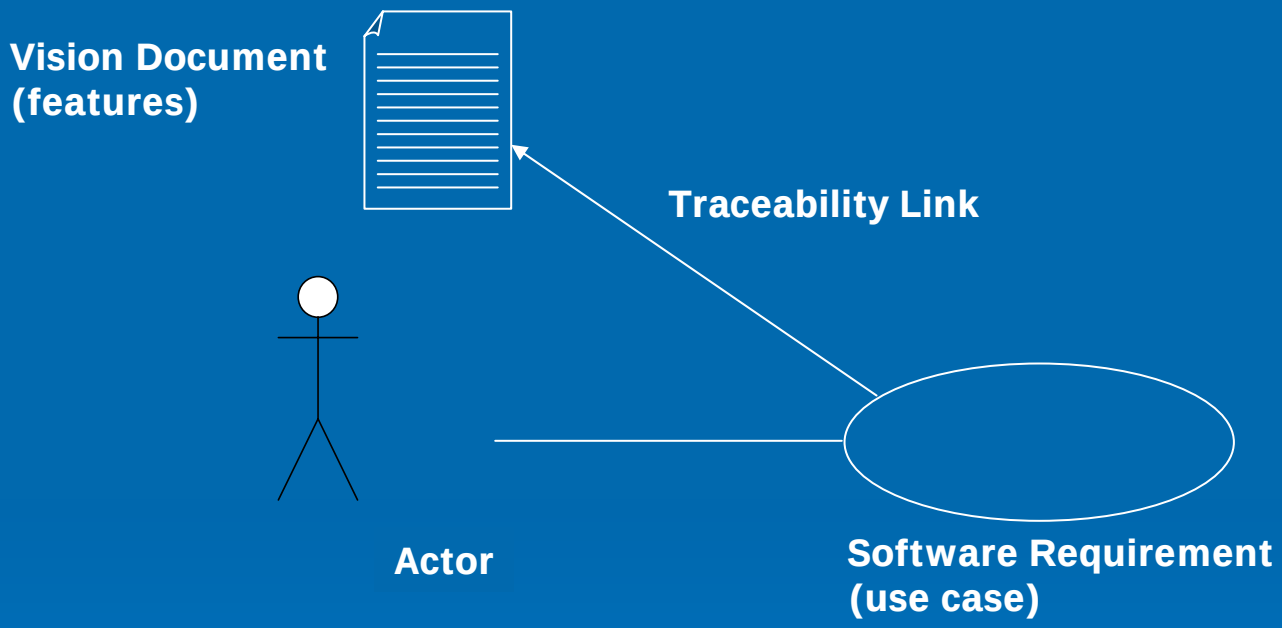
Dependence Relationship



Element B is in someway dependent on element A

One element may affect another, but the reverse is not necessarily true

Traceability Link from Vision Document to Software Requirement

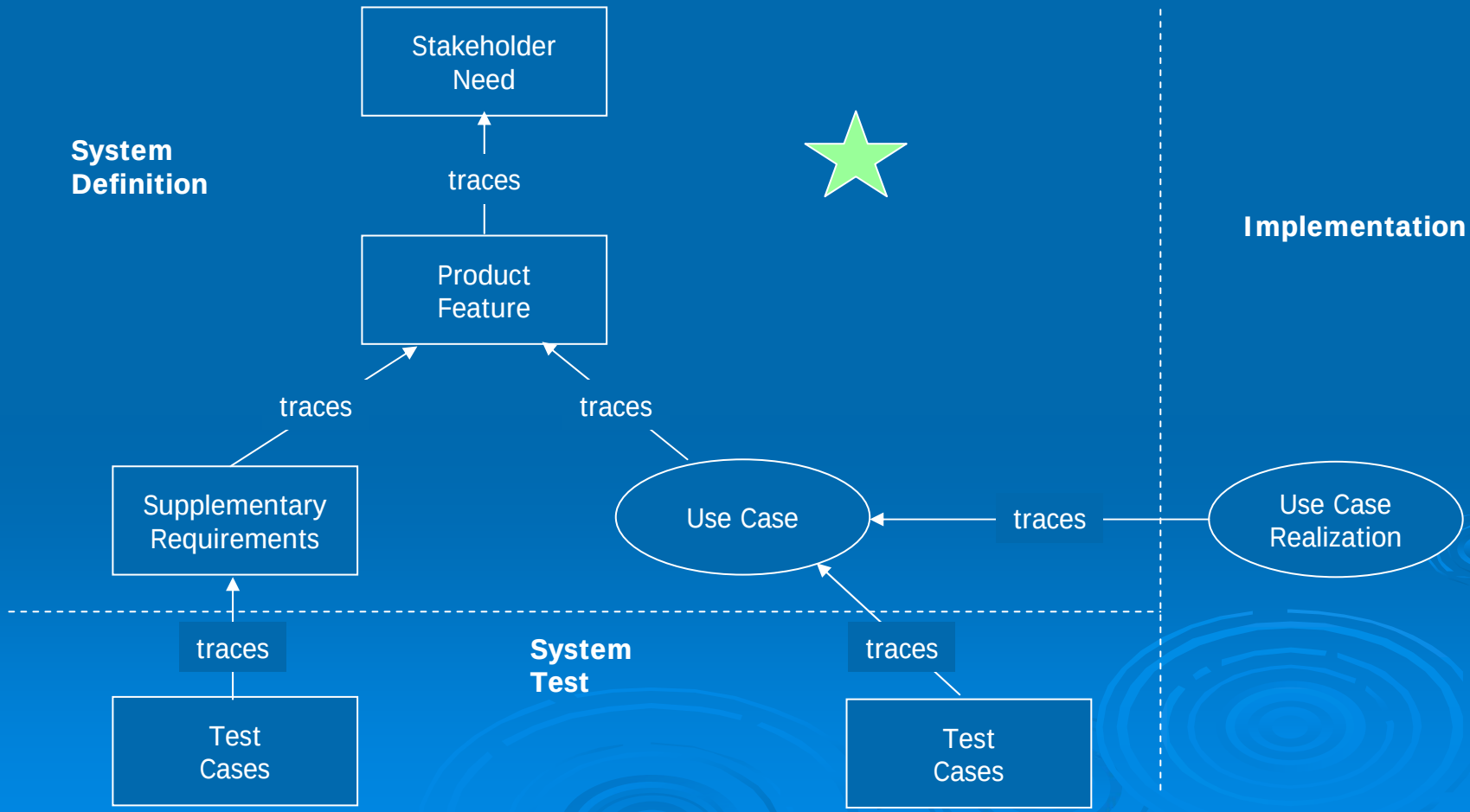


- Describes the predecessor-successor relationship
- Establishes the reason that the second element exists
  - To implement the future
- Additional meaning can be inferred
  - Software requirement traced to a test case suggests that the requirement is “tested by” the test case it is “traced to”

Tracing Requirements in the Systems Definition Domain

- System or product definition domain
  - Tracing user needs to features
  - Tracing features to use cases
  - Tracing features to supplementary requirements
- Requirement-to-requirement traceability
- A Generalized Traceability Model – Next slide

A Generalized Traceability Model



Tracing Requirements in the Systems Definition Domain

- Tracing user needs to features
  - Satisfy user and other stakeholder needs

	Feature 1	Feature 2	…	Feature n
Need 1	X			
Need 2		X		X
…		X	X	
Need m				X

Traceability Matrix: User Needs versus Features

Tracing Requirements in the Systems Definition Domain

- Inspecting the Traceability Matrix
  - If inspection of a *row* fails to detect an X
    - No feature may yet be defined to respond to a user need
  - If inspection of a *column* fails to detect an X
    - A feature may have been included for which there is no defined product need
  - Impact assessment process
    - Assess what special needs should be reconsidered if a user need changes

Tracing Requirements in the Systems Definition Domain

- Tracing features to use cases

	Use Case 1	Use Case 2	…	Use Case n
Feature 1	X			X
Feature 2		X		X
…			X	
Feature m		X		X

Traceability Matrix: Features versus Use Cases

Tracing Requirements in the Systems Definition Domain

- Inspecting the Traceability Matrix
  - If inspection of a *row* fails to detect an X
    - No Use Case may yet be defined to respond to a feature
  - If inspection of a *column* fails to detect an X
    - A use case has been included for which there is no known feature that requires it

Tracing Requirements in the Systems Definition Domain

- Tracing features to supplementary requirements

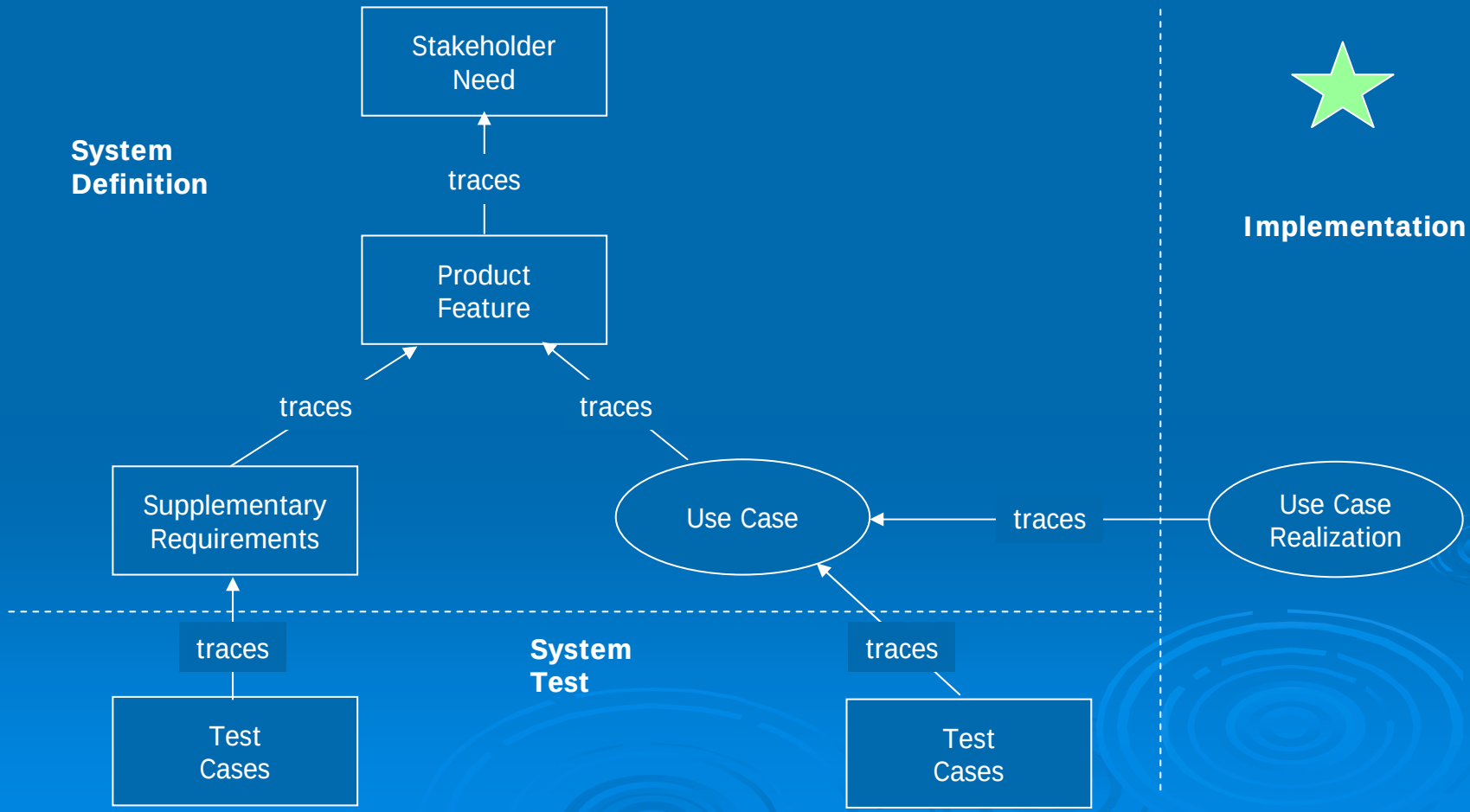
	Supplementary Requirements 1	Supplementary Requirements 2	…	Supplementary Requirements p
Feature or System Requirement 1	X			X
Feature or System Requirement 2		X		X
…		X	X	
Feature or System Requirement j				X

Traceability Matrix: Features versus supplementary requirements

Tracing Requirements in the Systems Definition Domain

- Inspecting the Traceability Matrix
  - If inspection of a *row* fails to detect an X
    - No supplementary requirement may yet be defined to respond to a feature
  - If inspection of a *column* fails to detect an X
    - A supplementary requirement has been included for which there is no known feature that requires it

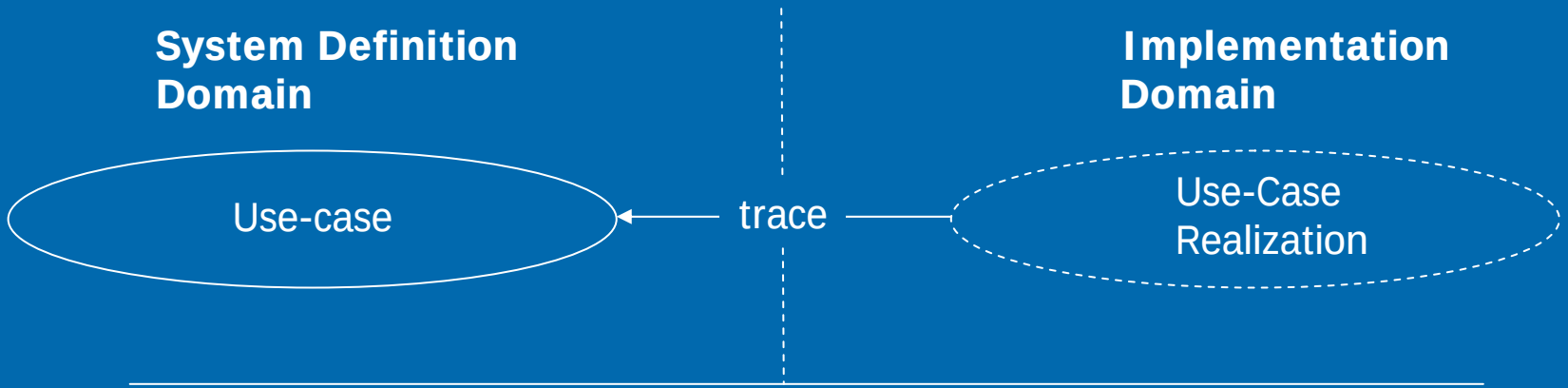
A Generalized Traceability Model



Tracing Requirements to Implementation

- Tracing requirements to code is extremely difficult and not recommended
- The following are perhaps the only pragmatic approaches
  - Tracing use cases to use-case realizations
  - Tracing from the use-case realizations into implementation
  - Tracing supplementary requirements into implementation

Tracing use cases to use-case realizations

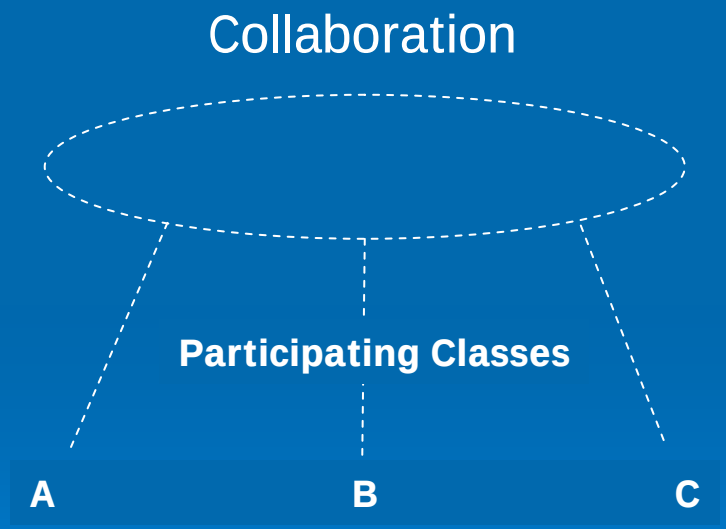


Tracing from Use Case to Use-Case Realization

- Simplified traceability problem
- One-to-one name space correspondence between a use case and its realization
- Both traceability requirements met
  - Relationship between entities is expressed directly by name sharing
  - Reason for the existence of the subordinate, or traced entity, the use case realization, is implicit in its very nature – No matrix is needed!
    - Its only purpose is to implement the use case

Tracing from the use-case realizations into implementation

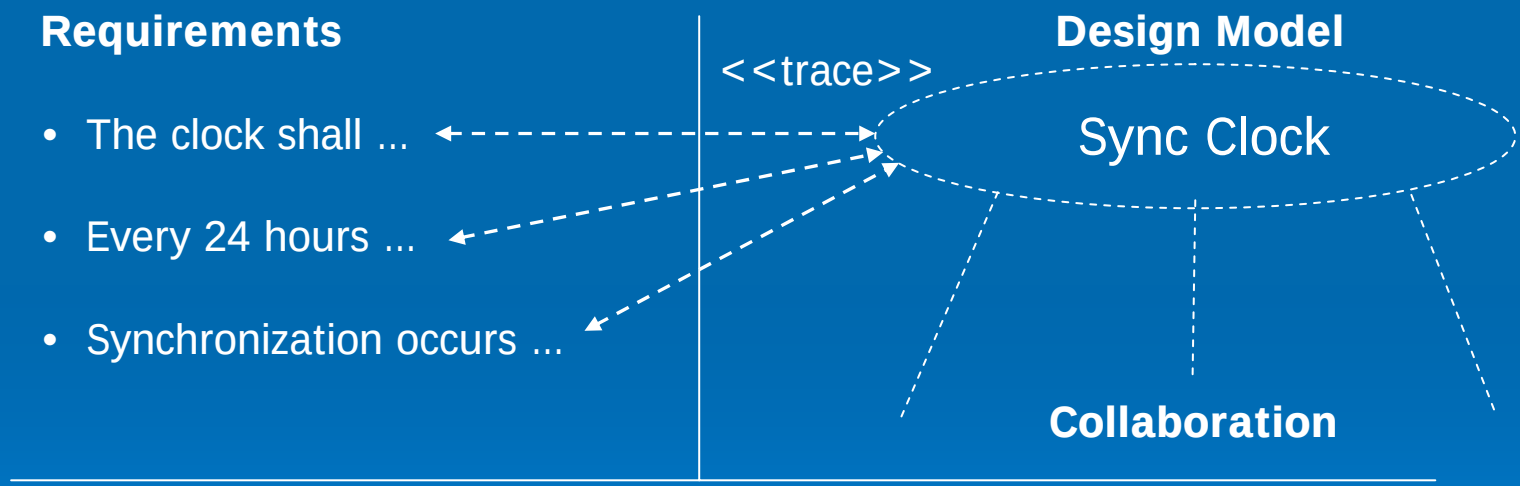
- When traceability to code is mandated:



Tracing from Use-Case Realization to Implementation

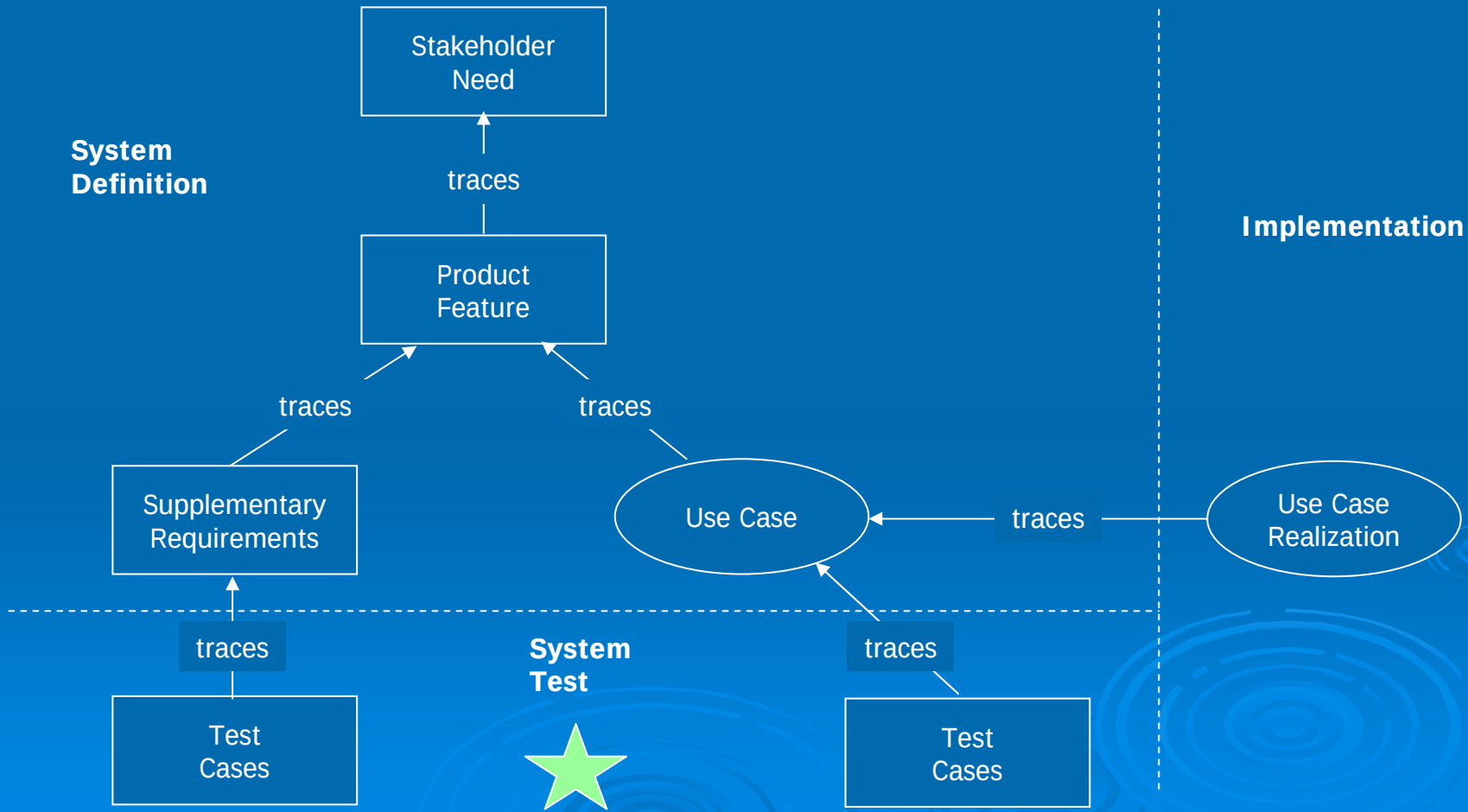
- Choice of tooling becomes important
  - You are dealing with dozens of use cases and hundreds of classes

Tracing Supplementary Requirements into Implementation



Tracing from supplemental requirements into implementation

A Generalized Traceability Model



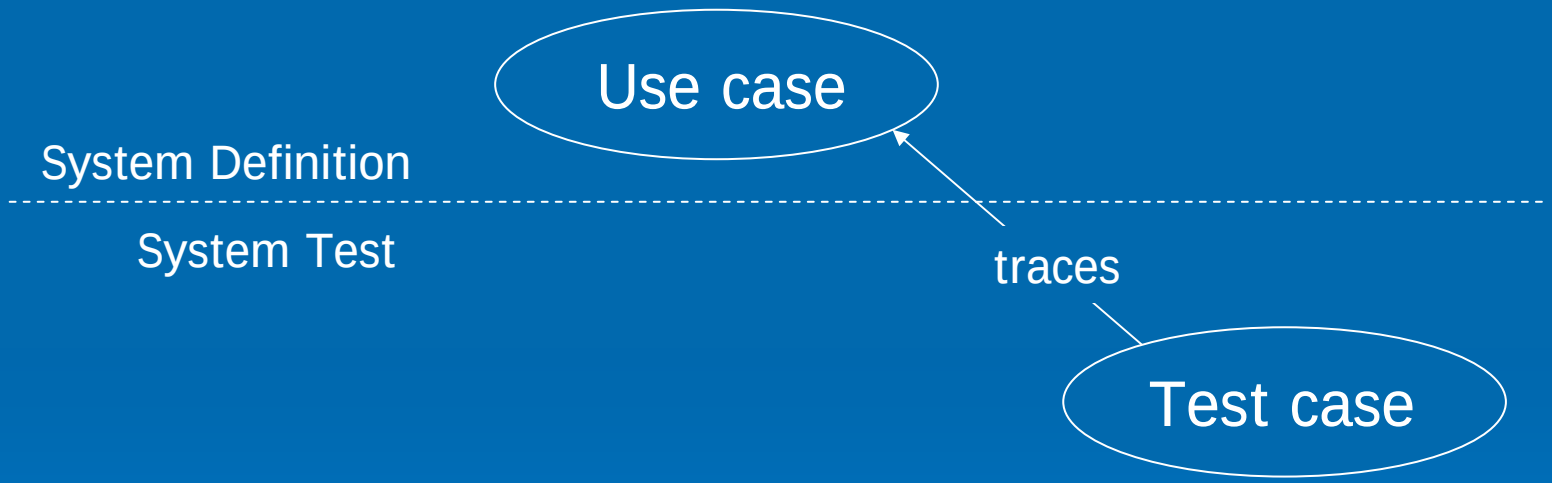
Tracing from Requirements to Testing

- Tracing from use case to test case
- Tracing from supplemental requirement to test case

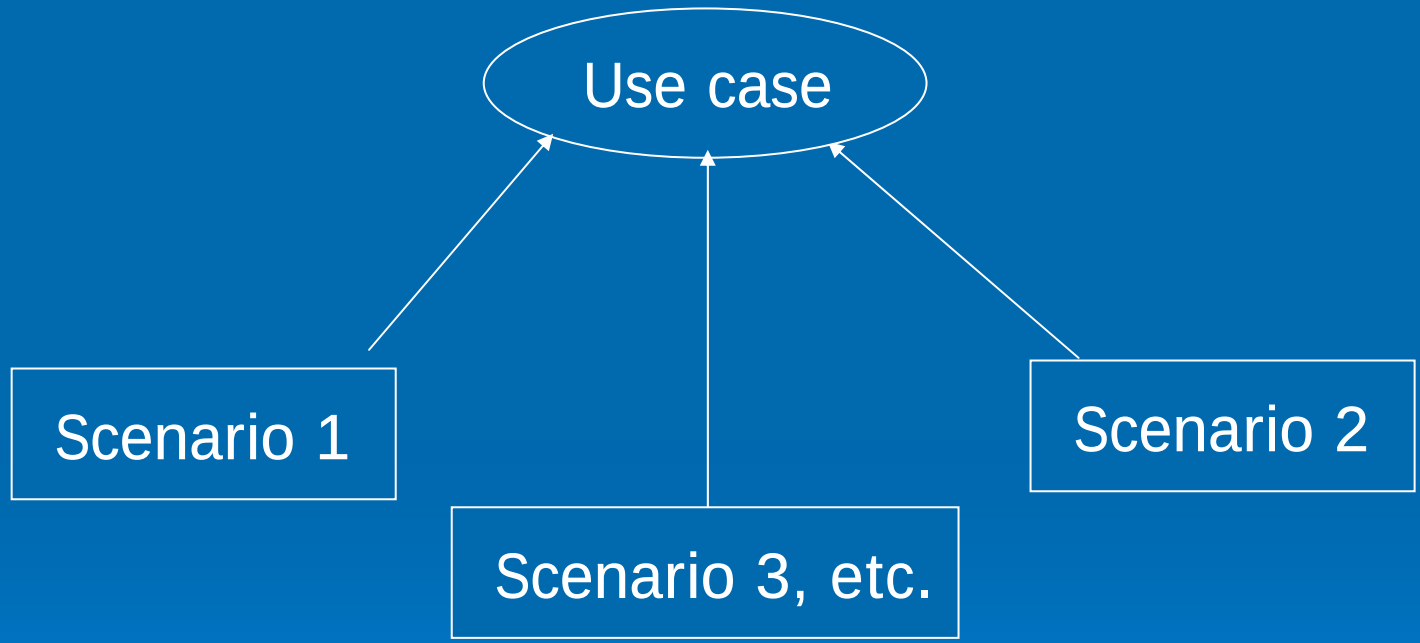
Tracing from Use Case to Test Case

- This is the bridge from the requirements domain to the testing domain
- Every use case must be tested by one or more test cases
- One to many relationship since the use case will have a variety of possible scenarios to test
  - Scenarios for a specific use case are represented by a matrix
- Further, each scenario can drive one or more test cases
  - An additional column is added to the matrix for the test cases

Tracing from Use Case to Test Case



Tracing from Use Case to Test Case

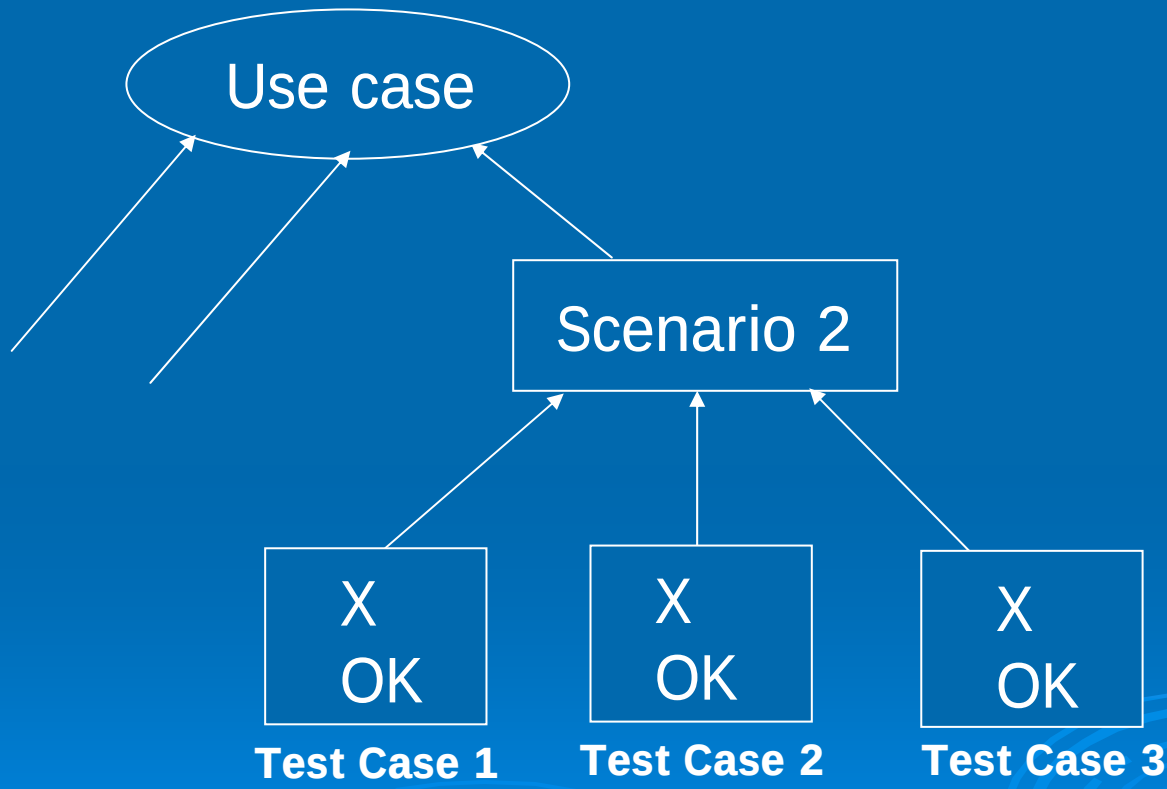


Tracing from Use Case to Test Case

Use Case	Scenario Number	Originating Flow	Alternate Flow	Next Alternate	Next Alternate
Control Light	1	Basic flow			
	2	Basic flow	Alternate Flow 1		
	3	Basic flow	Alternate Flow 1	Alternate Flow 2	
	4	Basic flow	Alternate Flow 1		
	5	Basic flow	Alternate Flow 1	Alternate Flow 1	
	6	Basic flow	Alternate Flow 1	Alternate Flow 1	Alternate Flow 2
	7	Basic flow	Alternate Flow 1		
	8	Basic flow	Alternate Flow 1	Alternate Flow 4	
Run Vacation Profile	1	Basic flow			

Traceability Matrix for Use Cases to Scenarios

Tracing from Use Case to Test Case



Tracing from Use Case to Test Case

Use Case	Scenario Number	...	Test Case ID
Control Light	1		1.1
	2		2.1
	3		3.1
	4		4.1
	4		4.2
	4		4.3
	5		5.1
	6		6.1
	7		7.1
	7		7.2
	8		8.1
Run Vacation Profile	1		1.1

Traceability Matrix for Use Cases to Test Cases

Tracing from Supplemental Requirement to Test Case

- Supplemental Requirements are either traced individually to scenarios and test cases or in groups called requirements packages
- Same matrix, except the far left column contains the requirement or requirement package

Using Traceability Tools

- Using Traceability Tools
  - Maintenance of Traceability Relationships
  - Displaying, managing and maintaining the matrices automatically
  - Implicit form of traceability
- Proceeding without Traceability Tools
  - Use spreadsheet
    - Maintenance of hierarchical relationships is difficult
  - Use a database
    - Fun, but time consuming
  - Difficult to manually manage
    - People fall into the pattern of resisting discussions to change relationships and
    - Abandon the matrices as the work becomes overwhelming
  - Not an option for large projects

Chapter 28

Managing Change

Managing Change

- External Factors
  - The problem changes
  - Users change their minds or their perceptions of what they want the system to do
  - The external environment changes which creates new constraints and/or opportunities
  - The vary existence of the new system causes the requirements to change (Yes, but…)

Managing Change

- Internal Factors
  - We failed to ask the right questions to the right people at the right time
  - We failed to create a practical process to help manage change
  - Iterating from requirements to design begets new requirements

Managing Change

- "We Have Met the Enemy, and They Is Us"
  - Enhancements mentioned by distributor who had been overheard by programmers at a sales convention
  - Direct customer requests by programmers
  - Mistakes that had been made and shipped and had to be supported
  - Hardware features that didn’t get in or didn’t work
  - Knee-jerk change of scope reactions to competitors
  - Functionality inserted by programmers with “careful consideration” of what is good for the customer
  - Programmer’s “Easter Eggs” (See Page 341)

Managing Change

- A Process for Managing Change
  - Step 1: Recognize that Change Is Inevitable, and Plan for It
  - Step 2: Baseline the Requirements
  - Step 3: Establish A Single Channel to Control Change
  - Step 4: Use a Change Control System to Capture Changes
  - Step 5: Manage Change Hierarchically

Managing Change

- Requirements Configuration Management
  - Tool-based Support for Change Management
  - Elements Impacted by Change
  - Audit Trail of Change History
  - Configuration Management and Change Management
    - Finest level of detail – all use cases and requirements
    - Middle level of detail – All documentation
    - General level of detail – change history for the project

Chapter 29

Assessing Requirements Quality in Iterative Development

Software Project Quality

- Purpose of the text:
  - To help assure that the results you produce have the requisite quality to meet the needs of your users and extended stakeholders
- RUP Defines Quality as:
  - The characteristic of having demonstrated the achievement of having produced a product that meets or exceeds agreed-on requirements-as measured by agreed-on measures and criteria-an that is produced by an agreed-on process

Assessing Quality in Iterative Development

- Iterative development in itself is a primary quality measure
  - Does it do what we said it would?
  - Does it appear to meet the requirements?
  - Did we do it about when we said we would?
  - Now that you can see a bit of the thing, is this what you really wanted?

Requirements Artifacts Sets

- Artifacts that the developer produces
  - Review Table 29-1, Pg. 359
- Assessing the completeness of artifacts by looking at the various aspects of quality that each artifact should contain at that point in the development process
- Use the checklist for your assessment
  - Table 29-1 thru 29-7, Pg. 361-8

Team Skill 6 Summary

- From Use Cases to Implementation
- From Use Cases to Test Cases
- Tracing Requirements
- Managing Change
- Assessing Requirements Quality in Iterative Development

Getting Started

- What you have learned so far
  - Team Skill 1: Analyzing the Problem
  - Team Skill 2: Understanding User Needs
  - Team Skill 3: Defining the System
  - Team Skill 4: Managing Scope
  - Team Skill 5: Refining the System Definition
  - Team Skill 6: Building the Right System
- Chapter 30: Agile Requirements Methods
- Chapter 31: Your Prescription for Agile Requirements Management

Chapter 30: Agile Requirements Methods

Chapter 31: Your Prescription for Agile Requirements Management

Agile Requirements Methods

- This section of the text paints a one-sided argument for agile methodologies
- Review this chapter with the understanding that RUP is the process of choice
- Present the Agile presentation here
- Present the XP Presentation here

Building the Right System Right: Overview

- Continually Confirm that the Development Is on Track
- Principles of Software Verification
- The Cost of Verification
- Verification at All Levels
- The Reason for Verification
- Confirm that the Development Results Are Correct
- Learn How to Cope with Change that Occurs During the Development Process

Quality Measures of Software Requirements

- Nine Quality Measures
- Correct Requirements
- Unambiguous Requirements
- Completeness of the Requirement Set
- Consistency in the Requirement Set
- Requirements Ranked for Importance and Stability
- Verifiable Requirement
- Modifiable Requirements Set
- Traceable Requirements
- Understandable Requirements
- Quality Measures for the Use-Case Model
- Use Case Specifications
- Use Case Actors
- Quality Measures of the Modern SRS Package
- A Good Table of Contents
- A Good Index
- A Revision History
- A Glossary

Validating the System

- Validation
- Validation Testing
- Validation Traceability
- Requirements-based Testing
- Case Study: Testing Use Cases
- Test Case 1 Description
- Tracing Test Cases
- Testing Discrete Requirements
- Omitted Validation Relationships
- Excess Validation Relationships
- Testing Design Constraints

Using ROI to Determine the V&V Effort

- Depth versus Coverage
- V&V Coverage
- What to Verify and Validate
- Option 1: Verify and Validate Everything
- Option 2: Use a Hazard Analysis to Determine V&V Necessities
- Hazard Analysis as Return On Investment (ROI)

Role Playing

- Getting to the root cause
  - Fishbone Diagram
  - Developer act as the user
    - Sit down and do what the user does
- Techniques Similar to Role Playing
  - Scripted Walkthroughs
    - Write script of function
    - Developers and customers play the role of users
    - They perform a scripted function on the prototype

Role Playing

- CRC (Class-Responsibility-Collaboration) Cards
  - Objected-oriented
  - Each participant has a set of cards describing
    - Class or object
    - Responsibilities or behaviors
    - Collaborations, or who the object communicates with, of each entity being modeled
      - Entities from the problem domain
      - Users, bush-buttons, lights, objects, etc.
    - As the initiator actor starts, each participant follows the behavior on cards

Prototyping Categories

- Types of Prototypes
  - Architectural Prototype
    - Used to establish feasibility of a technology
    - Throwaway or Evolutionary
  - Requirements Prototype
    - Use the fastest cheapest technology
      - Mainly for GUI development
    - Throwaway or Evolutionary
- Investment Strategy
  - Throwaway
    - To gain knowledge
  - Evolutionary
    - Will be used in the final system

Requirements Prototyping

- A partial implementation of a software system, built to help developers, users, and customers, better understand the requirements of the system
  - A Horizontal prototype has a wide range of functionality to a shallow depth
  - A Vertical prototype has a narrow range of functionality built to great depth and quality
  - A User Interface prototype implements the interface without the background logic or algorithms
- Build in a manner that consumes the fewest resources

Requirements Prototype

- Built by the developer, it can be used
  - To confirm customer understands requirements
  - As a catalyst to elicit more requirements
- Built by the customer, it can be used
  - To communicate requirements to developer
- What to Prototype
  - When requirements are well understood – No
  - When requirements are vague or unknown – Yes
    - Undiscovered ruins

Evaluating the Results of Prototyping

- Fuzzy becomes better understood
- Elicits a “Yes, But” response from the user
  - Unknown needs become known
- Summary
  - Select prototype based on project risk
  - Requirements prototype
    - Cheap and easy to make
    - Helps eliminate requirements risk