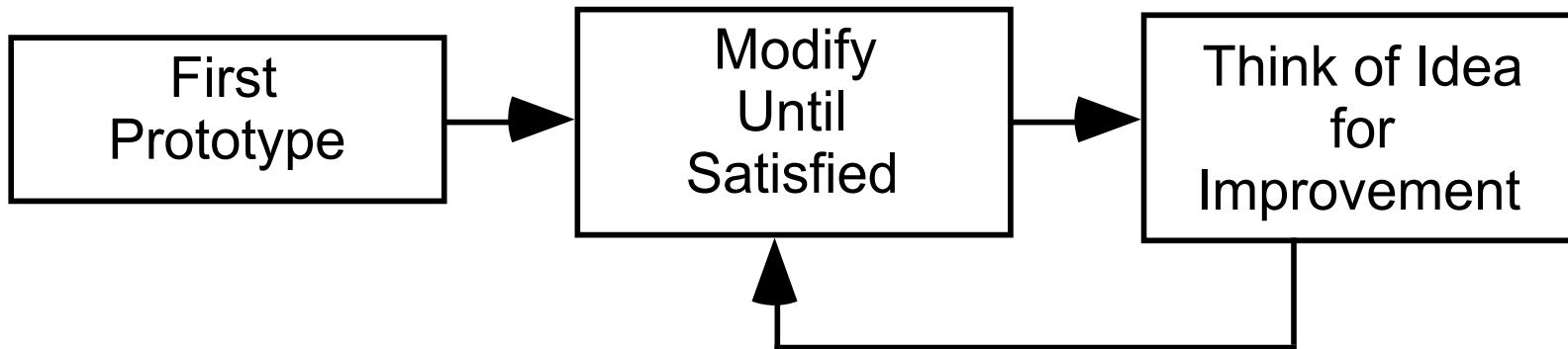


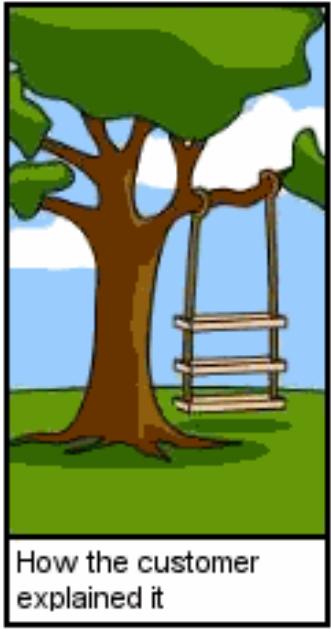
Software Development Life Cycle

DASS, Monsoon 2024
IIIT Hyderabad

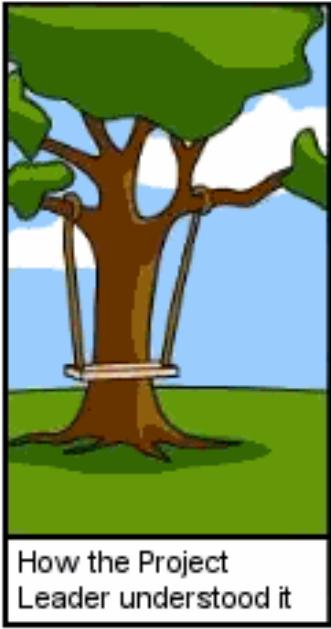
The Opportunistic approach



- OK for small, informal projects
- Inappropriate for professional environments/complex software where on-time delivery and high quality are expected



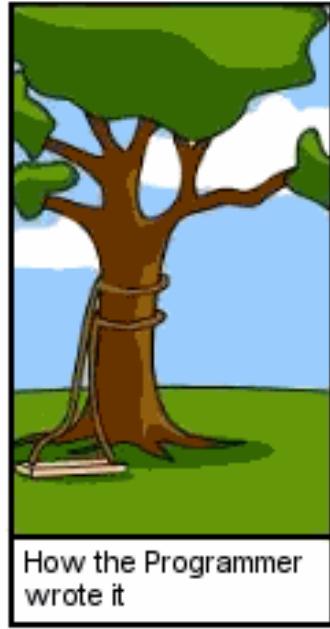
How the customer explained it



How the Project Leader understood it



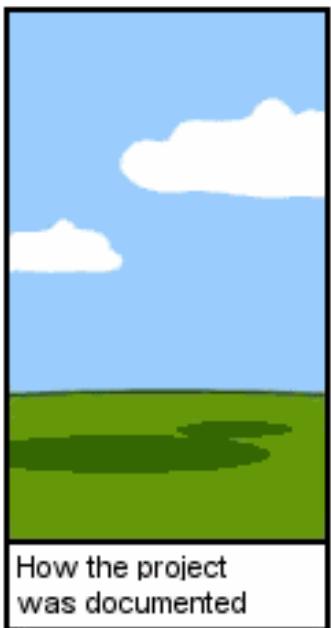
How the Analyst designed it



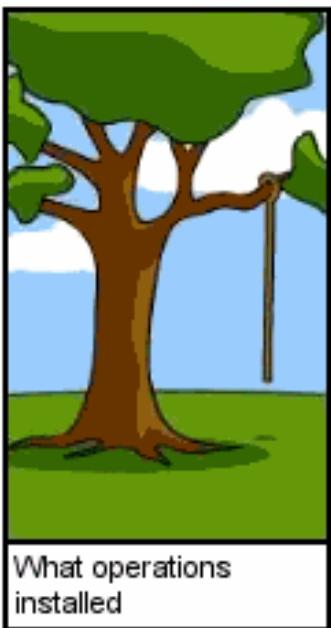
How the Programmer wrote it



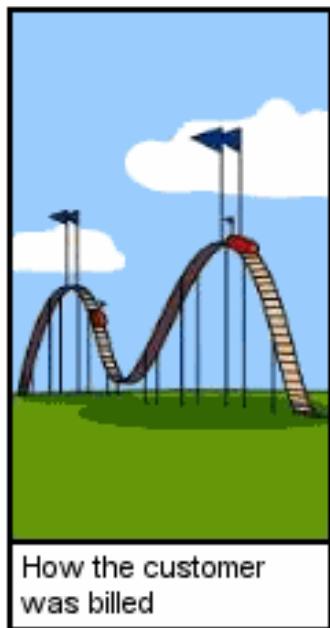
How the Business Consultant described it



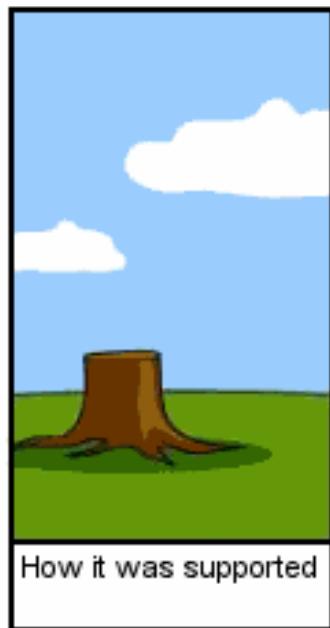
How the project was documented



What operations installed



How the customer was billed



How it was supported



What the customer really needed

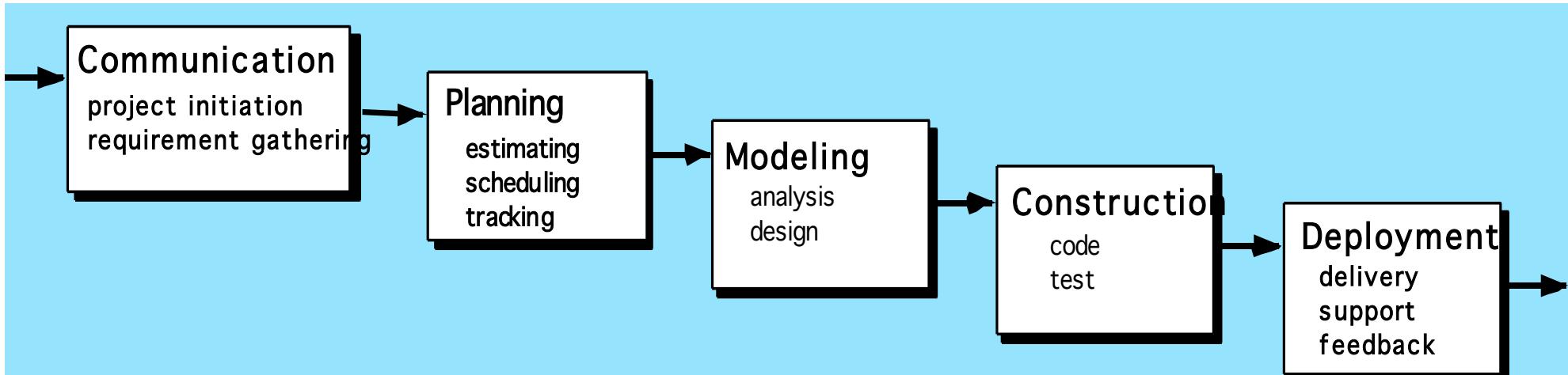
Why Lifecycle Model?

- A software project will never succeed if activities are not coordinated:
 - one engineer starts writing code,
 - another concentrates on writing the test document first,
 - yet another engineer first defines the file structure
 - another defines the I/O for his portion first

A software life cycle model (or process model):

- a descriptive and diagrammatic model of software life cycle:
- identifies all the activities required for product development
- establishes a precedence ordering among the different activities
- divides life cycle into phases.

Software Development Life Cycle (SDLC)



SDLC (simplified):

- Feasibility study/Planning (involves business case)
- Requirements Engineering
- Architecture/Design
- Implementation
- Testing
- Maintenance

Requirements Engineering

- Aim of this phase:

- understand the exact requirements of the customer,
- document them properly.

- Consists of following activities:

- Elicitation/Gathering
- Analysis
- Specification
- Verification
- Management

Design

- Design phase transforms requirements specification:
 - into a form suitable for implementation in some programming language.

High-level design (Architecture):

- decompose the system into *modules*,
- represent invocation relationships among the modules.

Detailed design:

- different modules designed in greater detail:
 - data structures and algorithms for each module are designed

IMPLEMENTATION

- During the implementation phase:
 - each module of the design is coded,

The end product of implementation phase:

- a set of program modules that have been tested individually.

INTEGRATION AND SYSTEM TESTING

- Different modules are integrated in a planned manner:
 - modules are almost never integrated in one shot.
 - Normally integration is carried out through a number of steps.
- During each integration step,
 - the partially integrated system is tested.

INTEGRATION AND SYSTEM TESTING

M1

M8

M2

M5

M7

M3

M4

M6

SYSTEM TESTING

- After all the modules have been successfully integrated and tested:
 - system testing is carried out.
- Goal of system testing:
 - ensure that the developed system functions according to its requirements as specified in the SRS document.

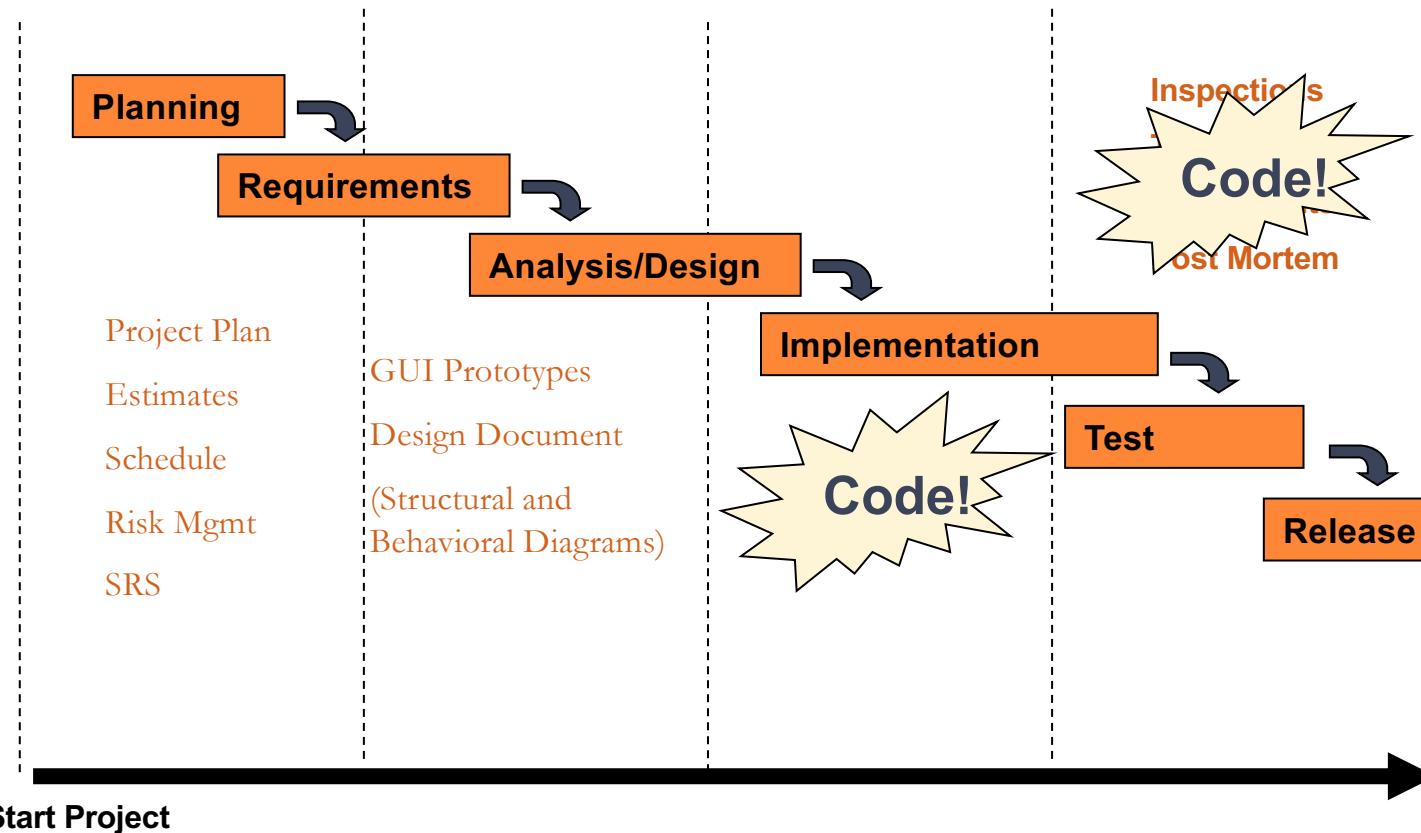
V Model – Popular lifecycle model

MAINTENANCE

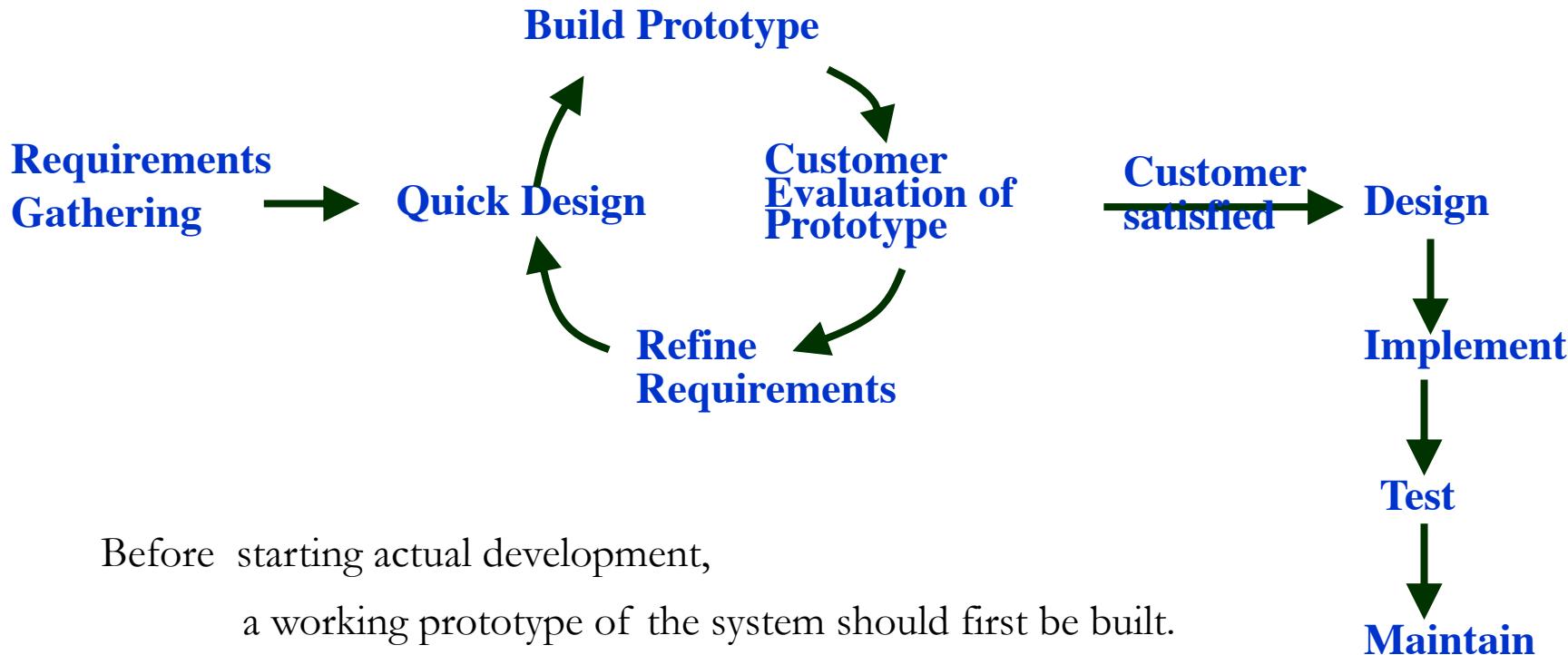
- Preventive maintenance
 - Making appropriate changes to prevent the occurrence of errors
- Corrective maintenance
 - Correct errors which were not discovered during the product development phases
- Perfective maintenance
 - Improve implementation of the system
 - enhance functionalities of the system
- Adaptive maintenance
 - Port software to a new environment

Process Models

TRADITIONAL SDLC. (E.G. WATERFALL PROCESS)



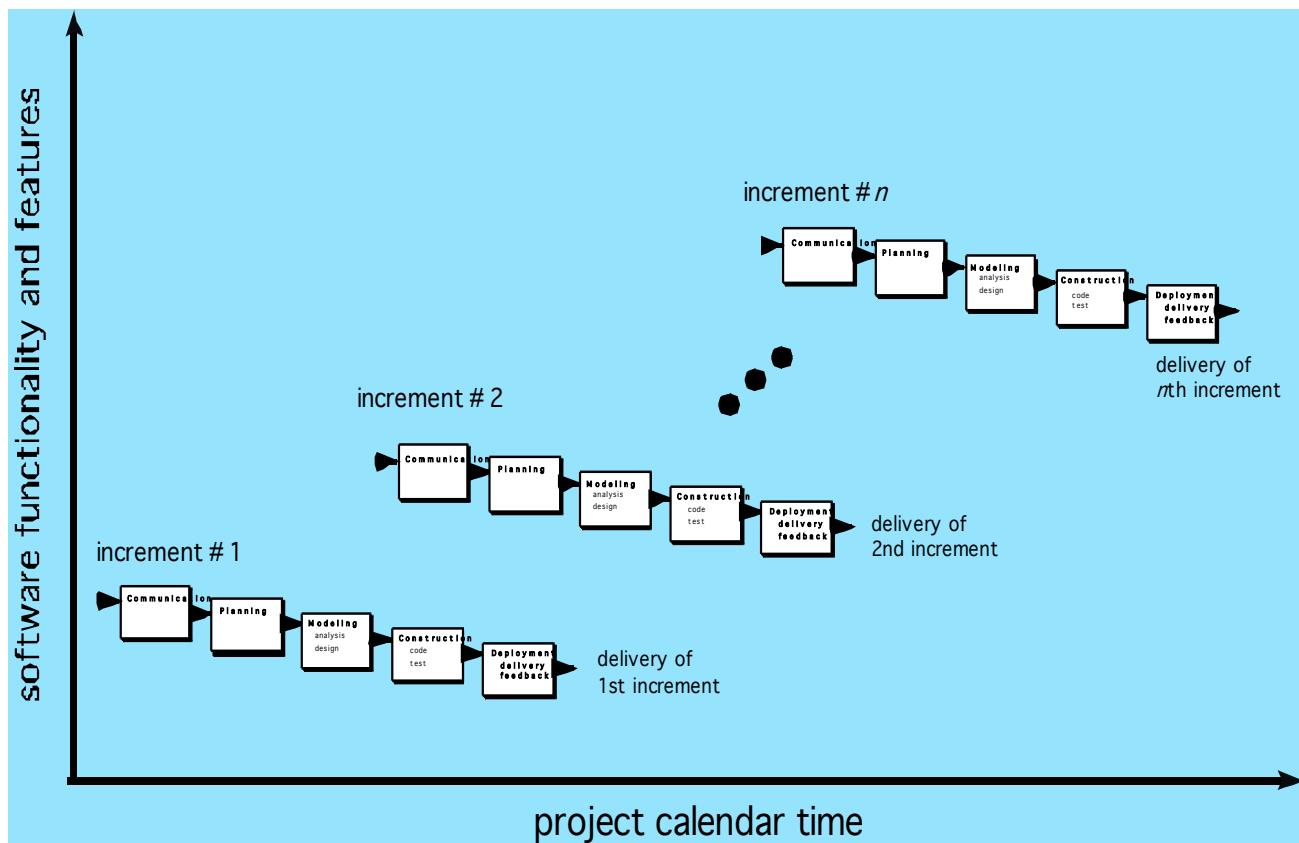
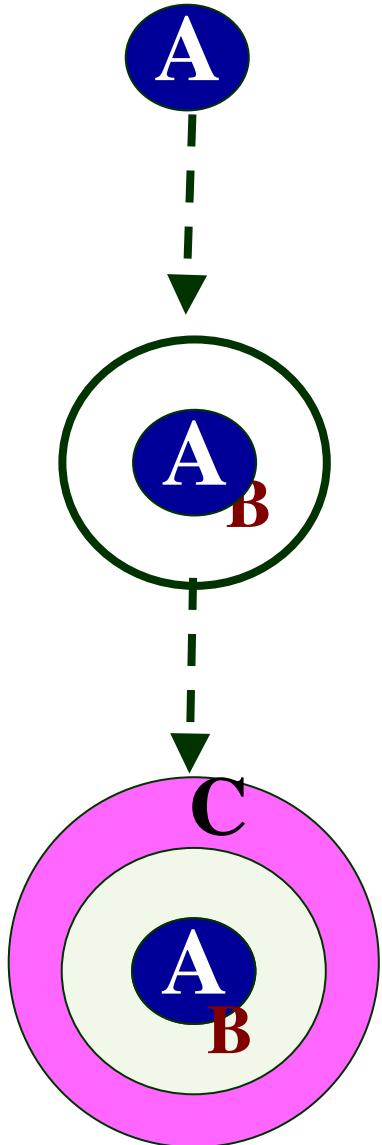
PROTOTYPING MODEL



A prototype is a toy implementation of a system:

- limited functional capabilities,
- low reliability,
- inefficient performance.

INCREMENTAL MODEL



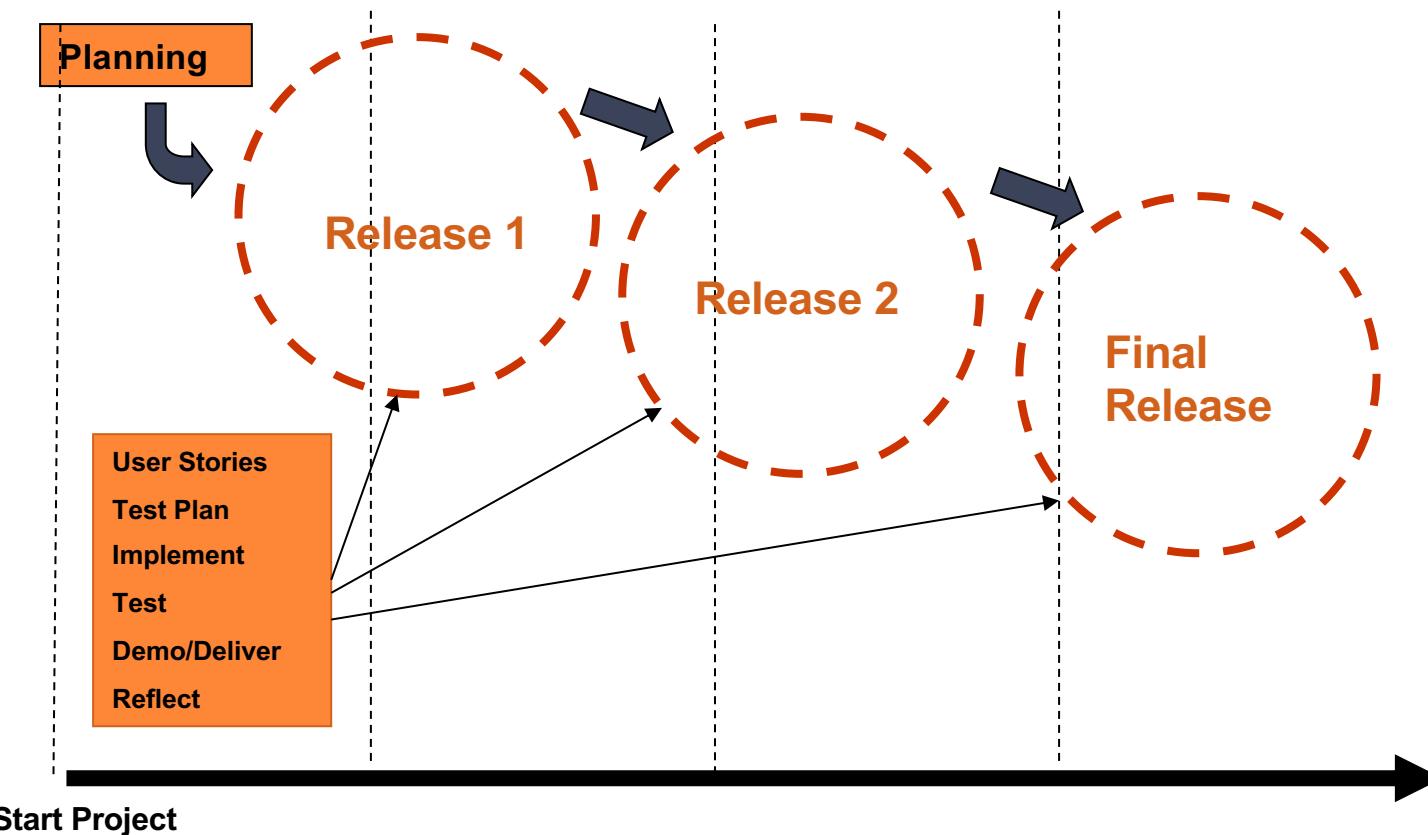
CHALLENGES WITH TRADITIONAL (PLAN-DRIVEN) APPROACHES

- Lightweight applications/heavyweight process
- Document intensive (perceived)
- Less flexible design
- Big bang approach to coding/integration
- Testing short-shifted
- One-shot delivery opportunity
- Limited opportunity for process improvement

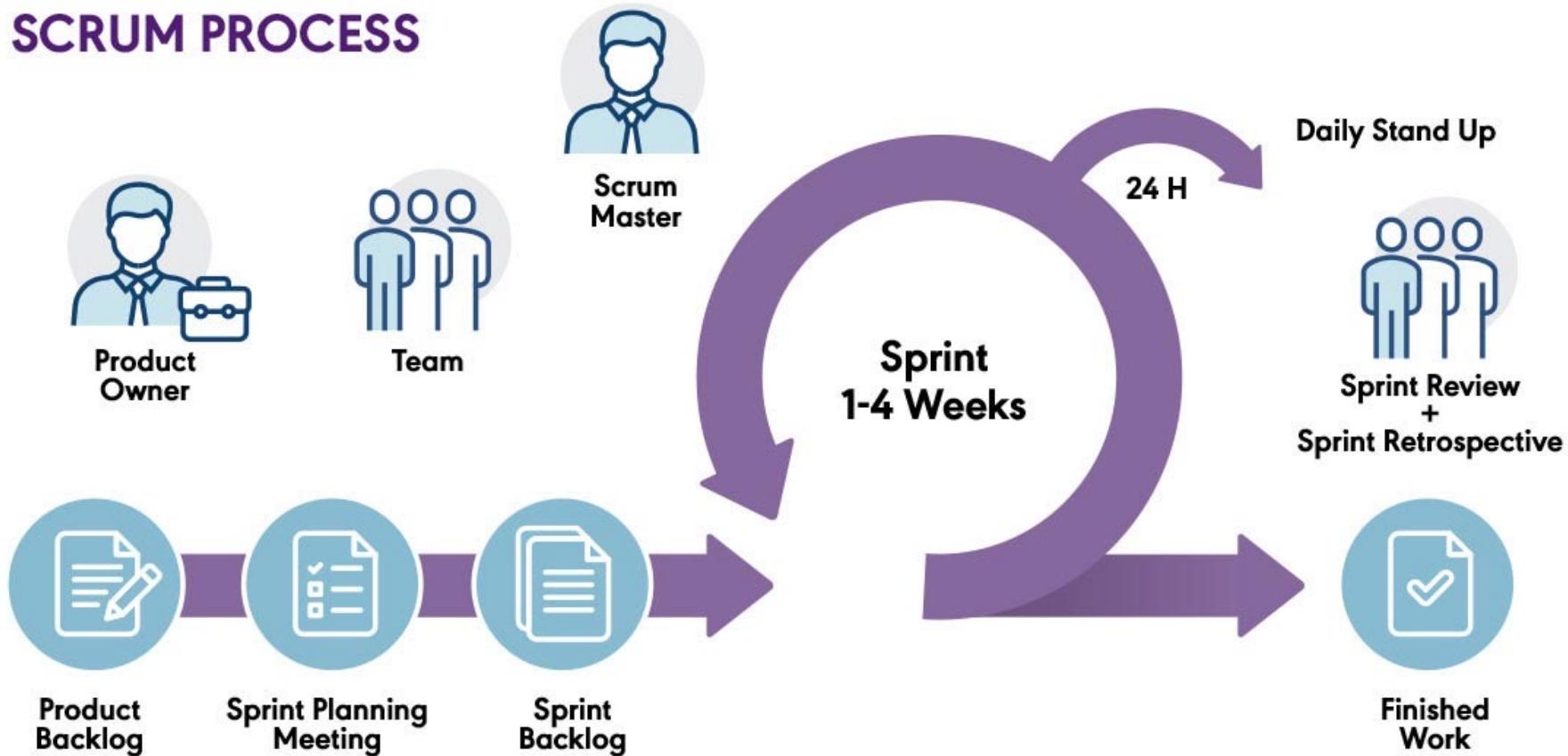
WHAT IS AGILE SOFTWARE DEVELOPMENT?

- In the late 1990's several methodologies began to get increasing public attention. All emphasized:
 - Close collaboration between developers and business experts
 - Face-to-face communication (as more efficient than written documentation)
 - Frequent delivery of new deployable business value
 - Tight, self-organizing teams
 - Ways to craft the code and the team such that the inevitable requirements churn was not a crisis.
- 2001 : Workshop in Snowbird, Utah, Practitioners of these methodologies met to figure out just what it was they had in common. They picked the word "**agile**" for an umbrella term and crafted the
 - Manifesto for Agile Software Development,

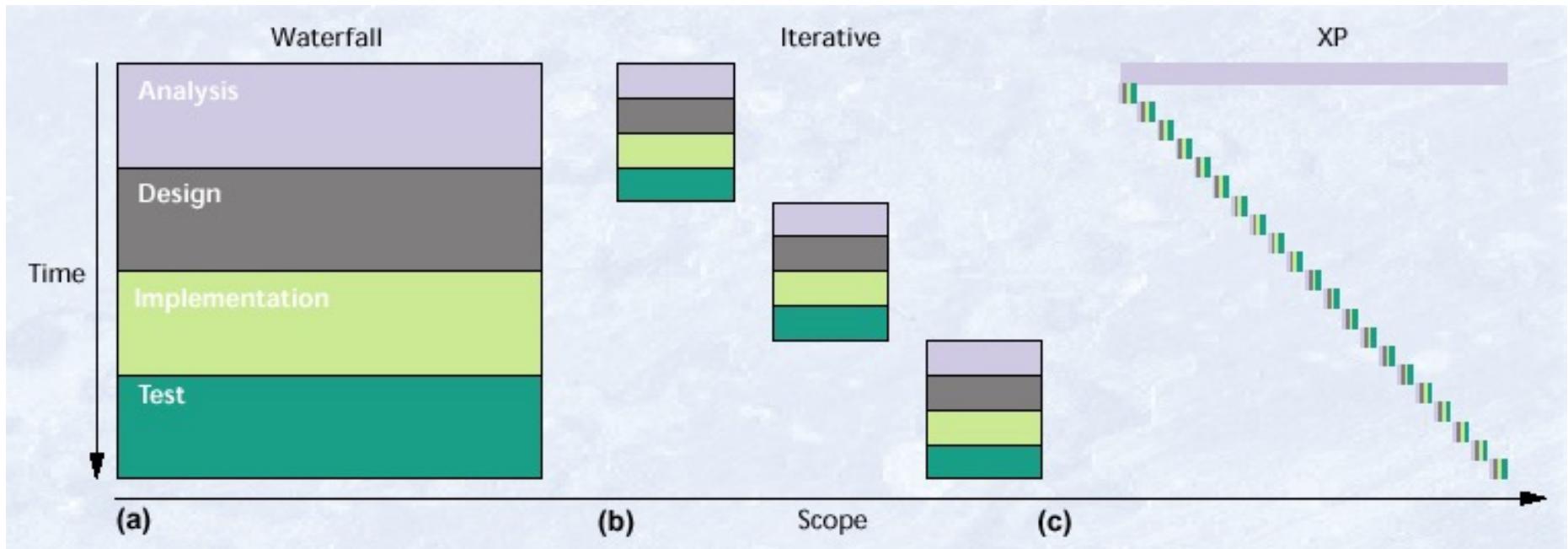
APPLYING AGILITY



SCRUM PROCESS

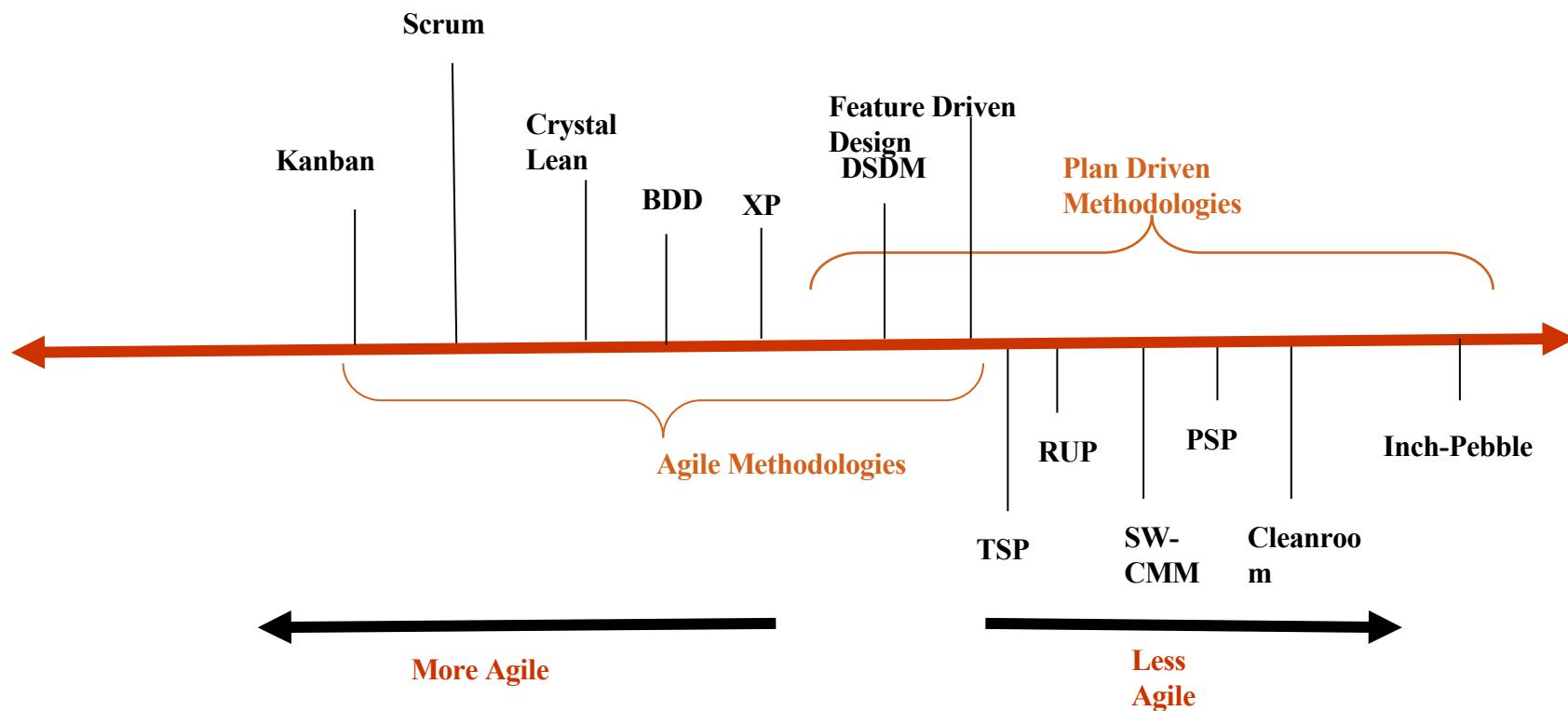


WATERFALL TO XP EVOLUTION



Source: "Embracing change with extreme programming" by Kent Beck, *IEEE Computer*, October 1999.

THE PROCESS METHODOLOGY SPECTRUM



It's not that black and white. The process spectrum spans a range of grey !

AGILE CHARACTERISTICS

- Incremental development – several releases
- Planning based on user stories
- Each iteration touches all life-cycle activities
- Testing – unit testing for deliverables; acceptance tests for each release
- Flexible Design – evolution vs. big upfront effort
- Reflection after each release cycle
- Several technical and customer focused presentation opportunities

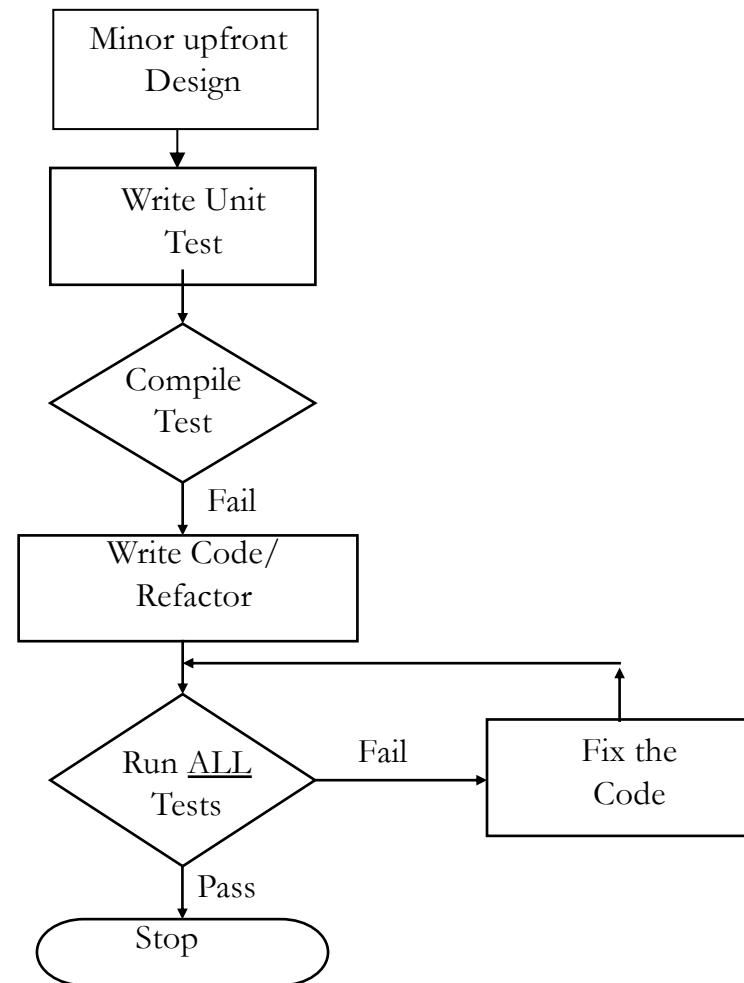
KEY AGILE COMPONENTS

- User Stories
 - Requirements elicitation
 - Planning – scope & composition
- Evolutionary Design
 - Opportunity to make mistakes
- **Test driven development**
 - Dispels notion of testing as an end of cycle activity
- **Continuous Integration**
 - Code (small booms vs big bang)
- **Refactoring**
 - Small changes to code base to maintain design entropy
- Team Skills
 - Collaborative Development (Pair programming)
 - Reflections (process improvement)
- Communication/shared ownership
 - Interacting with customer / team members

TEST DRIVEN DEVELOPMENT (TDD)

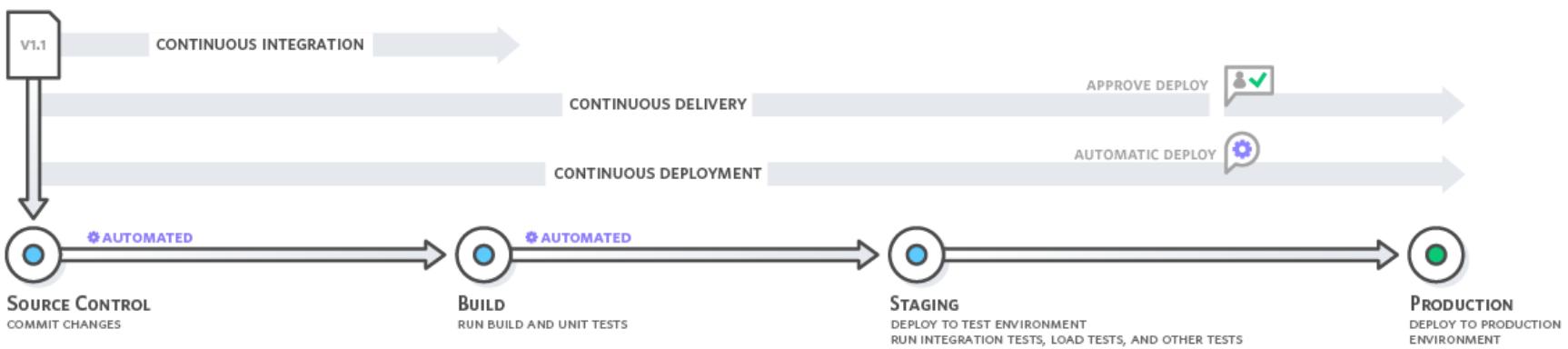
- “State of the art” → test-last
- “State of the practice” → test-whenever needed
- TDD → test-first
- Design evolves through coding/feedback
- Write unit tests for every piece of code that could possibly break
- Preferred testing tool are xUnits (open source)

TDD EXPLAINED



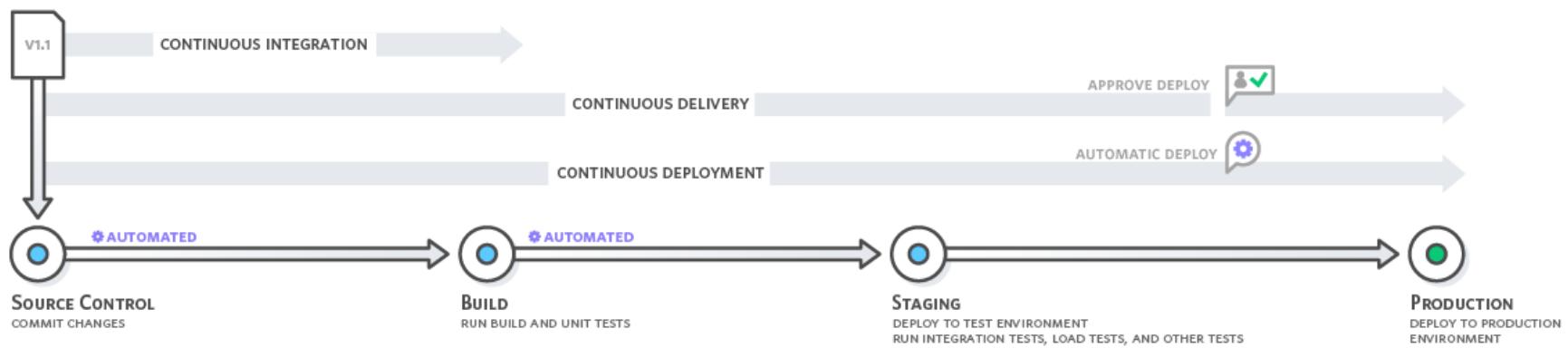
CONTINUOUS INTEGRATION

- Developers merge their code into the central repository regularly
- Automated builds and testing



CONTINUOUS DELIVERY

- Deploys all code changes to a testing and/or production environment after the build stage
- Deployment is manual



DEVELOPMENT PROBLEMS ADDRESSED – WHAT ABOUT RELEASE PROBLEMS ?

- Database issues
- OS issues
- Too slow in real settings
- Infrastructure issues
- Source from many repositories
- Different versions (libraries, compilers, local utilities, etc)
- Missing dependencies
- ...

Developers

- Designing
- Coding
- Testing, bug tracking, reviews
- Continuous Integration
- ...

Operations



Managing/Allocating
hardware/OS
updates/resources,
database



Monitoring
load spikes,
performance,
crashes
hardware
updates



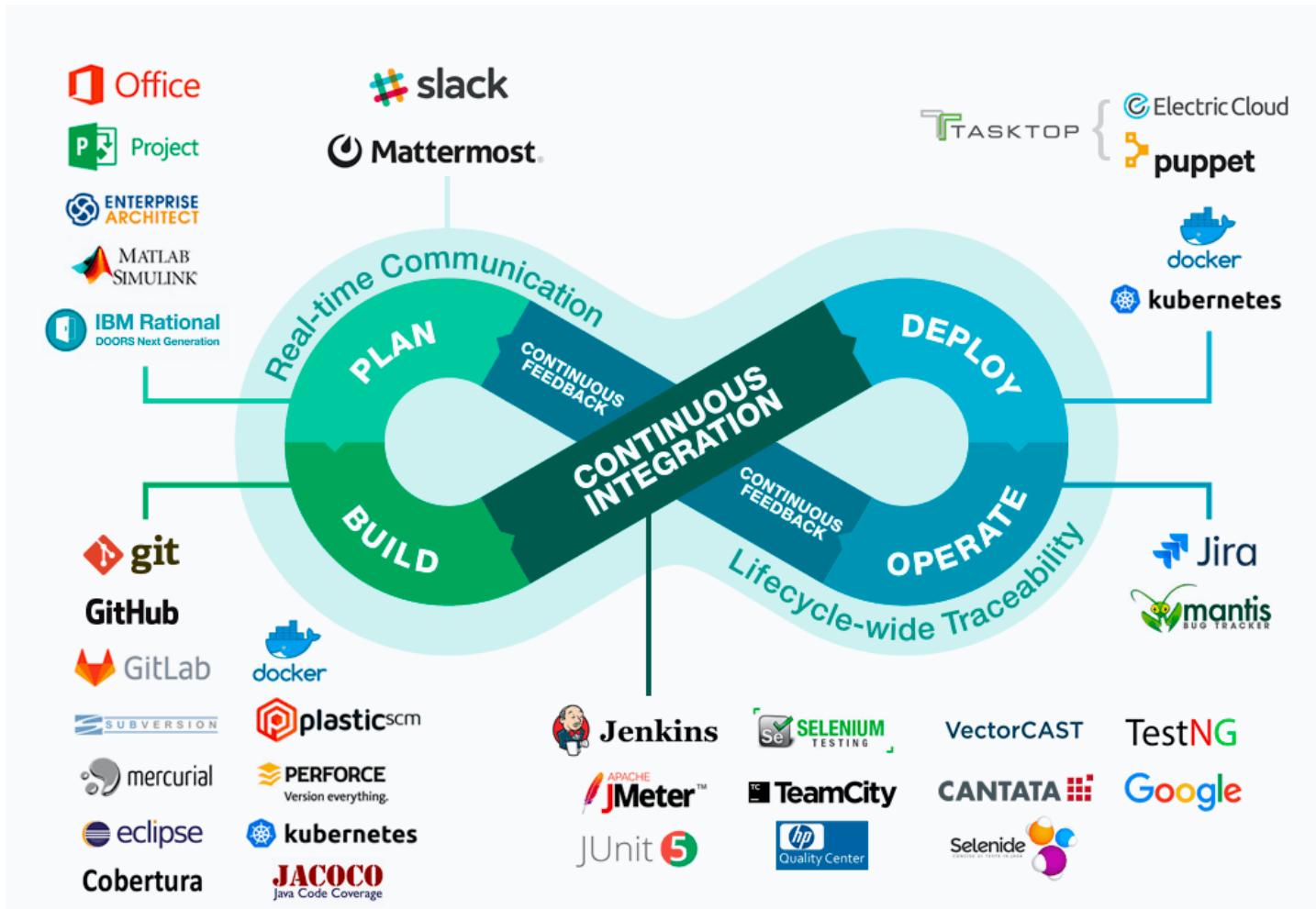
Backups,
Rollback
releases



etc.

- ✓ Can there be better coordination between Developers and Operators?
- ✓ Reduce issues while moving changes from development to production
- ✓ Configurations as code
- ✓ Automation (Delivery and Monitoring)

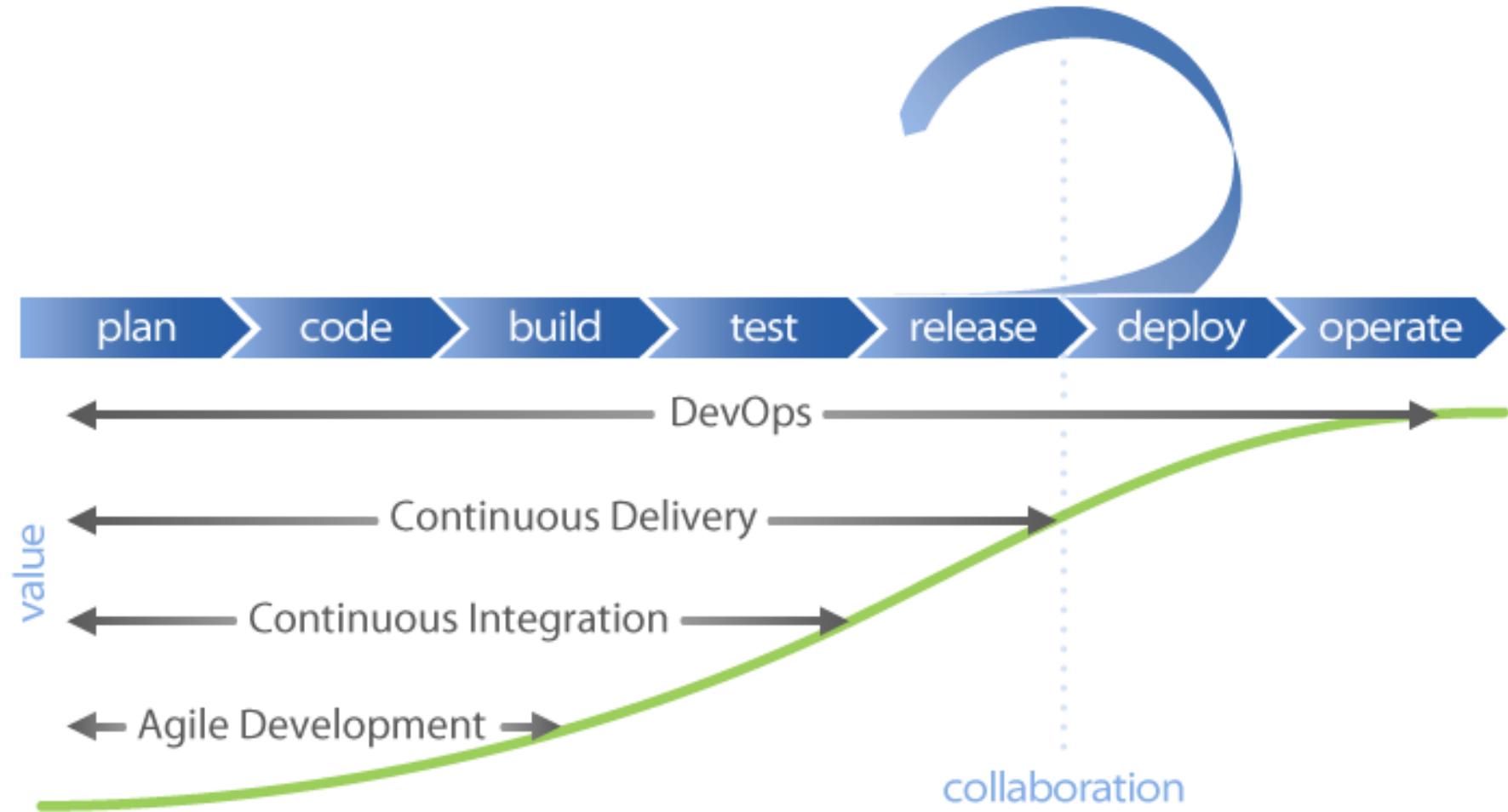
DEVOPS



DEVOPS – COMMON PRACTICES

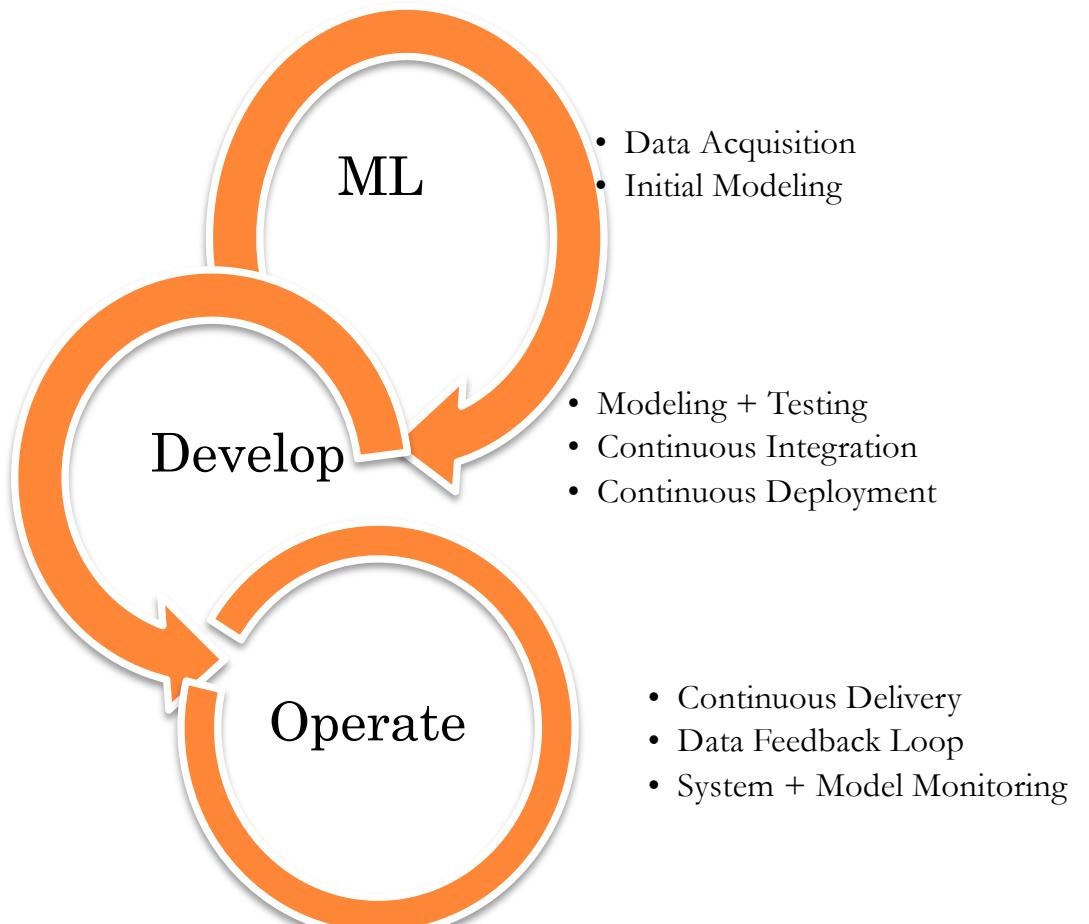
- Continuous Integration
- Continuous Delivery
- Infrastructure as code, test and deploy in containers
- Monitoring and logging
- Microservice architecture
- Communicate and Collaborate

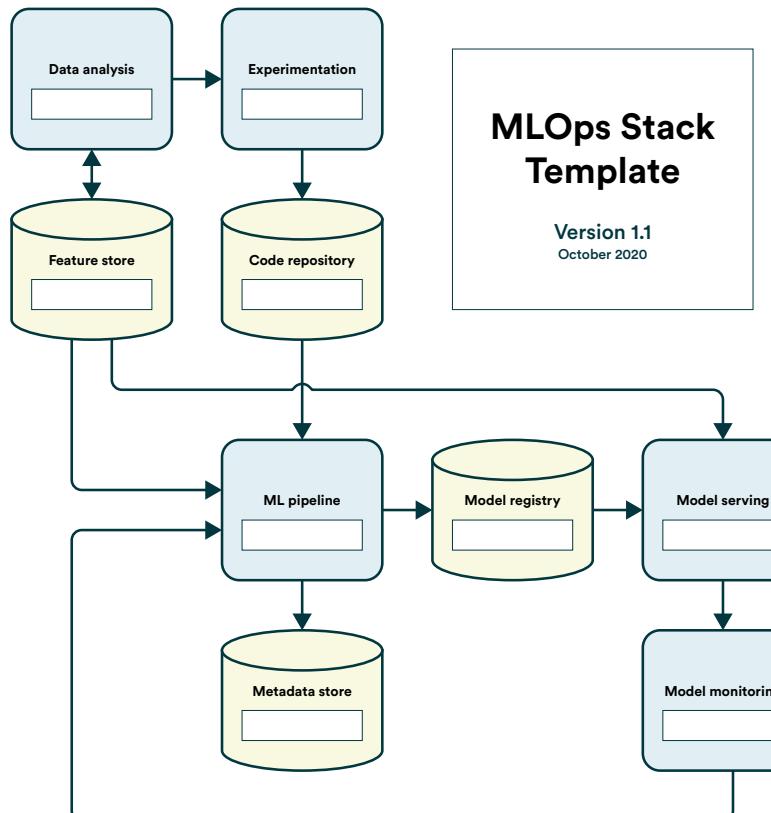
AGILE, CI, CD, DEVOPS...



MLOPS

- ML + Dev + Ops
- Adopts best practices of DevOps.
- Allows for experimentation (Canary Releases)



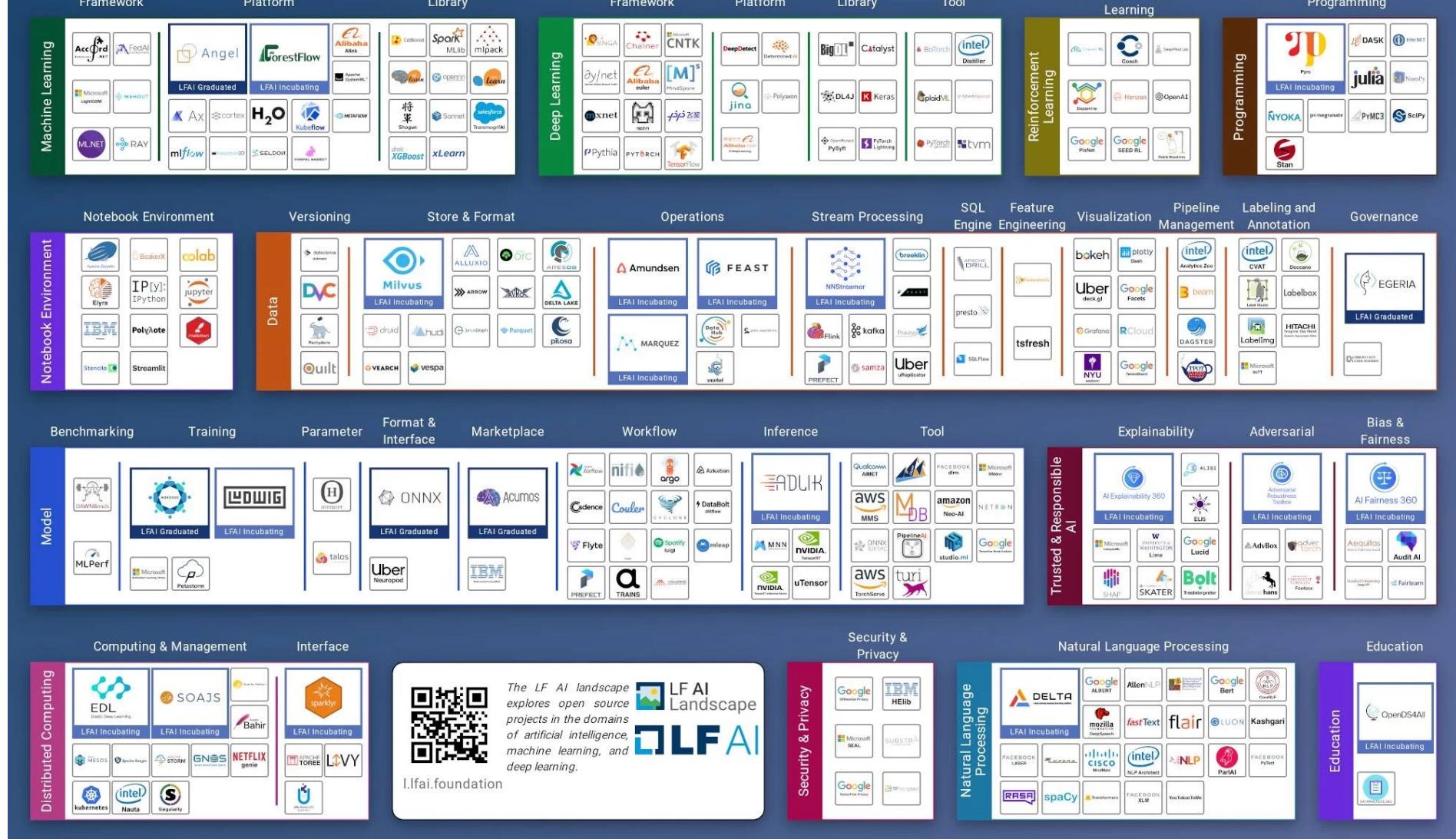


Component	Requirements	Tooling
Data analysis		
Experimentation		
Feature store		
Code repository		
ML pipeline		
Metadata store		
Model registry		
Model serving		
Model monitoring		

MLOps Setup Components

Tools
Python, Pandas
Git
PyTest & Make
Git, DVC
DVC [aws s3]
Project code library
DVC
DVC & Make

Source: ml-ops.org



Estimation, Scheduling & Tracking (Week 3)

Software Estimation

Some content adapted from “Rapid Development” by Steve McConnell



Can you estimate these?

1. Surface temperature of the sun (in degrees C)
2. Latitude of Hyderabad (in degrees)
3. Surface area of Asia (in km^2)
4. Birth date of Alexander The Great (year)
5. Global revenue of “Titanic” (in \$)
6. Length of the Pacific coastline (Ca, Or, Wa) (in km)
7. Weight of the largest whale (in tonnes)

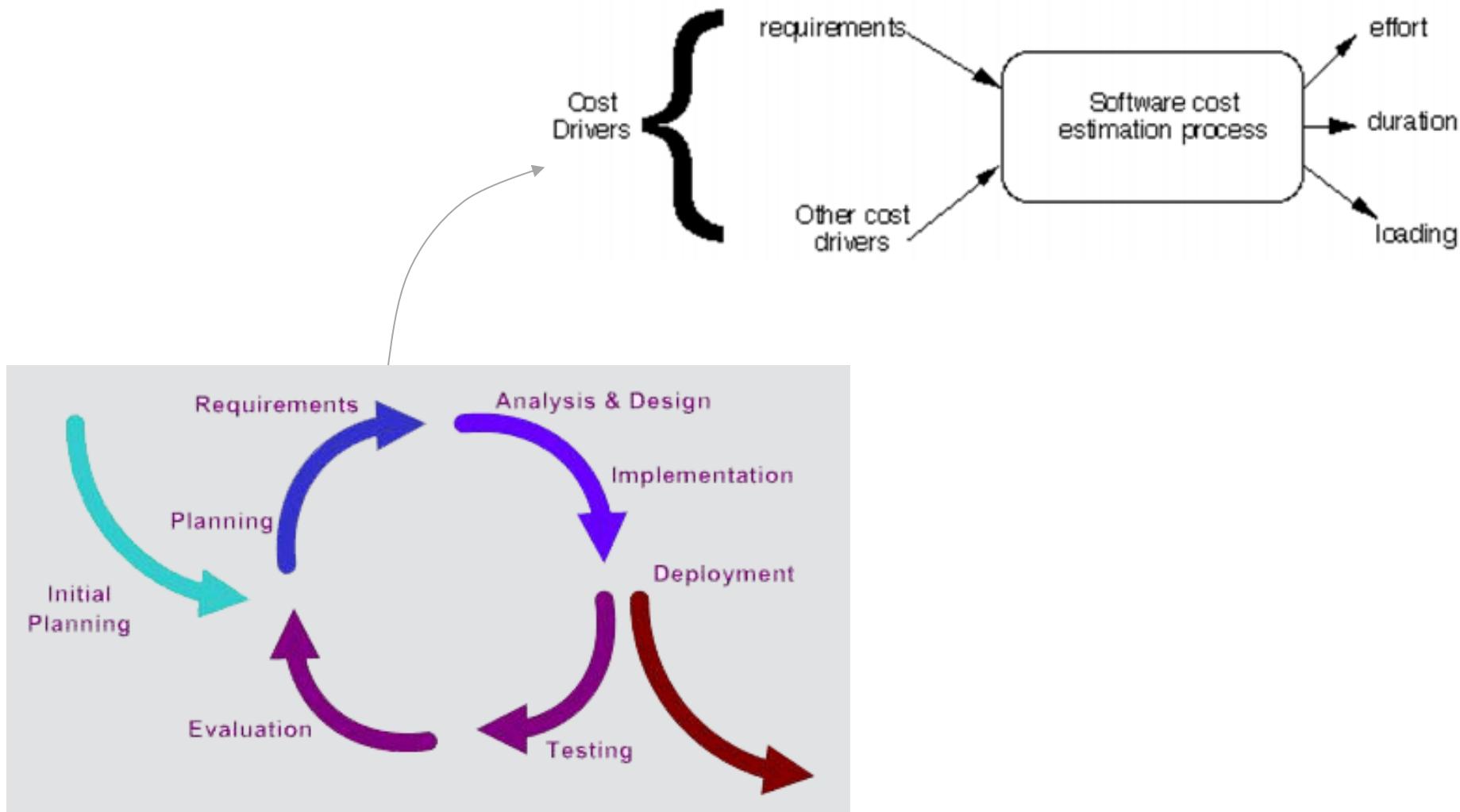


1

Always give a range
Never give them a number



Approach



#2

Always ask what the estimate will
be used for



Requirements is key



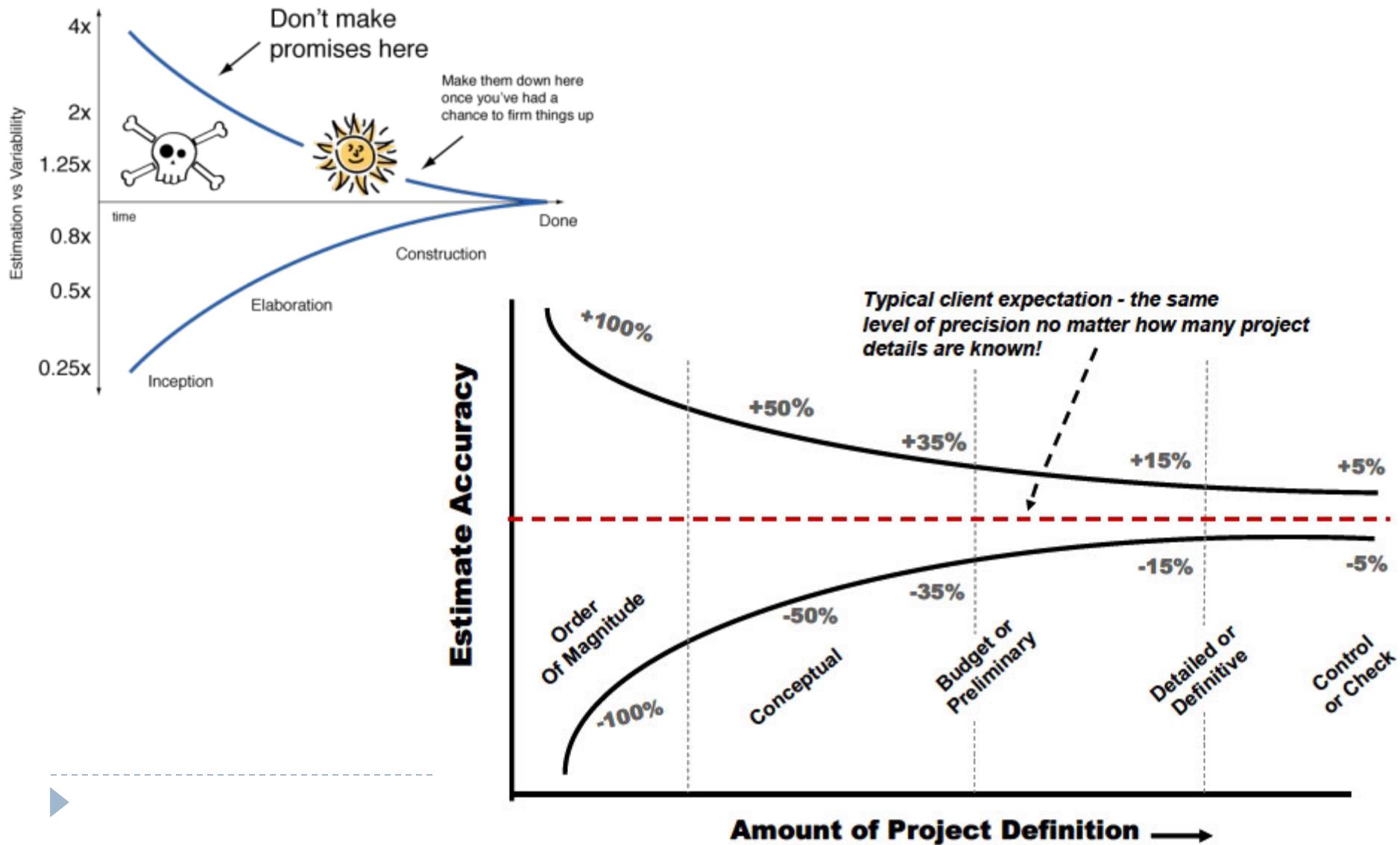
© Scott Adams, Inc./Dist. by UFS, Inc.

#3

Estimation != Commitment



Iteratively increasing clarity



#4

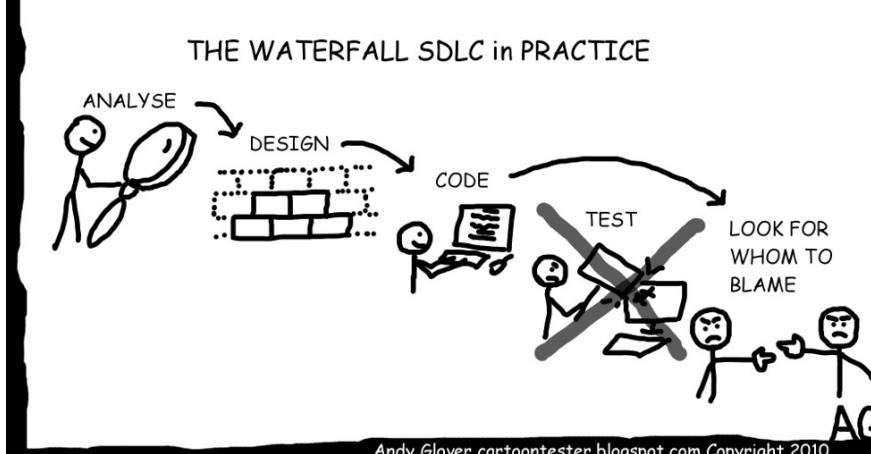
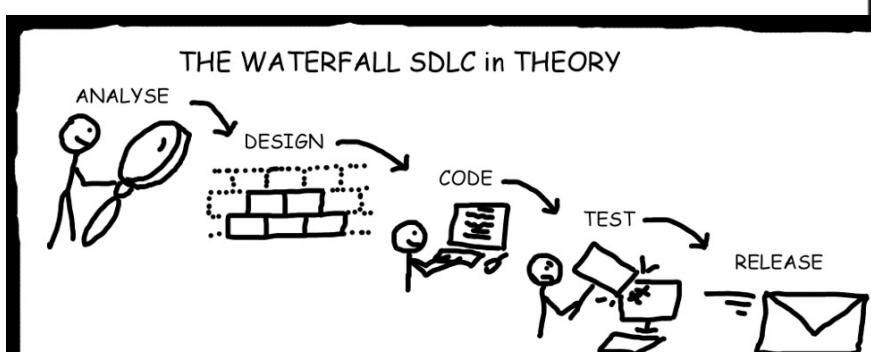
First try to measure, count and
compute

Estimate only when necessary



Reality

Estimation



#5

Aggregate independent estimates

“Wisdom of the Crowds”



Team effort

Project Approval Board

Don't worry! My team is hard at work coming up with an accurate cost estimation for the project.



I say we each get 3 tries and then we average the results.



The law of large numbers (or: statistics is on our side, for once)

- ▶ If we estimate with an error of $x\%$
- ▶ The estimate of each scope item will have an error of $x\%$
- ▶ But...
- ▶ Some items will be over-estimated, others under-estimated
(maybe....)

=> The error on the total estimate is likely $< x\%$



Estimation Methodologies

- ▶ Top-down
- ▶ Bottom-up
- ▶ Analogy
- ▶ Expert Judgment
- ▶ Priced to Win (request for quote – RFQ)
- ▶ Parametric or Algorithmic Method
 - ▶ Using formulas and equations



Function Point Analysis (FPA)

- ▶ Software size measured by number & complexity of functions it performs
- ▶ More methodical than LOC counts
- ▶ House analogy
 - ▶ House's Square Feet $\sim=$ Software LOC
 - ▶ # Bedrooms & Baths $\sim=$ Function points
 - ▶ Former is size only, latter is size & function
- ▶ Five basic steps



Function Point Analysis - Process

- ▶ 1. Count # of business functions per category
 - ▶ Categories: outputs, inputs, DB inquiries, files or data structures, and interfaces
- ▶ 2. Establish Complexity Factor for each and apply
 - ▶ Low, Medium, High
 - ▶ Set a weighting multiplier for each (0 → 15)
 - ▶ This results in the “unadjusted function-point total”
- ▶ 3. Compute an “influence multiplier” and apply
 - ▶ It ranges from 0.65 to 1.35; is based on 14 factors
- ▶ 4. Results in “function point total”
 - ▶ This can be used in comparative estimates
- ▶ 5. Estimating effort and time
 - ▶ Calculate based on per function point effort



Example - Online Bookstore Application

- ▶ **User Management:** Registration, Login, Profile Update, etc.
- ▶ **Product Management:** Adding new books, Editing book details, etc.
- ▶ **Order Management:** Placing orders, Viewing order history, etc.
- ▶ **Search Functionality:** Searching for books, etc.
- ▶ **Reviews and Ratings:** Adding and viewing reviews, etc.



Step 1: Identify and Classify Functions

- ▶ Inputs - e.g., User Registration, Adding new books
- ▶ Outputs - e.g., Displaying order confirmation
- ▶ Inquiries - e.g., Searching for books
- ▶ Logical Files - e.g., User account details, Book details
- ▶ Interface Files- e.g., Interfaces with a payment gateway



Step 2: Assign Complexity Weights

For example (for inputs):

- ▶ Low complexity: 3 FP
- ▶ Average complexity: 4 FP
- ▶ High complexity: 6 FP



Function point multipliers

Program Characteristic	Function Points		
	Low Complexity	Medium Complexity	High Complexity
Number of Inputs	x 3	x 4	x 6
Number of Outputs	x 4	x 5	x 7
Inquiries	x 3	x 4	x 6
Logical internal files	x 7	x 10	x 15
External interface files	x 5	x 7	x 10



Counting the Number of Function Points

Step 3: Calculate Unadjusted Function Points (UFP)

Step 4: Calculate Adjusted Function Points (AFP)

Influence multiplier is based on 14 general system characteristics like data communications, distributed functions, performance requirements, etc.

Each characteristic is rated on a scale from 0 (no influence) to 5 (strong influence), and the total is summed up

Program Characteristic	Function Points		
	Low Complexity	Medium Complexity	High Complexity
Number of Inputs	$5 \times 3 = 15$	$2 \times 4 = 8$	$3 \times 6 = 18$
Number of Outputs	$6 \times 4 = 24$	$6 \times 5 = 30$	$0 \times 7 = 0$
Inquiries	$0 \times 3 = 0$	$2 \times 4 = 8$	$4 \times 6 = 24$
Logical internal files	$5 \times 7 = 35$	$2 \times 10 = 20$	$3 \times 15 = 45$
External interface files	$8 \times 5 = 40$	$0 \times 7 = 0$	$2 \times 10 = 20$
Unadjusted function-point total		287	
Influence multiplier		1.20	
Adjusted function-point total		344	



Over and Under Estimation

- ▶ Over estimation issues
 - ▶ The project will not be funded
 - ▶ Conservative estimates guaranteeing 100% success may mean funding probability of zero.
 - ▶ Danger of feature and scope creep
 - ▶ Be aware of “double-padding”: team member + manager
- ▶ Under estimation issues
 - ▶ Quality issues (short changing key phases like testing)
 - ▶ Inability to meet deadlines
 - ▶ Morale and other team motivation issues



Wideband Delphi

- ▶ Group consensus approach
- ▶ Present experts with a problem and response form
- ▶ Conduct group discussion, collect anonymous opinions, then feedback
- ▶ Conduct another discussion & iterate until consensus
- ▶ Advantages
 - ▶ Easy, inexpensive, utilizes expertise of several people
 - ▶ Does not require historical data
- ▶ Disadvantages
 - ▶ Difficult to repeat
 - ▶ May fail to reach consensus, reach wrong one, or all may have same bias

PROBLEM STATEMENT

The system is an on-line version of the popular Monopoly board game. The game provides many of the features found in the board version of the game. Unless otherwise specified this game follows the standard rules of the board game.



FEATURES

Game Initialization
Move Player
Move Players in Turns
Pass Go
Free Parking
Go to Jail
Get Out of Jail
Purchase Property
Pay Rent to Property Owner
Unable to Pay Rent
Trade Properties
Buy Railroad
Pay Rent to Railroad Owner
Buy Utility
Pay Rent to Utility Owner
Buy House
Draw Jail Card
Draw Lose Money Card
Draw Gain Money Card
Draw Move Player Card



Scheduling & Tracking

How To Schedule

- ▶ **1. Identify “what” needs to be done**
 - ▶ Work Breakdown Structure (WBS)
- ▶ **2. Identify “how much” (the size)**
 - ▶ Size estimation techniques
- ▶ **3. Identify the dependency between tasks**
 - ▶ Dependency graph, network diagram
- ▶ **4. Estimate total duration of the work to be done**
 - ▶ The actual schedule



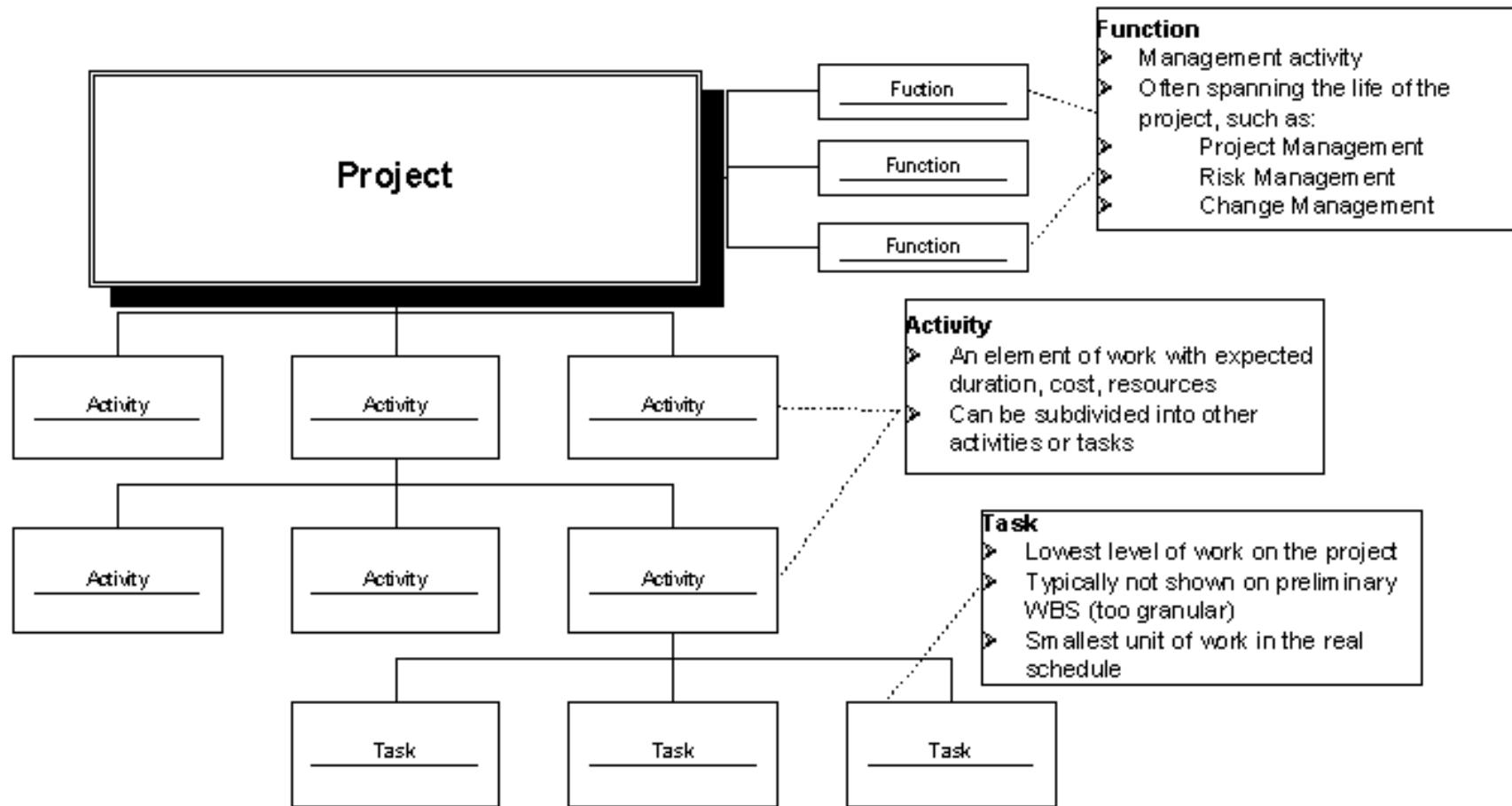
Partitioning Your Project

- ▶ You need to decompose your project into manageable chunks
- ▶ ALL projects need this step
- ▶ Divide & Conquer
- ▶ Two main causes of project failure
 - ▶ Forgetting something critical
 - ▶ Ballpark estimates become targets
- ▶ How does partitioning help this?



Project Elements

► A Project: functions, activities, tasks



Work Break Down Structure (WBS)

- ***Work Break Down Structure*** – a check list of the work that must be accomplished to meet the project objectives.
- The WBS lists the major project outputs and those departments or individuals primarily responsible for their completion.



WBS Outline Example

0.0 Retail Web Site

1.0 Project Management

2.0 Requirements Gathering

3.0 Analysis & Design

4.0 Site Software Development

 4.1 HTML Design and Creation

 4.2 Backend Software

 4.2.1 Database Implementation

 4.2.2 Middleware Development

 4.2.3 Security Subsystems

 4.2.4 Catalog Engine

 4.2.5 Transaction Processing

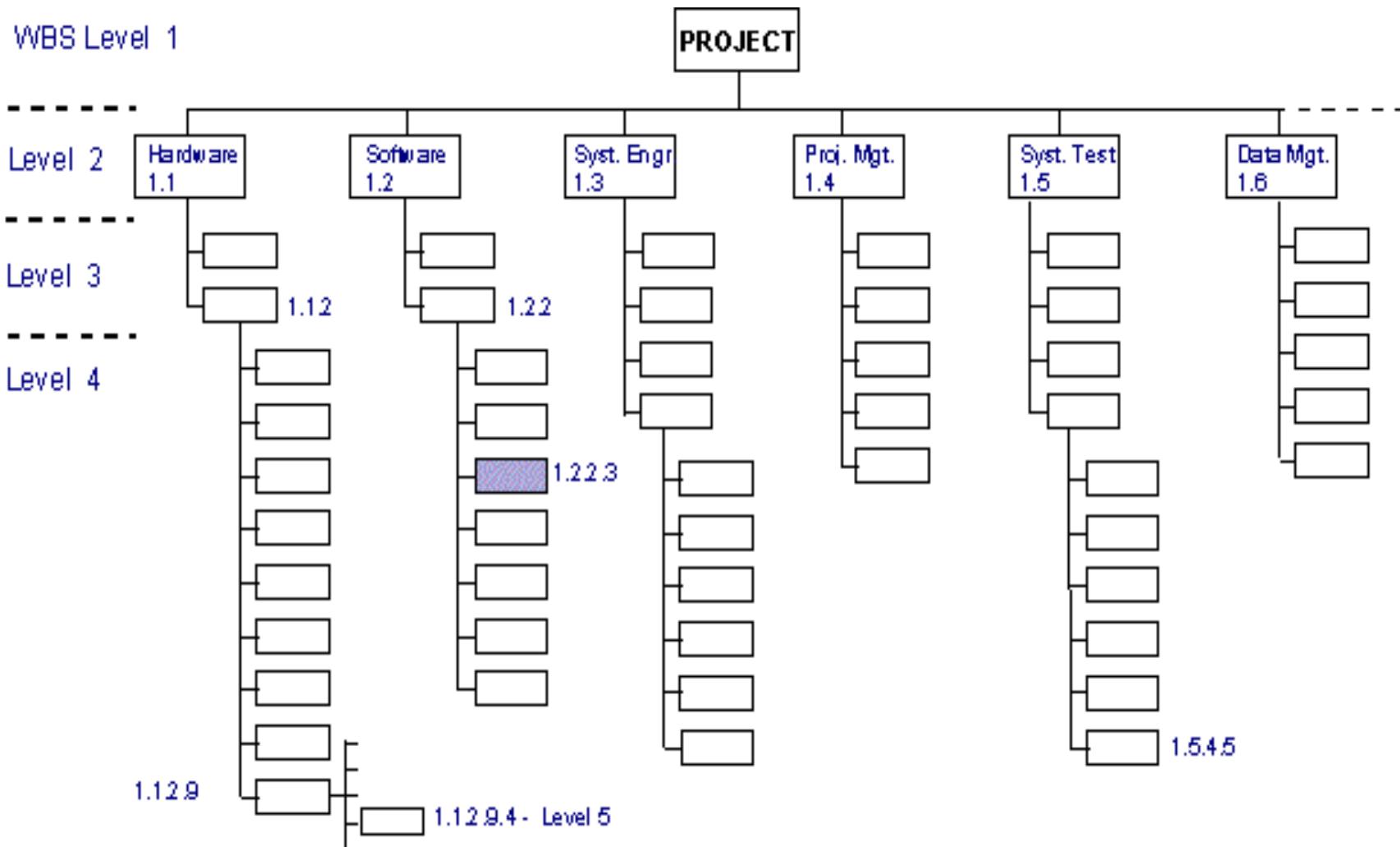
 4.3 Graphics and Interface

 4.4 Content Creation

5.0 Testing and Production



WBS Level 1



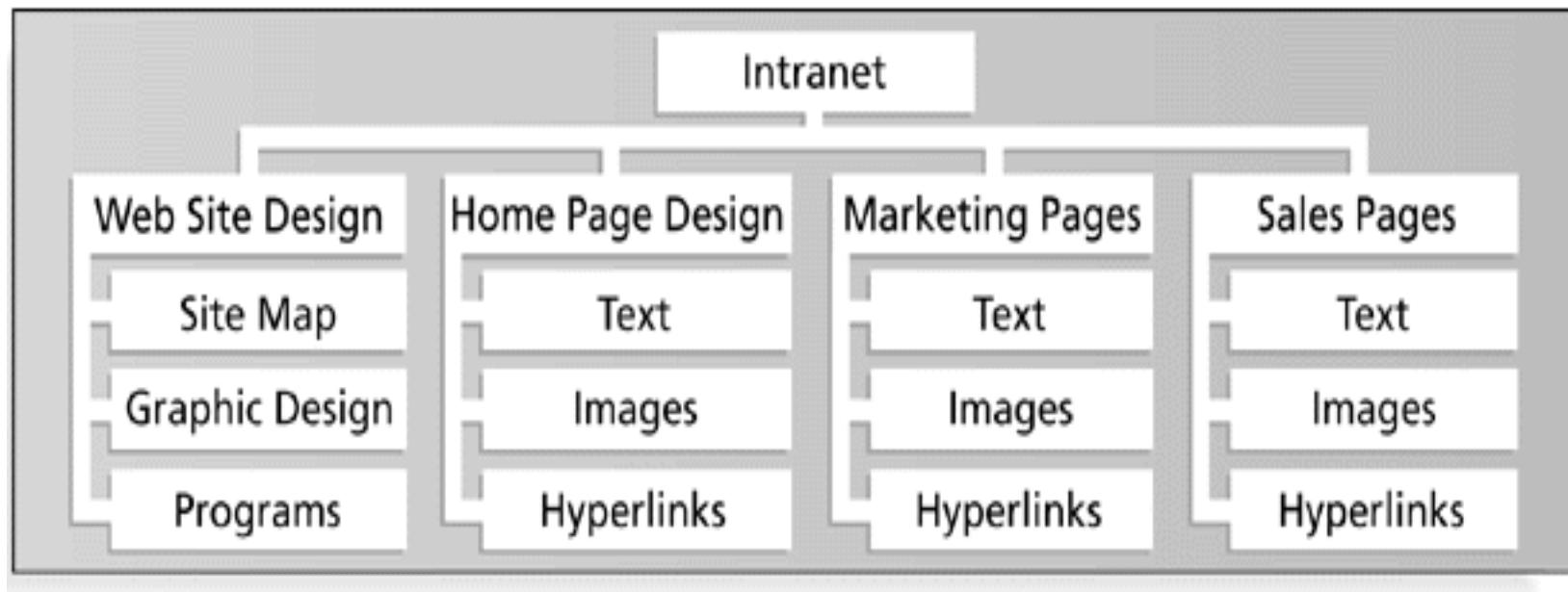
From: http://www.hyperthot.com/pm_wbs.htm

WBS Types

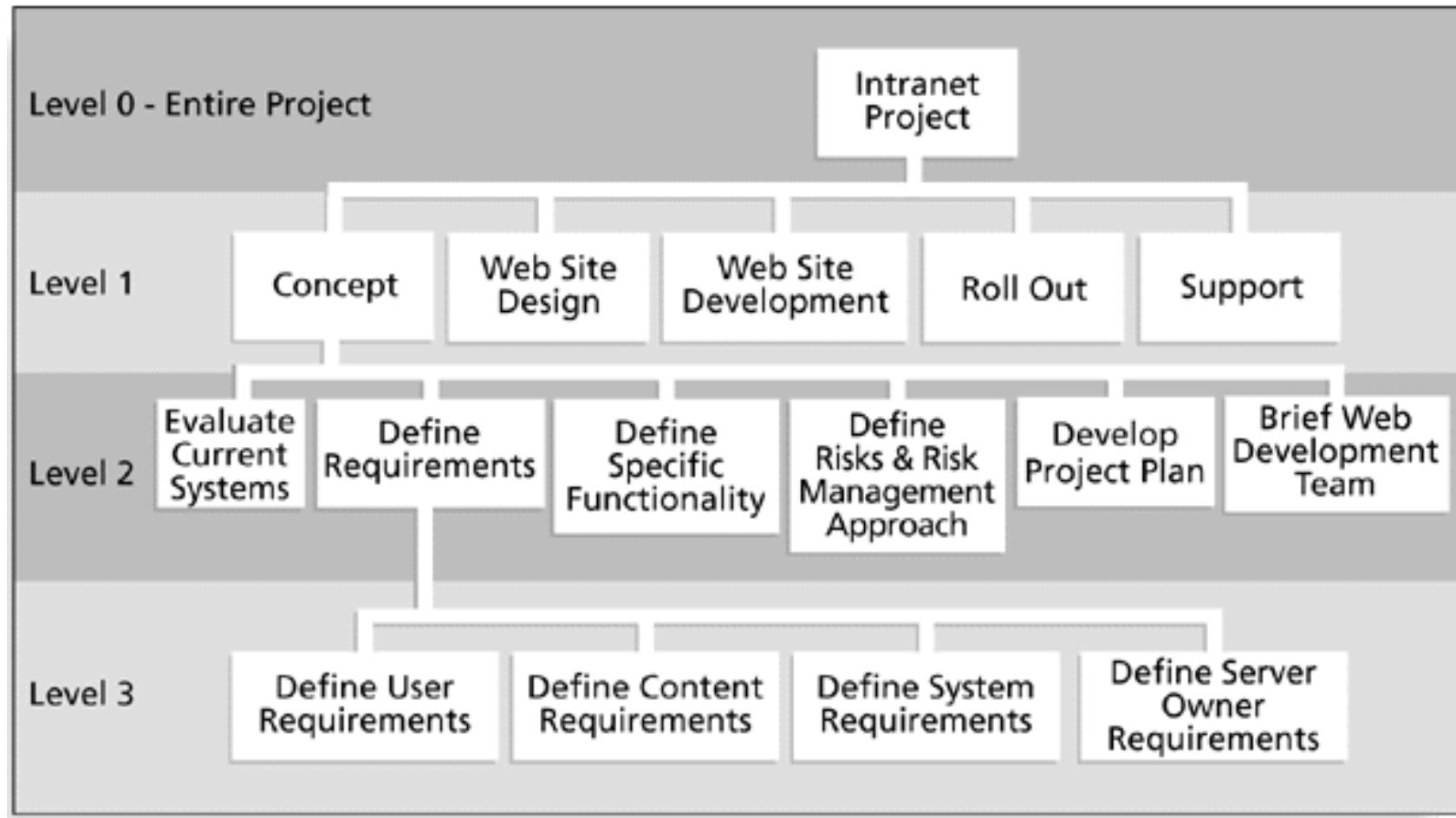
- ▶ **Process WBS**
 - ▶ a.k.a Activity-oriented
 - ▶ Ex: Requirements, Analysis, Design, Testing
 - ▶ Typically used by PM
- ▶ **Product WBS**
 - ▶ a.k.a. Entity-oriented
 - ▶ Ex: Financial engine, Interface system, DB
 - ▶ Typically used by engineering manager
- ▶ **Hybrid WBS: both above**
 - ▶ This is not unusual
 - ▶ Ex: Lifecycle phases at high level with component or feature-specifics within phases
 - ▶ Rationale: processes produce products



Product WBS



Process WBS



WBS

- ▶ List of Activities, not Things
- ▶ List of items can come from many sources
 - ▶ SOW, Proposal, brainstorming, stakeholders, team
- ▶ Describe activities using “bullet language”
 - ▶ Meaningful but terse labels
- ▶ All WBS paths do not have to go to the same level
- ▶ Do not plan more detail than you can manage



Work Packages (Tasks)

- ▶ Generic term for discrete **tasks** with definable end results
- ▶ The “one-to-two” rule
 - ▶ Often at: 1 or 2 persons for 1 or 2 weeks
- ▶ Basis for monitoring and reporting progress
 - ▶ Can be tied to budget items (charge numbers)
 - ▶ Resources (personnel) assigned
- ▶ Ideally shorter rather than longer
 - ▶ Not so small as to micro-manage



WBS Techniques

- ▶ Top-Down
- ▶ Bottom-Up
- ▶ Analogy
- ▶ Rolling Wave
 - ▶ 1st pass: go 1-3 levels deep
 - ▶ Gather more requirements or data
 - ▶ Add more detail later
- ▶ Post-its on a wall



WBS Techniques

▶ Analogy

- ▶ Base WBS upon that of a “similar” project
- ▶ Use a template
- ▶ Analogy also can be estimation basis
- ▶ Pros
 - ▶ Based on past actual experience
- ▶ Cons
 - ▶ Needs comparable project



Sequence the Work Activities

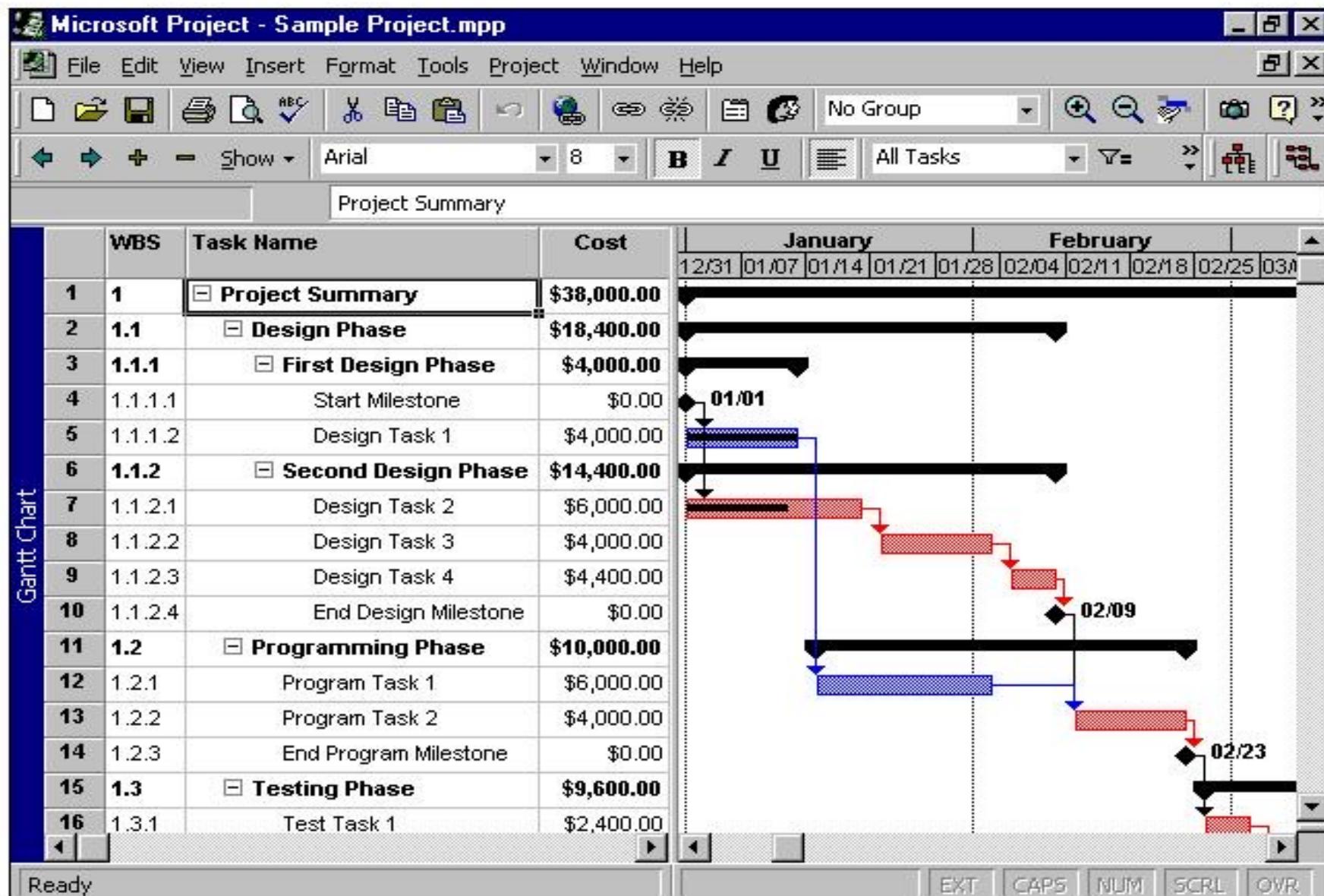
- ▲ Milestone Chart
- ▲ Gantt chart
- ▲ Network Techniques
 - CPM (Critical Path Method)
 - PERT (Program Evaluation and Review Technique)



Gantt Chart

- Gantt chart is a means of displaying simple activities or events plotted against time or dollars
- Most commonly used for exhibiting program progress or for defining specific work required to reach an objective
- Gantt charts may include listing of activities, activity duration, scheduled dates, and progress-to-date





Gantt Chart

- Advantages:
 - Easy to understand
 - Easy to change
- Disadvantages:
 - only a vague description of the project
 - does not show interdependency of activities
 - cannot show results of an early or late start of an activity

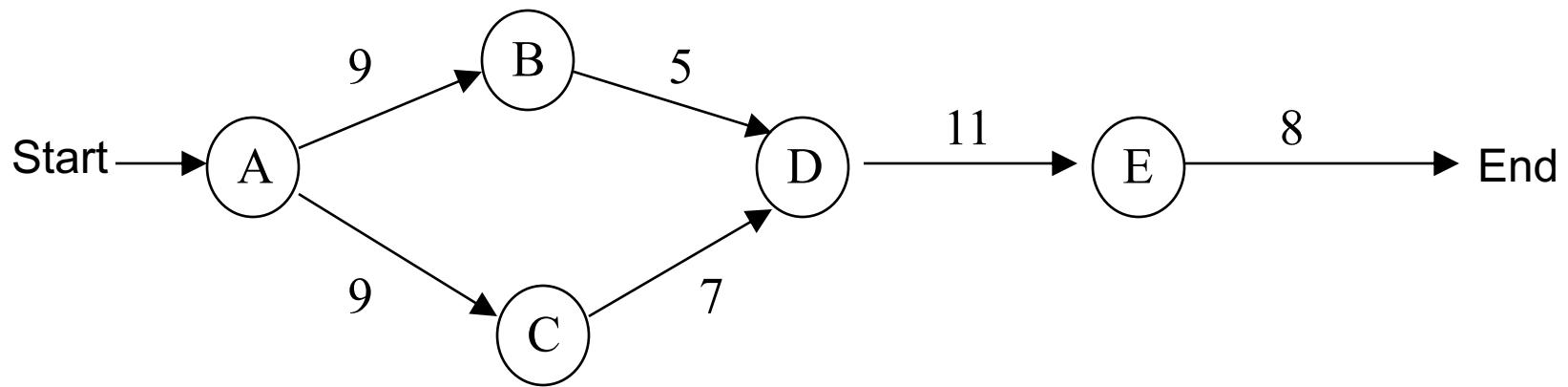


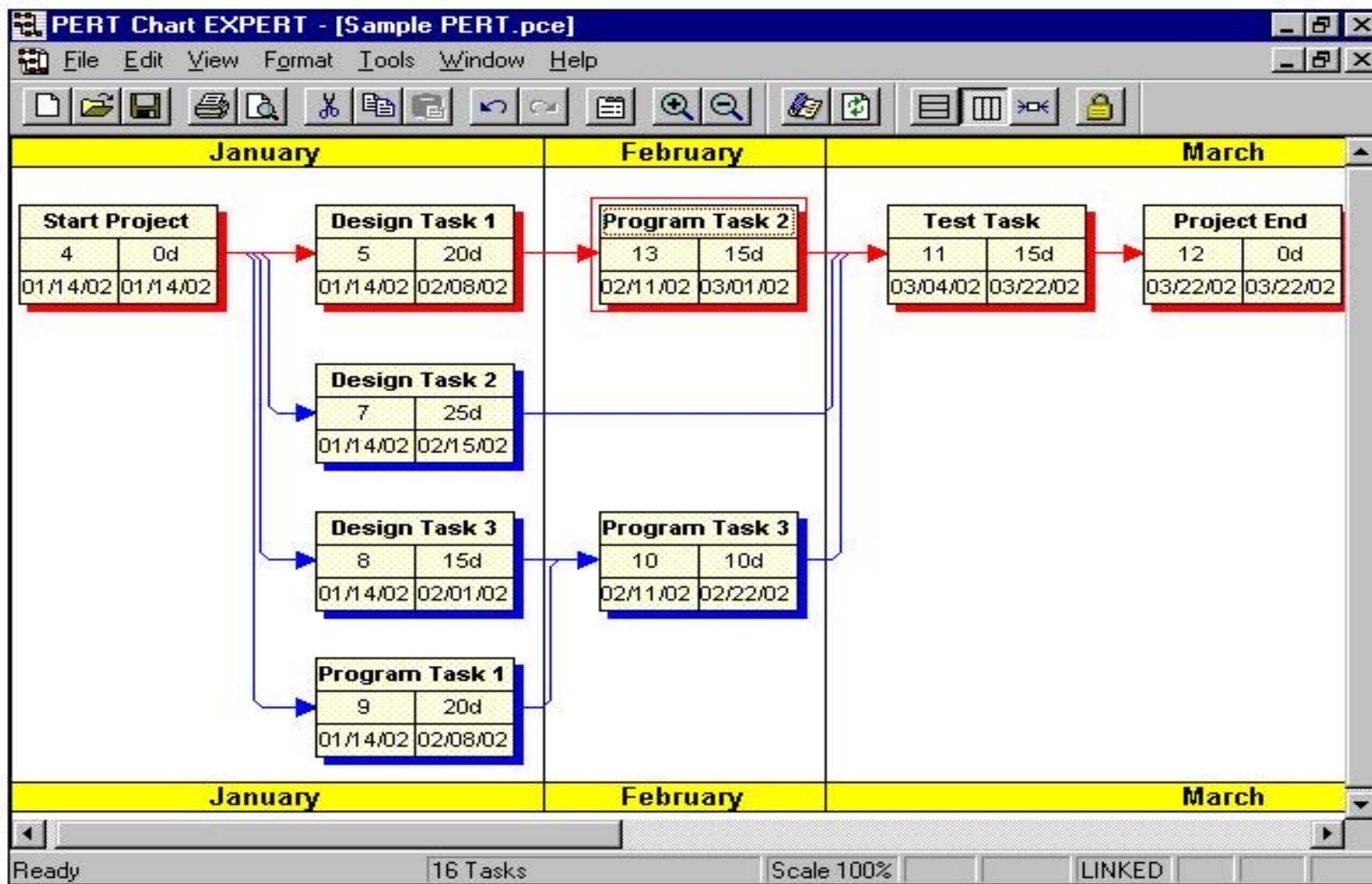
Network Techniques

- A *precedence network* diagram is a graphic model portraying the sequential relationship between key events in a project.
- Initial development of the network requires that the project be defined and thought out.
- The network diagram clearly and precisely communicates the plan of action to the project team and the client.



Task	Duration	Dependencies
A - Architecture & design strategy	9	start
B - Decide on number of releases	5	A
C - Develop acceptance test plan	7	A
D - Develop customer support plan	11	B,C
E - Final sizing & costing	8	D





Critical Path Method (CPM)

CPM tries to answer the following questions:

1. What is the duration of the project?
2. By how much (if at all) will the project be delayed if any one of the activities takes N days longer?
3. How long can certain activities be postponed without increasing the total project duration?

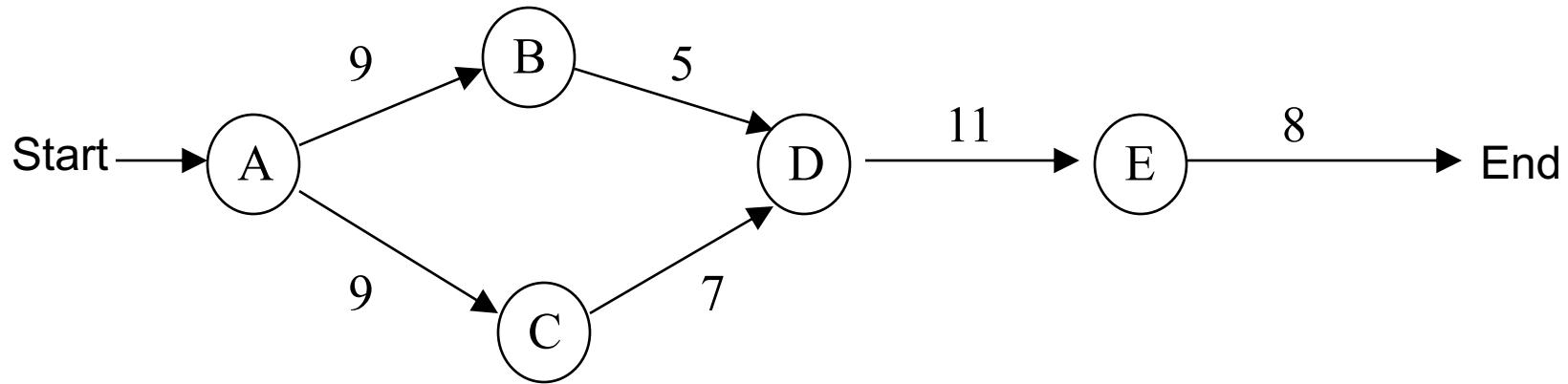


Critical Path

- Sequence of activities that have to be executed one after another
- Duration times of these activities will determine the overall project time, because there is no slack/float time for these activities
- If any of the activities on the critical path takes longer than projected, the entire project will be delayed by that same amount
- Critical path = Longest path in the precedence network (generally, the longest in time)



Critical Path

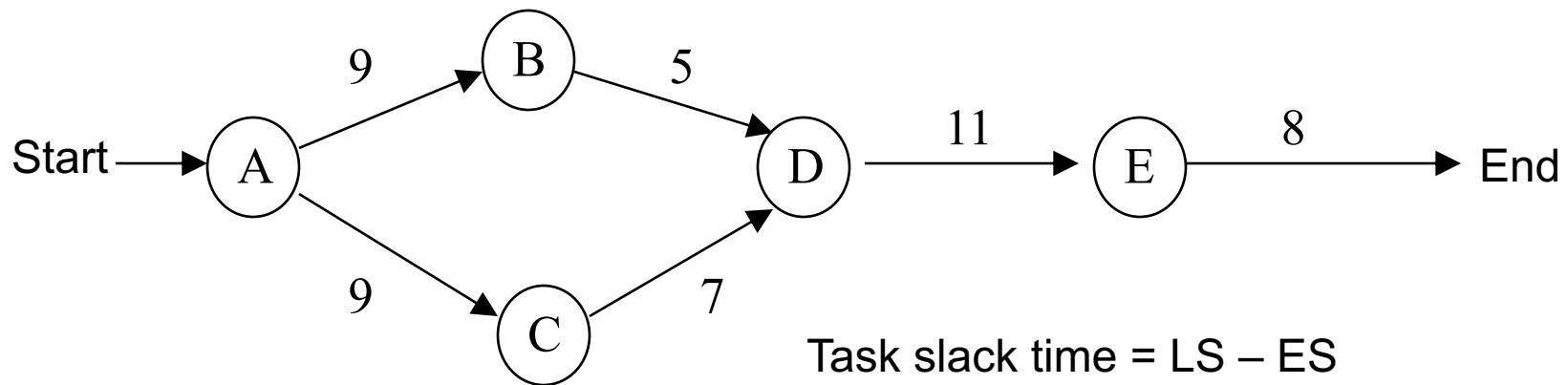


Critical Path = A – C – D – E (35 time units)

Critical Tasks = A,C,D,E

Non-Critical Path = A-B-D-E

Task	Duration	Depend	Earliest Start	Earliest Finish	Latest Start	Latest Finish
A	9	none	0	9	0	9
B	5	A	9	14	11	16
C	7	A	9	16	9	16
D	11	B,C	16	27	16	27
E	8	D	27	35	27	35



Slack time – maximum allowable delay for a non-critical activity.

Task slack time = LS – ES
- or -
Task slack time = LF - EF
Task B has 2 time units of **slack time**

Requirements Gathering and Analysis (Week 4)

Some content adapted from Rajib Mall's book and Craig Larman's book.

Requirements Phase

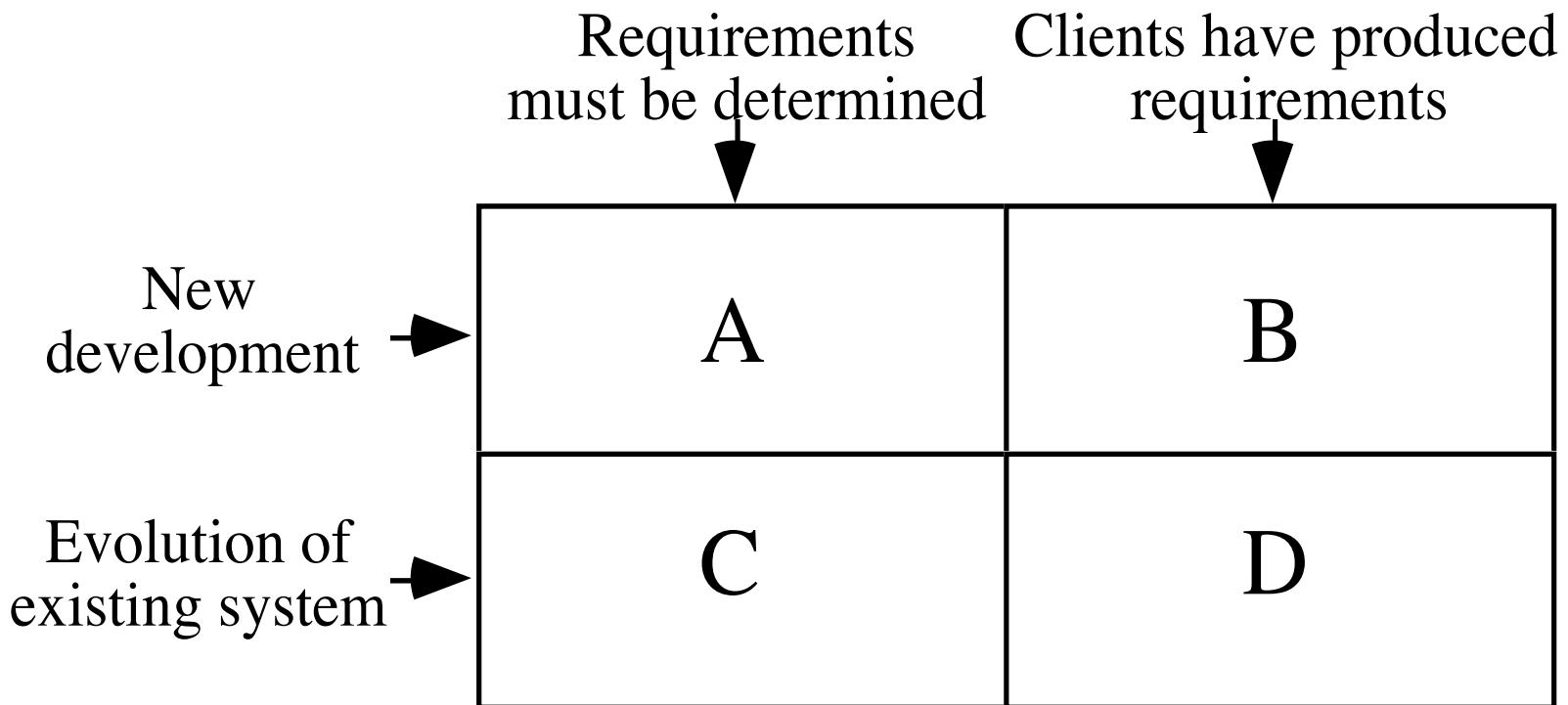
- Many projects fail:
 - Because they start implementing the system.
 - Without determining whether they are building what the customer really wants.

Why Requirements analysis and specification?

- Factors that cause projects to fail:
 - Lack of User Input 12.8%
 - Incomplete Requirements & Specifications 12.3%
 - Changing Requirements & Specifications 11.8%
 - Lack of Executive Support 7.5%
 - Technology Incompetence 7.0%
 - Lack of Resources 6.4%
 - Unrealistic Expectations 5.9%
 - Unclear Objectives 5.3%
 - Unrealistic Time Frames 4.3%
 - New Technology 3.7%
 - Other 23.0%

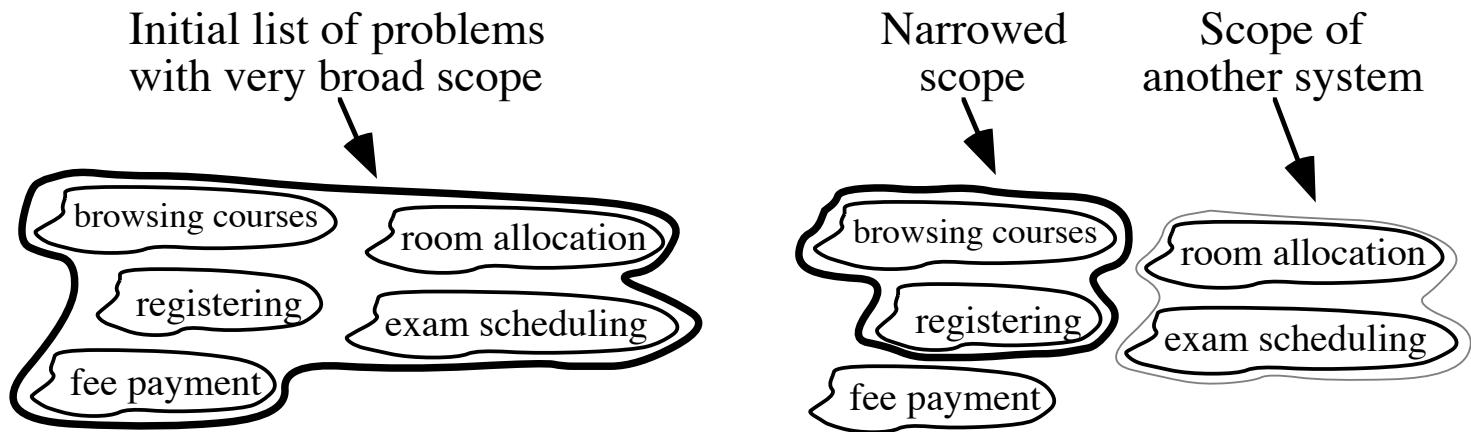
• Standish Group, *The Standish Group Report: Chaos*, 1995, <http://www.scs.carleton.ca/~beau/PM/Standish-Report.html>

The Starting Point for Software Projects



Defining the Scope

- Narrow the *scope* by defining a more precise problem
 - List all the things you might imagine the system doing
 - Exclude some of these things if too broad
 - Determine high-level goals if too narrow
- Example: A university registration system



What is a Requirement?

- Requirement: A statement about the proposed system that all stakeholders agree must be made true in order for the customer's problem to be adequately solved.
 - Short and concise piece of information
 - Says something about the system
 - All the stakeholders have agreed that it is valid
 - It helps solve the customer's problem
- A collection of requirements is a *requirements document*.

Types of Requirements

- Business requirements
 - High-level objectives of the organization or customer who requests the system.
- Functional requirements
 - Describe *what* the system should do
For example, features (use cases)
- Non-functional requirements
 - *Constraints* that must be adhered to during development
For example, quality constraints, technology constraints, process constraints, etc.

Requirements Phase

- Goals of requirements phase:
 - Fully understand the user requirements.
 - Remove inconsistencies, anomalies, etc. from requirements.
 - Document requirements properly in an SRS document.

Requirements Phase

- Consists of two distinct activities:
 - Requirements Gathering and Analysis
 - Requirements Specification

Requirements Gathering

- Also known as requirements elicitation.
- If the project is to automate some existing procedures
 - e.g., automating existing manual accounting activities,
 - The task of the system analyst is a little easier
 - Analyst can immediately obtain:
 - input and output formats
 - accurate details of the operational procedures

Requirements Gathering (CONT.)

- In the absence of a working system,
 - Lot of imagination and creativity are required.
- Interacting with the customer to gather relevant data:
 - Requires a lot of experience.

Analysis of the gathered requirements

- Main purpose of requirements analysis:
 - Clearly understand the user requirements,
 - Detect inconsistencies, ambiguities, and incompleteness.
- Incompleteness and inconsistencies:
 - Resolved through further discussions with the end-users and the customers.

Inconsistent Requirement

- Some part of the requirement:
 - contradicts with some other part.

- Example (E-commerce site):

1. Product Availability:

1. Requirement A: Products are displayed as available even if there's only one left in stock to create a sense of urgency.
2. Requirement B: Users cannot add a product to their cart if the stock quantity is less than five.

2. Pricing Rules:

1. Requirement C: Discounted prices should be displayed with the original price crossed out and the new discounted price prominently shown.
2. Requirement D: Prices are automatically adjusted based on user behavior, with the goal of maximizing revenue

Incomplete Requirement

- Some requirements have been omitted:
 - Possibly due to oversight.
- Example (E-commerce site):
 - Implement user authentication for the e-commerce website.

Enhanced Requirement:

Precondition: User must be registered in the system via an email id and password.

"Users should be able to log in with their registered email and password. After three unsuccessful login attempts, users should be locked out of their accounts for 30 minutes. Implement password reset functionality that sends a password reset link to the user's registered email address"

Analysis of the gathered requirements (contd.)

- Requirements analysis involves:
 - Obtaining a clear, in-depth understanding of the product to be developed,
 - Remove all ambiguities and inconsistencies from the initial customer perception of the problem.

Analysis of gathered requirements (contd.)

- Experienced analysts take considerable time:
 - To understand the exact requirements the customer has in his mind.
- Experienced systems analysts know - often as a result of past (painful) experiences

Analysis of gathered requirements (contd.)

- Several things about the project should be clearly understood by the analyst:
 - What is the problem?
 - Why is it important to solve the problem?
 - What are the possible solutions to the problem?
 - What complexities might arise while solving the problem?

Analysis of gathered requirements (contd.)

- After collecting all data regarding the system to be developed,
 - Remove all inconsistencies and anomalies from the requirements,
 - Systematically organize requirements into a Software Requirements Specification (SRS) document.

Bad Requirements: A Simplified Example

- *A mail should be displayed within 3 seconds of clicking on mail*
- *User should be able to add a new mail server during peak hours within a small downtime*
- *Business services should not be interrupted during the peak hours*
- *User should be able to customize all the mailbox settings*
- *User should be able to change the look and feel of how the mailbox is displayed*

Quality Requirements

- Correct – only user representative can determine
- Feasible – get reality check on what can or cannot be done technically or within given cost constraints.
- Necessary – trace each requirement back to its origin
- Unambiguous – one interpretation
- Verifiable – how do you know if the requirement was implemented properly?
- Prioritized – function of value provided to the customer

Writing Example #1

“The product shall provide status messages at regular intervals not less than every 60 seconds.”

Writing Example #1

“The product shall provide status messages at regular intervals not less than every 60 seconds.”

- Incomplete – What are the status messages and how are they supposed to be displayed?
- Ambiguous – What part of the product? Regular interval?
- Not verifiable

Alternative #1

1. Status Messages.
 - 1.1. The Background Task Manager shall display status messages in a designated area of the user interface at intervals of 60 plus or minus 10 seconds.
 - 1.2. If background task processing is progressing normally, the percentage of the background task processing that has been completed shall be displayed.
 - 1.3. A message shall be displayed when the background task is completed.
 - 1.4. An error message shall be displayed if the background task has stalled.

Writing Example #2

“The product shall switch between displaying and hiding non-printing characters instantaneously.”

Writing Example #2

“The product shall switch between displaying and hiding non-printing characters instantaneously.”

- Not Feasible – computers cannot do anything instantaneously.
- Incomplete – conditions which trigger state switch
- Ambiguous – “non-printing character”

Alternative #2

“The user shall be able to toggle between displaying and hiding all HTML markup tags in the document being edited with the activation of a specific triggering condition.”

- Note that “triggering condition” is left for design

The Specification Trap

The Landing Pilot is the Non-Landing Pilot until the 'decision altitude' call, when the Handling Non-Landing Pilot hands the handling to the Non-Handling Landing Pilot, unless the latter calls 'go-around,' in which case the Handling Non-Landing Pilot continues handling and the Non-Handling Landing Pilot continues non-handling until the next call of 'land,' or 'go-around' as appropriate . In view of recent confusions over these rules, it was deemed necessary to restate them clearly.

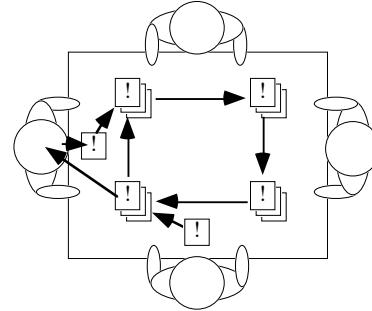
- British Airways memorandum, quoted in *Pilot Magazine*.

Techniques - Gathering and Analyzing Requirements

- Observation
 - Read documents and discuss requirements with users
 - Shadowing important potential users as they do their work
 - ask the user to explain everything he or she is doing
 - Session videotaping
- Interviewing
 - Conduct a series of interviews
 - Ask about specific details
 - Ask about the stakeholder's vision for the future
 - Ask if they have alternative ideas
 - Ask for other sources of information
 - Ask them to draw diagrams

Gathering and Analyzing Requirements

- Brainstorming
 - Appoint an experienced moderator
 - Arrange the attendees around a table
 - Decide on a ‘trigger question’
 - Ask each participant to write an answer and pass the paper to its neighbour



- *Joint Application Development (JAD)* is a technique based on intensive brainstorming sessions

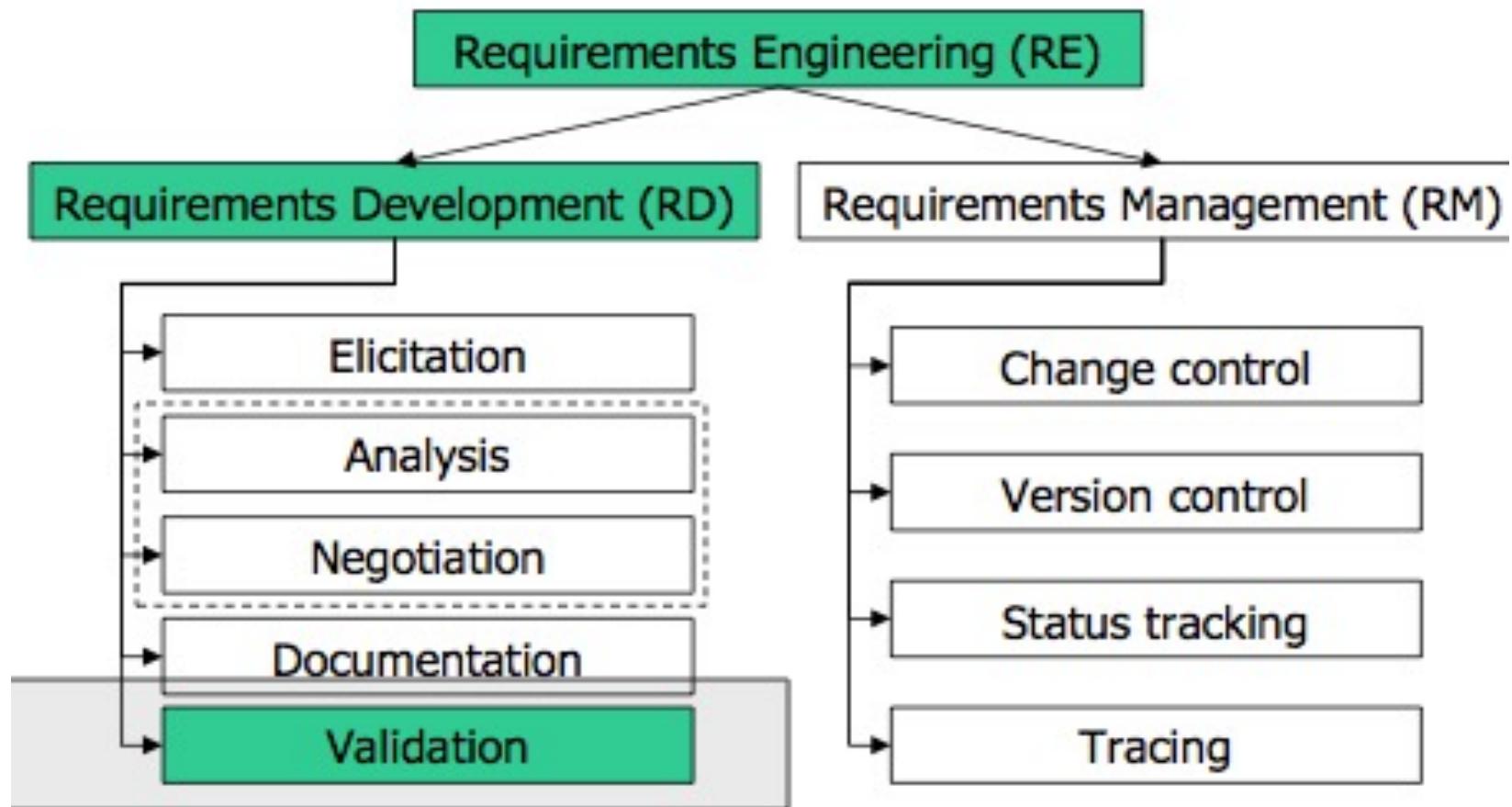
Gathering and Analyzing Requirements

- Prototyping
 - The simplest kind: *paper prototype*.
 - a set of pictures of the system that are shown to users in sequence to explain what would happen
 - The most common: a mock-up of the system's UI
 - Written in a rapid prototyping language
 - Does *not* normally perform any computations, access any databases or interact with any other systems
 - May prototype a particular aspect of the system

Difficulties and Risks in Domain and Requirements analysis

- Lack of understanding of the domain or the real problem
 - *Do domain analysis and prototyping*
- Requirements change rapidly
 - *Perform incremental development, build flexibility into the design, do regular reviews*
- Attempting to do too much
 - *Document the problem boundaries at an early stage, carefully estimate the time*
- It may be hard to reconcile conflicting sets of requirements
 - *Brainstorming, JAD sessions, competing prototypes*
- It is hard to state requirements precisely
 - *Break requirements down into simple sentences and review them carefully, look for potential ambiguity, make early prototypes*

Requirements Processes



Requirements Specification

(Week 4)

Software Requirements Specification (SRS)

- Main aim of requirements specification:
 - Systematically organize the requirements arrived during requirements analysis.
 - Document requirements properly.

Software Requirements Specification

- The SRS document is useful in various contexts:
 - Statement of user needs
 - Contract document
 - Reference document
 - Definition for implementation

Software Requirements Specification: A Contract Document

- Requirements document is a reference document.
- SRS document is a contract between the development team and the customer.
 - Once the SRS document is approved by the customer,
 - Any subsequent controversies are settled by referring the SRS document.

Software Requirements Specification: A Contract Document

- Once customer agrees to the SRS document:
 - Development team starts to develop the product according to the requirements recorded in the SRS document.
- The final product will be acceptable to the customer:
 - As long as it satisfies all the requirements recorded in the SRS document.

SRS Document (CONT.)

- The SRS document is known as black-box specification:
 - The system is considered as a black box whose internal details are not known.
 - Only its visible external (i.e. input/output) behavior is documented.



SRS Document (CONT.)

- SRS document concentrates on:
 - What needs to be done
 - Carefully avoids the solution (“how to do”) aspects.
- The SRS document serves as a contract
 - Between development team and the customer.
 - Should be carefully written

SRS Document (CONT.)

- The requirements at this stage:
 - Written using end-user terminology.
- If necessary:
 - Later a formal requirement specification may be developed from it.

Properties of a Good SRS Document

- It should be concise
 - and at the same time should not be ambiguous.
- It should specify what the system must do
 - and not say how to do it.
- Easy to change.,
 - i.e. it should be well-structured.
- It should be consistent
- It should be complete

Properties of a Good SRS Document (cont...)

- It should be traceable
 - You should be able to trace which part of the specification corresponds to which part of the design, code, etc and vice versa.
- It should be verifiable
 - e.g. “system should be user friendly” is not verifiable

SRS Document (CONT.)

- SRS document, normally contains three important parts:
 - Functional requirements,
 - Non-functional requirements,
 - Goals of Implementation.

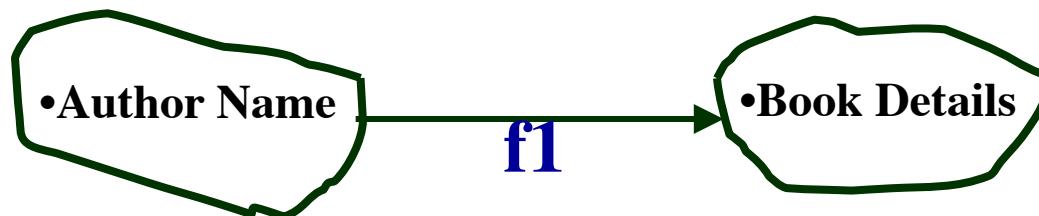
SRS Document (CONT.)

- It is desirable to consider every system:
 - Performing a set of functions $\{f_i\}$.
 - Each function f_i considered as:
 - Transforming a set of input data to corresponding output data.



Example: Functional Requirement

- F1: Search Book
 - Input:
 - an author's name:
 - Output:
 - details of the author's books and the locations of these books in the library.



Functional Requirements

- Functional requirements describe:
 - A set of high-level requirements
 - Each high-level requirement:
 - takes in some data from the user
 - outputs some data to the user
 - Each high-level requirement:
 - might consist of a set of identifiable functions

Functional Requirements

- For each high-level requirement:
 - Every function is described in terms of:
 - Input data set
 - Output data set
 - Processing required to obtain the output data set from the input data set.

Nonfunctional Requirements

- Characteristics of the system which can not be expressed as functions:
 - Maintainability,
 - Portability,
 - Usability, etc.

Nonfunctional Requirements

- Nonfunctional requirements include:
 - Reliability issues,
 - Performance issues:
 - Example: How fast the system can produce results
 - so that it does not overload another system to which it supplies data, etc.
 - Human-computer interface issues,
 - Interface with other external systems,
 - Security, maintainability, etc.

Non-Functional Requirements

- Hardware to be used,
- Operating system
 - or DBMS to be used
- Capabilities of I/O devices
- Standards compliance
- Data representations
 - by the interfaced system

Goals of Implementation

- Goals describe things that are desirable of the system:
 - But, would not be checked for compliance.
 - For example,
 - Reusability issues
 - Functionalities to be developed in future

Organization of the SRS Document

- Introduction.
- Functional Requirements
- Nonfunctional Requirements
 - External interface requirements
 - Performance requirements
- Goals of implementation

Functional Requirements

- A high-level function is one:
 - Using which the user can get some useful piece of work done.
- Can the receipt printing work during withdrawal of money from an ATM:
 - Be called a useful piece of work?
- A high-level requirement typically involves:
 - Accepting some data from the user,
 - Transforming it to the required response, and then
 - Outputting the system response to the user.

High-Level Function

- A high-level function:
 - Usually involves a series of interactions between the system and one or more users.
- Even for the same high-level function,
 - There can be different interaction sequences (or scenarios)
 - Due to users selecting different options or entering different data items.

Example Functional Requirements

- List all functional requirements
 - with proper numbering.
- Req. 1:
 - Once the user selects the “search” option,
 - he is asked to enter the key words.
 - The system should output details of all books
 - whose title or author name matches any of the key words entered.
 - Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.

Example Functional Requirements

- Req. 2:
 - When the “renew” option is selected,
 - The user is asked to enter his membership number and password.
 - After password validation,
 - The list of the books borrowed by him are displayed.
 - The user can renew any of the books:
 - By clicking in the corresponding renew box.

Req. 1:

- R.1.1:
 - Input: “search” option,
 - Output: user prompted to enter the key words.
- R1.2:
 - Input: key words
 - Output: Details of all books whose title or author name matches any of the key words.
 - Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.
 - Processing: Search the book list for the keywords

Alternatively/Additionally...

- **Use cases** can be used for representing functional requirements

Req. 2:

- R2.1:
 - Input: “renew” option selected,
 - Output: user prompted to enter his membership number and password.
- R2.2:
 - Input: membership number and password
 - Output:
 - list of the books borrowed by user are displayed. User prompted to enter books to be renewed or
 - user informed about bad password
 - Processing: Password validation, search books issued to the user from borrower list and display.

Req. 2:

- R2.3:
 - **Input:** user choice for renewal of the books issued to him through mouse clicks in the corresponding renew box.
 - **Output:** Confirmation of the books renewed
 - **Processing:** Renew the books selected by the in the borrower list.

Examples of Bad SRS Documents

- Unstructured Specifications:
 - Narrative essay --- one of the worst types of specification document:
 - Difficult to change,
 - Difficult to be precise,
 - Difficult to be unambiguous,
 - Scope for contradictions, etc.

Examples of Bad SRS Documents

- Noise:
 - Presence of text containing information irrelevant to the problem.
- Silence:
 - aspects important to proper solution of the problem are omitted.

Examples of Bad SRS Documents

- Overspecification:
 - Addressing “how to” aspects
 - For example, “Library member names should be stored in a sorted descending order”
 - Overspecification restricts the solution space for the designer.
- Contradictions:
 - Contradictions might arise
 - if the same thing described at several places in different ways.

Examples of Bad SRS Documents

- Ambiguity:
 - Literary expressions
 - Unquantifiable aspects, e.g. “good user interface”
- Forward References:
 - References to aspects of problem
 - defined only later on in the text.
- Wishful Thinking:
 - Descriptions of aspects
 - for which realistic solutions will be hard to find.

User Stories – example structure

As a [Type of USER],

[Function to Perform (some goal)]

so that [Business Value (some reason)]

Example: As a user, I can indicate folders not to backup so that my backup isn't filled up with things I don't need saved

UML use cases (week 5)

Some content adapted Applying UML and patterns by Craig Larman

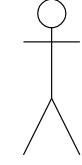
Use Cases

- *Use cases* tell stories of actors using a system.
- A *use case scenario* is a sequence of actions a system performs that yields an observable result of value to a particular actor.
- It describes an end-to-end process --- from when a user starts to use the system for a particular purpose until they are done (for that purpose).
- *Use cases* are an excellent mechanism to express functional requirements. They emphasize thinking from the viewpoint of the users.
- We use *use cases* to "drive" the process. Our goal during development is to make specific use cases operational.

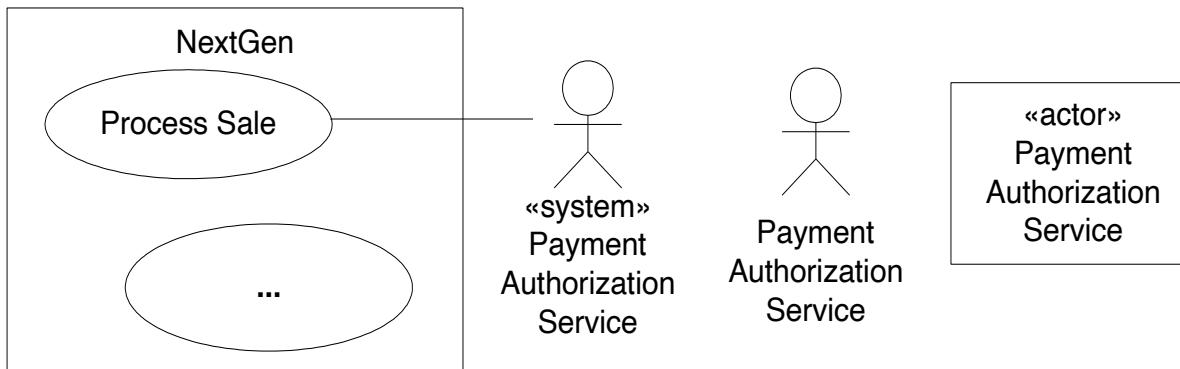
Use Cases in UML 2.X

- Use cases are associated with subjects
 - A subject can be a system, a subsystem in a system, or a class
- A use case describes interactions between users (clients) and a subject

Use Case Modeling: Core Elements

Construct	Description	Syntax
use case	A sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.	 UseCaseName
actor	A role played by an entity that interacts with the subject (e.g., system, subsystem, class).	 ActorName
System/subject boundary	Represents the boundary between the subject and the actors who interact with the subject.	

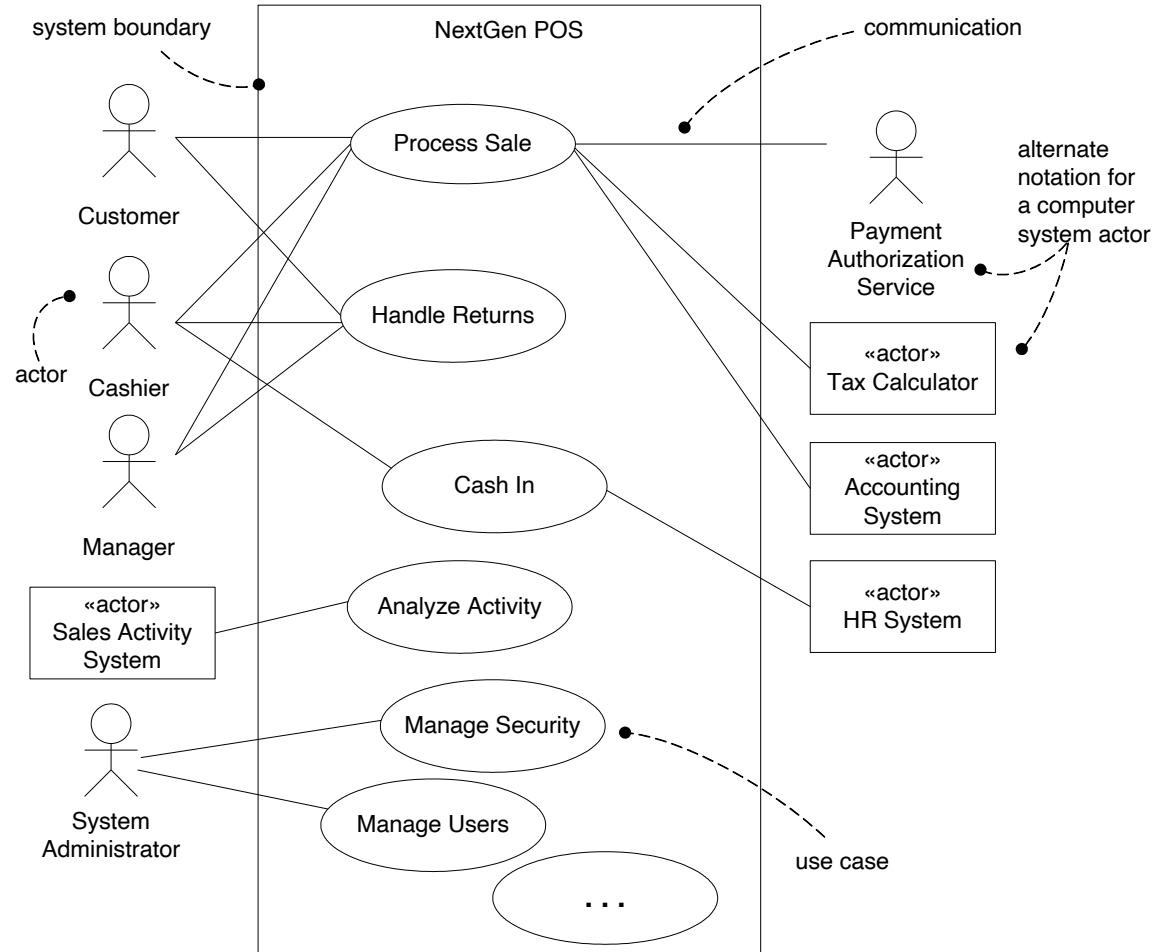
Depicting actors



Some UML alternatives to illustrate external actors that are other computer systems.

The class box style can be used for any actor, computer or human. Using it for computer actors provides visual distinction.

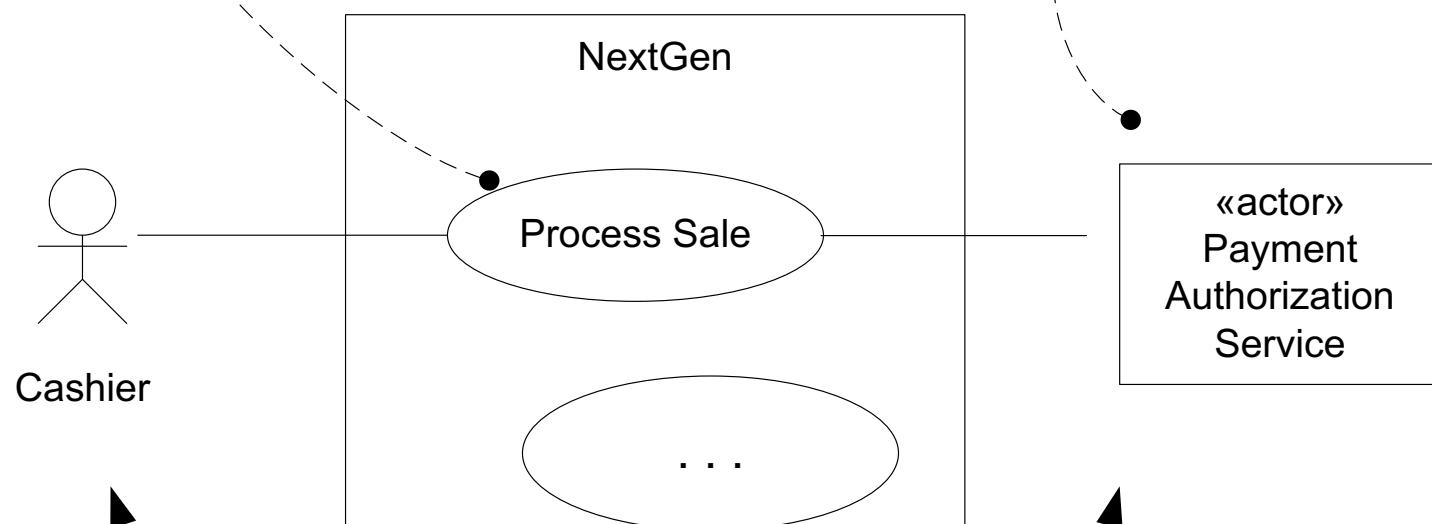
Use Case Diagram



Use Case Diagram Style

For a use case context diagram, limit the use cases to user-goal level use cases.

Show computer system actors with an alternate notation to human actors.



primary actors on
the left

supporting actors
on the right

Use Cases as Requirements

- Use cases can be used to capture functional requirements
 - System attributes associated with a system operation can be documented in a use case
- Not all requirements can be captured by use cases
 - System attributes that span use cases are documented as supplementary requirements

Use Case Instance (Scenario)

- A **scenario** is a particular sequence of actions in a use case.
 - A use case is a related set of scenarios that yields an observable result of value to a particular actor
- A **use case instance** is an execution of a scenario.
 - Often use case instance and scenario are used synonymously in informal discussions.

Levels of Rigor

- **Brief:** One paragraph summaries of functionality
- **Casual:** Multiple paragraphs that cover multiple scenarios
- **Fully-dressed (Detailed):** Structured, detailed description of scenarios

Essential vs. Concrete Use Cases

- **Essential Use Cases** describe functionality in implementation independent terms
 - Requirements level use cases must be essential
- **Concrete Use Cases** describe external functionality in system dependent terms
 - Use cases can be used during design to document externally observable behavior of subsystems

Requirements Use Case Template - 1

Use Case Number	EU-xxxx : Indicates an essential use case, i.e., a use case that describes activity in system independent terms
Use Case Name	<i>Enter name of Use Case.</i>
Overview	<i>Describe the purpose of the Use Case and give a brief description.</i>
Type	<i>Enter Use Case priority (primary, secondary, optional)</i>
Actors	<i>•List all actors that participate in this Use Case. Indicate the actor that initiates the use case by placing “initiator” in brackets after the actor name. Also, indicate primary actors by placing “primary” in brackets after actor name.</i>
Properties	<i>Performance:</i>
	<i>Security:</i>
	<i>Other:</i>

Use Case Template – cont'd

Pre-condition	<i>Enter the condition that must be true when the main flow is initiated. This should reference the conceptual model.</i>
Flow	Main Flow: Steps should be numbered.
	Subflows: Break down of main flow steps
	Alternate Flows: Include the post condition for each alternate flow if different from the main flow.
Post Condition	<i>Enter the condition that must be true when the main flow is completed. This should reference the conceptual model. Include the following information in this section:</i>
Cross References	<i>References to other Use Cases or textual requirements that relate to this Use Case.</i>

Use case example

Use Case Number:	EU-0001
Use Case Name:	Withdraw Money
Overview:	Customer withdraws money from an ATM .
Type:	primary
Actors:	Customer
Pre-condition:	Customer has selected withdraw option
Main flow:	<ol style="list-style-type: none">1. System requests customer PIN2. Customer keys in PIN and submits to system3. If the PIN is valid ten the system acknowledges valid entry and asks for amount to be withdrawn4. Customer enters amount to be withdrawn5. If amount entered in less than amount in Customer's account, system dispenses the cash, else system informs customer that amount cannot be withdrawn
Alternate flow	3. If the PIN is invalid then the use case is restarted. If this occurs 3 consecutive times then the system cancels the transaction and prevents the customer from interacting for 60 seconds
Alternate flow	2. The customer can cancel the PIN validation at anytime. Cancellation causes the use case to restart
Alternate flow	2. The customer can clear and reenter the PIN any number of times before submitting the PIN
Post-condition	True
Cross-reference:	Validate PIN

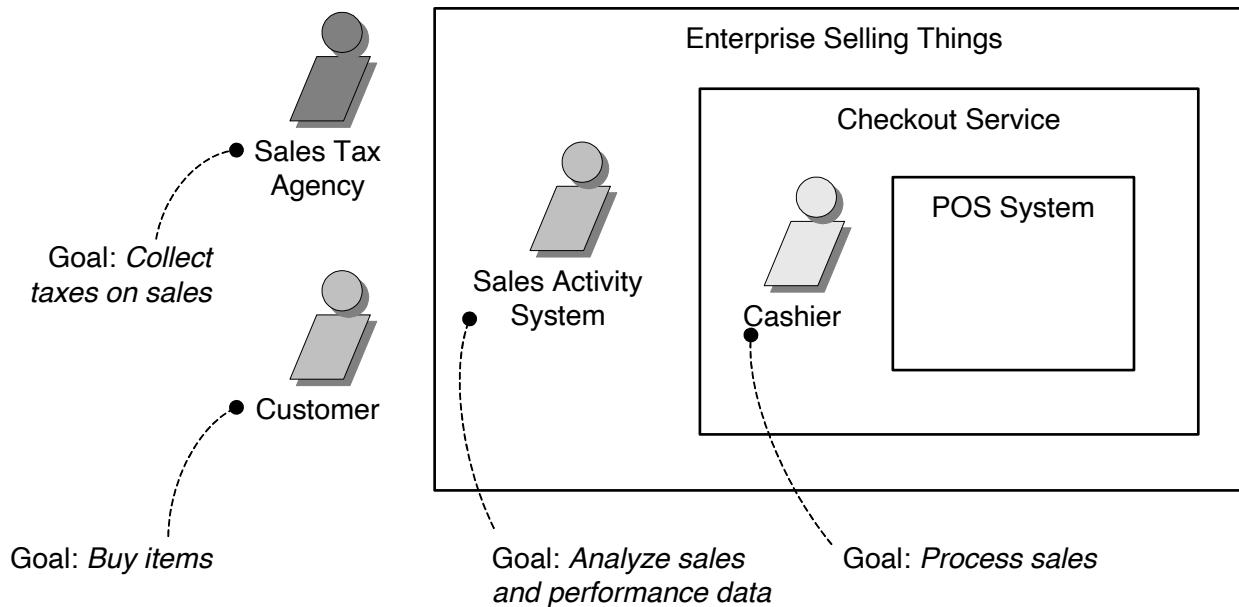
Actor Types

- **Primary:** actor whose goal is accomplished by the use case
- **Supporting:** actor that provides services to the system
 - E.g., authorization service
- **Offstage:** an actor that has an interest in the use case but is not primary or supporting
 - E.g., regulating agency

Use Case Scope

A use case should describe end-to-end functionality

- Should describe a task carried in response to an event generated by an actor that produces a result of value to a subset of its actors and leaves the system in a stable state (one in which it is not waiting for a restricted set of inputs)



Developing Use Cases

- Scope system and identify primary actors that interact with the system
- Determine goals of primary actors (can be documented in an Actor-Goal list)
- For each actor, consider the ways that the actor typically interacts with the system to accomplish goals
- Consider exceptional behaviors

Use Case Modeling Tips

- Writing essential use cases: Focus on intent
 - Keep user interface terms out
 - Ask “what is the goal?”
- Write “black-box” use cases
 - Do not describe internal operations (e.g., storing to a data base)
- Focus only on interactions between system and actors
 - Ignore interactions between actors
- Focus on text description
 - Use diagrams for presentation purposes only
- A use case diagram should
 - contain only use cases at the same level of abstraction
 - include only required actors

How do I know I have a good use case?

- Use case should describe an activity that yields an observable result of value to an actor.
- A use case can describe an elementary business process: a sequence of tasks performed to handle a business event
- Use cases are typically not single steps or single low-level actions.

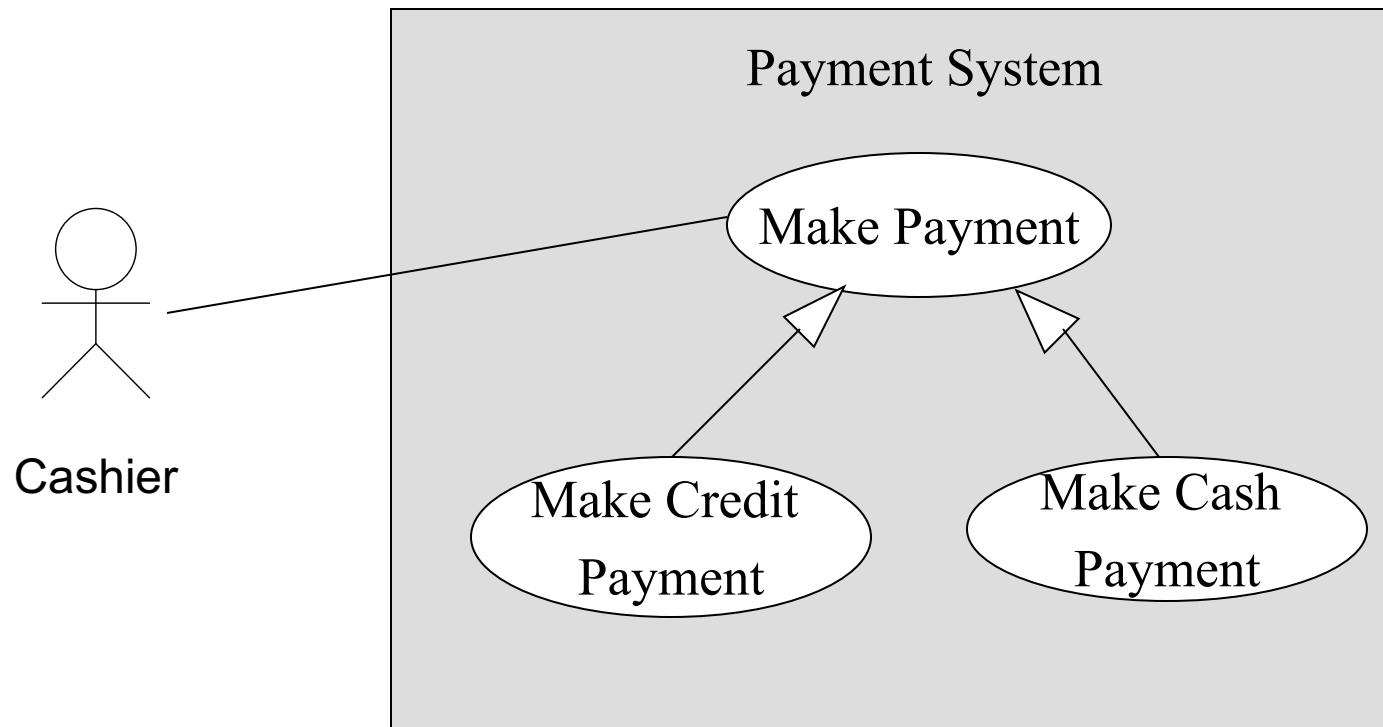
Organizing Use Cases

- Specializing/generalizing use cases
- Including use cases
- Extending use cases

Specializing Use Cases

- Generalizing/Specializing use cases
 - A specialized use case inherits the behavior (sequence of actions) of its parent(s)
 - A specialized use case can override some of the behavior of its parent(s). It can also add to the behavior
 - A specialized use case can be used anywhere the general use case is expected.

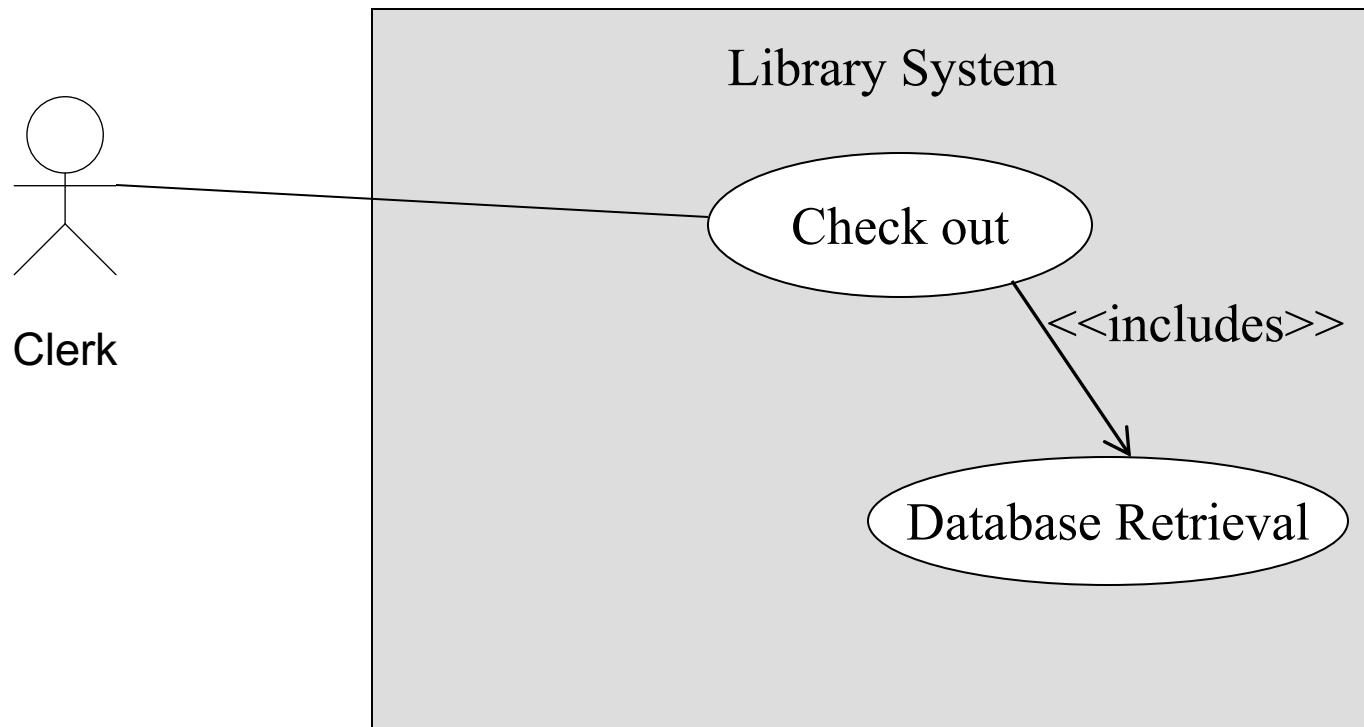
Specializing Use Cases



Including Use Cases

- A use case can include another use case at a specified location
 - Used to avoid writing the same flow of events across a number of use cases.
 - The included use case must not be a stand-alone use case.

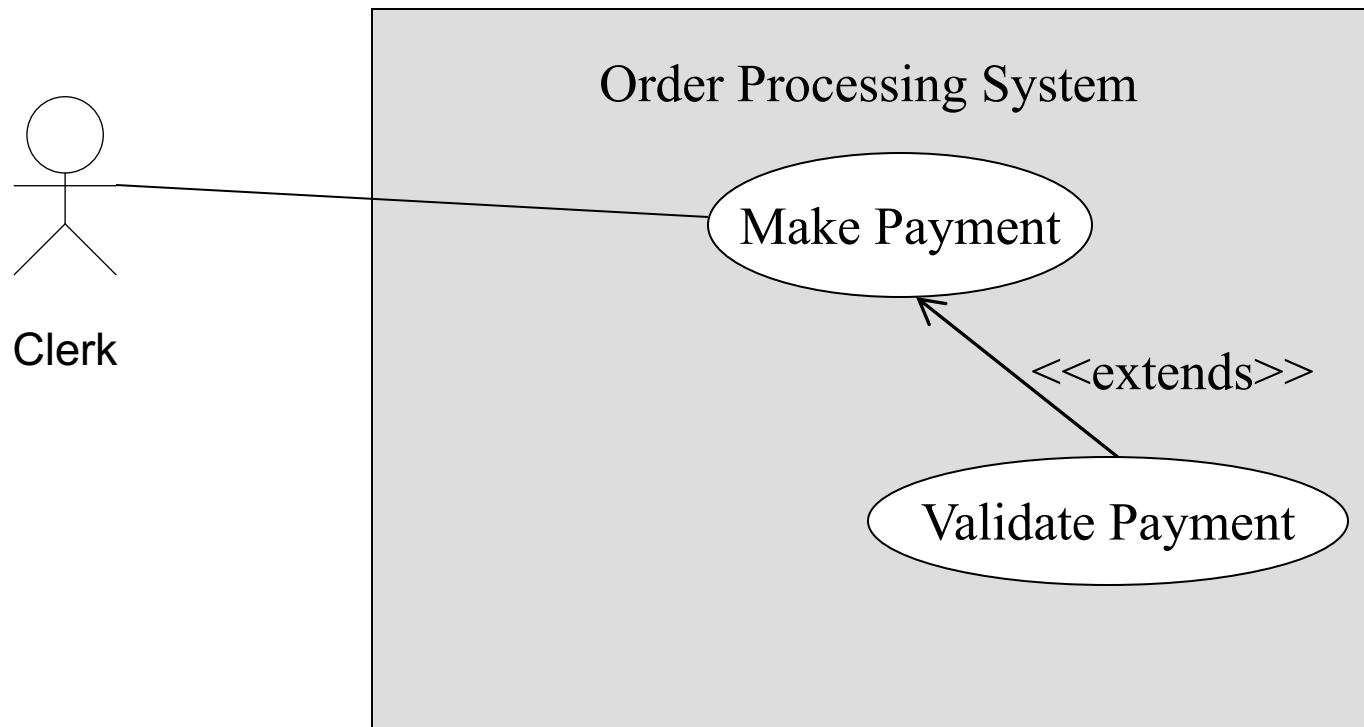
Including Use Cases



Extending Use Cases

- A use case can extend a base use case by incorporating additional behavior at specified locations of the base use case
 - The base use case can act as a stand-alone use case.
 - The base use case can only be extended at specified points called *extension points*.
 - Often used to separate optional behavior from mandatory behavior
 - Also used to model a separate flow that is executed under certain conditions

Including Use Cases



Summary

- Use case model the behavior of the system
 - Functional requirements are mapped to use cases
 - Non-functional requirements can be specified as constraints (not done in most of the cases)

Modelling using UML Class Diagrams

Let's revisit Process Sale Use case

Preconditions: Cashier is identified and authenticated on a sales terminal.

Main flow:

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sales line item and presents item description, price, and running total. Price calculated from a set of price rules.
< Cashier repeats steps 3-4 until indicates done >
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, requests payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting System and Inventory System.
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Alternative flow:

- 3a. Customer asks cashier to remove an item from the purchase.
 1. Cashier enters item identifier for removal from sale.
 2. System displays updated running total.

Moving from Use cases to solution design

1. **Customer** arrives at **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. Cashier enters **item identifier**.
4. System records **sales line item** and presents **item description**, **price**, and **running total**. Price calculated from a set of price rules.
< Cashier repeats steps 3-4 until indicates done >
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, requests **payment**.
7. Customer pays and System handles payment.
8. System logs completed **sale** and sends sale and payment information to the external **Accounting System** and **Inventory System**.
9. System presents **receipt**.
10. Customer leaves with receipt and goods (if any).

Conceptual Classes

Register

Item

Store

Sale

Sales
LineItem

Cashier

Customer

Manager

Payment

Product
Catalog

Product
Specification

Note: These are not software classes

Representing Problem/Domain Concepts using class diagrams



visualization of a real-world concept in
the domain of interest

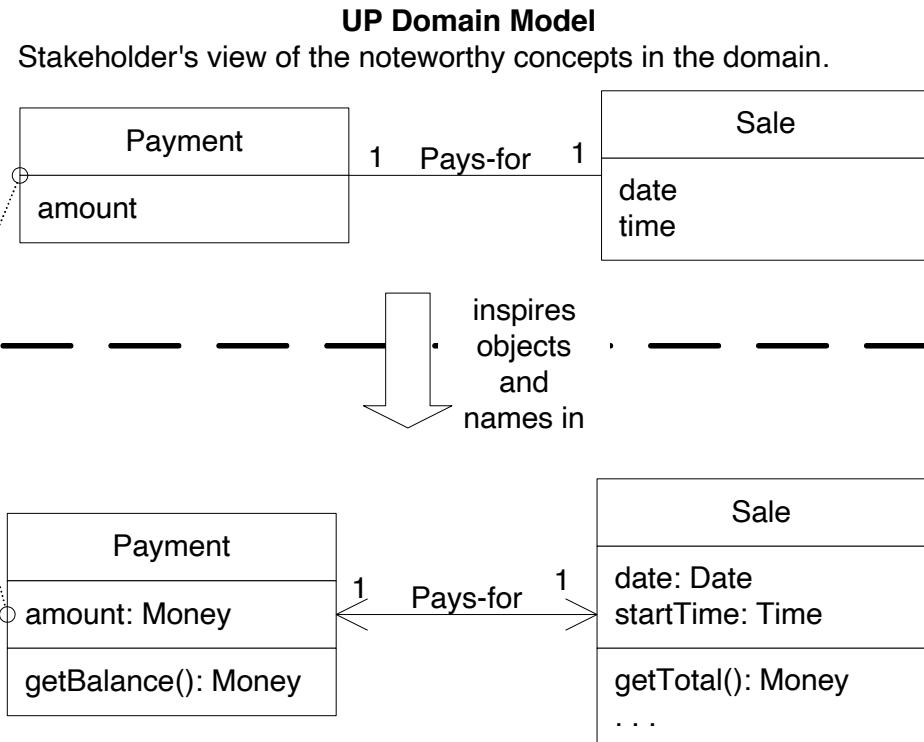
it is a *not* a picture of a software class

Requirements class vs. design class

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.



Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Rendering Concepts

- A depicted class has the following structure:
 - Name compartment (mandatory)
 - Attributes compartment (optional)
- Every class must have a distinguishing name.

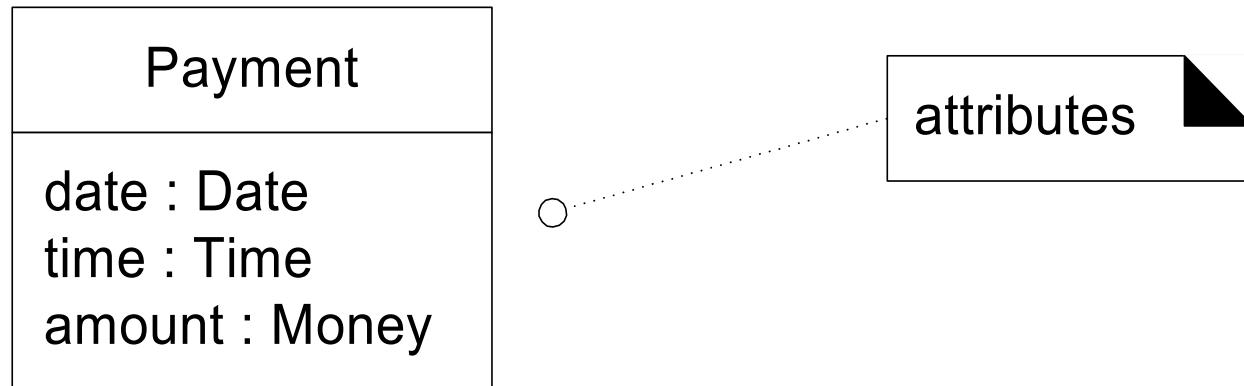
Attributes

- An attribute is a named property. Each concept instance associates value(s) with each attribute of a concept.
- *OOA syntax*

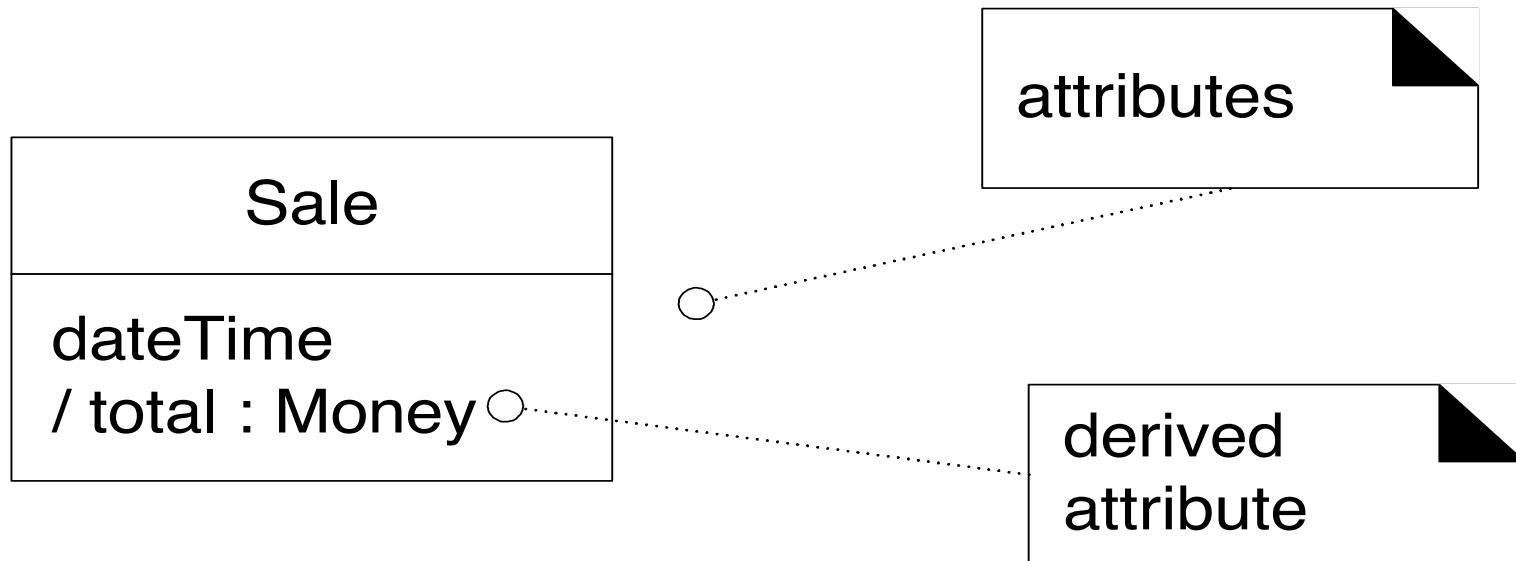
name [multiplicity] [: type] [{property-string}]

Attributes

- Show only “simple” relatively primitive types as attributes.
- Connections to other concepts are to be represented as associations, not attributes.



Derived Attributes



Access Modifiers

Sale
- dateTime : Date
- / total : Money

Private visibility
attributes

Math
+ pi : Real = 3.14 {readOnly}

Public visibility readonly
attribute with initialization

Person
firstName
middleName : [0..1]
lastName

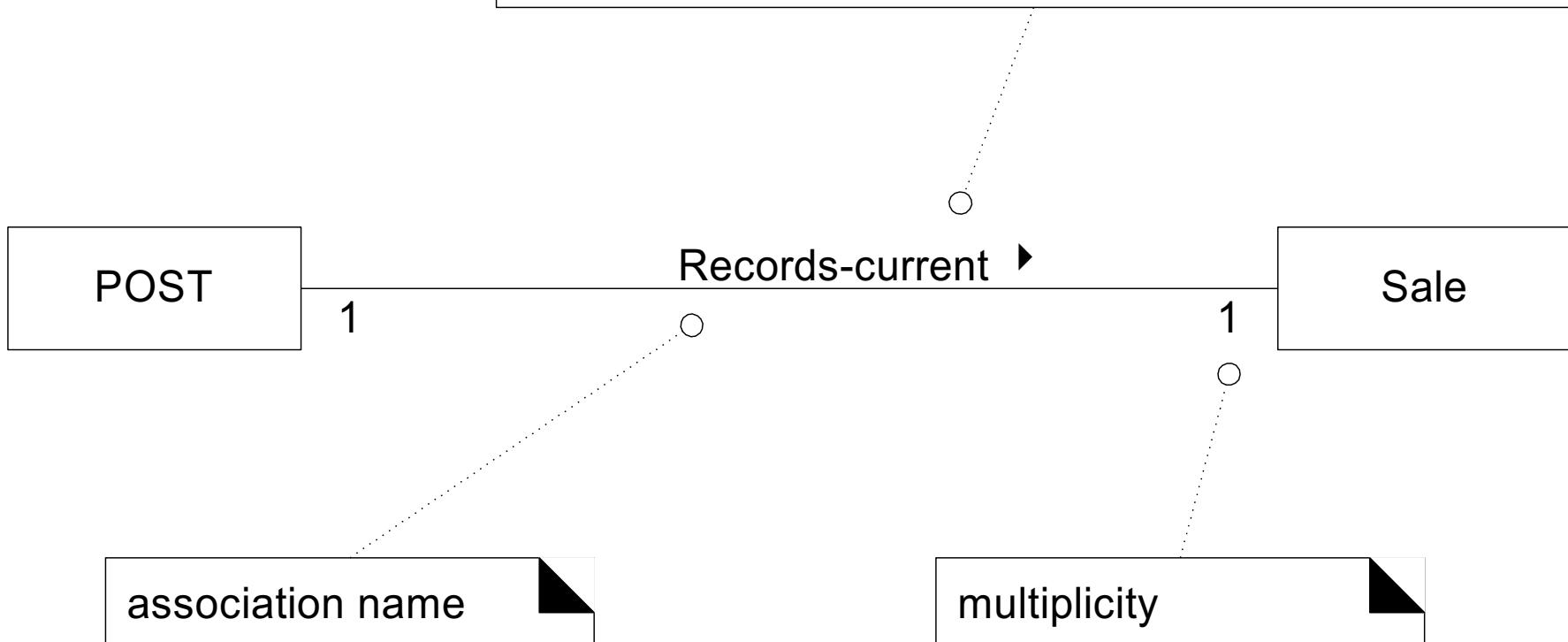
Optional value

Modeling Static Relationships

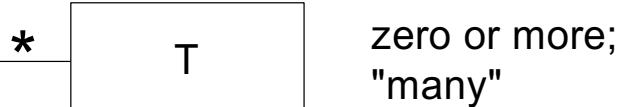
- Two kinds of static relationships:
 - Associations
 - Represent structural relationships among objects
 - Generalizations
 - Represent generalization/specialization class structures
- The two kinds of relationships are orthogonal

Associations

- "direction reading arrow"
- it has **no** meaning except to indicate direction of reading the association label
- often excluded



Multiplicity

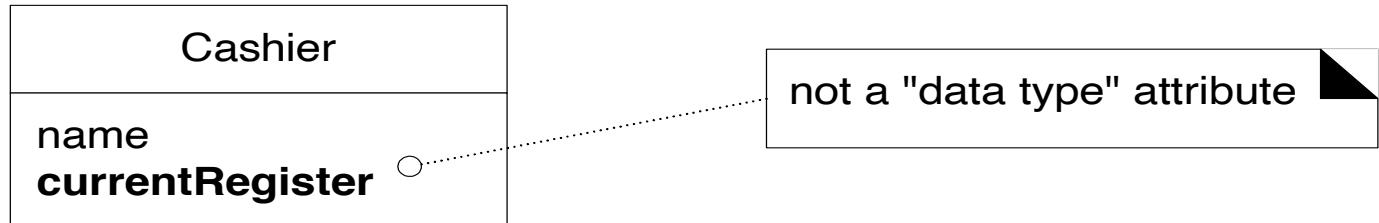


One instance of a Customer may be renting zero or more Videos.

One instance of a Video may be being rented by zero or one Customers.

Association vs. Attribute

Worse

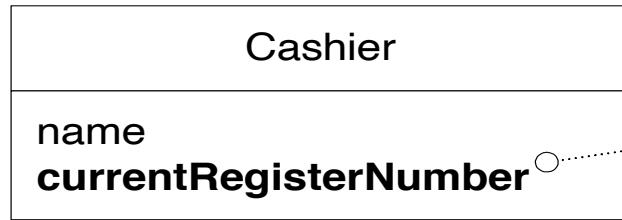


Better



Do not use attributes to represent foreign keys

Worse

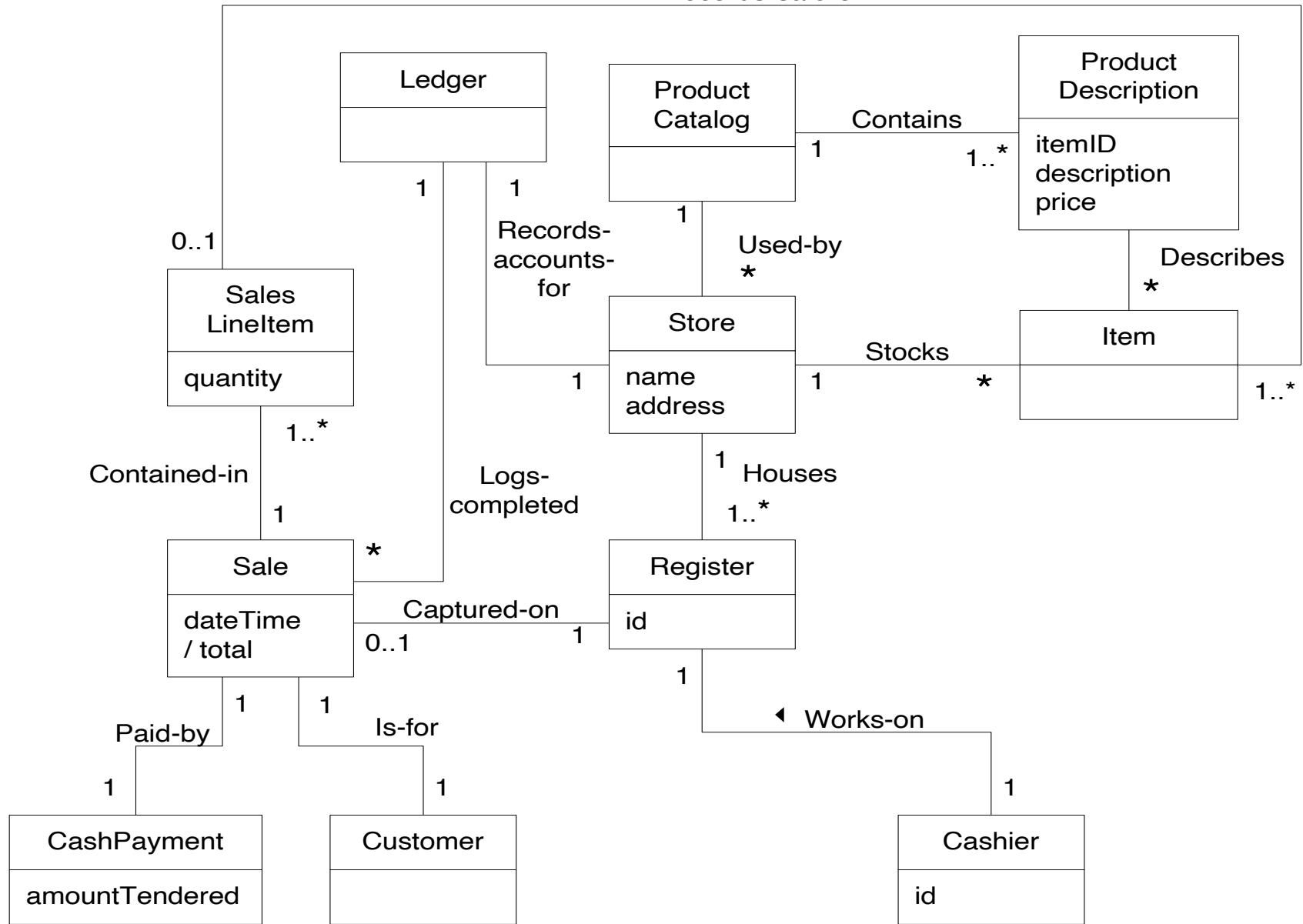


a "simple" attribute, but being used as a foreign key to relate to another object

Better



Records-sale-of



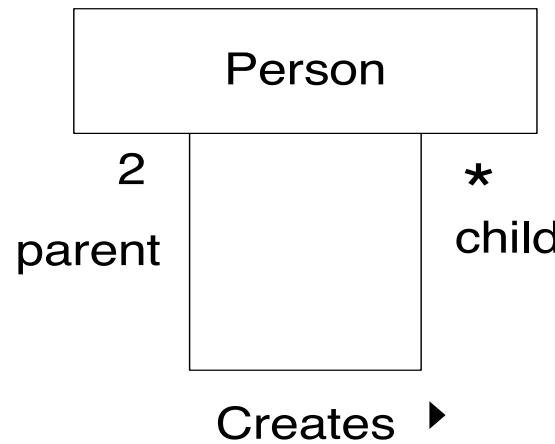
Association Roles

- When a class is part of an association it plays a *role* in the relationship.
- You can name the role that a class plays in an association by placing the name at the class's association end.
- Formally, a class role is the set of objects that are linked via the association.

Examples of Association Roles



role name
describes the role of a city in the
Flies-to association



Association Constraints

- Users can define constraints on how objects are linked via associations.
- Constraints can be expressed in the *Object Constraint Language* (OCL).

Aggregation

- Aggregation is a special form of association
 - reflect whole-part relationships
- The whole delegates responsibilities to its parts
 - the parts are subordinate to the whole
 - unlike associations in which classes have equal status

UML Forms of Aggregation

- Composition (strong aggregation)
 - parts are existent-dependent on the whole
 - parts are generated at the same time, before, or after the whole is created (depending on cardinality at whole end) and parts are deleted before or at the same time the whole dies
 - multiplicity at whole end must be 1 or 0..1

Composition

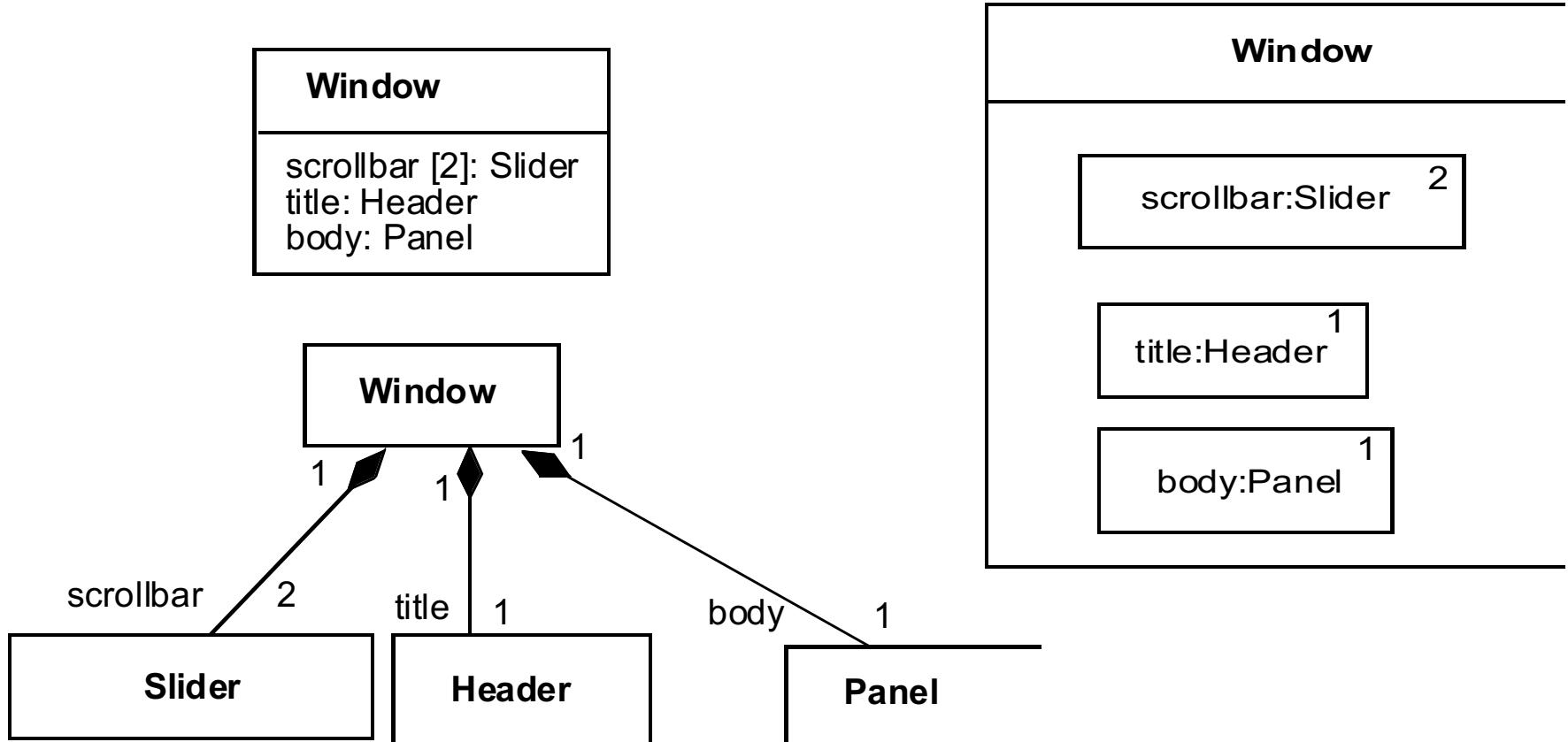


Fig. 3-45, UML Notation Guide

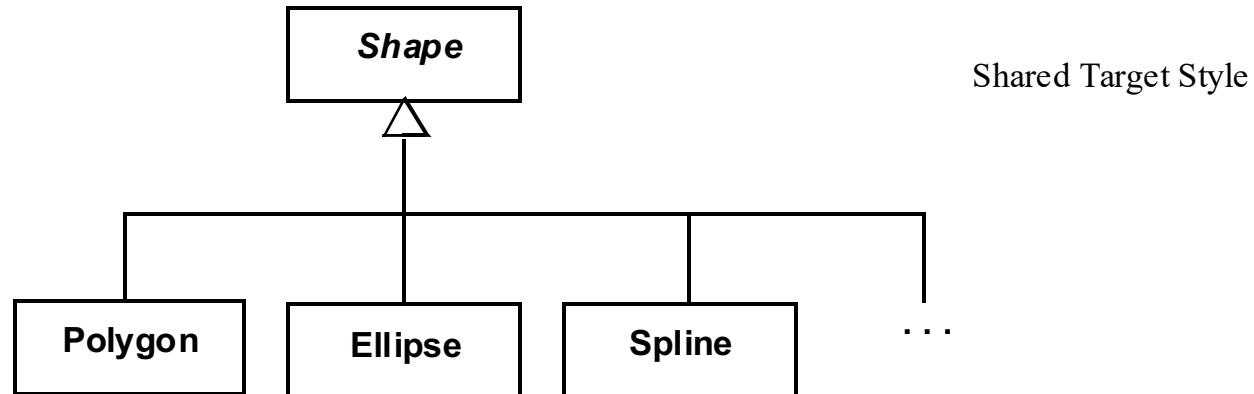
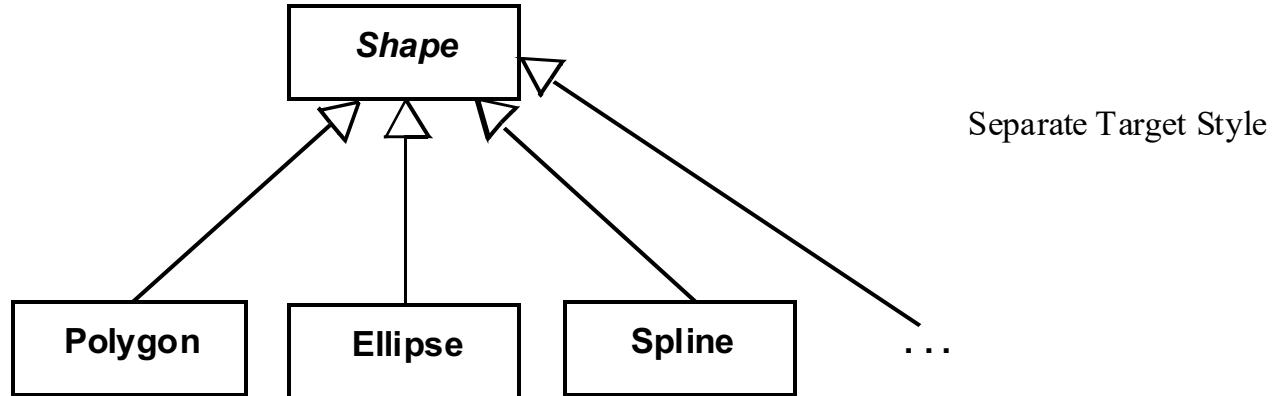
Generalization/Specialization

- A generalization (or specialization) is a relationship between a general concept and its specializations.
 - Objects of specializations can be used anywhere an object of a generalization is expected (but not vice versa).
- Example: *Mechanical Engineer* and *Aeronautical Engineer* are specializations of *Engineer*

Rendering Generalizations

- Generalization is rendered as a solid directed line with a large open arrowhead.
 - Arrowhead points towards generalization
- A discriminator can be used to identify the nature of specializations

Generalization



Generalization

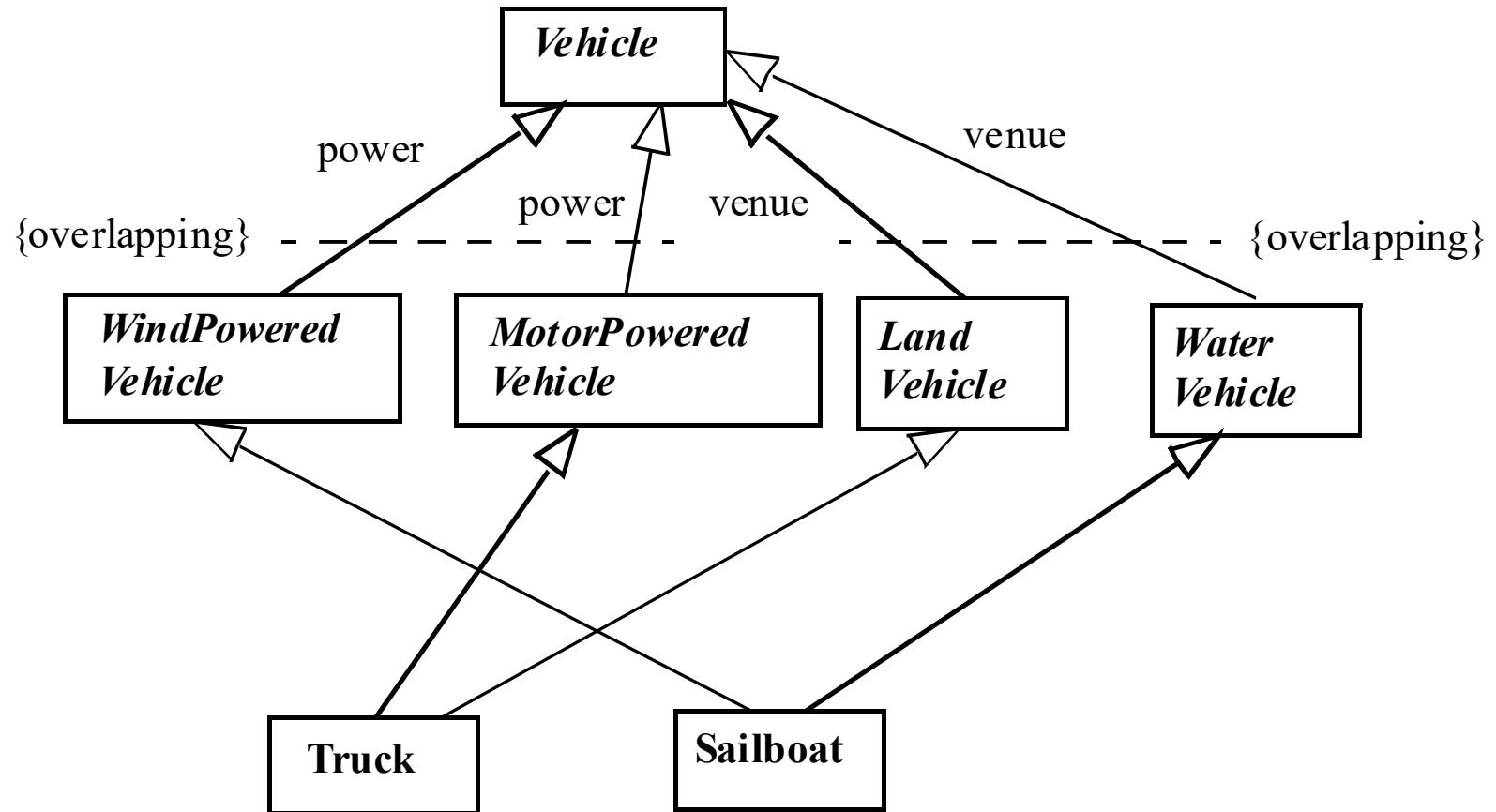
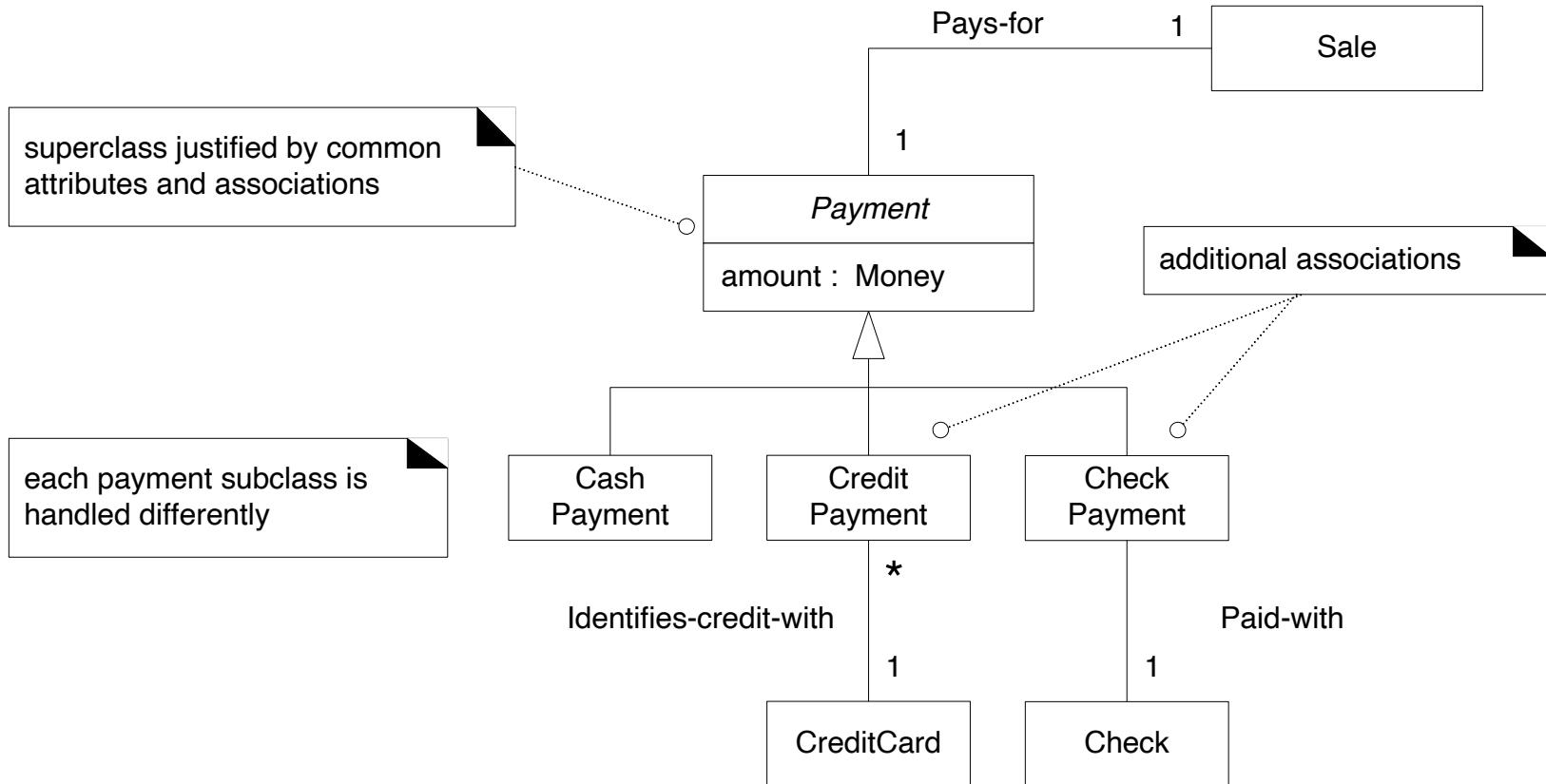
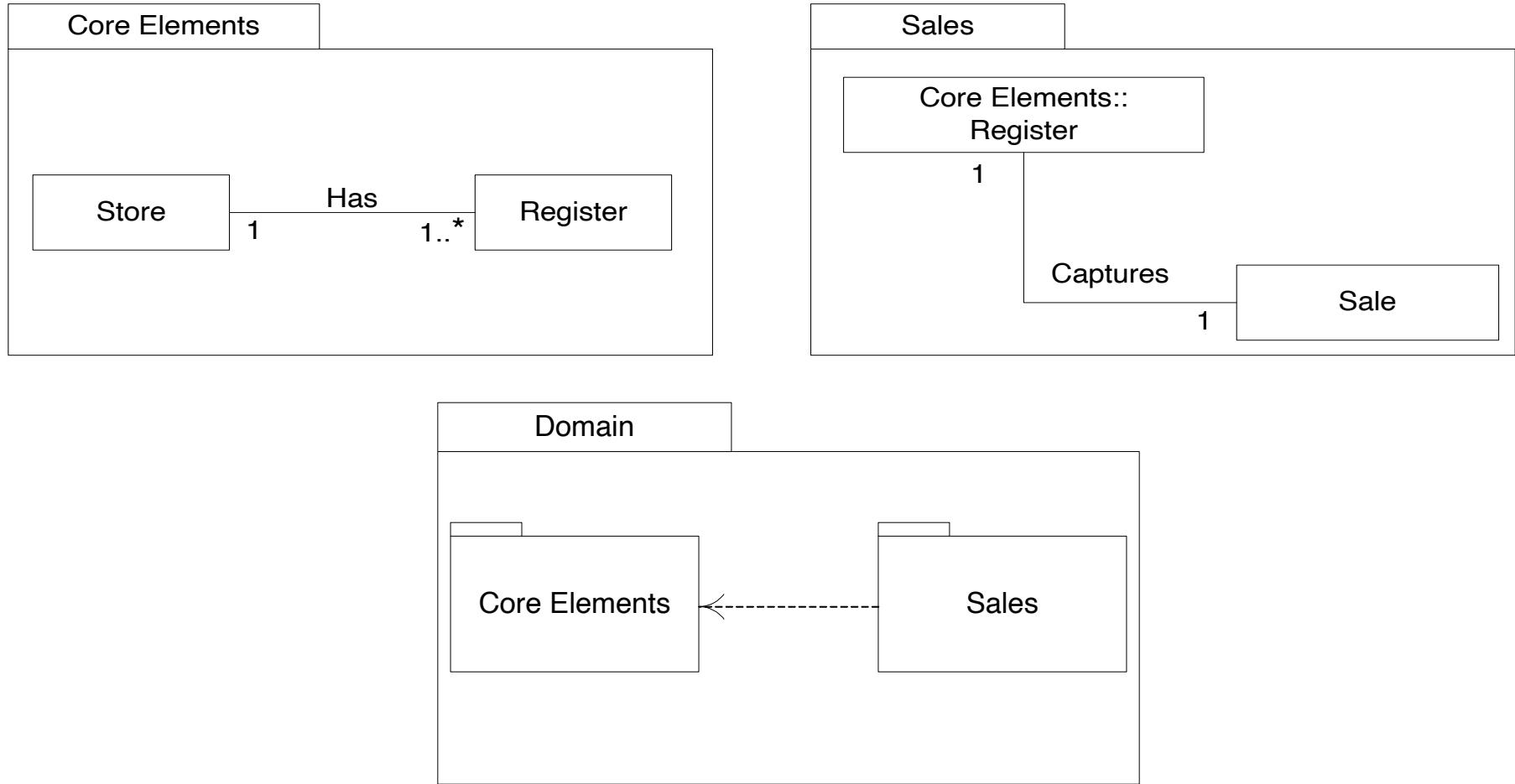


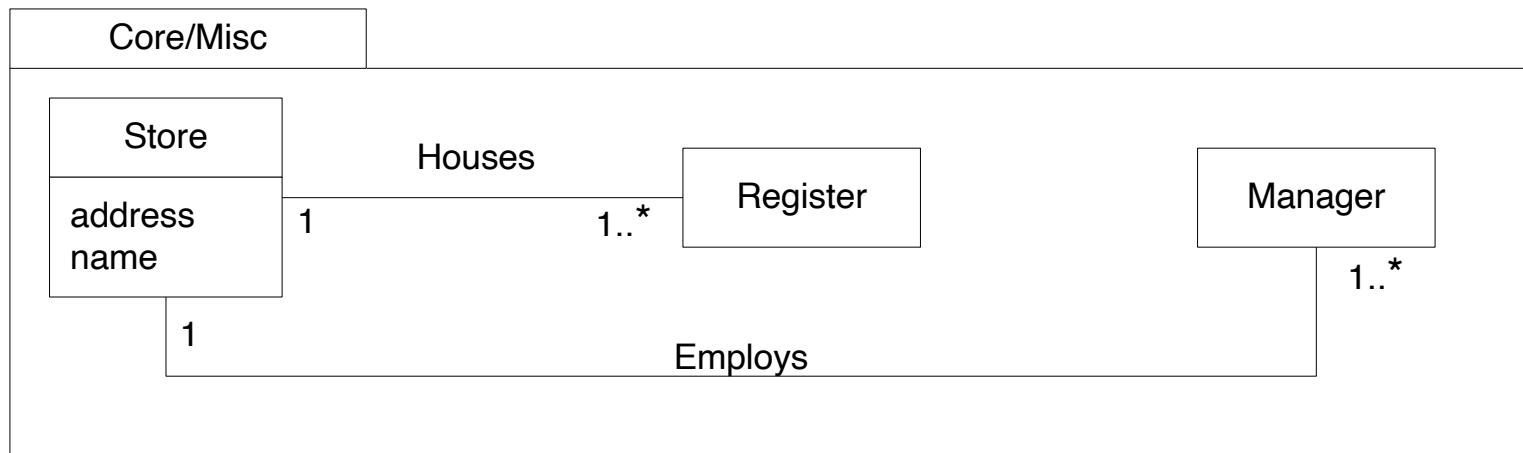
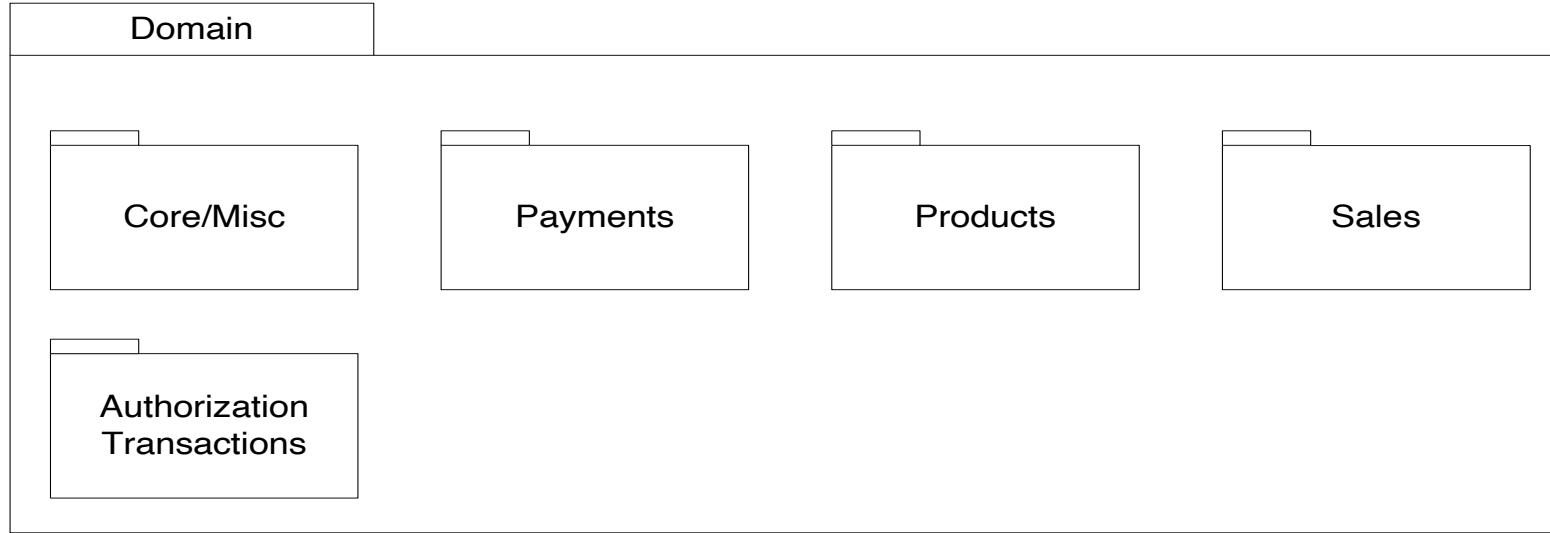
Fig. 3-48, *UML Notation Guide*

Inheritance of associations

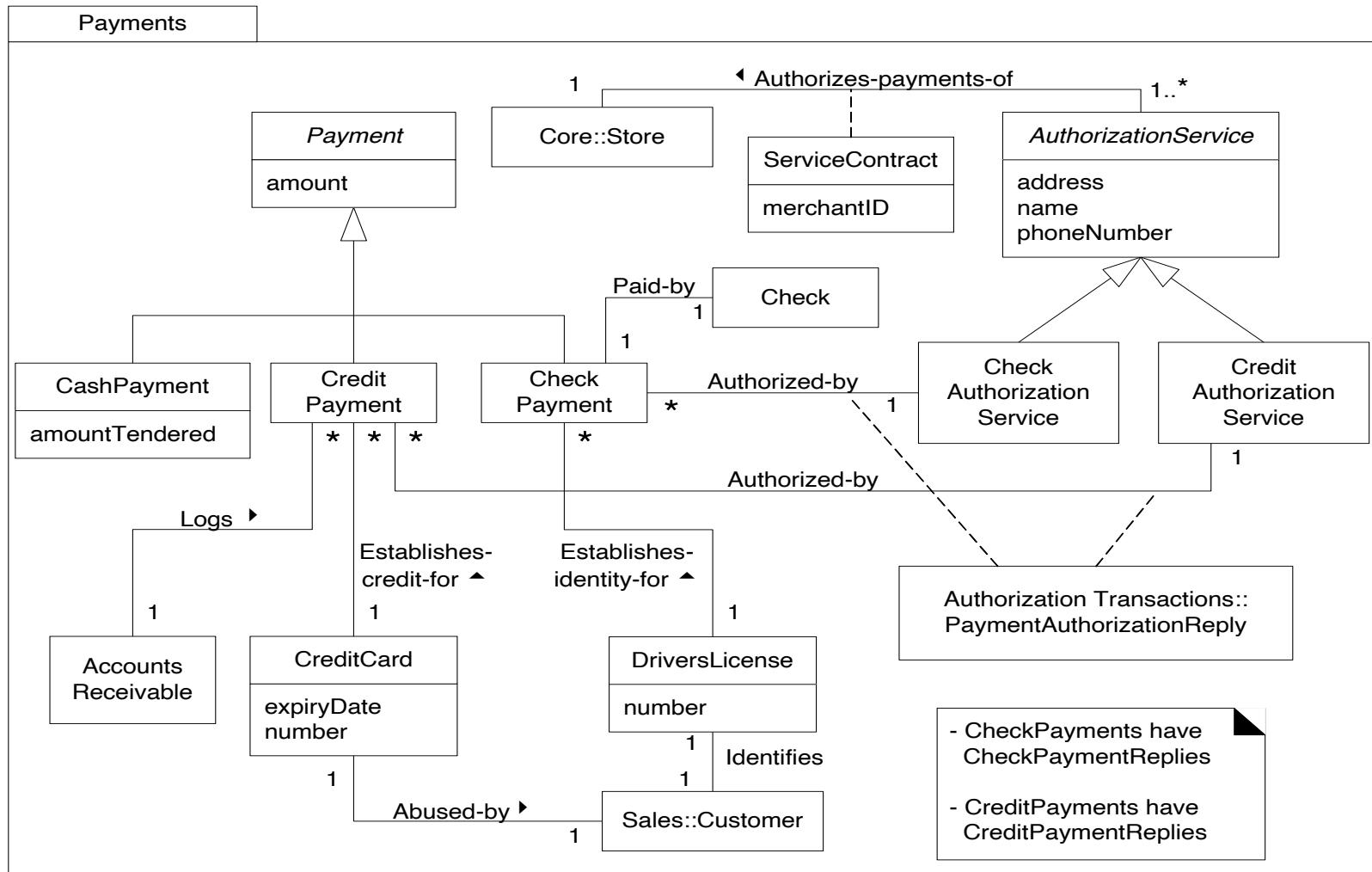


Handling Large Domain Models

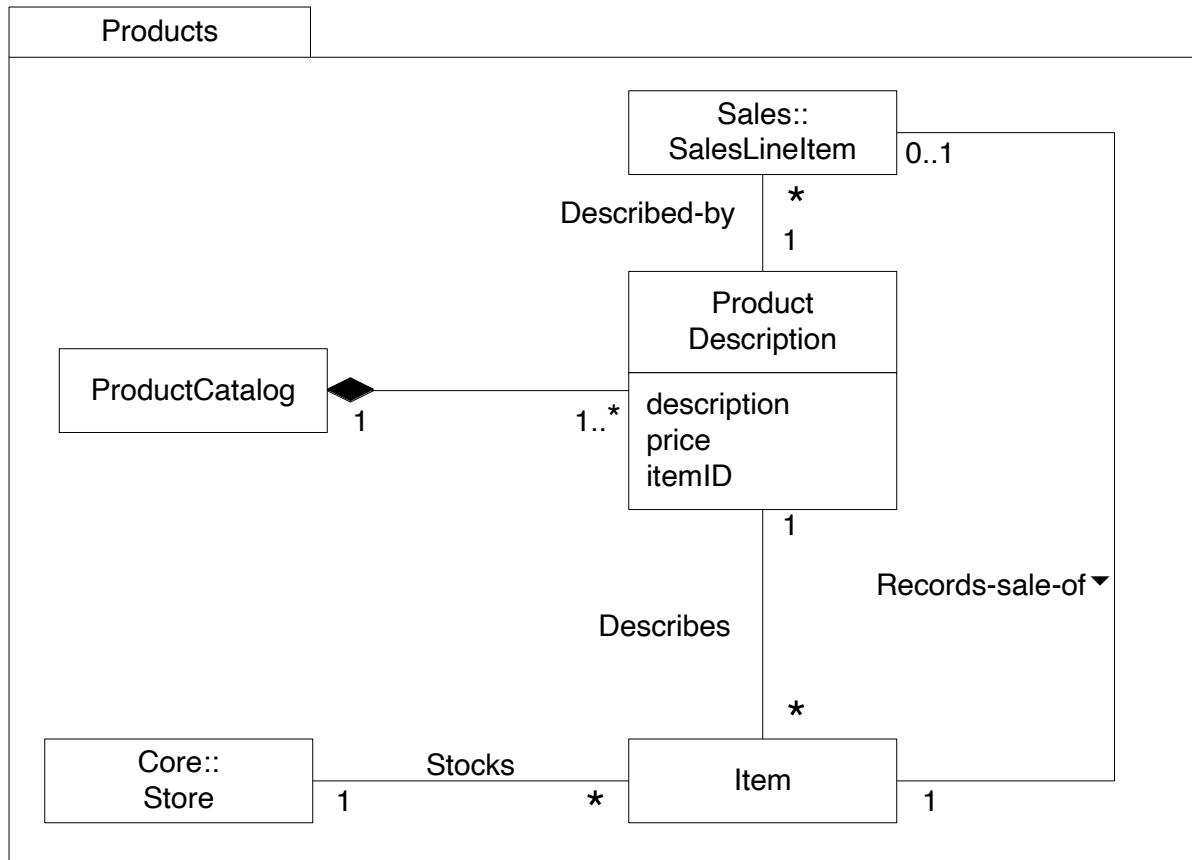




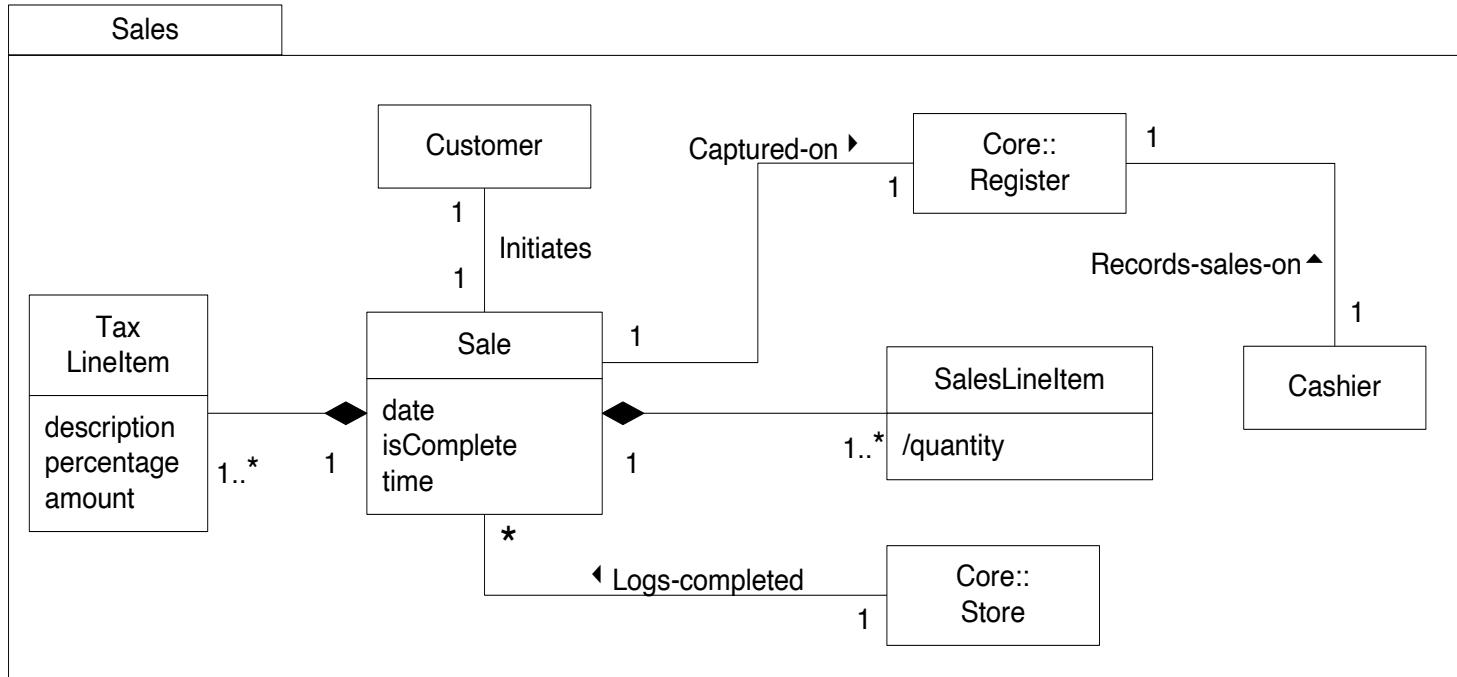
Payments Package



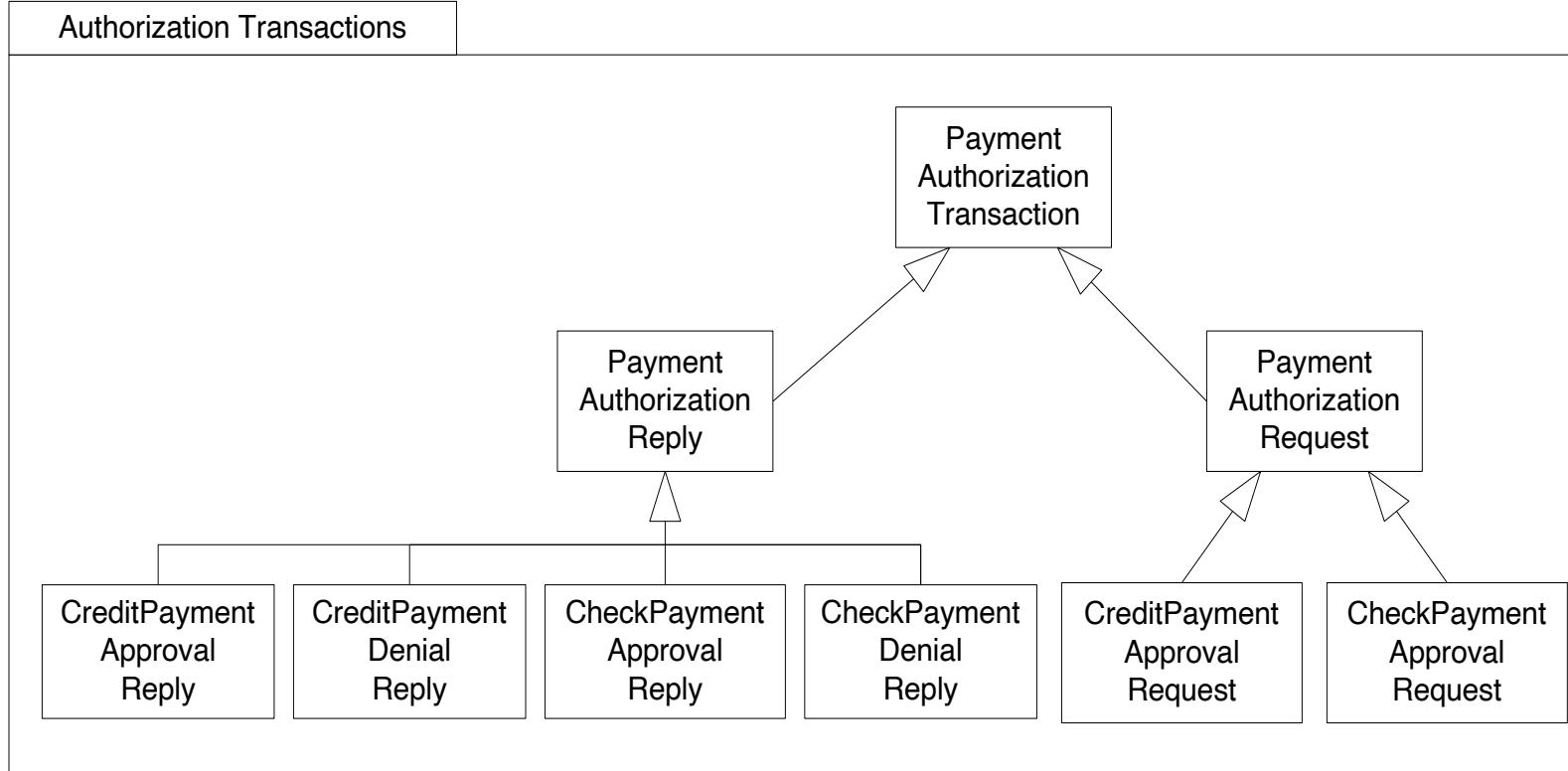
Products Package



Sales Package



Authorization Transactions Package



Summary

- Requirements models are used to represent conceptual elements and their relationships at the problem level
- UML may be used as a specification language. However UML is semi-formal
- How do we write formal specifications?
 - For example, constraints may be added to make it more formal. Object Constraint Language (OCL) may be used for this purpose

Code Inspections

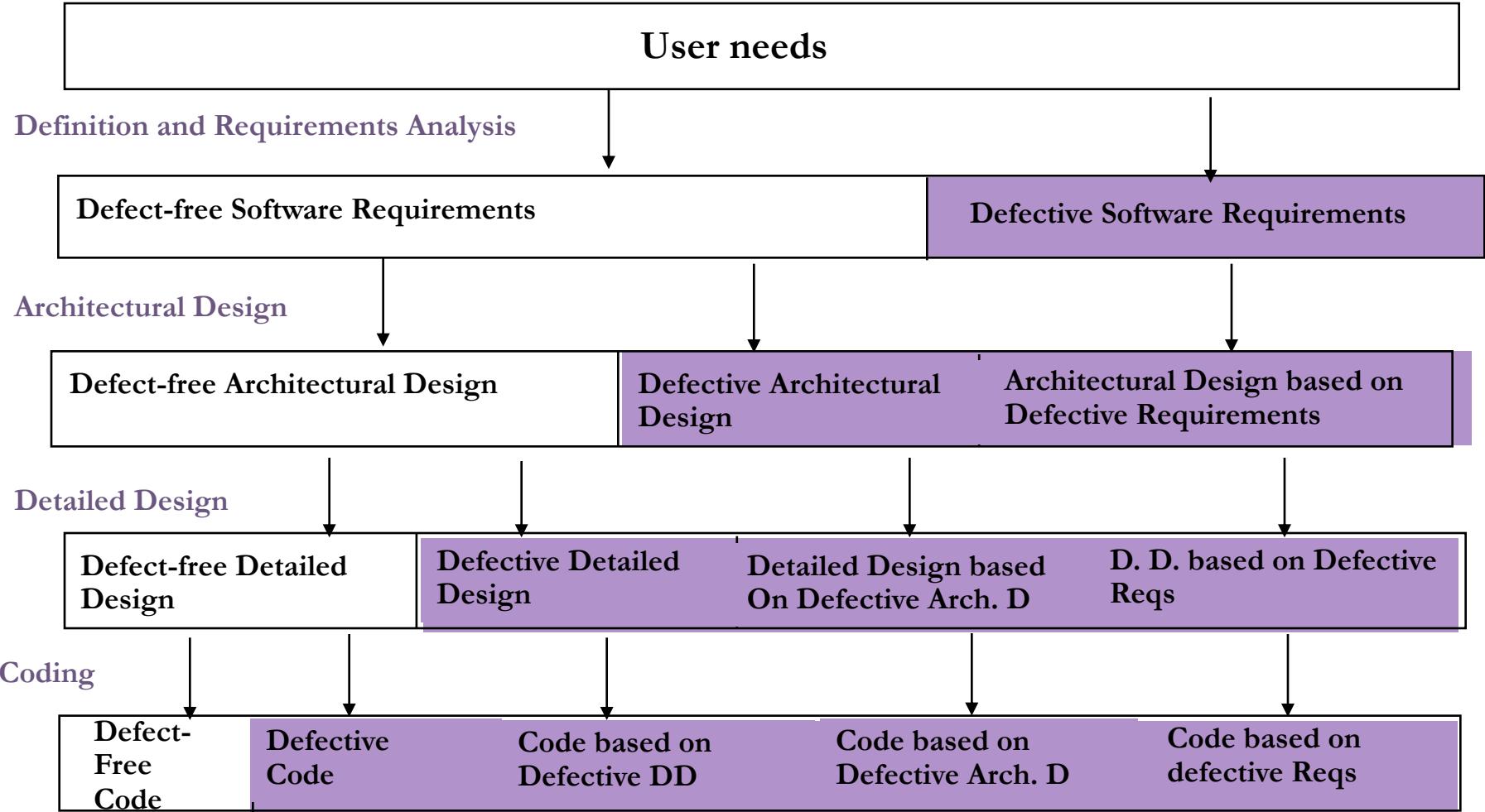
DASS , Spring 2024
IIIT Hyderabad

Motivation

- ▶ Detecting errors late in the development cycle is expensive
 - ▶ E.g. A requirements defect that is found only at testing costs almost 100 times more to fix than if it had been found and fixed at requirements itself
 - ▶ Need to rework not only the requirements doc, but all other deliverables produced from it: change the design, change the code, rerun tests!
 - ▶ The earlier in the lifecycle we find problems, the cheaper they are to fix



Error Introduction and Propagation



Motivation (2)

- ▶ Multiple stages of defect removal
 - ▶ If we inspect each deliverable (requirements, design, code), and then do multiple stages of testing (unit tests, integration tests, system tests), then we get many chances to find defects
- ▶ Like filtering multiple times: the result is much cleaner!

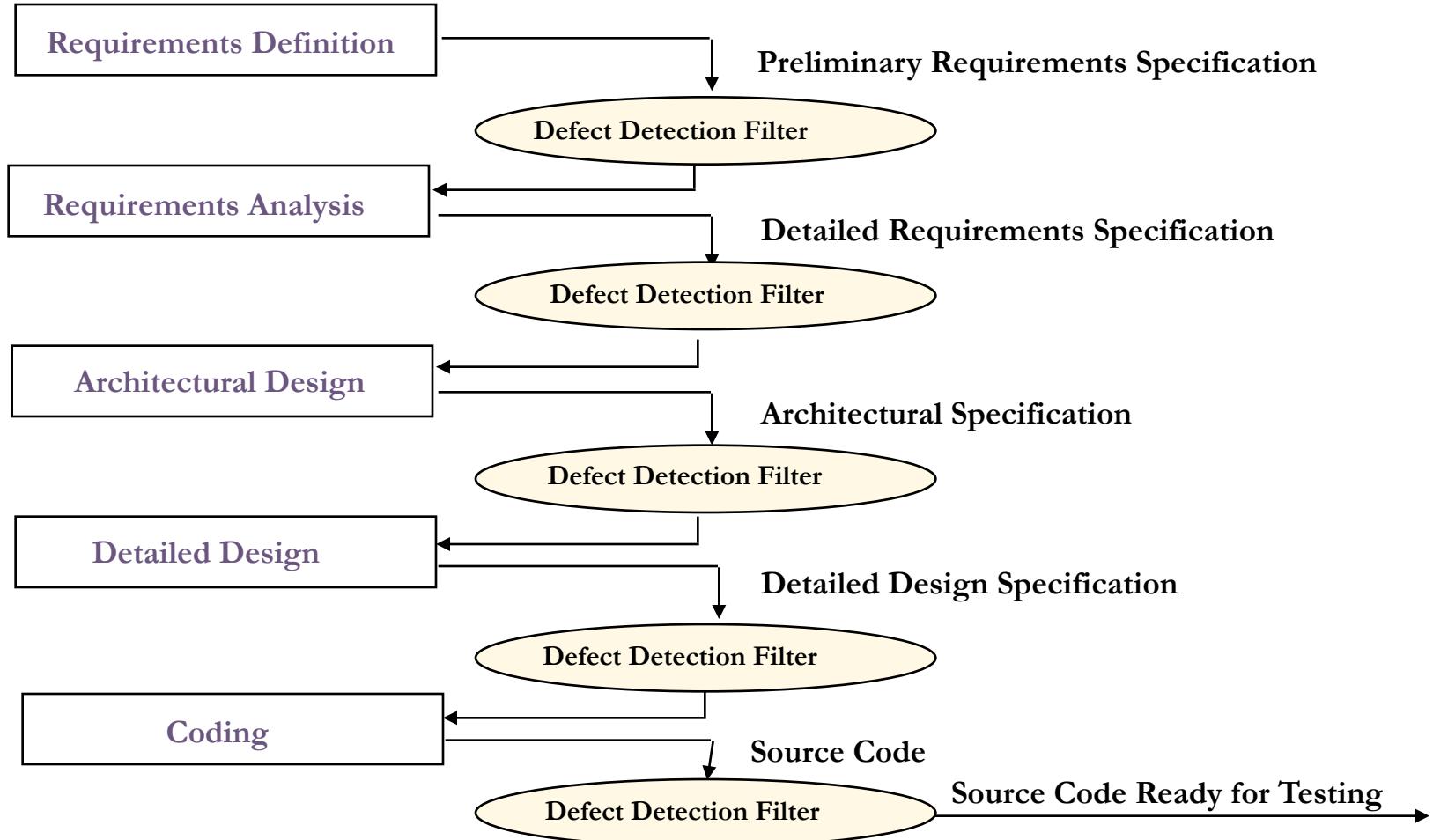


Effectiveness of inspections

- A study showed that, as a rule of thumb, each defect identified during inspection saves around 9 hours of time downstream.
- AT&T found inspections led to 14% increase in productivity and tenfold increase in quality.
- HP found 80% of the errors detected during inspections were unlikely to be caught by testing.
- HP, Shell Research, Bell Northern, and AT&T all found inspections 20 to 30 times more efficient at testing in detecting errors.
- IBM found inspections gave a 23% increase in productivity and a 38% reduction in bugs detected after unit test.



Multiple Stages of Defect Removal

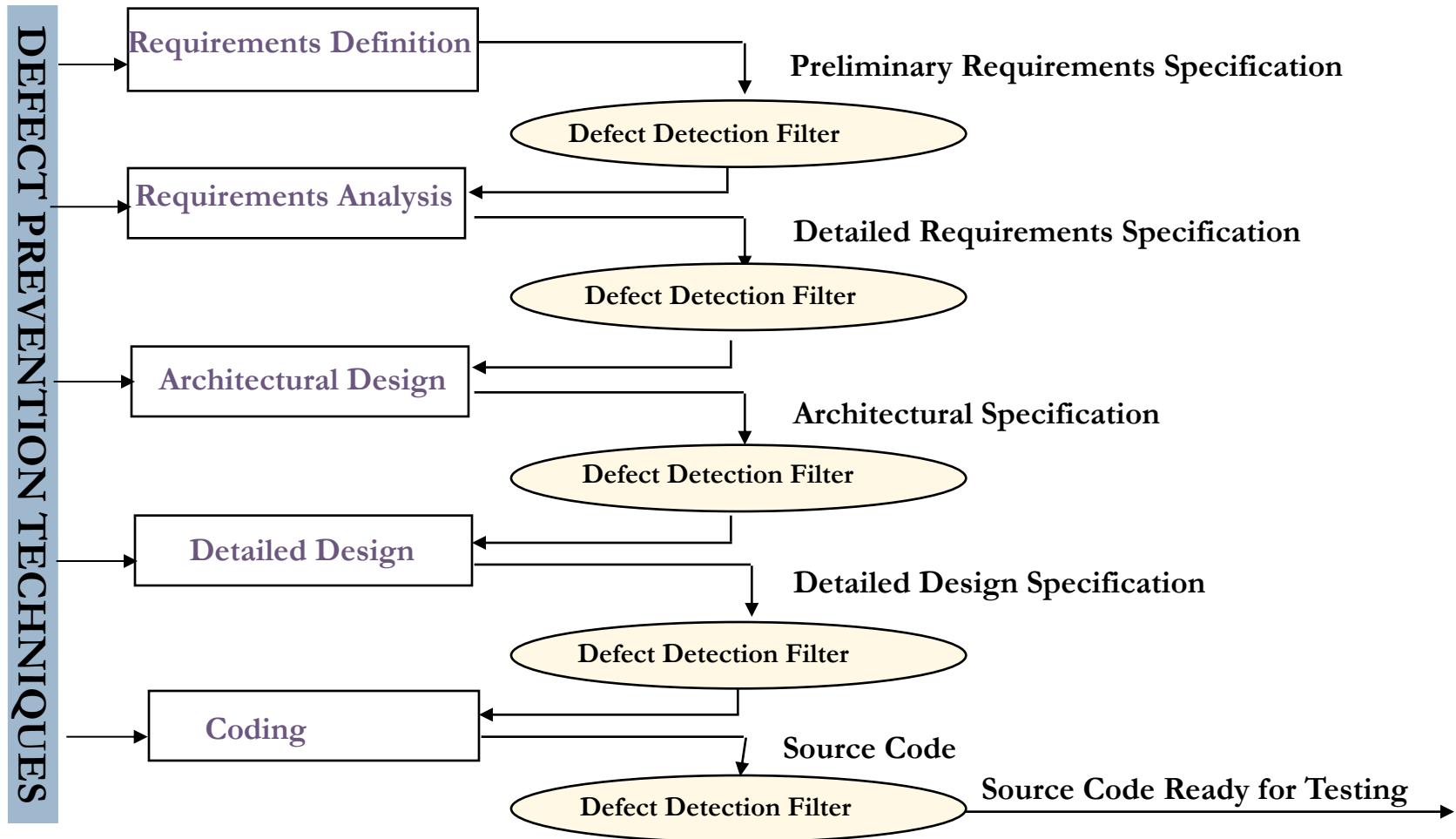


Defect Prevention

- ▶ In addition to removing defects through inspections, we can eliminate defects using
 - ▶ Checklists: common mistakes, concerns to address, activities to do
 - ▶ Templates: standard document formats that list the different aspects to be covered
 - ▶ Reduce work and avoid incompleteness
 - ▶ Tools and workflow automation
 - ▶ Avoid errors, inconsistencies and missing steps
 - ▶ Reduce effort too!



Quality-Centered Development

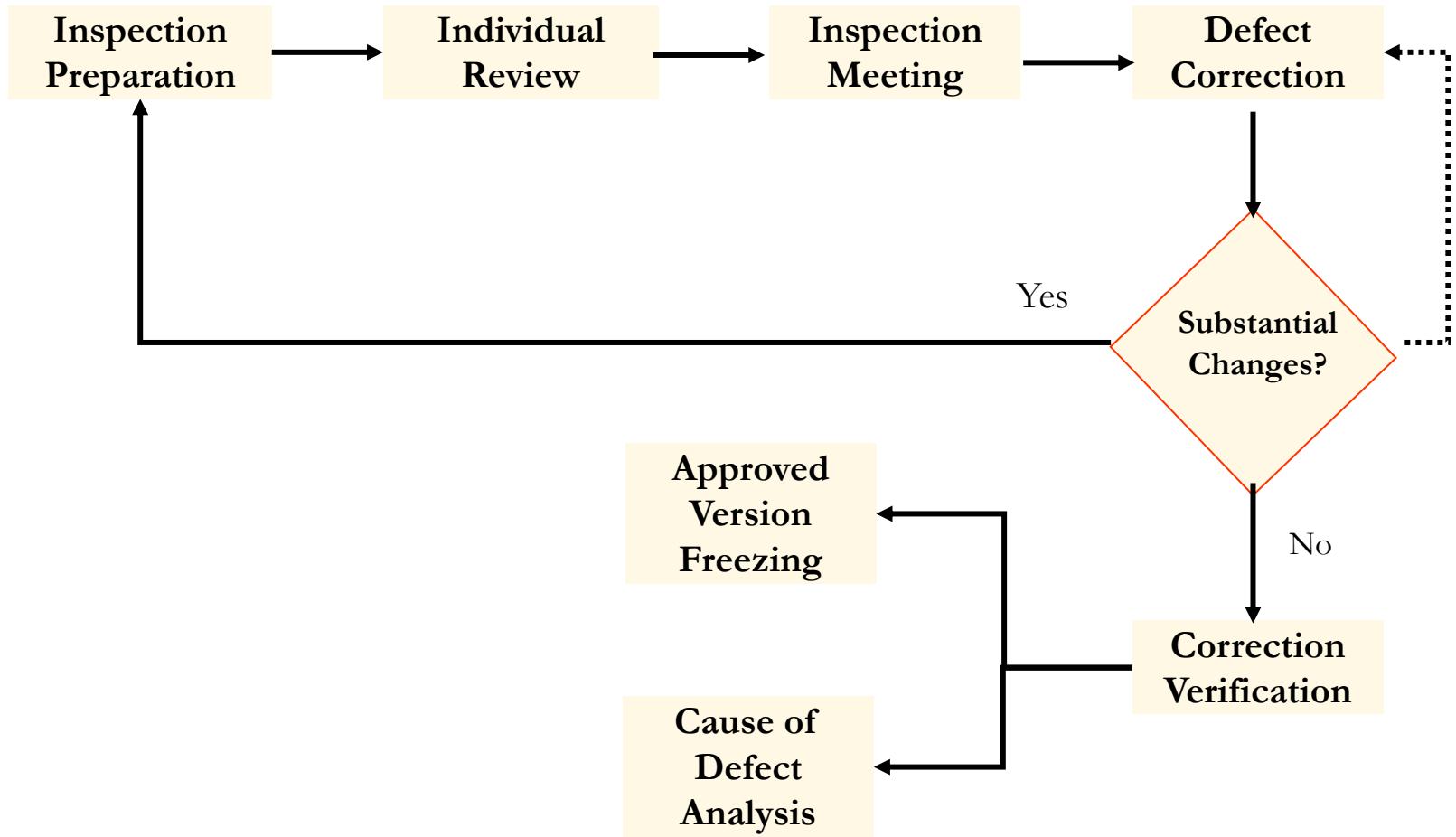


Inspections

- ▶ A group of people review an artifact (code, documents) to find defects and identify opportunities for improvement
- ▶ Can be used for any document or code produced during the development
 - ▶ Preferably, **all** major development artifacts should be inspected
- ▶ Each reviewer spends several hours going through the artifact and finding problems and possible improvements
- ▶ Hold a review meeting to discuss the inputs from each reviewer and identify the problems that need fixing
- ▶ Fix problems, re-review if necessary



Inspection Phases



Formal inspection process

- ▶ Defined roles: Author, Reader, Moderator, Scribe, Inspectors
- ▶ **Author** - distributes the artifact that he/she has coded
- ▶ **Reader** interprets the code for the inspectors
 - ▶ If reader is different from author, reduces possibility of author propagating their own misunderstandings
- ▶ **Inspectors** prepare comments before meeting, provide their inputs and contribute to discussions during meeting
- ▶ **Moderator** keeps the discussions on track, also responsible for checking later that the problems found have been fixed
- ▶ **Scribe** ensures that problems found get recorded



Code Inspection Report

Team / Project	<Team name / Project name in Black.>
Component	<Identification of inspected component in Black.>
Date	<Date in black (e.g., Jan 21, 2008) in Black.>
Author	<Name of the component's primary author in Black.>
Moderator	<Name of inspection moderator in Black.>
Reader	<Name of inspection reader in Black.>
Scribe	<Name of inspection scribe in Black.>
Other Inspectors	<Names of other inspectors in Black.>

Defect No.	Line #	Severity H,M,L	Description
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			



```
1  /*
2   * PrintSpooler - Models a print spooler using a StringBuffer. Allows the user to get
3   * an instance of the spooler and add strings or characters. The
4   * contents of the spooler can then be sent to the system output
5   * device.
6   *
7   * Note that PrintSpooler is a Singleton - only a single instance
8   * of it can exist at anytime. The method getPrintSpooler insures
9   * this by only performing one instance creation and then simply
10  * returning the same object reference to subsequent requests.
11 */
12 public class PrintSpooler {
13
14     private static PrintSpooler spooler = new PrintSpooler(); // the one and only
15
16     public StringBuffer spoolBuffer = null; // instance of PrintSpooler
17
18     public PrintSpooler(){ // constructor for object creation
19         };
20
21     public PrintSpooler getPrintSpooler(){ // only allow one-time creation of
22         PrintSpooler mySpooler = spooler;
23         if ( spooler == null)
24             mySpooler = new PrintSpooler();
25         return mySpooler;
26     }
27
28     public void addString( String s){ // add a String to the spooler
29         spoolBuffer.append( s );
30     }
31
32     public void addChar( char theChar, int length){ // add a character, allow repeating
33         for ( int i=1; i<length ; i++)
34             spoolBuffer.append( theChar);
35     }
36
37     public void print(){ // send to system.out device, add header info
38         spoolBuffer.insert(0, "PrintSpooler - Version 1.3 Header ");
39         System.out.print(spoolBuffer);
40     }
41
42     public void clear(){ // clear the spooler
43         spooler = new PrintSpooler();
44     }
}
```

```
1  /*
2   * PrintSpooler - Models a print spooler using a StringBuffer. Allows the user to get
3   * an instance of the spooler and add strings or characters. The
4   * contents of the spooler can then be sent to the system output
5   * device.
6   *
7   * Note that PrintSpooler is a Singleton - only a single instance
8   * of it can exist at anytime. The method getPrintSpooler insures
9   * this by only performing one instance creation and then simply
10  * returning the same object reference to subsequent requests.
11 */
12 public class PrintSpooler {
13
14     private static PrintSpooler spooler = new PrintSpooler(); null; // the one and only
15
16     public private StringBuffer spoolBuffer = null; // instance of PrintSpooler
17
18     public private PrintSpooler(){ // model using a StringBuffer
19         spoolBuffer = new StringBuffer();
20     }
21
22     public static PrintSpooler getPrintSpooler(){ // constructor for object crea-
23         PrintSpooler mySpooler = spooler; // only allow one-time creation of
24         if ( spooler == null) // PrintSpooler, else return referece,
25             mySpooler spooler = new PrintSpooler();
26         return mySpooler spooler;
27     }
28
29     public void addString( String s){ // add a String to the spooler
30         spoolBuffer.append( s );
31     }
32
33     public void addChar( char theChar, int length){ // add a character, allow repeating
34         for ( int i=1; i < length i <= length; i++) // of same character
35             spoolBuffer.append( theChar);
36     }
37
38     public void print(){ // send to system.out device, add header info
39         spoolBuffer.insert(0, "PrintSpooler - Version 1.3 Header"); // make a constant
40         System.out.print(spoolBuffer);
41     }
42
43     public void clear(){ // clear the spooler
44         spooler = new PrintSpooler();
45         spoolBuffer = new StringBuffer();
46     }

```

Other benefits of Inspections

- ▶ Team members get familiar with the code
 - ▶ Backup if someone is unavailable
- ▶ More uniform design and coding practices across team
- ▶ Knowledge sharing
- ▶ Shared understanding & improved communication
 - ▶ Identify miscommunication and misperceptions
- ▶ More perspective on how everything comes together

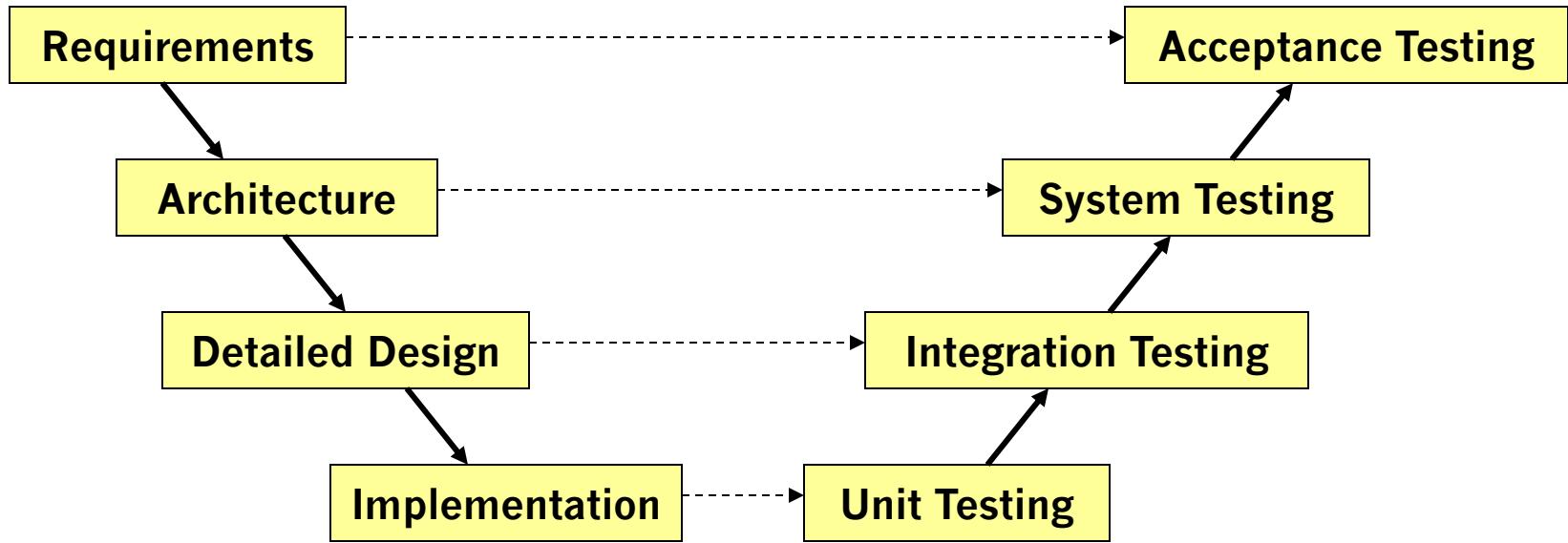




Software Testing

Some Material adapted from Lethbridge & Laganiere;
Some Material adapted from Pressman.

Testing phases: V model



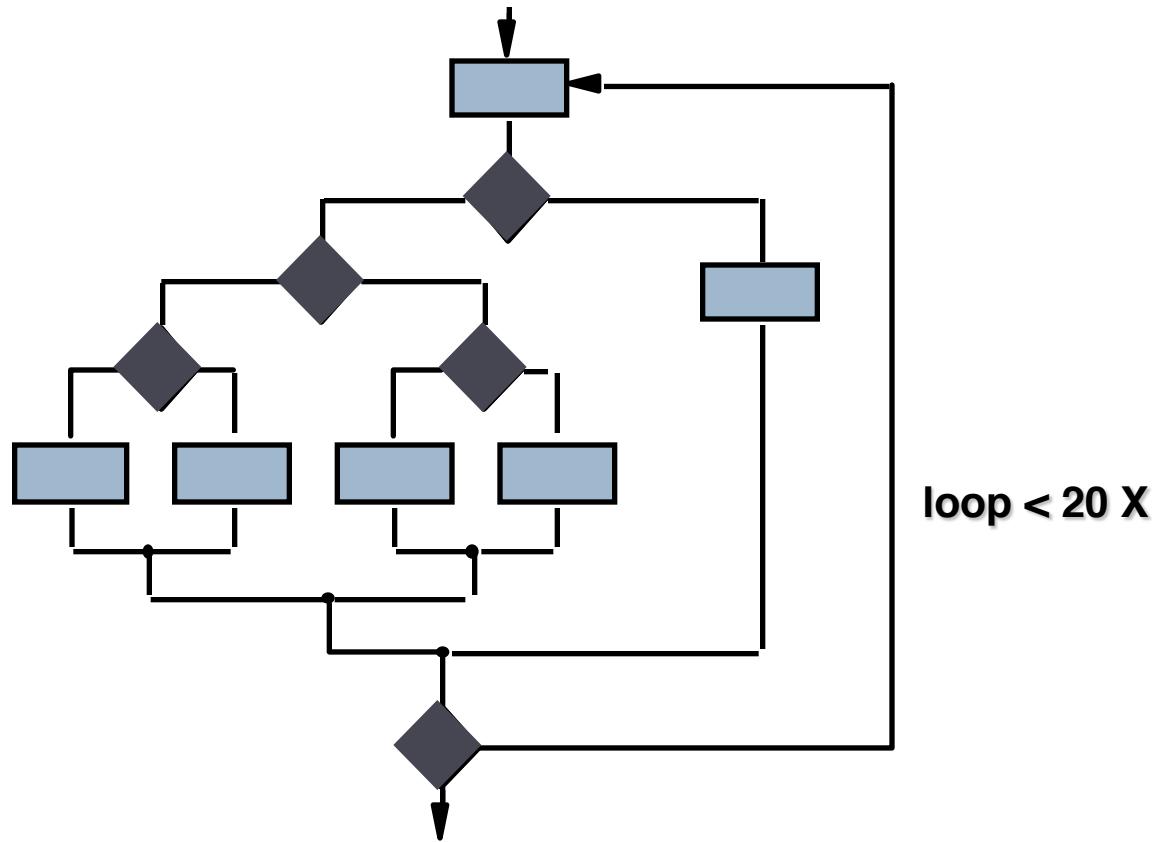
A lifecycle view that shows relationships between development and test phases

Testing Phases

- ▶ Unit Testing
 - ▶ Developer tests individual modules
- ▶ Integration testing
 - ▶ Put modules together, try to get them working together
 - ▶ Integration testing is complete when the different pieces are able to work together
- ▶ System testing
 - ▶ Testing of entire deliverable against specs
- ▶ Acceptance testing
 - ▶ Testing against user needs, often by the user

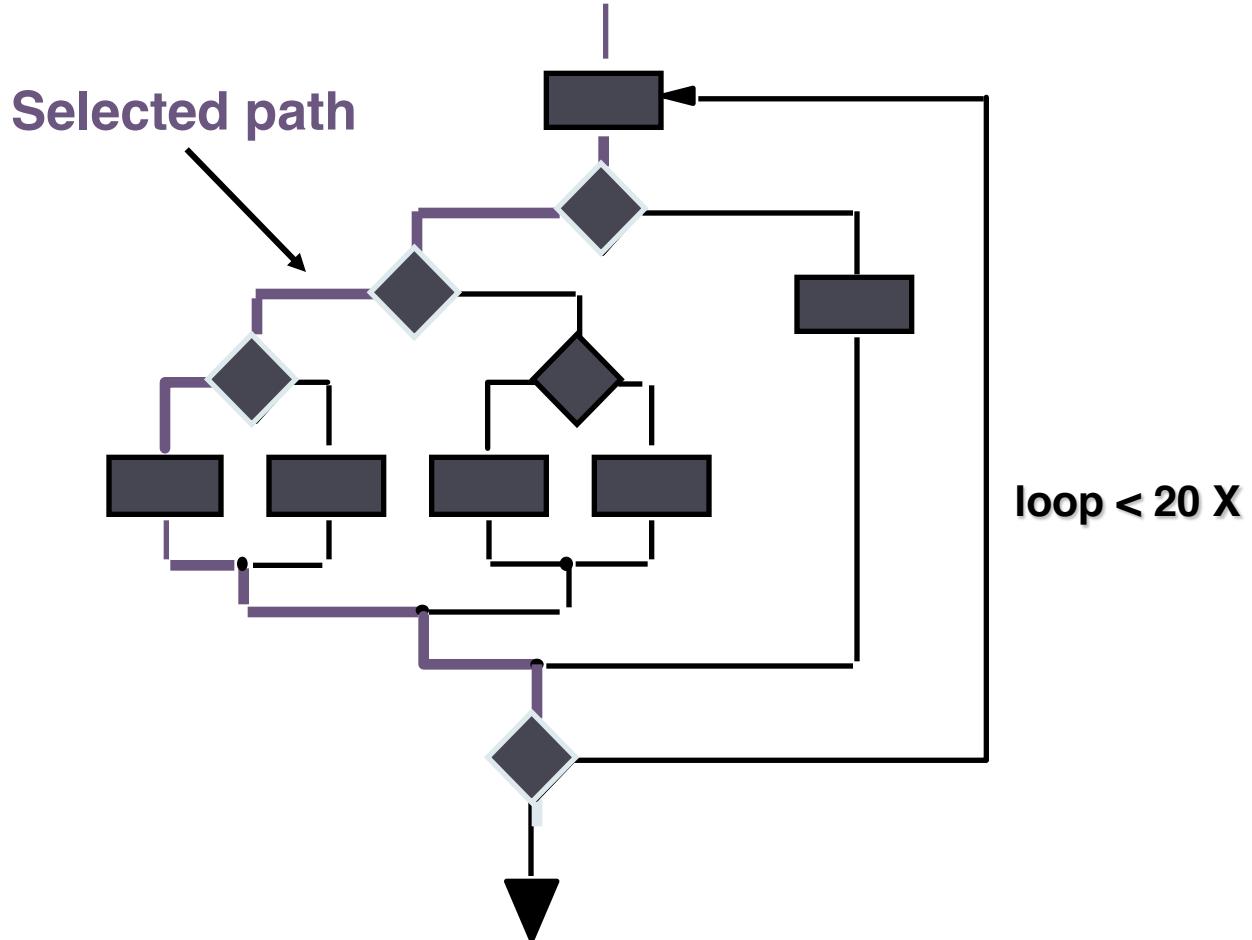


Exhaustive Testing



There are 10 possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Selective Testing



Coverage

- ▶ Function coverage: Each function/method executed by at least one test case
- ▶ Statement coverage: Each line of code covered by at least one test case (need more test cases than above)
- ▶ Path coverage: Every possible path through code covered by at least one test case (need lots of test cases)



Equivalence classes

- ▶ It is inappropriate to test by *brute force*, using *every possible* input value
 - ▶ Takes a huge amount of time
 - ▶ Is impractical
 - ▶ Is pointless!
- ▶ You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms.
 - ▶ Such groups are called *equivalence classes*.
 - ▶ A tester needs only to run one test per equivalence class
 - ▶ The tester has to
 - understand the required input,
 - appreciate how the software may have been designed



Example of equivalence classes

Valid input is a month number (1-12)

- ▶ Equivalence classes are: $[-\infty..0]$, $[1..12]$, $[13..\infty]$



Combinations of equivalence classes

- ▶ Combinatorial explosion means that you cannot realistically test every possible system-wide equivalence class.
 - ▶ If there are 4 inputs with 5 possible values there are 5^4 (i.e.625) possible system-wide equivalence classes.
- ▶ You should first make sure that at least one test is run with every equivalence class of every individual input.
- ▶ You should also test all combinations where an input is likely to *affect the interpretation* of another.
- ▶ You should test a few other random combinations of equivalence classes.



Example equivalence class combinations

- ▶ One valid input is either ‘Metric’ or ‘US/Imperial’
 - ▶ Equivalence classes are:
 - ▶ Metric, US/Imperial, Other
- ▶ Another valid input is maximum speed: 1 to 750 km/h or 1 to 500 mph
 - ▶ Validity depends on whether metric or US/imperial
 - ▶ Equivalence classes are:
 - ▶ $[-\infty..0]$, $[1..500]$, $[501..750]$, $[751..\infty]$
- ▶ Some test combinations
 - ▶ Metric, $[1..500]$ valid
 - ▶ US/Imperial, $[501..750]$ invalid
 - ▶ Metric, $[501..750]$ valid
 - ▶ Metric, $[501..750]$ valid



Testing at boundaries of equivalence classes

- ▶ More errors in software occur at the boundaries of equivalence classes
- ▶ The idea of equivalence class testing should be expanded to specifically test values at the extremes of each equivalence class
 - ▶ E.g. The number 0 often causes problems
- ▶ *E.g.*: If the valid input is a month number (1-12)
 - ▶ Test equivalence classes as before
 - ▶ Test 0, 1, 12 and 13 as well as very large positive and negative values



Test Case

- ▶ A triplet of [Input, Steps, Expected behavior]
- ▶ (IEEE Standard 829-1983) *Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.”*
- ▶ **Purpose:** to **get some information about the system behavior**



Information that can be gained from Test Cases

- ▶ Find defects
 - ▶ A test is run in order to trigger failures that expose defects.
- ▶ Assess conformance to specification.
- ▶ Assess conformance to standards or regulations.
- ▶ Find safe scenarios for use of the product
- ▶ *Assess quality*
- ▶ Block premature product releases.
- ▶ Help managers make ship / no-ship decisions.
- ▶ Minimize technical support costs.

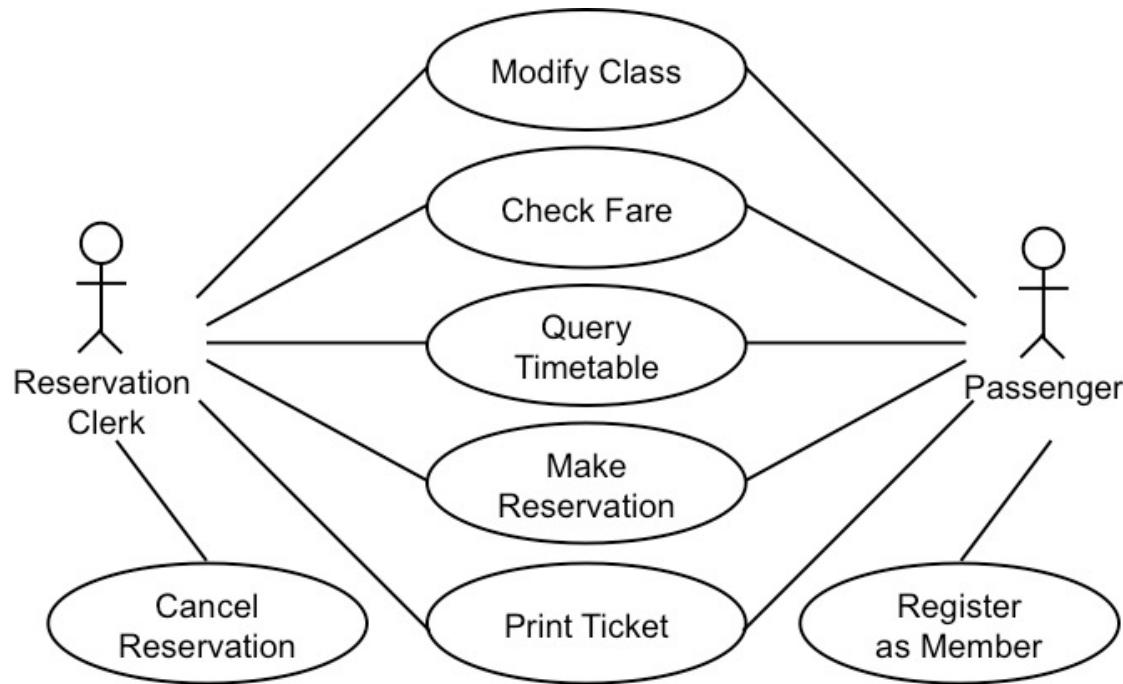


Identifying Scenarios

- ▶ Use case has one basic flow (B) and 0..* alternate flows (A1..An)
- ▶ Notation:
 - ▶ Basic flow: B
 - ▶ Alternative flows: A1, A2, A3, ...
 - ▶ Steps in a basic flow: B1, B2, B3, ...
 - ▶ Steps in alternative flow 1: A1.1, A1.2, A1.3, ...
 - ▶ Steps in alternative flow 2: A2.1, A2.2, A2.3, ...
- ▶ Scenario - A possible path through basic and alternate flows that an actor can take.
- ▶ Can be Usage, growth or exploratory.
 - ▶ Usage from use cases, mostly for functional testing
 - ▶ Growth and exploratory for system level testing



Writing test cases



Assume the simplified use case from IRCTC.co.in.



Make Reservation Basic Flow

Use Case	Make Reservation
Actors	Passenger, Reservation Clerk
Purpose	Reserve a seat or berth
Pre-condition	Authorized user is logged in
Overview	Allows the user to make a reservation for a journey.
Normal Flow	<ol style="list-style-type: none">1. User logs in2. User specifies the train and journey details.3. User specifies passenger details4. User specifies payment details5. User selects proceed for transaction
Post Condition	1. User is taken to the payment gateway. 2. Use case make payment initiates
Non-functional requirements	Performance, scalability



Alternate Flows

- ▶ A1 No matching trains found
 - ▶ A2 Source Destination require modification
 - ▶ A3 Session Expired
 - ▶ A4 Continue booking another ticket
 - ▶ A5 Leave in between
 - ▶ A6 Booking not permitted right now (time, date constraints due to business rules)
 - ▶ A7 Booking senior citizen ticket
 - ▶ A8 Changing boarding point
 - ▶ A9 Check availability
 - ▶ A10 Close transaction in between
 - ▶ ...Many more
-

Identifying Scenarios from use cases

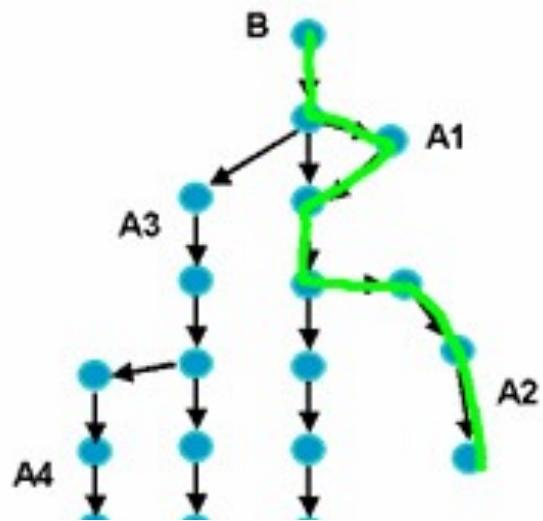
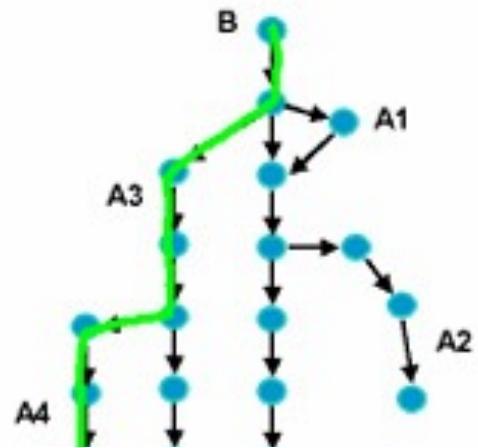
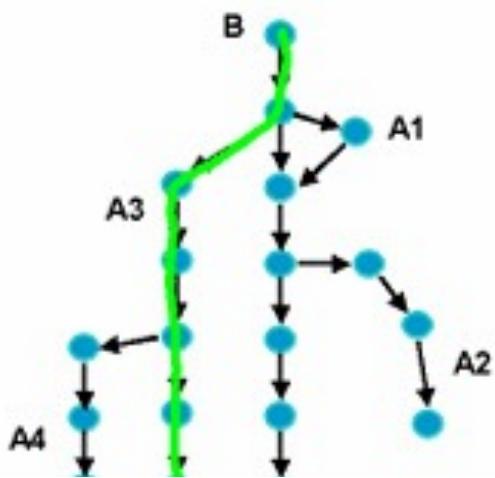
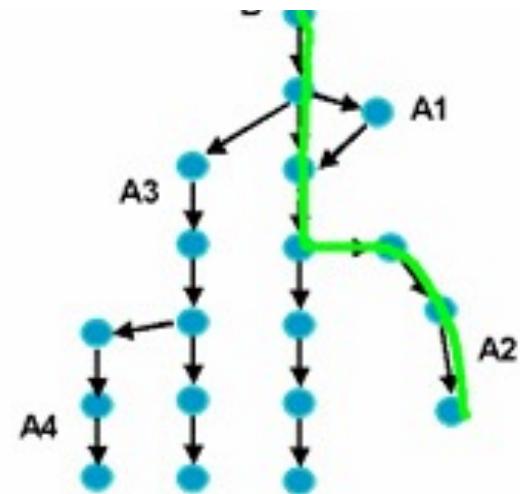
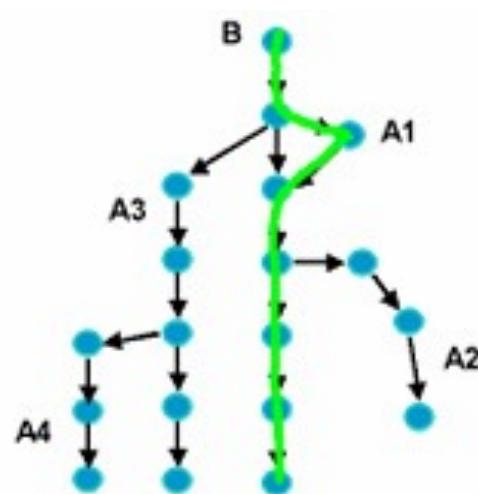
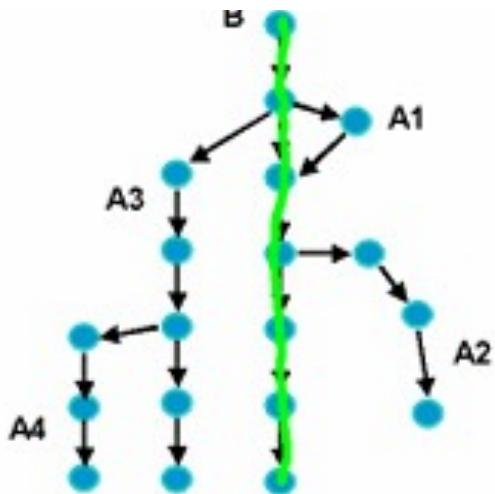
Should be feasible

Every basic +
alternate flows individually
+
combinations of alternate
flows.

- ▶ Scenario 1: Basic
- ▶ Scenario 2: A1
- ▶ Scenario 3: A2
- ▶ Scenario 4: A3
- ▶ Scenario 5: A4
- ▶ Scenario 6: A5
- ▶ Scenario 7 A6
- ▶ Scenario 8 A7
- ▶ Scenario 9 A8
- ▶ Scenario 10: A1, A2
- ▶ Scenario 11: A3, A4
- ▶ Scenario: 12 A4, A5
- ▶ Scenario 13: A3, A5
- ▶ Scenario 14 A6, A7
- ▶ Scenario 15 A7, A8



Possible Scenarios



Writing Formal Test Cases and Test Plans

- ▶ A *test case* is an explicit set of instructions designed to detect a particular class of defect in a software system.
- ▶ A test case can give rise to many tests (test case instances).
- ▶ Each test is a particular running of the test case on a particular version of the system.



Test plans

- ▶ A *test plan* is a document that contains a complete set of test cases for a system
 - ▶ Along with other information about the testing process.
 - ▶ The test plan is one of the standard forms of documentation.
 - ▶ If a project does not have a test plan:
 - ▶ Testing will inevitably be done in an ad-hoc manner.
 - ▶ Leading to poor quality software.
 - ▶ The test plan should be written long before the testing starts.
 - ▶ You can start to develop the test plan once you have developed the requirements.



Information to include in a formal test case

A. Identification and classification:

- ▶ Each test case should have a number, and may also be given a descriptive title.
- ▶ The system, subsystem or module being tested should also be clearly indicated.
- ▶ The importance of the test case should be indicated.

B. Instructions:

- ▶ Tell the tester exactly what to do.
- ▶ The tester should not normally have to refer to any documentation in order to execute the instructions.

C. Expected result:

- ▶ Tells the tester what the system should do in response to the instructions.
- ▶ The tester reports a failure if the expected result is not encountered.

D. Cleanup (when needed):

- ▶ Tells the tester how to make the system go ‘back to normal’ or shut down after the test.



Detailed Example: Test cases for Phase 2 of the SimpleChat

GGeneral Setup for Test Cases in the 2000 Series

SSystem: SimpleChat/OCSF **Phase:** 2

IInstructions:

1. Install Java, minimum release 1.2.0, on Windows 95, 98 or ME.
2. Install Java, minimum release 1.2.0, on Windows NT or 2000.
3. Install Java, minimum release 1.2.0, on a Solaris system.
4. Install the SimpleChat - Phase 2 on each of the above platforms.



Test cases for Phase 2 of the SimpleChat

TTest Case 2001

SSystem: SimpleChat **P**hase: 2

SServer startup check with default arguments

SSeverity: 1

Instructions:

1. At the console, enter: **java EchoServer**.

Expected result:

1. The server reports that it is listening for clients by displaying the following message:

Server listening for clients on port 5555

2. The server console waits for user input.

Cleanup:

1. Hit CTRL+C to kill the server.



Test cases for Phase 2 of the SimpleChat

TTest Case 2002

SSystem: SimpleChat **P**hase: 2

CClient startup check without a login

SSeverity: 1

Instructions:

1. At the console, enter: **java ClientConsole**.

Expected result:

1. The client reports it cannot connect without a login by displaying:

ERROR - No login ID specified. Connection aborted.

2. The client terminates.

Cleanup: (if client is still active)

1. Hit CTRL+C to kill the client.



Test cases for Phase 2 of the SimpleChat

Test Case 2019

System: SimpleChat **Phase:** 2

Different platform tests

Severity: 3

Instructions:

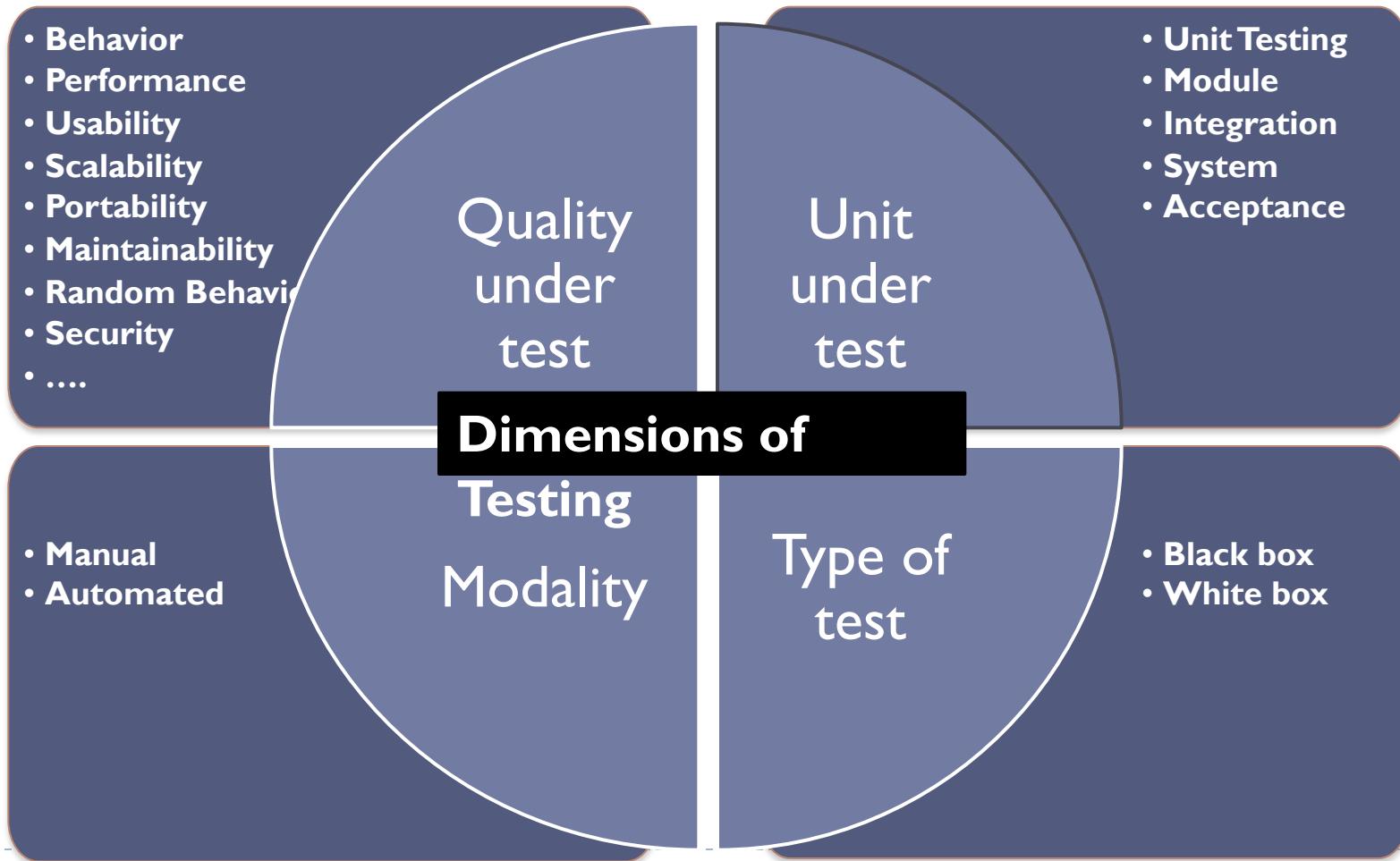
1. Repeat test cases 2001 to 2018 on Windows 95, 98, NT or 2000, and Solaris

Expected results:

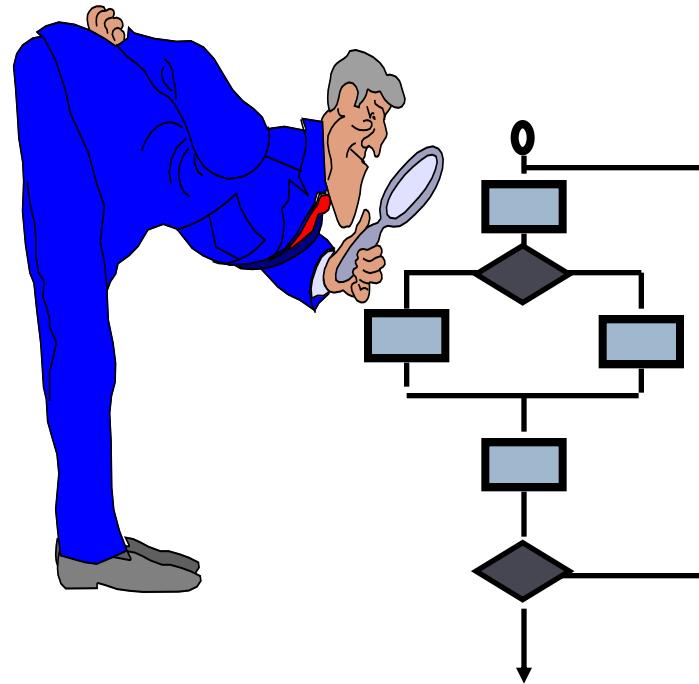
1. The same as before.



Various Dimensions of Testing



White-Box Testing



**... our goal is to ensure that all
statements and conditions have
been executed at least once ...**

White-box testing

- ▶ Also called ‘glass-box’ or ‘structural’ testing
- ▶ Testers have access to the system design
 - ▶ They can
 - ▶ Examine the design documents
 - ▶ View the code
 - ▶ Observe at run time the steps taken by algorithms and their internal data
- ▶ Individual programmers often informally employ glass-box testing to verify their own code



White Box Methods

- ▶ Can be applied at all levels of system development – unit, integration, and system
- ▶ Coverage (Control flow)
 - ▶ Statement
 - ▶ Branch
 - ▶ Condition
- ▶ Dataflow
- ▶ Mutation

Statement Coverage

- ▶ Execute each statement in the program
- ▶ Considered minimum criterion for most unit testing
- ▶ May be difficult to achieve for error cases

Example Program

```
1: if (a < 0) {  
2:     return 0 }  
3: r = 0;  
4: c = a;  
5: while (c > 0) {  
6:     r = r + b;  
7:     c = c - 1; }  
8: return r;
```

Statement tests

a = 3, b = 4

executes 1, 3, 4, 5, 6, 7,
5, 6, 7, 5, 6, 7, 5, 8

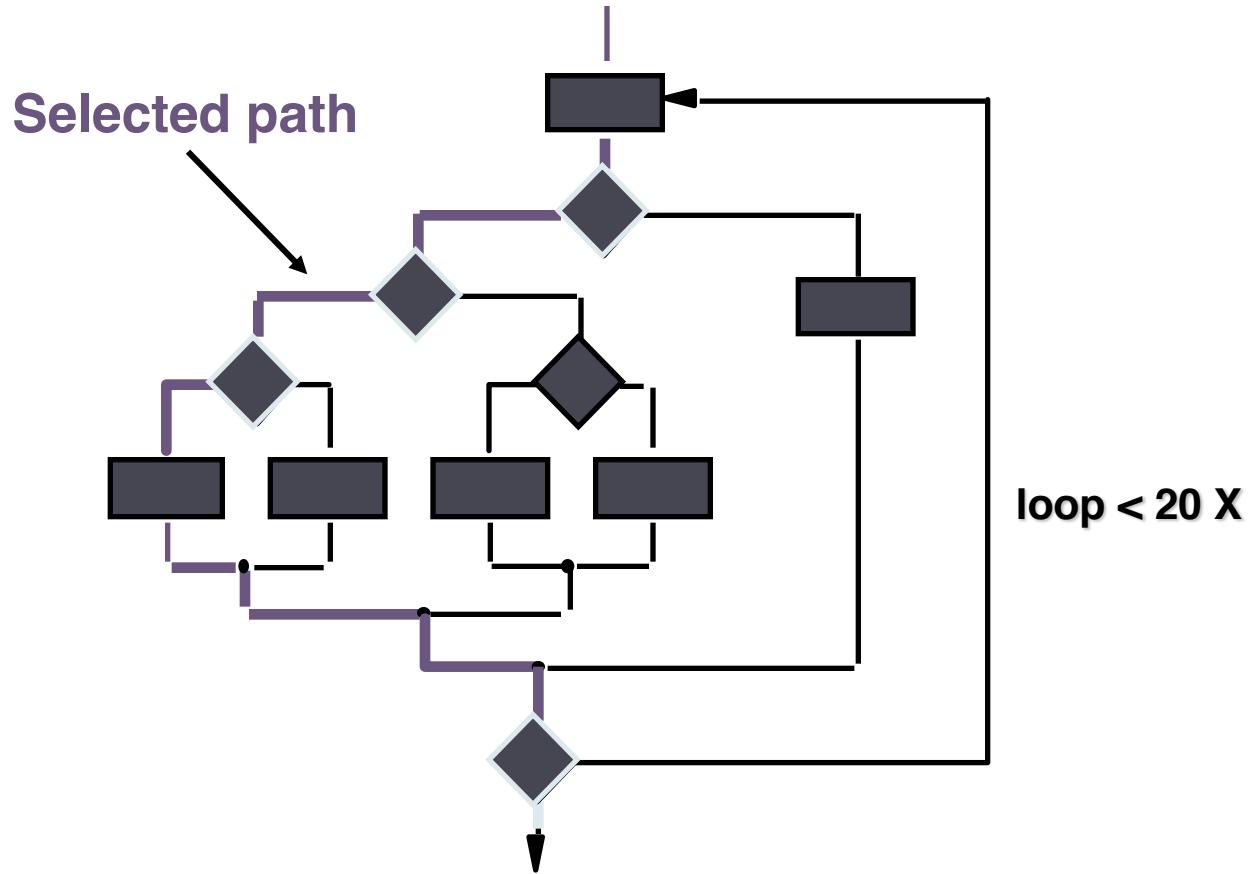
a = -3, b = 2

executes 1, 2

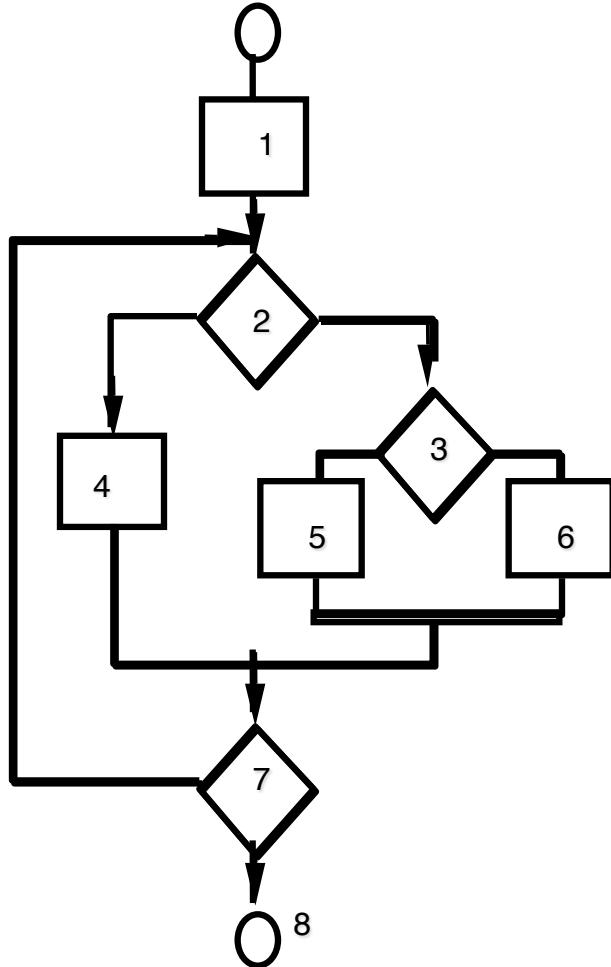
Branch Coverage

- ▶ Execute each branch of the program at least once
- ▶ Differs from statement coverage only for "if" statements without "else"s and case statements without default cases.

Structured Testing



Basis Path Testing



Next, we derive the independent paths:

**Since $V(G) = 4$,
there are four paths**

Path 1: 1,2,3,6,7,8

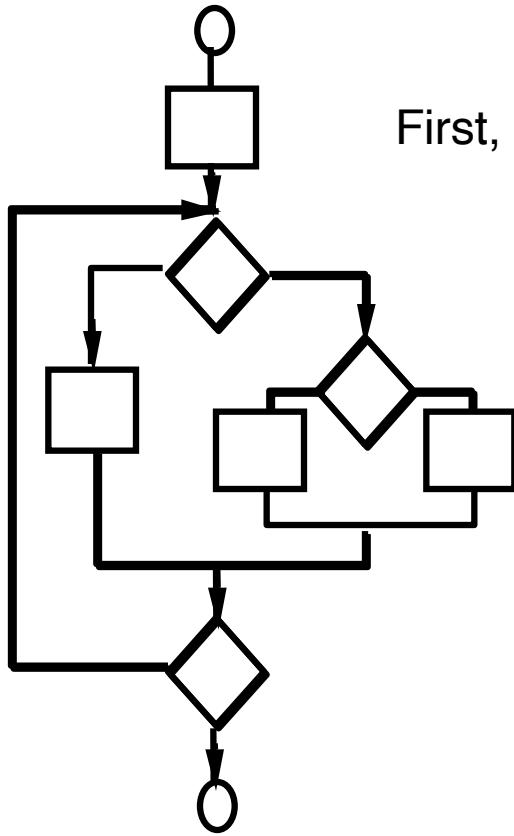
Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2..,8

Finally, we derive test cases to exercise these paths.

Basis Path Testing



First, we compute the **McCabe' s cyclomatic complexity**:

number of simple decisions + 1

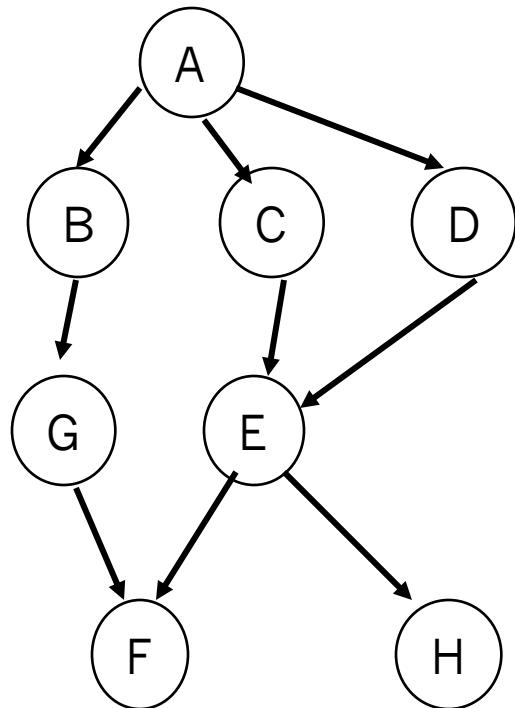
or

number of enclosed areas + 1

In this case, $V(G) = 4$

If decisions are not binary, this can't be used for computation of cyclomatic complexity

Cyclomatic complexity



- ▶ Cyclomatic complexity of a graph is

$$C = \text{edges} - \text{nodes} + 2$$

Why Data flow testing?

- ▶ Look at this simple C program

```
#include <stdio.h>

main() {
    int x;
    printf ("%d",x);
}
```

What value for x will be printed ?

Dataflow Testing

Definitions and Uses

- ▶ Defining node
 - ▶ input statement
 - ▶ lhs of assignment
- ▶ Usage node
 - ▶ output statement
 - ▶ rhs of assignment
 - ▶ control statement
- ▶ **DU testing:** execute each du-path of each variable

Example Program

1: r = 0;	Def (c) = {2, 5}	Use (c) = {3, 5}
2: c = a;	Def (r) = {1, 4}	Use (r) = {4, 6}
3: while (c > 0) {		
4: r = r + b;		
5: c = c - 1; }		
6: return r;		

Other White box testing techniques

- ▶ Loop Boundary Analysis
- ▶ Data Dependency Testing
- ▶ Path Predicate Testing
- ▶ Combinatorial Testing
- ▶ Error Guessing
- ▶ Mutation Testing
- ▶ Model Based Testing
- ▶ Static Program Analysis



Loop Boundary Analysis

Testing the boundary conditions within loops

```
def calculate_average(numbers):
    total = 0
    count = 0
    for num in numbers:
        total += num
        count += 1
    return total / count
```

Test cases:

- Test Case 1: Empty input array
- Test Case 2: Array with a single element
- Test Case 3: Array with multiple elements



Data Dependency Testing

- ▶ Focuses on identifying and testing relationships between different variables and data elements

Example: Testing a function that calculates the discounted price based on a product's original price and discount percentage.

```
def calculate_discounted_price(original_price, discount_percentage):  
    discounted_price = original_price - (original_price * discount_percentage / 100)  
return discounted_price
```

Test cases:

- ▶ Test Case 1: Positive original price and discount percentage
- ▶ Test Case 2: Zero original price with non-zero discount percentage
- ▶ Test Case 3: Negative discount percentage



Path Predicate Testing

Analyzing the predicates (Boolean expressions) within the code to create test cases that cover specific paths based on predicate outcomes

Example: Testing a function that determines the category of a product based on its price.

```
def determine_category (price):
    if price < 0:
        return "Invalid"
    elif price < 100:
        return "Low"
    elif price < 1000:
        return "Medium"
    else:
        return "High"
```

Test cases:

- Test Case 1: Price is negative
- Test Case 2: Price is between 0 and 99
- Test Case 3: Price is between 100 and 999
- Test Case 4: Price is 1000 or greater



Combinatorial Testing

- ▶ Also known as pairwise testing, involves a subset of test cases that cover all possible combinations of input parameters.

Example: Testing a function that validates user login credentials.

```
def validate_login(username, password):  
    # Validation logic  
    pass
```

Test cases:

- Test Case 1: Valid username and password
- Test Case 2: Valid username with incorrect password
- Test Case 3: Invalid username with valid password
- Test Case 4: Invalid username and password



Error Guessing

- ▶ Test cases are designed based on common mistakes or areas of concern that specific coverage criteria

Example: Testing a function that sorts a list of numbers.

```
def bubble_sort (numbers):  
    # Sorting logic
```

Test cases:

- Test Case 1: Empty list
- Test Case 2: Already sorted list
- Test Case 3: List with duplicate numbers



Mutation Testing

- ▶ Making small deliberate changes (mutations) to assess effectiveness of test suite in detecting the changes

Example: Testing a function that calculates the factorial of a number.

```
def factorial(n):  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

Mutation:

- Change ‘result *= i’ to ‘result += i’
- Test Case: ‘factorial (5)’ should fail due to incorrect result.



Model-based Testing

- ▶ Creating formal model of the system's behavior and using it to generate test cases

Example: Testing a banking application's transfer funds feature.

- ▶ Formal model includes states such as "Account Verified", "Sufficient Balance", and "Transfer Successful".
- ▶ Test cases are generated based on transitions between these states, covering scenarios like successful transfer, insufficient balance, and invalid account.



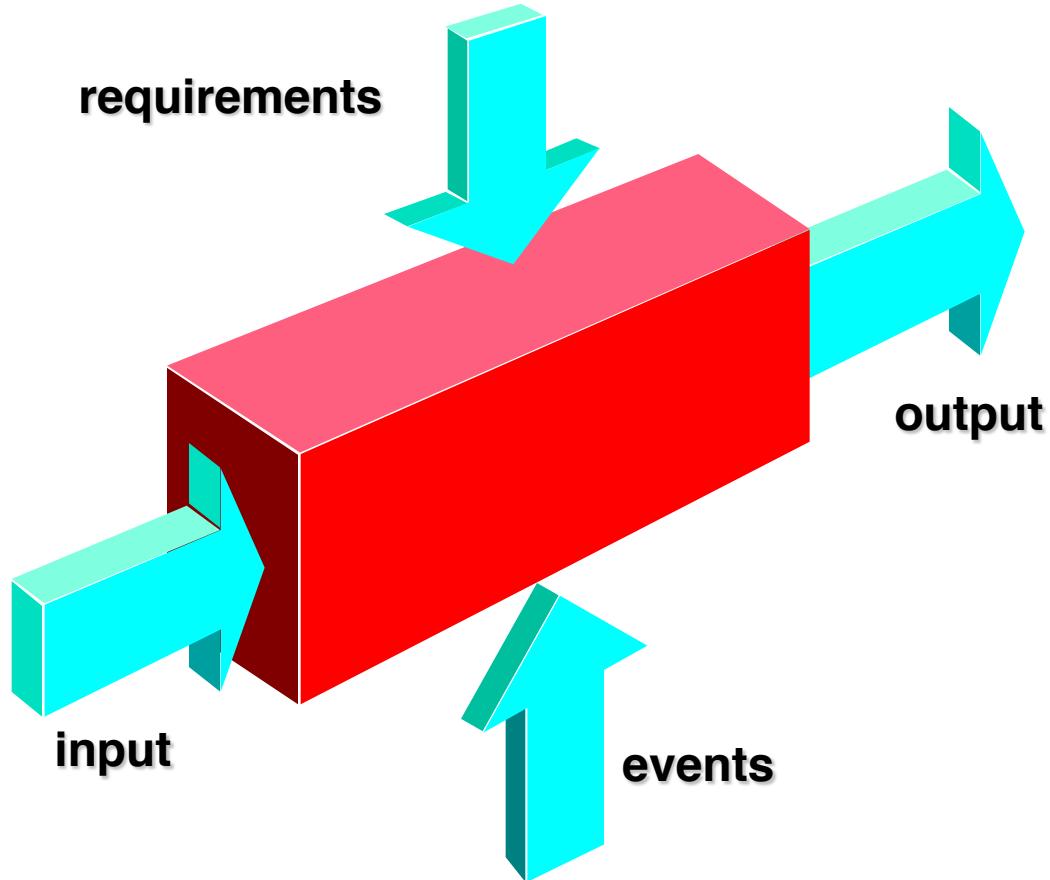
Static Program Analysis

- ▶ Analyze the source code without executing it to identify potential defects, vulnerabilities and performance issues
- ▶ Involves data flow analysis, control flow analysis, and semantic analysis

Example: Using a static analysis tool like pylint to analyze a Python codebase for potential issues such as unused variables, missing docstrings, or code complexity violations.



Black-Box Testing



Testing

- ▶ Software products are tested at four levels:
 - ▶ Unit testing
 - ▶ Integration testing
 - ▶ System testing
 - ▶ Acceptance testing



Unit testing

- ▶ During unit testing, modules are tested in isolation:
 - ▶ If all modules were to be tested together:
 - ▶ it may not be easy to determine which module has the error.
- ▶ Unit testing reduces debugging effort several folds.
 - ▶ Programmers carry out unit testing immediately after they complete the coding of a module.



Unit Testing involves...

- ▶ Writing Test Cases
- ▶ Framework Usage
- ▶ Isolation of the Unit
- ▶ Mocking/Stubbing
- ▶ Regression Testing
- ▶ Test Coverage



Unit Testing Benefits

- ▶ Facilitating change by ensuring the module works as expected.
- ▶ Simplifying integration by verifying that individual units work beforehand.
- ▶ Documentation, as they provide a written understanding of how units are supposed to work.
- ▶ Design improvement and specification clarification.



TaskManager - Example

```
import java.util.ArrayList;
import java.util.List;

public class TaskManager {
    private List<String> tasks;

    public TaskManager() {
        this.tasks = new ArrayList<>();
    }

    public void addTask(String task) {
        tasks.add(task);
    }

    public void markTaskAsCompleted(String task) {
        tasks.remove(task);
    }

    public List<String> getTasks() {
        return tasks;
    }
}
```

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.List;

public class TaskManagerTest {
    private TaskManager taskManager;

    @Before
    public void setUp() {
        taskManager = new TaskManager();
        taskManager.addTask("Task 1");
        taskManager.addTask("Task 2");
        taskManager.addTask("Task 3");
    }

    @Test
    public void testAddTask() {
        taskManager.addTask("New Task");
        List<String> tasks = taskManager.getTasks();
        assertEquals(4, tasks.size());
        assertTrue(tasks.contains("New Task"));
    }

    @Test
    public void testMarkTaskAsCompleted() {
        taskManager.markTaskAsCompleted("Task 2");
        List<String> tasks = taskManager.getTasks();
        assertEquals(2, tasks.size());
        assertFalse(tasks.contains("Task 2"));
    }

    @Test
    public void testGetTasks() {
        List<String> tasks = taskManager.getTasks();
        assertEquals(3, tasks.size());
        assertTrue(tasks.contains("Task 1"));
        assertTrue(tasks.contains("Task 2"));
        assertTrue(tasks.contains("Task 3"));
    }
}
```

Writing effective unit tests...

1. Understanding Requirements
2. Complex Logic
3. Dependencies (Mocking/Stubbing)
4. State Management
5. Test Maintainability
6. Performance Overhead
7. False Positives/Negatives
8. Test Coverage
9. Testing Edge Cases
10. Maintaining Test Independence



Complex logic - Factorial example

```
public class MathUtils {  
    public int computeFactorial(int n) {  
        if (n < 0) {  
            throw new IllegalArgumentException("Factorial is not  
defined for negative numbers");  
        } else if (n == 0) {  
            return 1;  
        } else {  
            return n * computeFactorial(n - 1);  
        }  
    }  
}
```

```
// Test factorial of 0  
assertEquals(1, mathUtils.computeFactorial(0));  
  
// Test factorial of positive numbers  
assertEquals(1, mathUtils.computeFactorial(1));  
assertEquals(6, mathUtils.computeFactorial(3));  
assertEquals(24, mathUtils.computeFactorial(4));  
assertEquals(120, mathUtils.computeFactorial(5));
```

```
// Test factorial of negative number  
try {  
    mathUtils.computeFactorial(-1);  
    fail("Expected IllegalArgumentException");  
} catch (IllegalArgumentException e) {  
    assertEquals("Factorial is not defined for negative numbers",  
e.getMessage());  
}
```



Mocking/Stubbing

```
public class PaymentProcessor {  
    private PaymentGateway paymentGateway;  
  
    public PaymentProcessor(PaymentGateway paymentGateway) {  
        this.paymentGateway = paymentGateway;  
    }  
  
    public boolean processPayment(double amount) {  
        // Call the payment gateway to process the payment  
        return paymentGateway.charge(amount);  
    }  
}
```

```
public interface PaymentGateway {  
    boolean charge(double amount);  
}
```

```
import static org.junit.Assert.*;  
import static org.mockito.Mockito.*;  
  
import org.junit.Test;  
  
public class PaymentProcessorTest {  
  
    @Test  
    public void testProcessPayment() {  
        // Create a mock PaymentGateway  
        PaymentGateway mockGateway = mock(PaymentGateway.class);  
  
        // Configure the mock to return true when charge method is called with any  
        double value  
        when(mockGateway.charge(anyDouble())).thenReturn(true);  
  
        // Create PaymentProcessor with the mock PaymentGateway  
        PaymentProcessor paymentProcessor = new  
        PaymentProcessor(mockGateway);  
  
        // Call the method under test  
        boolean result = paymentProcessor.processPayment(100.0);  
  
        // Verify that the payment was processed successfully  
        assertTrue(result);  
  
        // Verify that the charge method of the mock was called exactly once with  
        the specified amount  
        verify(mockGateway, times(1)).charge(100.0);  
    }  
}
```

False Positives

```
public class EmailValidator {  
    public boolean isValidEmail(String email) {  
        // Simplified email validation logic  
        return email != null && email.contains("@") && email.contains(".");  
    }  
}
```

```
import static org.junit.Assert.*;  
import org.junit.Test;  
  
public class EmailValidatorTest {  
  
    @Test  
    public void testIsValidEmail() {  
        EmailValidator emailValidator = new EmailValidator();  
        assertTrue(emailValidator.isValidEmail("test@@example.com")); // Incorrect  
        assertion  
    }  
}
```



Maintaining Test independence

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        assertEquals(5, calculator.add(2, 3));
    }

    @Test
    public void testSubtraction() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.subtract(5, 3));
    }

    @Test
    public void testMultiplication() {
        Calculator calculator = new Calculator();
        assertEquals(15, calculator.multiply(5, 3));
    }

    @Test
    public void testDivision() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.divide(6, 3));
    }

    @Test(expected = IllegalArgumentException.class)
    public void testDivisionByZero() {
        Calculator calculator = new Calculator();
        calculator.divide(6, 0);
    }
}
```



Integration testing

- ▶ After different modules of a system have been coded and unit tested:
 - ▶ modules are integrated in steps according to an integration plan
 - ▶ partially integrated system is tested at each integration step.

Objectives:

- Gain confidence in the integrity of overall system design
- Ensure proper interaction of components



Simple example...

- ▶ Let's consider a scenario where a software application is being developed, consisting of three separate modules: User Interface (UI), Database (DB), and Payment Gateway (PG).
- ▶ UI: Handles user interactions and displays information.
- ▶ DB: Manages data storage and retrieval.
- ▶ PG: Processes payment transactions.

Each module is first tested individually (Unit Testing) to ensure it works correctly on its own.

- ▶ UI Test: Checking if the UI displays data correctly.
- ▶ DB Test: Ensuring data is stored and retrieved correctly.
- ▶ PG Test: Verifying that transactions are processed accurately.



Simple example... integration testing

After unit testing, the modules are combined and tested together.

UI + DB Integration: Checking if the UI correctly displays data fetched from the DB.

DB + PG Integration: Ensuring the DB correctly processes data received from and sent to the PG.

UI + PG Integration: Verifying that the UI can initiate and display results of payment transactions through the PG.

Scenario for Integration Testing:

- A user performs an action on the UI, like placing an order.
- The order data is sent from the UI to the DB for storage.
- The payment details are sent from the UI to the PG for transaction processing.
- The PG sends transaction status back to the UI.
- The UI displays the order and payment status to the user.



Integration Testing Strategies

- ▶ Big-bang
- ▶ Top-down
- ▶ Bottom-up
- ▶ Critical-first
- ▶ Function-at-a-time
- ▶ As-delivered
- ▶ Sandwich



Big Bang Integration Testing

- ▶ Big bang approach is the simplest integration testing approach:
 - ▶ all the modules are simply put together and tested.
 - ▶ this technique is used only for very small systems.

Issues:

- ▶ avoids cost of scaffolding (stubs or drivers)
- ▶ does not provide any locality for finding faults



Top-down integration testing

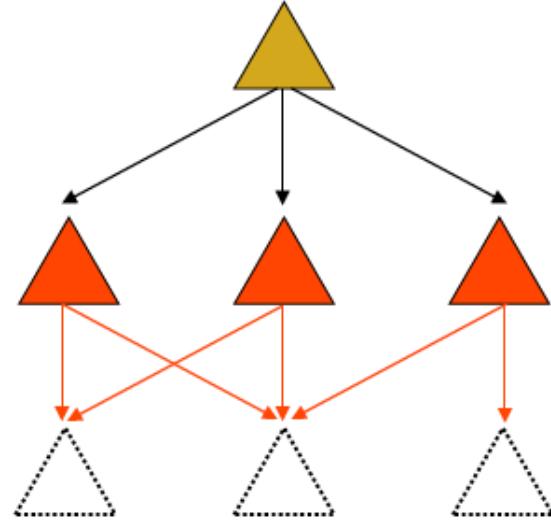
- ▶ Start with top-level modules
- ▶ Use stubs for lower-level modules
- ▶ As each level is completed, replace stubs with next level of modules

Pros:

- ▶ Always have a top-level system
- ▶ Stubs can be written from interface specifications

Cons:

- ▶ May delay performance problems until too late
- ▶ Stubs can be expensive



Bottom-up Integration Testing

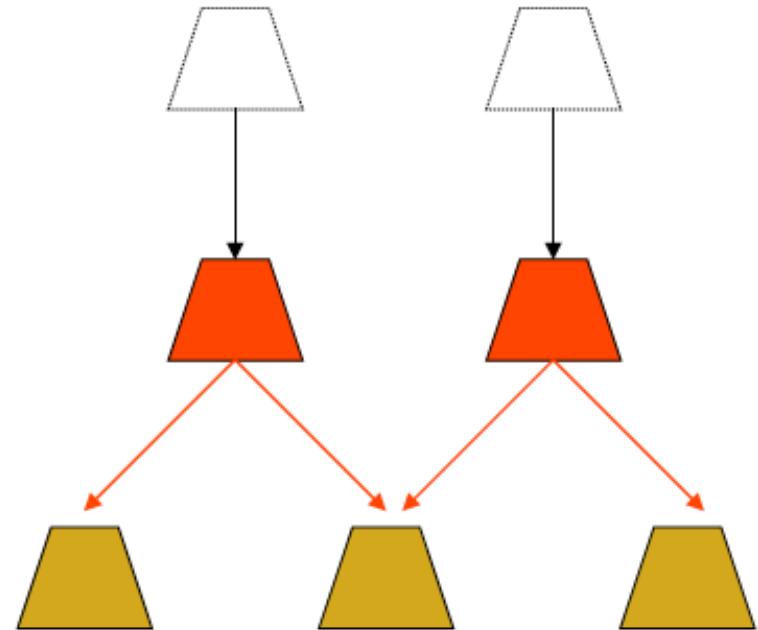
- Start with bottom-level modules
- Use drivers for upper-level modules
- As each level is completed, replace drivers with next level of modules

Pros:

- Primitive functions get most testing
- Drivers are usually cheap

Cons:

- Only have a complete system at the end



Critical-first Integration

- ▶ Integrate the most critical components first
- ▶ Add the remaining pieces later

Issues:

- Guarantees that the most important components work
- May be difficult to integrate



Function-at-a-time Integration

- ▶ Integrate all modules needed to perform a particular function
- ▶ For each function, add another set of modules

Issues

- ▶ Makes for easier test generation
- ▶ May postpone function interaction for too long
 - ▶ Dependencies may create a problem



As-Delivered Integration

- ▶ Integrate the modules as and when they become available

Issues

- ▶ Efficient – Just-in-time Integration
- ▶ Lazy – may lead to missed schedules



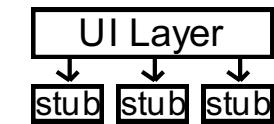
Sandwich Integration Testing

- ▶ Mixed (or sandwiched) integration testing:
 - ▶ uses both top-down and bottom-up testing approaches.
 - ▶ Most common approach

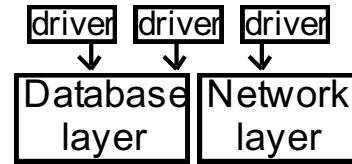


Example of different integration strategies

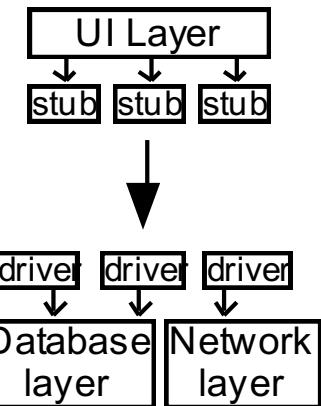
Top-down testing



Bottom-up testing



Sandwich testing



Fully
integrated
system

System Testing

Objectives

- ▶ Gain confidence in the integrity of the system as a whole
 - ▶ Ensure compliance with functional requirements
 - ▶ Ensure compliance with non-functional requirements (for example, performance requirements)



Testing Functional Requirements

1. Prepare a test plan from the functional specification of the system
2. Prepare tests for all areas of functionality
3. Review test plan and tests
4. Execute tests
5. Monitor fault rate



Testing Performance Requirements

1. Identify stress points of system
2. Create or obtain load generators
 - ▶ might use existing system
 - ▶ might buy/make special purpose tools
3. Run stress tests
4. Monitor system performance
 - ▶ usually needs instrumentation



Acceptance Testing

- ▶ Testing performed by the customer or end-user himself:
 - ▶ to determine whether the system should be accepted or rejected.

Other paths to acceptance:

- ▶ Beta testing
 - ▶ Distribute system to volunteers
 - ▶ Collect change requests, fix, redistribute
 - ▶ Collect statistics on beta use
- ▶ Shadowing
 - ▶ Collect or redistribute real-time use of existing system
 - ▶ Compare results
 - ▶ Collect statistics



The test-fix-test cycle

- ▶ When a failure occurs during testing:
 - ▶ Each failure report is entered into a failure tracking system.
 - ▶ It is then screened and assigned a priority.
 - ▶ Low-priority failures might be put on a *known bugs list* that is included with the software's *release notes*.
 - ▶ Some failure reports might be merged if they appear to result from the same defects.
 - ▶ Somebody is assigned to investigate a failure.
 - ▶ That person tracks down the defect and fixes it.
 - ▶ Finally a new version of the system is created, ready to be tested again.



Deciding when to stop testing

- ▶ All of the level 1 (“critical”) test cases must have been successfully executed.
- ▶ Certain pre-defined percentages of level 2 and level 3 test cases must have been executed successfully.
- ▶ The targets must have been achieved and are maintained for at least two cycles of ‘builds’.
 - ▶ A *build* involves compiling and integrating all the components.
 - ▶ Failure rates can fluctuate from build to build as:
 - Different sets of regression tests are run.
 - New defects are introduced.



The roles of people involved in testing

- ▶ The first pass of unit and integration testing is called *developer testing*.
 - ▶ Preliminary testing performed by the software developers who do the design.
- ▶ *Independent testing* may be performed by separate group.
 - ▶ They do not have a vested interest in seeing as many test cases pass as possible.
 - ▶ They develop specific expertise in how to do good testing, and how to use testing tools.



Test planning

- ▶ Decide on overall test strategy
 - ▶ What type of integration
 - ▶ Whether to automate system tests
 - ▶ Whether there is an independent test team
- ▶ Decide on the coverage strategy for system tests
 - ▶ Compute the number of test cases needed
- ▶ Identify the test cases and implement them
 - ▶ The set of test cases constitutes a “test suite”
 - ▶ May categorize into critical, important, optional tests (level 1, 2, 3)
- ▶ Identify a subset of the tests as regression tests



Testing performed by users and clients

- ▶ *Alpha testing*
 - ▶ Performed by the user or client, but under the supervision of the software development team.
- ▶ *Beta testing*
 - ▶ Performed by the user or client in a normal work environment.
 - ▶ Recruited from the potential user population.
 - ▶ An *open beta release* is the release of low-quality software to the general population.
- ▶ *Acceptance testing*
 - ▶ Performed by users and customers.
 - ▶ However, the customers do it on their own initiative.

