Software Testing

Introduction

- What is it?
- Why is it important?
- What is the work product?

A Test case

TEST CASE ID	TEST SCENARIO	TEST CASE	PRE-CONDITION	TEST STEPS	TEST DATA	EXPECTED RESULT	POST CONDITION	ACTUAL RESULT	STATUS (PASS/ FAIL)
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Valid User Name>	Successful login	Gmail inbox is shown		
				2. Enter Password	<Valid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Valid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Invalid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Invalid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Valid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Invalid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Invalid Password>				
				3. Click "Login" button					

Test Characteristics

- *A good test has a high probability of finding an error.*
- *A good test is not redundant*
- *A good test should be "best of breed"*
- *A good test should be neither too simple nor too complex*

"It does not guarantee absence of errors"

Testing Principles

- All tests should be *traceable to customer/user requirements*
- Tests should be *planned long before* testing begins
- The *Pareto principle* applies to software testing
- Testing *should begin "in the small" and progress toward testing "in the large"*
- *Exhaustive testing* is not possible
- To be most effective, testing should be conducted by an *independent third party*

What criteria to use?

- Should the testing be done based on externally observable behavior ?

Or

- Should the code be seen to design testcases?

External Vs Internal View

-Black box testing

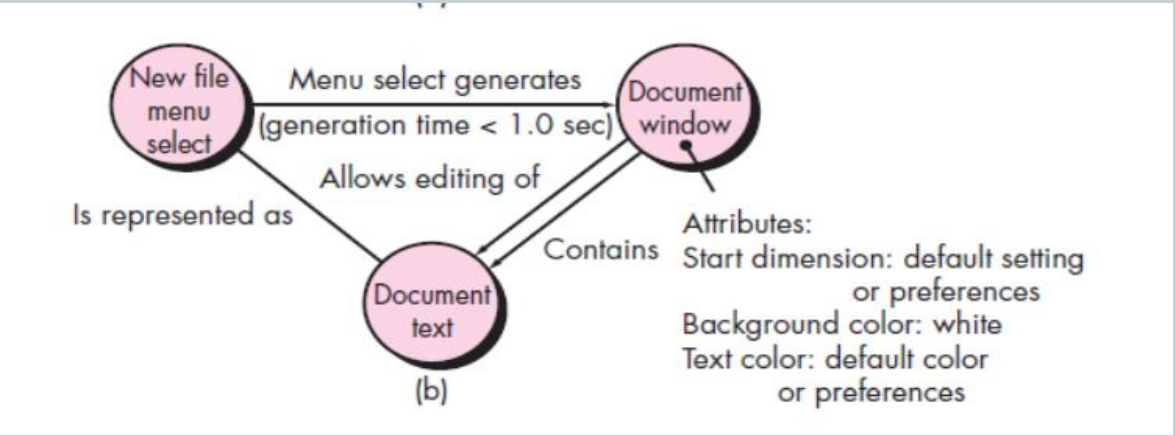
- White box testing

Black box testing

- Also called as *functional testing* or *behavioral testing*,
- Tests in the external point of view
- Specifications are used to generate testcases
- Tests for absence of features
- No programming knowledge is required

Black box testing techniques

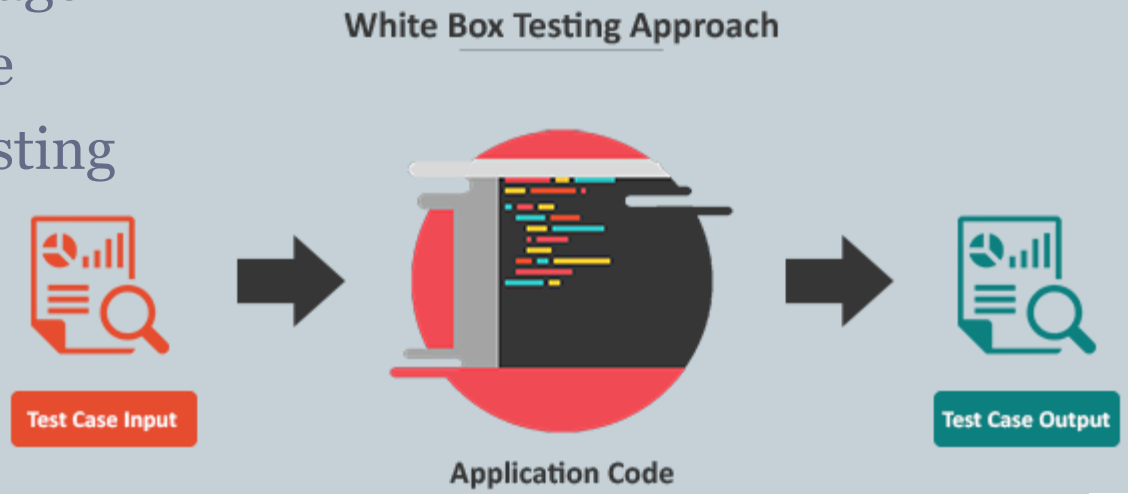
- Graph-Based Testing Methods



- Equivalence Partitioning
- Boundary Value Analysis

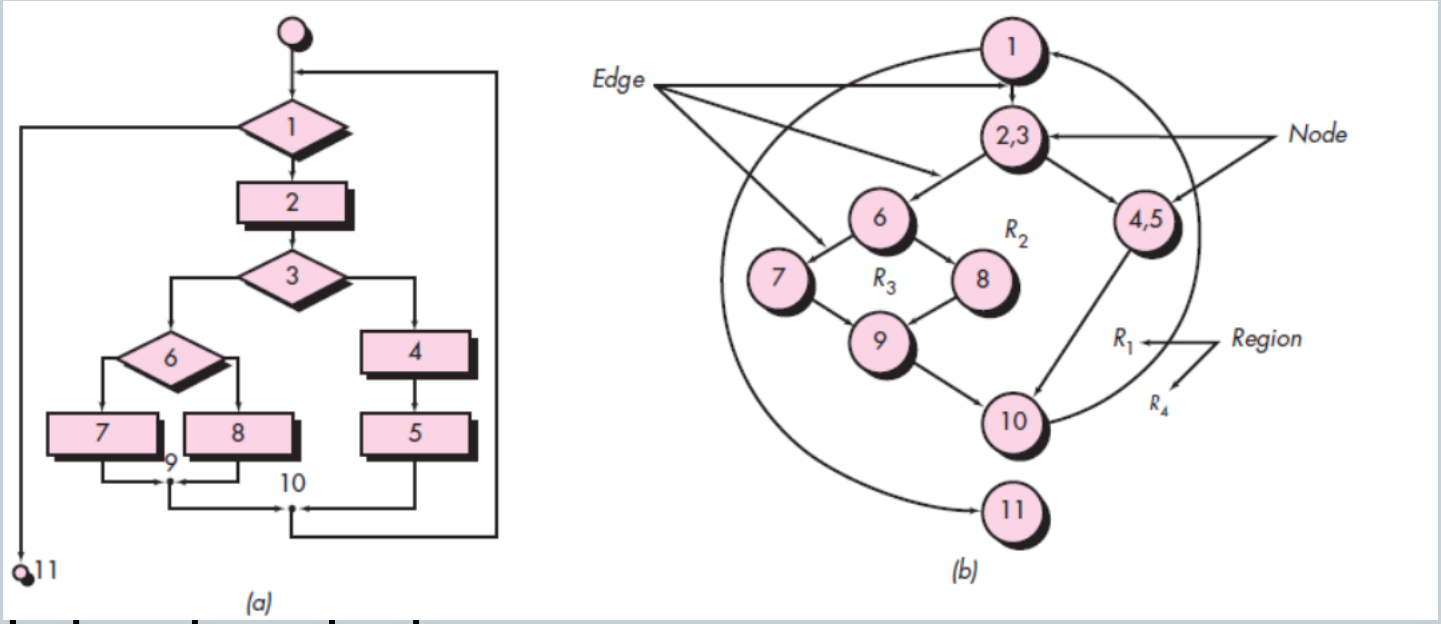
White box testing

- Also called as *glass box testing* or *structural testing*
- Tests the internal point of view or implementation
- Cannot detect absence of features
- Coverage measures are used
  - Statement coverage
  - Branch Coverage
  - Path oriented testing



White box testing techniques

- Basis Path Testing



- Control structure testing
  - Condition testing
  - Loop testing

Testing Strategies for Software Process

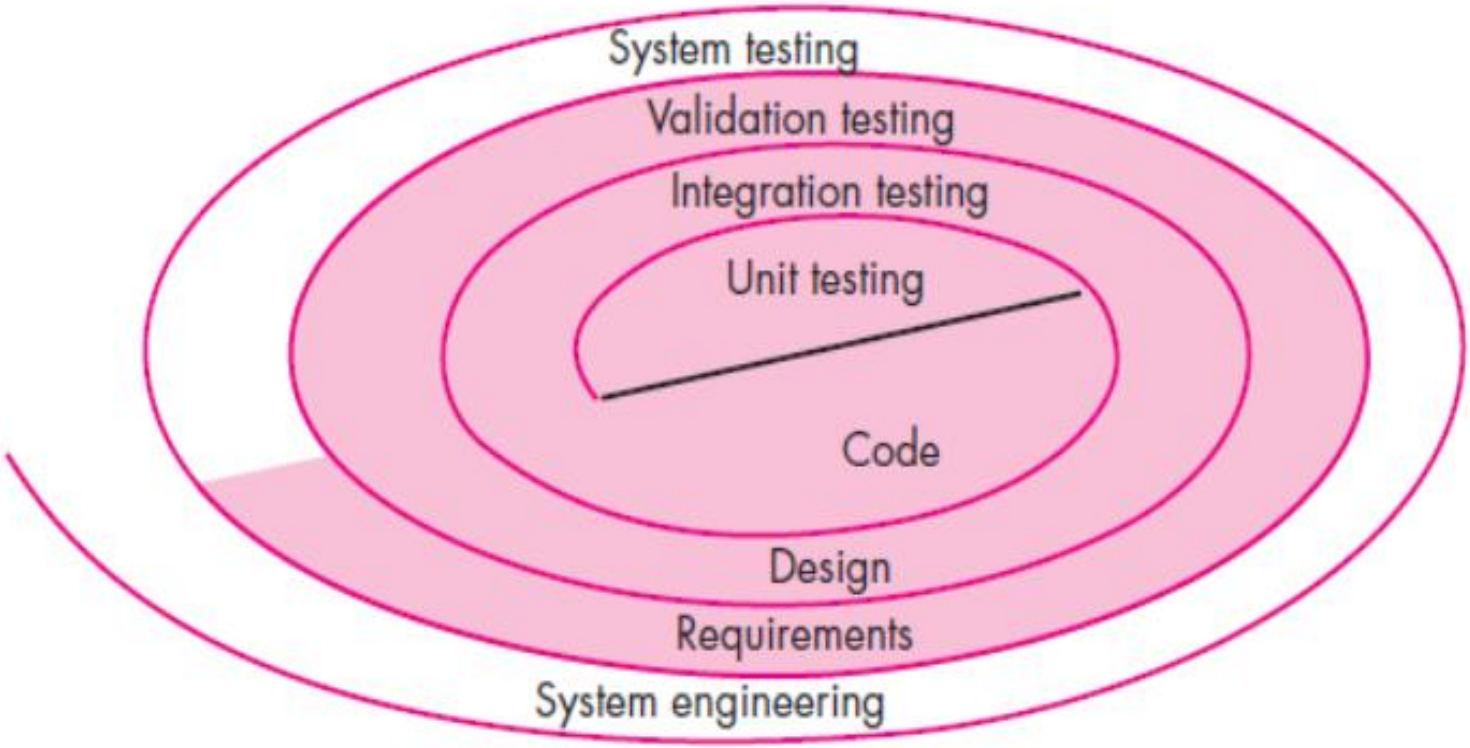
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.

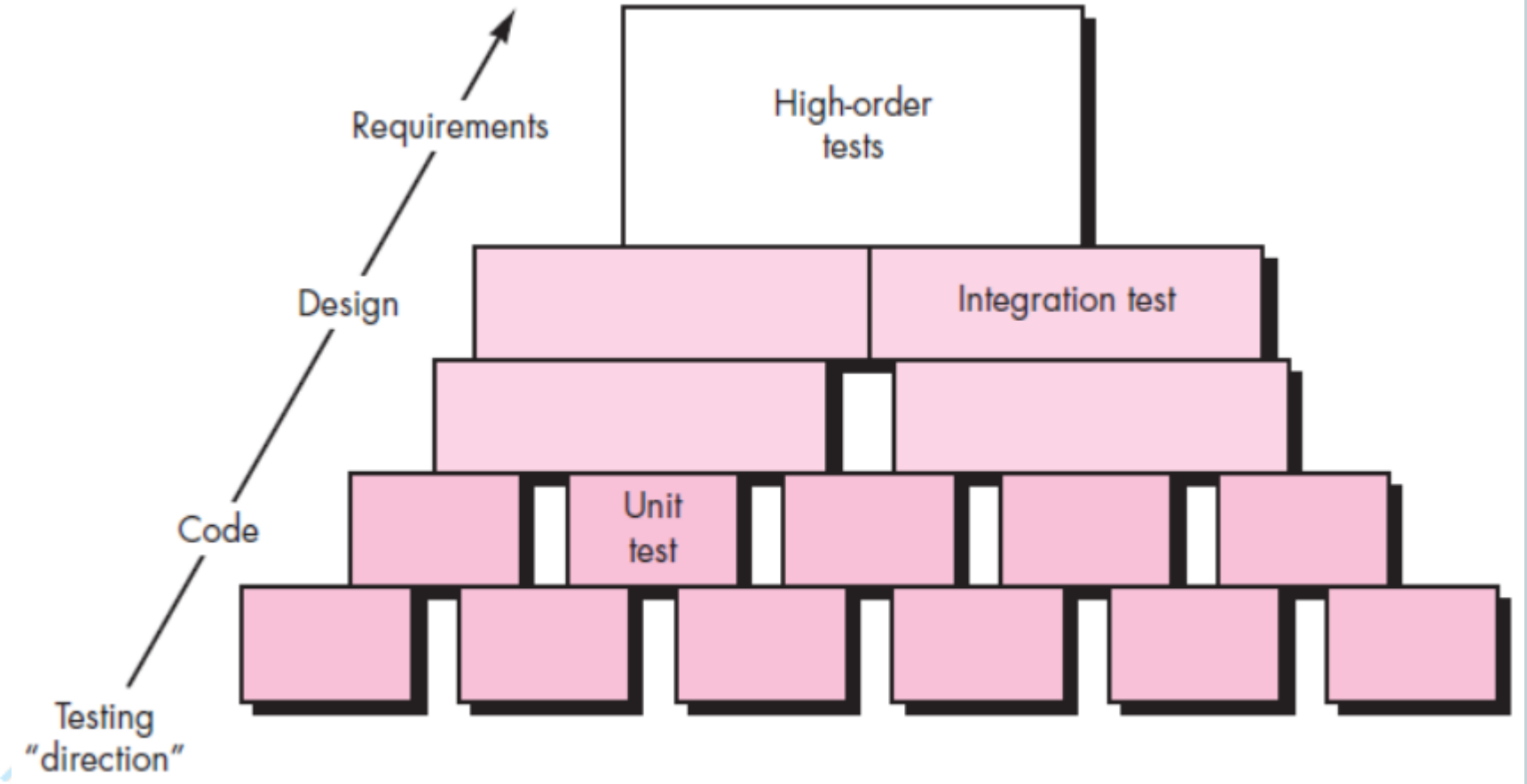
Verification and Validation (V&V)

Verification: “Are we building the product right?”

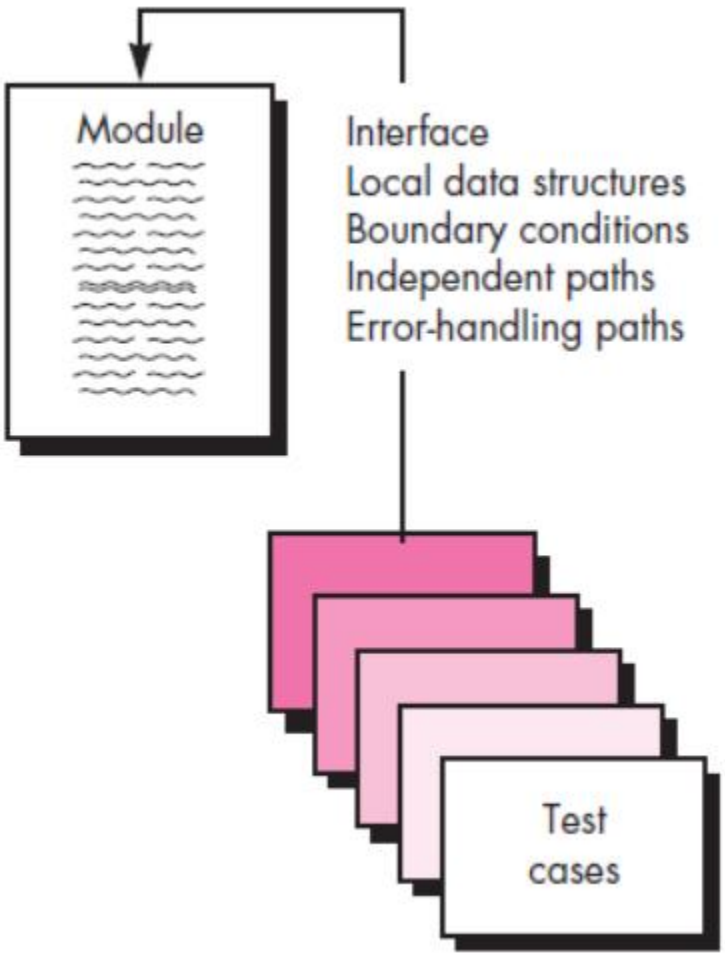
Validation: “Are we building the right product?”

Software Testing Strategy—The Big Picture

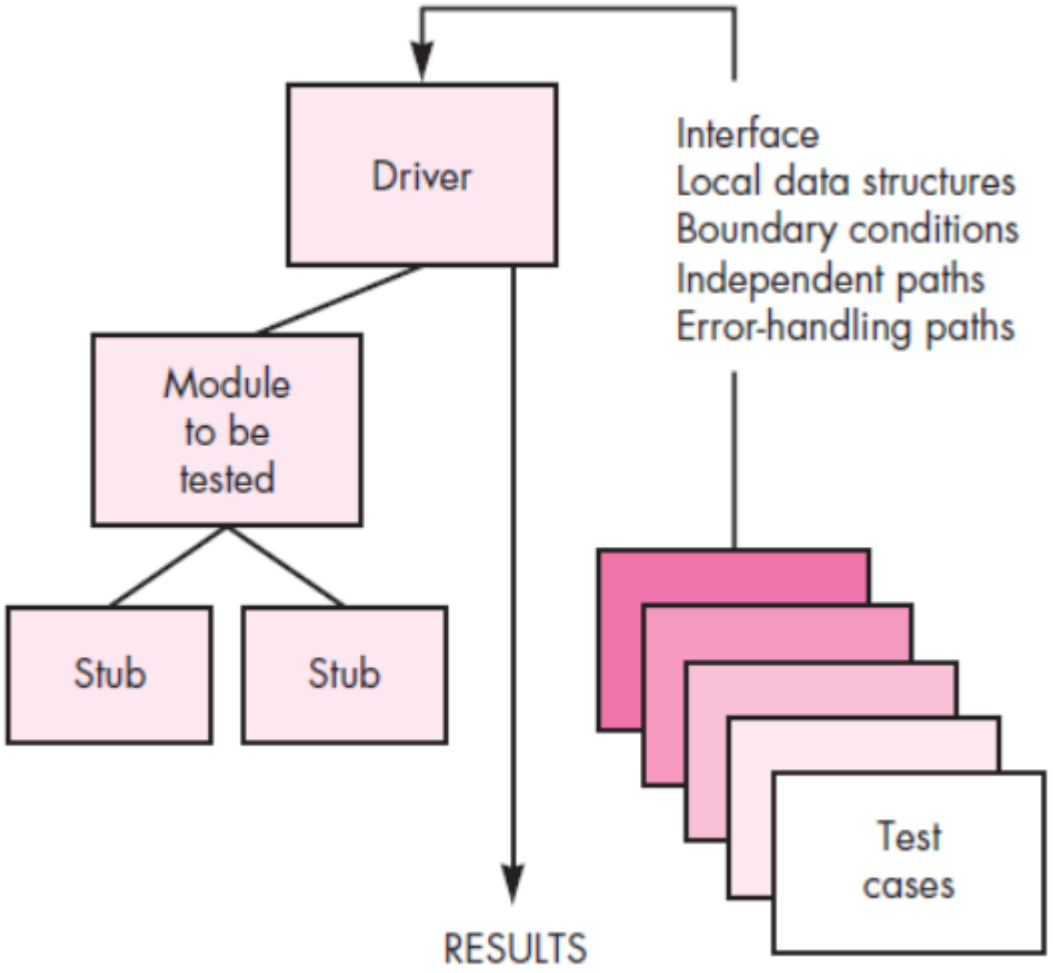



Requirements
Design
Code
Testing
"direction"
High-order
tests
Integration test
Unit
test

Unit Testing



Unit Test Procedure



Integration Testing

Once all modules have been unit tested:

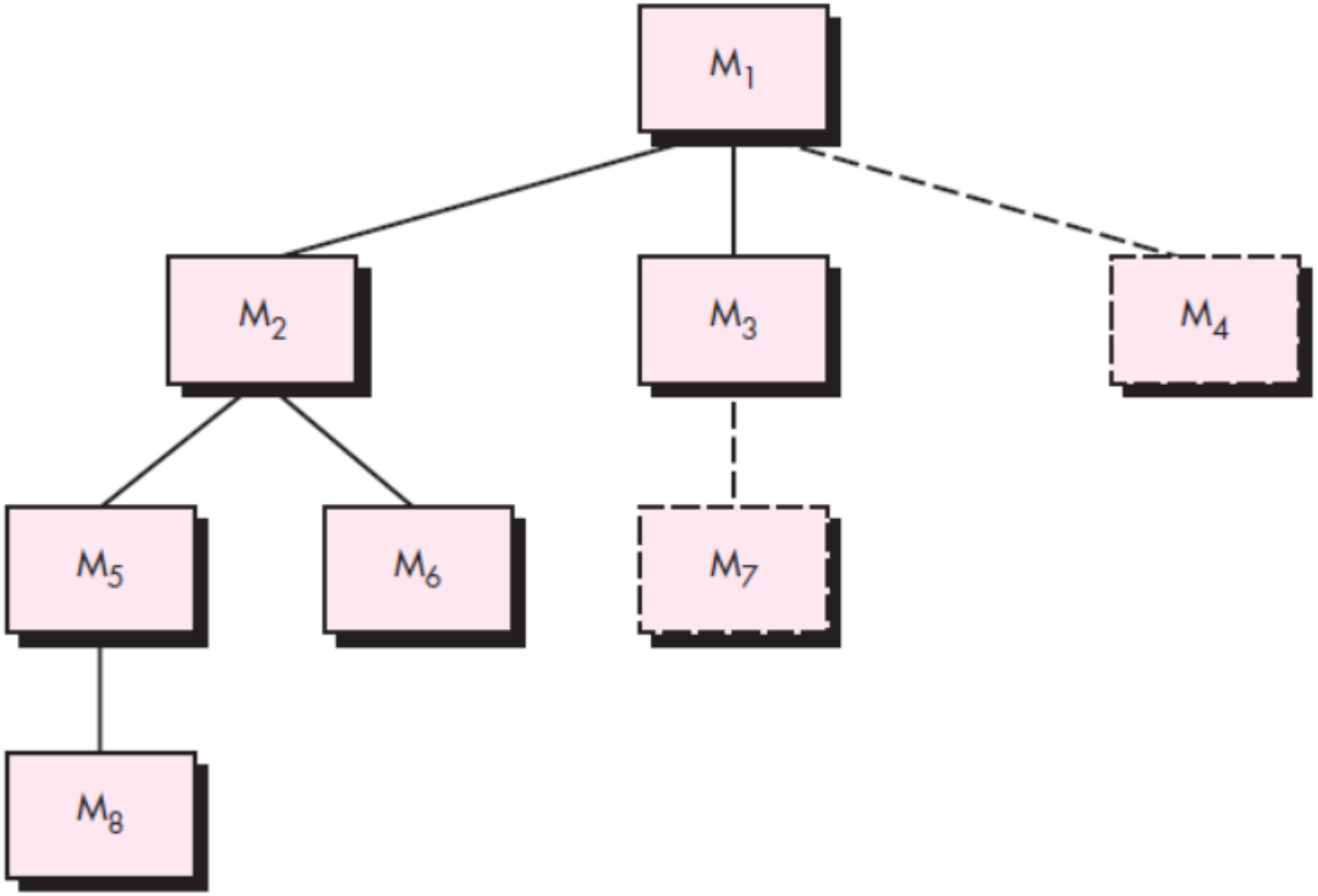
"If they all work individually, why do you doubt that they'll work when we put them together?"

The problem, of course, is

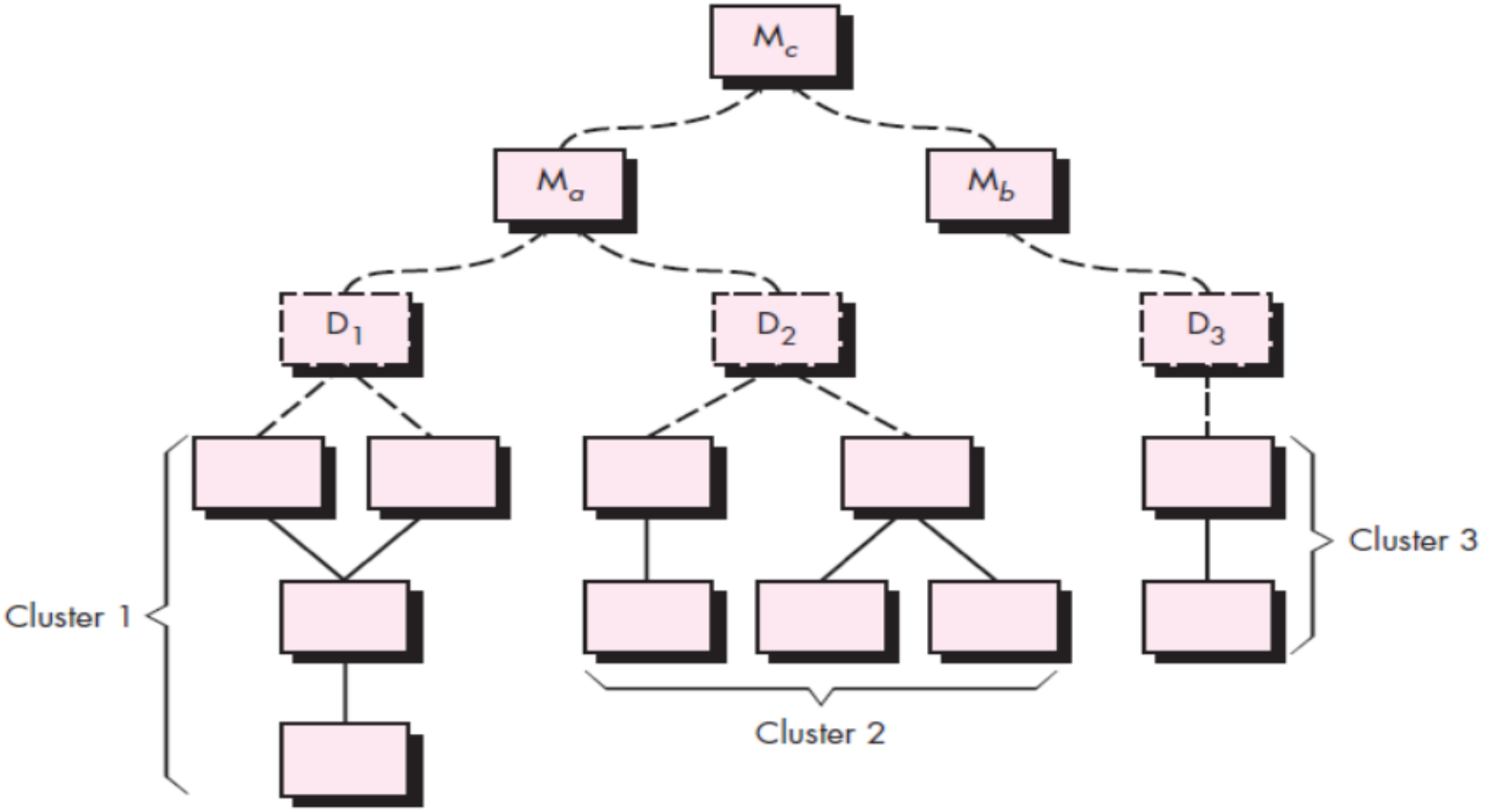
"putting them together"—interfacing.

- Incremental Approach is desirable
  - Top-Down Integration
  - Bottom-Up Integration

Top-down Integration



Bottom-up integration



Regression testing

- *Re-execution of some subset of tests that have already been* conducted to ensure that changes have not propagated unintended side effects.
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors.

Validation Testing

- Testing focuses on user-visible actions and user-recognizable output from the system.
- Validation **succeeds when software functions in a manner that can be reasonably expected by the customer**
- If Software Requirements Specification has been developed, it forms the basis for a validation-testing approach

Acceptance Test

- Alpha Testing
  - Conducted at the developer's site by a representative group of end users in a controlled environment.
  - The software is used in a natural setting with the developer and records errors and usage problems.
- Beta Testing
  - Conducted at one or more end-user sites
  - The developer generally is not present.
  - It is a "live" application of the software in an environment that cannot be controlled by the developer.
  - The customer records all problems.

System Testing

- Recovery Testing
  - Systems must recover from faults and resume processing with little or no downtime.
  - It is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security Testing
  - Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress Testing
  - Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - It test the run-time performance of software within the context of an integrated system.

Debugging

- Debugging is not testing but often occurs as a consequence of testing
- When a test case uncovers an error, debugging is the process that results in the removal of the error
- Attempts to match symptom with cause, thereby leading to error correction
- The activity must track down the cause of an error
- Debugging is difficult

Code Inspection

- An inspection is an activity in which one or more people systematically examine source code or documentation, looking for defects.
- Both testing and inspection rely on different aspects of human intelligence
- Inspecting allows you to get rid of many defects quickly.

THANK YOU

Introduction to Metrics

Introduction

- **Measure:**

  It provides a quantitative indication of the extent, dimension, size and the capacity of a product.

- **Measurement:**

  It is defined as the act of evaluating a measure.

- **Metric:**

  It is a quantitative measure of the degree to which a system or its component possesses a given attribute.

A Quote on Measurement

- "When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science."

LORD WILLIAM KELVIN (1824 – 1907)

What are Software Metrics?

A **software metric** is a measure of **software** characteristics which are measurable or countable.

Software metrics are valuable for many reasons, including measuring **software** performance, planning work items, measuring productivity, and many other uses.

What are Metrics?

- Software metrics are quantitative measures
- They are a management tool
- They offer insight into the effectiveness of the software process and the projects.
- Basic quality and productivity data are collected
- These data are analyzed, compared against past averages, and assessed
- The goal is to determine whether quality and productivity improvements have occurred
- The data can also be used to pinpoint problem areas
- Remedies can then be developed and the software process can be improved

Uses of Measurement

- Can be applied to the software process with the intent of improving it on a continuous basis
- Can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control
- Can be used to help assess the quality of software work products and to assist in tactical decision making as a project proceeds

What to be measured?

- **Characteristics of the software product:**
  - Software size and complexity
  - Software reliability and quality
  - Functionalities
  - Performance
- **Characteristics of the software project:**
  - Number of software developer
  - Staffing pattern over the life cycle of software
  - Cost and schedule
  - Productivity
- **Characteristics of software process:**
  - Methods, tools, and techniques used for software development
  - Efficiency of detection of fault

Reasons to Measure

- To characterize in order to
  - Gain an understanding of processes, products, resources, and environments
  - Establish baselines for comparisons with future assessments
- To evaluate in order to
  - Determine status with respect to plans
- To improve in order to
  - Identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance

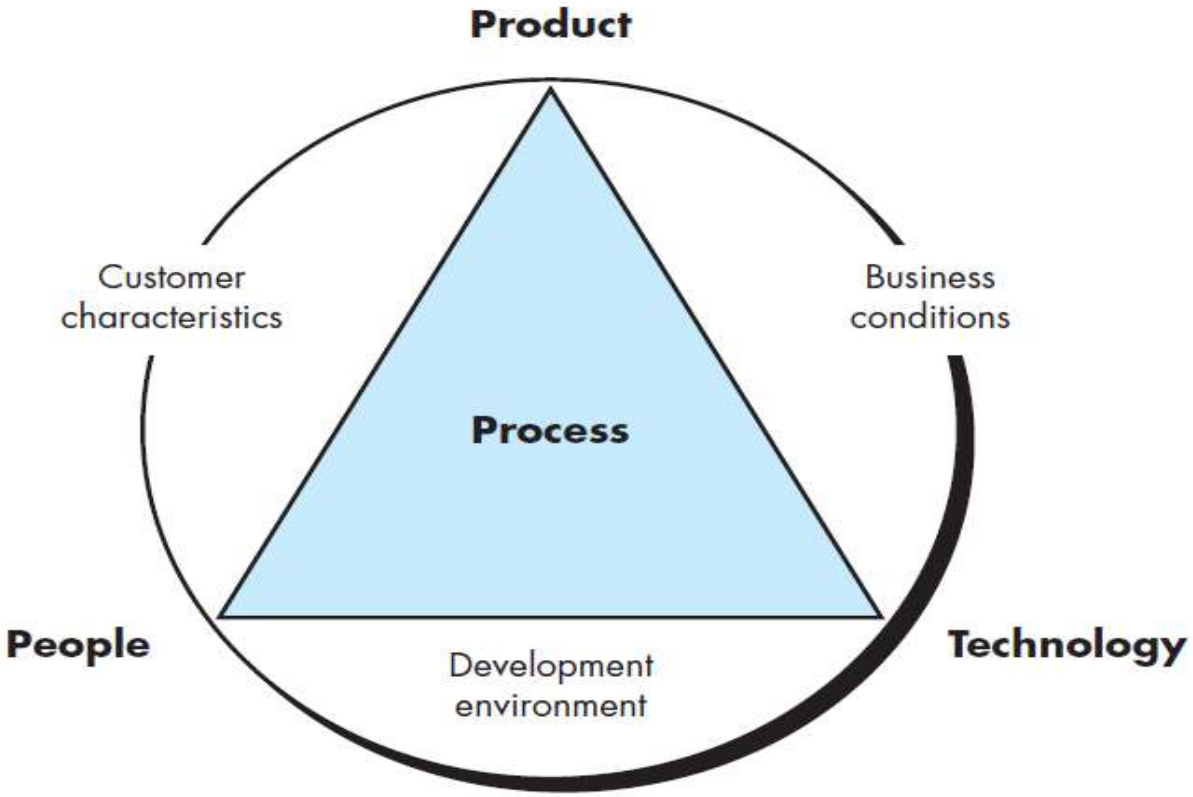
Role of Metrics in the Process Domain

- Process metrics are collected across all projects and over long periods of time
- They are used for making strategic decisions
- The intent is to provide a set of process indicators that lead to long-term software process improvement
- The only way to know how/where to improve any process is to
  - Measure specific attributes of the process
  - Develop a set of meaningful metrics based on these attributes
  - Use the metrics to provide indicators that will lead to a strategy for improvement

Metrics in the Process Domain.....

- We measure the effectiveness of a process by deriving a set of metrics based on outcomes of the process such as
  - Errors uncovered before release of the software
  - Defects delivered to and reported by the end users
  - Work products delivered
  - Human effort expended
  - Calendar time expended
  - Conformance to the schedule
  - Time and effort to complete each generic activity

Determinants for software quality an organizational effectiveness.



Etiquette of Process Metrics

- Use common sense and organizational sensitivity when interpreting metrics data
- Provide regular feedback to the individuals and teams who collect measures and metrics
- Don't use metrics to evaluate individuals
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them
- Never use metrics to threaten individuals or teams
- Metrics data that indicate a problem should not be considered "negative"
  - Such data are merely an indicator for process improvement
- Don't obsess on a single metric to the exclusion of other important metrics

Role of Metrics in the Project Domain

- Project metrics enable a software project manager to
  - Assess the status of an ongoing project
  - Track potential risks
  - Uncover problem areas before their status becomes critical
  - Adjust work flow or tasks
  - Evaluate the project team's ability to control quality of software work products
- Many of the same metrics are used in both the process and project domain
- Project metrics are used for making tactical decisions
  - They are used to adapt project workflow and technical activities

Use of Project Metrics

- The first application of project metrics occurs during estimation
  - Metrics from past projects are used as a basis for estimating time and effort
- As a project proceeds, the amount of time and effort expended are compared to original estimates
- As technical work commences, other project metrics become important
  - Production rates are measured (represented in terms of models created, review hours, function points, and delivered source lines of code)
  - Error uncovered during each generic framework activity (i.e, communication, planning, modeling, construction, deployment) are measured

Use of Project Metrics……

- Project metrics are used to
  - Minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
  - Assess product quality on an ongoing basis and, when necessary, to modify the technical approach to improve quality
- In summary
  - As quality improves, defects are minimized
  - As defects go down, the amount of rework required during the project is also reduced
  - As rework goes down, the overall project cost is reduced

Categories of Software Measurement

- Two categories of software measurement
  - Direct measures of the
    - Software process (cost, effort, etc.)
    - Software product (lines of code produced, execution speed, defects reported over time, etc.)
  - Indirect measures of the
    - Software product (functionality, quality, complexity, efficiency, reliability, maintainability, etc.)

Size-oriented Metrics

- Derived by normalizing quality and/or productivity measures by considering the size of the software produced
- Thousand lines of code (KLOC) are often chosen as the normalization value
- Based on the LOC/KLOC count of software, many other metrics can be computed**:**
  - Errors/KLOC.
  - $/ KLOC.
  - Defects/KLOC.
  - Pages of documentation/KLOC.
- You can also compute other things like,
  - Errors/PM.
  - Productivity = KLOC/PM (effort is measured in person-months).
  - $/ Page of documentation.

Size-oriented Metrics

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

Size-oriented Metrics.....

- Size-oriented metrics are not universally accepted as the best way to measure the software process
- Opponents argue that KLOC measurements
  - Are dependent on the programming language
  - Penalize well-designed but short programs
  - Require a level of detail that may be difficult to achieve

Function-Based Metrics

- In 1977, A. J. Albrecht of IBM developed a method of software metrics based on the functionality of the software delivered by an application as a normalization value.
- He called it the **Function Points (FPs).**
- They are derived using an empirical relationship based on direct measures of software's information domain and assessments of software complexity.
- FPs try to quantify the functionality of the system, i.e., what the system performs.
- This is taken as the method of measurement as FPs cannot be measured directly.
- **FP is not a single characteristic but is a combination of several software features/characteristics.**

Function-Based Metrics

- The effort required to develop the project depends on what the software does.
- FP is programming language independent.
- FP method is used for data processing systems, business systems like information systems.

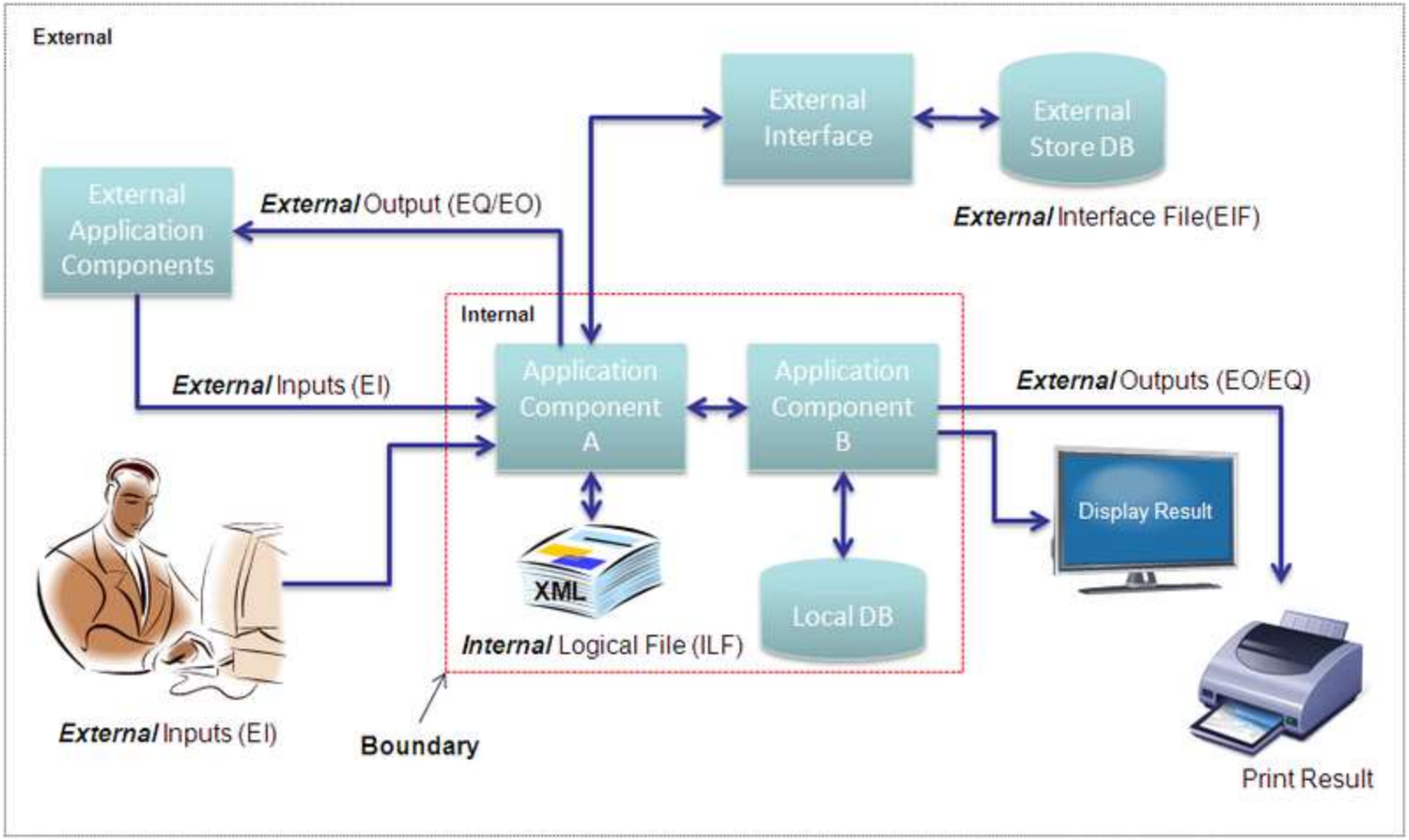
Function Point Computation

- FPs of an application is found out by counting the number and types of functions used in the applications.
- Various functions used in an application can be put under five types as shown in Table:

Measurement Parameter	*Examples*
1. Number of external inputs (EI)	Input screen and tables.
2. Number of external outputs (EO)	Output screens and reports.
3. Number of external inquiries (EQ)	Prompts and interrupts.
4. Number of internal files (ILF)	Databases and directories.
5. Number of external interfaces (EIF)	Shared databases and shared routines.

- The 5 parameters mentioned are also known as information domain characteristics.
- All these parameters are then individually assessed for complexity.

FPA components



Function Point Computation.....

- The calculation begins with the counting of the five function types of a project or application:
- These 5 function types are then ranked according to their complexity: Low, Average or High, using a set of prescriptive standards.
- These counts are then multiplied with the corresponding weights(complexity) values and the values are added up to determine the **UFP** (Unadjusted Function Point) or **count total** of the subsystem.

Function Point Computation.....

- Organizations that use FP methods, develop criteria for determining whether a particular entry is Low, Average or High.
- Nonetheless, the determination of complexity is somewhat subjective.

Table 2.3 Computing FPs

Measurement Parameter	*Count*		*Weighing factor*			
			Simple	*Average*	*Complex*	
1. Number of external inputs (EI)	—	*	3	4	6 =	—
2. Number of external outputs (EO)	—	*	4	5	7 =	—
3. Number of external inquiries (EQ)	—	*	3	4	6 =	—
4. Number of internal files (ILF)	—	*	7	10	15 =	—
5. Number of external interfaces (EIF)	—	*	5	7	10 =	—
Count-total →						—

Function Point Computation.....

complexity adjustment value/ factor

- The last step involves assessing the environment and processing complexity of the project or application as a whole.
- In this step, the impact of 14 general system characteristics is rated on a scale from 0 to 5 in terms of their likely effect on the project or application.

complexity adjustment value/ factor

- complexity adjustment value/ factor are calculated based on responses to the following questions

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated online?

Computing Value Adjustment Factor....

9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

complexity adjustment value/ factor

- Each of these questions is answered using an ordinal scale that ranges from 0 (not important or applicable) to 5 (absolutely essential).
    - 0 – No influence
    - 1 – Incidental
    - 2 – Moderate
    - 3 – Average
    - 4 – Significant
    - 5 - Essential
- The constant values in FP Equation and the weighting factors that are applied to information domain counts are determined empirically.

Function Point Computation.....

- The Function Point (FP) is thus calculated with the following formula

$$\mathbf{FP = Count\text{-}total * [0.65 + 0.01 * \sum(Fi)]}$$

$$\mathbf{= Count\text{-}total * CAF}$$

where Count-total is obtained from the table 2.3.

$$CAF = [0.65 + 0.01 * \sum(Fi)]$$

- $\sum(Fi)$ is the sum of all 14 questionnaires and show the **complexity adjustment value/ factor-CAF** (where i ranges from 1 to 14).

Function Point Computation.....

- $\sum$(Fi) ranges from 0 to 70,

  i.e., 0 <= $\sum$(Fi) <=70 and CAF ranges from 0.65 to 1.35

- because

  (a) When $\sum$(Fi) = 0 then CAF = 0.65

  (b) When $\sum$(Fi) = 70 then CAF = 0.65 + (0.01 * 70) = 0.65 + 0.7 = 1.35

Function Point Computation.....

- Based on the FP measure of software many other metrics can be computed:

  (a) Errors/FP

  (b) $/FP.

  (c) Defects/FP

  (d) Pages of documentation/FP

  (e) Errors/PM.

  (f) Productivity = FP/PM (effort is measured in person-months).

  (g) $/Page of Documentation.

Example

Given the following values, compute function point and productivity when all complexity adjustment factor (CAF) and weighting factors are **average**.

User Input = 50

User Output = 40

User Inquiries = 35

User Files = 6

External Interface = 4

Effort = 36.9 p-m

Example.....

- **Solution:**
- **Step-1:**

  As complexity adjustment factor is average (given in question), hence, scale = 3.

$$\sum(Fi) = 14 * 3 = 42$$

- **Step-2:**

$$CAF = [0.65 + 0.01 * \sum(Fi)]$$

$$CAF = 0.65 + (0.01 * 42) = 1.07$$

Example.....

- **Step-3:**

  As weighting factors are also average (given in question) hence we will multiply each individual function point to corresponding values in function point table 2.3

Count Total = (50*4) + (40*5) + (35*4) + (6*10) + (4*7)
= 628

- **Step-4:**

FP = Count-total * [0.65 + 0.01 * ∑(Fi)]
= Count-total * CAF
= 628 * 1.07 = 671.96

Example.....

Productivity = FP/ Effort

= 671.96/36.9

= 18.21

Function Point Controversy

- Like the KLOC measure, function point use also has proponents and opponents
- Proponents claim that
  - FP is programming language independent
  - FP is based on data that are more likely to be known in the early stages of a project, making it more attractive as an estimation approach
- Opponents claim that
  - FP requires some "sleight of hand" because the computation is based on subjective data.
  - FPA has been criticized as not being universally applicable to all types of software.
    - For example, FPA doesn't capture all functional characteristics of real-time software

Object-oriented Metrics

- If you are planning to develop a software through an object oriented approach, then you can go for object oriented metrics.

- Conventional metrics are not provide granularity for schedule and effort estimation for incremental or evolutionary projects.

Object-oriented Metrics.....

- Lorenz and Kidd [Lor94] suggest the following set of metrics for OO projects:
- **Number of scenario scripts** (i.e., use cases)
  - detailed sequence of steps that describes the interaction between the user and the application.
  - Each script is organized into triplets of the form

    { **initiator,** ***action, participant*** }
  - This number is directly related to the size of an application and to the number of test cases required to test the system

Object-oriented Metrics

- **Number of key classes** (the highly independent components)
  - Key classes are defined early in object-oriented analysis and are central to the problem domain
  - This number indicates the amount of effort required to develop the software
  - It also indicates the potential amount of reuse to be applied during development
- **Number of support classes**
  - Support classes are required to implement the system but are not immediately related to the problem domain (e.g., user interface, database, computation)

Object-oriented Metrics...

- **Average number of support classes per key class**
  - Key classes are identified early in a project (e.g., at requirements analysis)
  - Estimation of the number of support classes can be made from the number of key classes
  - GUI applications have between two and three times more support classes as key classes
  - Non-GUI applications have between one and two times more support classes as key classes
- **Number of subsystems**
  - A subsystem is an aggregation of classes that support a function that is visible to the end user of a system

Use Case-Oriented Metrics

- Describes customer-level or business domain requirements that imply software features and functions.
- Use case is defined early in the software process, allowing it to be used for estimation before significant modeling and construction activities are initiated.
- Independent of programming language.
- The number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Use Case-Oriented Metrics...

- Use cases can be created at vastly different levels of abstraction, there is no standard "size" for a use case.
- *use-case points (UCPs) as a mechanism for estimating* project effort and other characteristics.
- The UCP is a function of the number of actors and transactions implied by the use-case models.

Metrics for Software Quality

- You can use measurement to assess the quality of the requirements and design models, the source code, and the test cases that have been created as the software is engineered.
- A project manager must also evaluate quality as the project progresses.
- Private metrics collected by individual software engineers are combined to provide project-level results.
- Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team.

Metrics for Software Quality.....

- Correctness
  - This is the number of defects per KLOC, where a defect is a verified lack of conformance to requirements
  - Defects are those problems reported by a program user after the program is released for general use
- Maintainability
  - This describes the ease with which a program can be corrected if an error is found, adapted if the environment changes, or enhanced if the customer has changed requirements
  - Mean time to change (MTTC) : the time to analyze, design, implement, test, and distribute a change to all users
    - Maintainable programs on average have a lower MTTC

Metrics for Software Quality.....

- **integrity**
  - measures system's ability to withstand attacks on its security
- To measure integrity, two additional attributes must be defined:
    - threat
    - security.
- *Threat is the* probability that an attack of a specific type will occur within a given time.
- *Security* is the probability that the attack of a specific type will be repelled.

Metrics for Software Quality.....

- The integrity of a system can then be defined as:

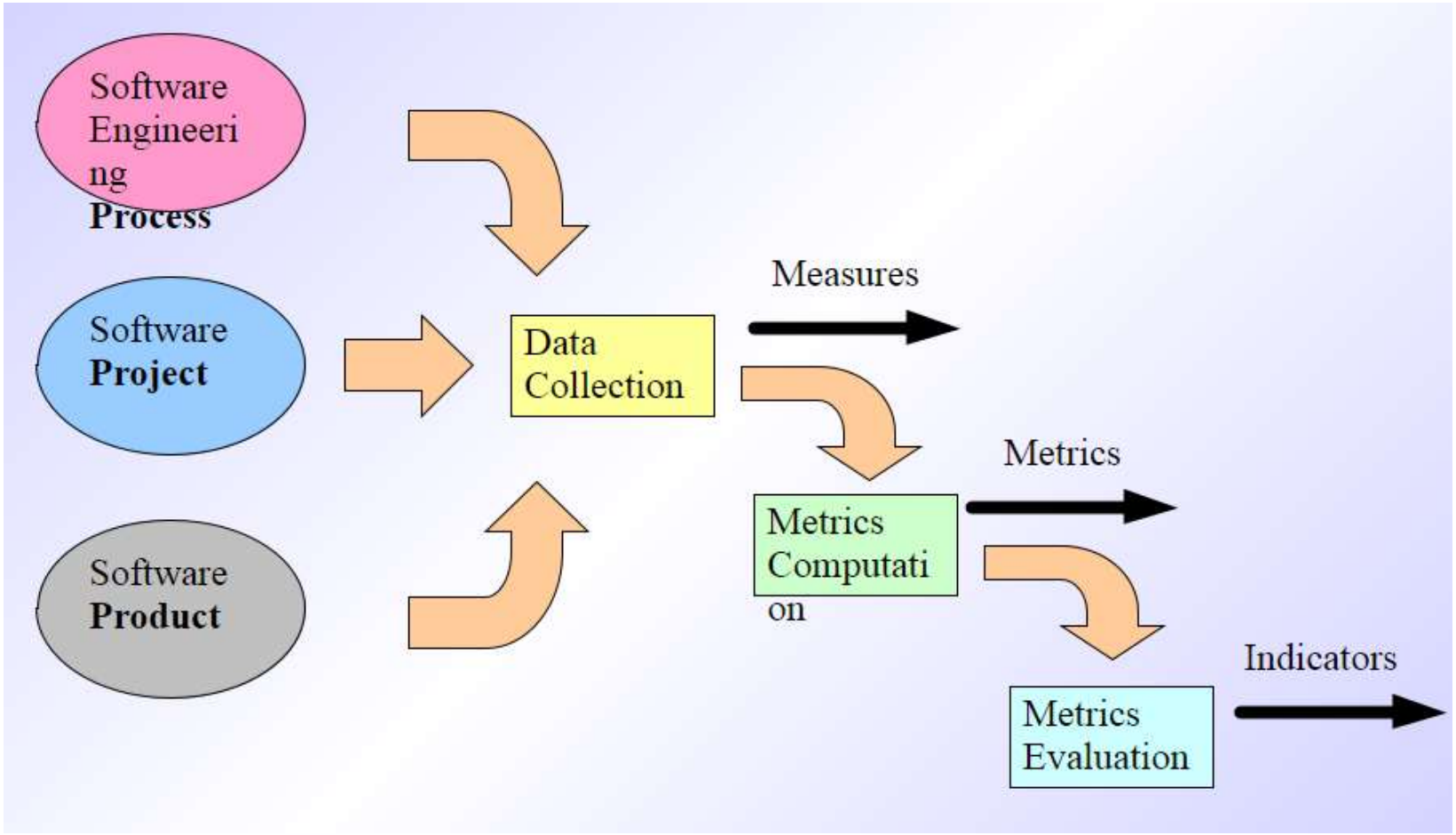
  $$\text{Integrity} = \sum [1 - (\text{threat} * (1 - \text{security}))]$$

- For example, if threat (the probability that an attack will occur) is 0.25 and security (the likelihood of repelling an attack) is 0.95, the integrity of the system is 0.99 (very high).
- **Usability.**
  - Usability is an attempt to quantify ease of use and can be measured in terms of the characteristics

Arguments for Software Metrics

- Most software developers do not measure, and most have little desire to begin
- Establishing a successful company-wide software metrics program can be a multi-year effort
- But if we do not measure, there is no real way of determining whether we are improving
- Measurement is used to establish a process baseline from which improvements can be assessed
- Software metrics help people to develop better project estimates, produce higher-quality systems, and get products out the door on time

Software Metrics Baseline Process



Establishing a Metrics Baseline

- By establishing a metrics baseline, benefits can be obtained at the software process, product, and project levels
- The same metrics can serve many masters
- The baseline consists of data collected from past projects
- Baseline data must have the following attributes
  - Data must be reasonably accurate (guesses should be avoided)
  - Data should be collected for as many projects as possible
  - Measures must be consistent (e.g., a line of code must be interpreted consistently across all projects)
  - Past applications should be similar to the work that is to be estimated
- After data is collected and metrics are computed, the metrics should be evaluated and applied during estimation, technical work, project control, and process improvement

Costing ,Scheduling and Tracking Techniques

Cost estimation

- **To estimate how much software-engineering time will be required to do some work.**
  - *Elapsed time*
    - The difference in time from the start date to the end date of a task or project.
  - *Development effort*
    - The amount of labour used in *person-months* or *person-days*.
    - To convert an estimate of development effort to an amount of money:

      You multiply it by the *weighted average cost* (*burdened* cost) of employing a software engineer for a month (or a day).

Example

- *In your organization, although the average salary is $4,000 /month, the weighted average salary for cost estimation purposes is $11,000/month.You have determined that a particular project will take 7 person-months to complete. How much would you estimate this project will cost financially?*

You estimate that the project will cost 7×$11,000 = $77,000.

Principles of effective cost estimation

- **Principle 1: Divide and conquer.**
  - To make a better estimate, you should divide the project up into individual subsystems.
  - Then divide each subsystem further into the activities that will be required to develop it.
  - Next, you make a series of detailed estimates for each individual activity.
  - And sum the results to arrive at the grand total estimate for the project.

Principles of effective cost estimation

- **Principle 2: Include all activities when making estimates.**
  - The time required for *all* development activities must be taken into account.
  - Including:
    - Prototyping
    - Design
    - Inspecting
    - Testing
    - Debugging
    - Writing user documentation
    - Deployment.

Principles of effective cost estimation

- **Principle 3: Base your estimates on past experience combined with knowledge of the current project.**
  - If you are developing a project that has many similarities with a past project:
    - You can expect it to take a similar amount of work.
  - Base your estimates on the *personal judgement* of your experts

    or
  - Use *algorithmic models* developed in the software industry as a whole by analyzing a wide range of projects.
    - They take into account various aspects of a project's size and complexity, and provide formulas to compute anticipated cost

Algorithmic models

- Allow you to systematically estimate development effort.
  - Project managers base their estimates on factors such as the following
    - The number of use cases
    - The number of distinct requirements
    - The number of classes in the domain model
    - The number of widgets in the prototype user interface
    - An estimate of the number of lines of code

Algorithmic models

- A typical algorithmic model uses a formula like the following:
  - COCOMO
  - Functions Points:

COCOMO MODEL

(Constructive Cost **Model**)

Most widely used software estimation model.

COCOMO predicts the efforts and schedule of a software product.

COCOMO Models

- COCOMO is defined in terms of three different models:
  - the **Basic model**,
  - the **Intermediate model**, and
  - the **Detailed model**.
- The more complex models account for more factors that influence software projects, and make more accurate estimates.

The Development mode

- the most important factors contributing to a project's duration and cost is the Development Mode
  - **Organic Mode:** The project is developed in a familiar, stable environment, and the product is similar to previously developed products. The product is relatively small, and requires little innovation.
  - **Semidetached Mode:** It is an intermediate (in terms of size and complexity) project. The project's characteristics are intermediate between Organic and Embedded.

The Development mode

Embedded Mode:

- The project is characterized by tight, inflexible constraints and interface requirements.
- An embedded mode project will require a great deal of innovation.
- This project having a high level of complexity with a large team size by considering all sets of parameters (software, hardware and operational).

The Development mode

Mode/Details	Project Size	Project Nature	Innovation	Deadline
Organic	2-50 KLOC	Small sized and experienced developers	Little	Flexible (not tight)
Semi-detached	50-300 KLOC	Medium size project and team	Medium	Medium
Embedded	Over 300 KLOC	Large projects	Significant	Tight

Basic COCOMO model

- Computes software development effort (and cost) as function of program size expressed in estimated lines of code
- Model:

Category	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Basic COCOMO Equations

$$E = akLOC^{b}$$

$$D = cE^{d}$$

where

- E is effort in person-months
- D is development time in months
- kLOC is estimated number of lines of code

Basic COCOMO Equations

- P=E/D

- P- Total number of persons required to accomplish the project

Merits

- Good for quick,early,rough order of estimates
- Limitations:
  - Accuracy is limited
  - Does not consider certain factors(personalquality,experience,tools)

Example

- For a given project was estimated with a size of 300 KLOC. Calculate the Effort, Scheduled time for development. Also, calculate the Average staff size and Productivity of the software for Organic project type.
- **Ans:** Given estimated size of project is: 300 KLOC
- **For Organic**
- Effort (E) = $a*(KLOC)^b = 2.4*(300)^{1.05} = 957.61$ PM
  Scheduled Time (D) = $c*(E)^d = 2.5*(957.61)^{0.38} =$ 33.95 Months(M)

Example

- Avg. staff Size(P) = E/D = 957.61/33.95 = 28.21 Persons

  Productivity of Software = KLOC/E = 300/957.61 = 0.3132 KLOC/MM = 313 LOC/MM

Intermediate COCOMO

- Computes software development effort as a function of program size and a set of "cost drivers" that include subjective assessments of product, hardware, personnel, and project attributes.
- Give rating to 15 attributes, from "very low" to "extra high", find effort multipllier (from table) and **product of all effort multipliers gives an *effort adjustment factor (EAF)***

Cost Driver Attributes

- Product attributes
  - Required reliability
  - Database size
  - Product complexity
- Computer attributes
  - Execution time constraint
  - Main storage constraint
  - Virtual machine volatility
  - Computer turnaround time

Cost Driver Attributes (Continued)

- Personnel attributes
  - Analyst capability, Programmer capability
  - Applications experience
  - Virtual machine experience
  - Programming language experience
- Project attributes
  - Use of modern programming practices
  - Use of software tools
  - Required development schedule

Intermediate COCOMO Equation

Category	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

$$E = akLOC^{b} \times EAF$$

- where
- E is effort in person-months,
- kLOC is estimated number of lines of code

EAF Parameter values

COST DRIVERS PARAMETERS	VERY LOW	LOW	NOMINAL	HIGH	VERY HIGH
Product Parameter					
Required Software	0.75	0.88	1	1.15	1.4
Size of Project Database	NA	0.94		1.08	1.16
Complexity of The Project	0.7	0.85		1.15	1.3

EAF Parameter values

Hardware Parameter					
Performance Restriction	NA	NA	1	1.11	1.3
Memory Restriction	NA	NA		1.06	1.21
virtual Machine Environment	NA	0.87		1.15	1.3
Required Turnabout Time	NA	0.94		1.07	1.15

EAF Parameter values

Personnel Parameter					
Analysis Capability	1.46	1.19	1	0.86	0.71
Application Experience	1.29	1.13		0.91	0.82
Software Engineer Capability	1.42	1.17		0.86	0.7
Virtual Machine Experience	1.21	1.1		0.9	NA
Programming Experience	1.14	1.07		0.95	NA

EAF Parameter values

Project Parameter					
Software Engineering Methods	1.24	1.1	1	0.91	0.82
Use of Software Tools	1.24	1.1		0.91	0.83
Development Time	1.23	1.08		1.04	1.1

Example

- For a given project was estimated with a size of 300 KLOC. Calculate the Effort, Scheduled time for development by considering developer having very high application experience and very low experience in programming.

  Ans:

- Given the estimated size of the project is: 300 KLOC
  Developer having highly application experience: 0.82 (as per above table)
  Developer having very low experience in programming: 1.14(as per above table)

Example

- EAF = 0.82*1.14 = 0.9348
  Effort (E) = a*(KLOC)^{b} *EAF = 3.0*(300)$^{1.12}$ *0.9348 = 1668.07 PM
  Scheduled Time (D) = c*(E)^{d} = 2.5*(1668.07)$^{0.35}$ = 33.55 Months(M)

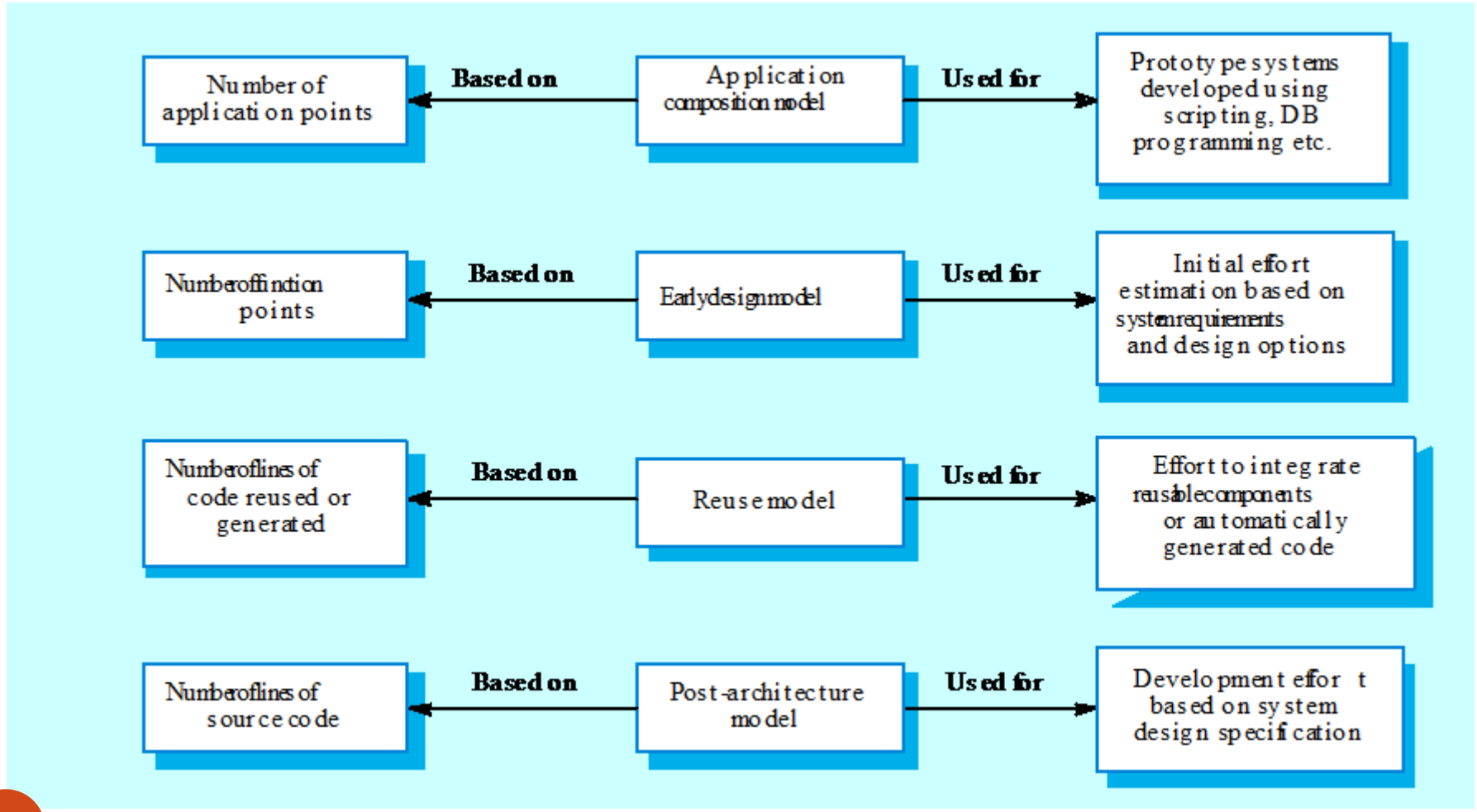
Advanced COCOMO

- Incorporates all characteristics of intermediate COCOMOwith an assessment of the cost driver's impact on each step of software engineering process.

COCOMO 2 models

- COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.
- The sub-models in COCOMO 2 are:
  - Application composition model. Used when software is composed from existing parts.
  - Early design model. Used when requirements are available but design has not yet started.
  - Reuse model. Used to compute the effort of integrating reusable components.
  - Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

Use of COCOMO 2 models



Principles of effective cost estimation

- Principle 4: **Be sure to account for *differences* when extrapolating from other projects.**
  - Different software developers
  - Different development processes and maturity levels
  - Different types of customers and users
  - Different schedule demands
  - Different technology
  - Different technical complexity of the requirements
  - Different domains
  - Different levels of requirement stability

Principles of effective cost estimation

- **Principle 5: Anticipate the worst case and plan for contingencies.**
  - Develop the most critical use cases first
    - If the project runs into difficulty, then the critical features are more likely to have been completed
  - Make three estimates:
    - Optimistic (O)
      - Imagining a everything going perfectly
    - Likely (L)
      - Allowing for typical things going wrong
    - Pessimistic
      - Accounting for everything that could go wring

Principles of effective cost estimation

- **Principle 6: Combine multiple independent estimates.**
  - Use several different techniques and compare the results.
  - If there are discrepancies, analyze your calculations to discover what factors causing the differences.
  - Use the **Delphi technique.**
    - Several individuals initially make cost estimates in private.
    - They then share their estimates to discover the discrepancies.
    - Each individual repeatedly adjusts his or her estimates until a consensus is reached.

Principles of effective cost estimation

- **Principle 7: Revise and refine estimates as work progresses**
  - As you add detail.
  - As the requirements change.
  - As the risk management process uncovers problems.

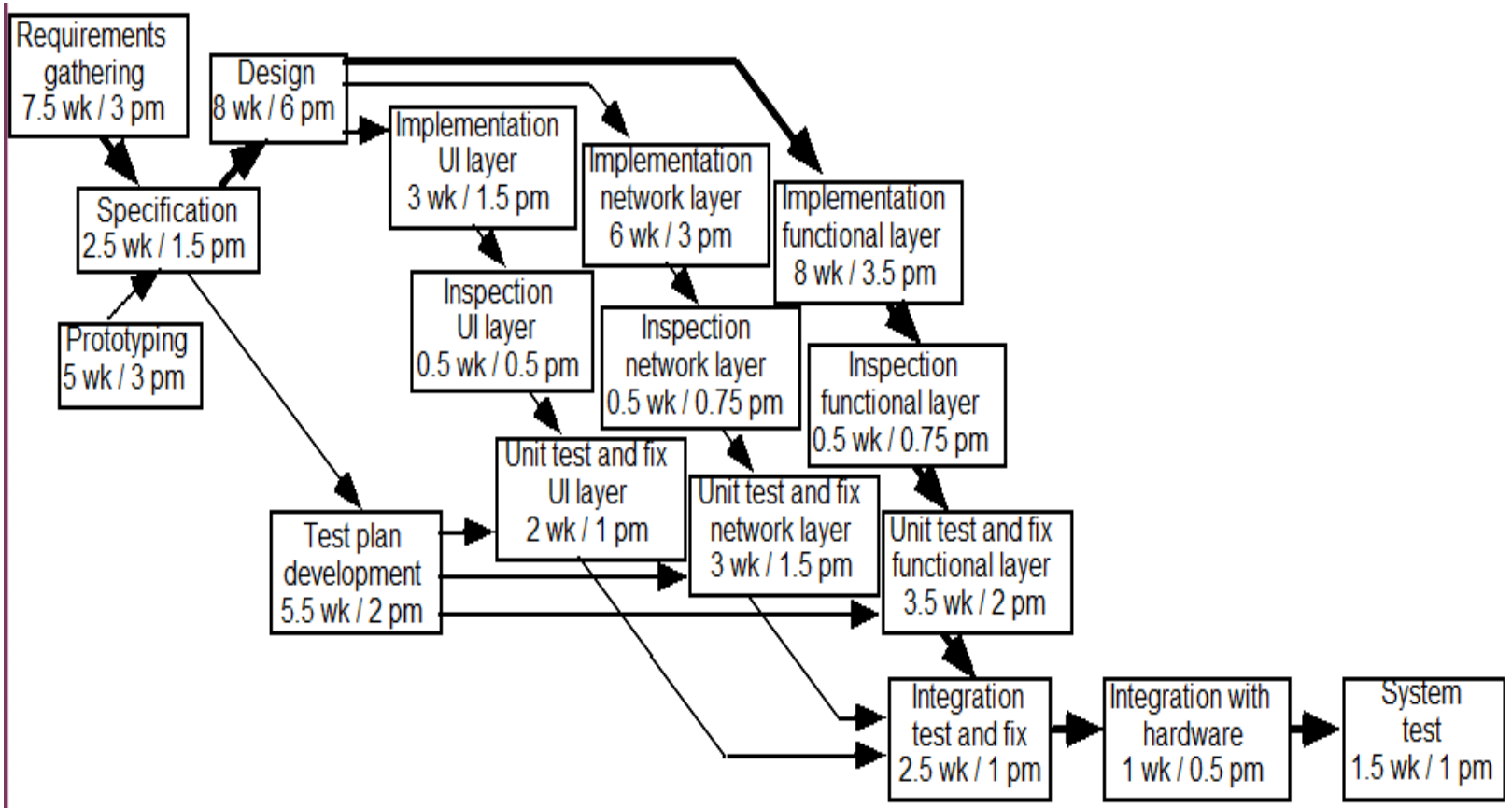
Project Scheduling and Tracking

- *Scheduling* is the process of deciding:
  - In what sequence a set of activities will be performed.
  - When they should start and be completed.
  - Two important scheduling techniques
    - PERT chart
    - Gantt chart
- *Tracking* is the process of determining how well you are sticking to the cost estimate and schedule.
- Important tracking technique
  - Earned value charts

PERT charts(Program Evaluation Review Technique)

- A PERT chart shows the sequence in which tasks must be completed.
  - In each node of a PERT chart, you typically show the elapsed time and effort estimates.
  - One of the most important uses of a PERT chart is to determine the ***critical path***.
    - The ***critical path*** indicates the minimum time in which it is possible to complete the project.
    - It is computed by searching for the path through the chart that has the greatest cumulative elapsed time and no idle time.

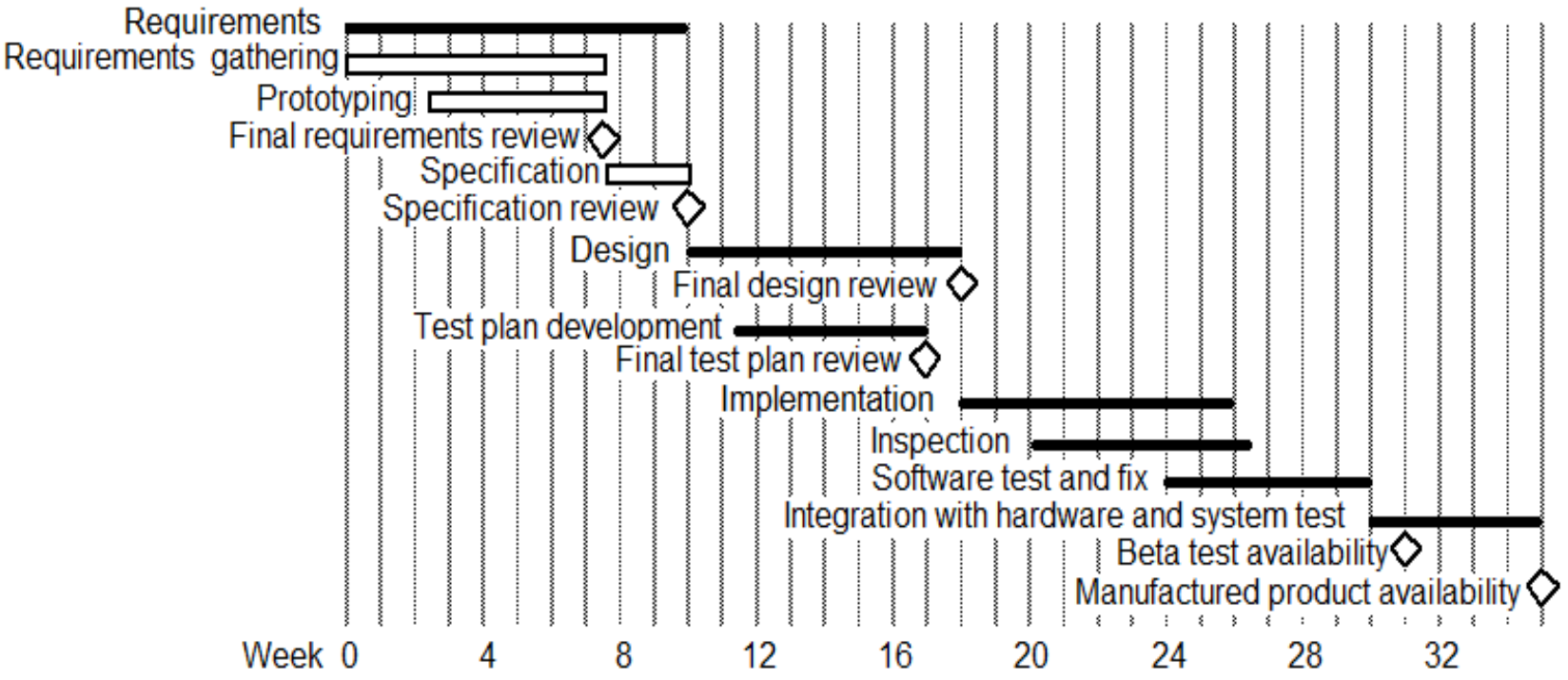
Example of a PERT chart



Gantt charts

- A Gantt chart is used to graphically present the start and end dates of each software engineering task
  - One axis shows time.
  - The other axis shows the activities that will be performed.
  - The black bars are the top-level tasks.
  - The white bars are subtasks
  - The diamonds are *milestones:*
    - Important deadline dates, at which specific events may occur

Example of a Gantt chart



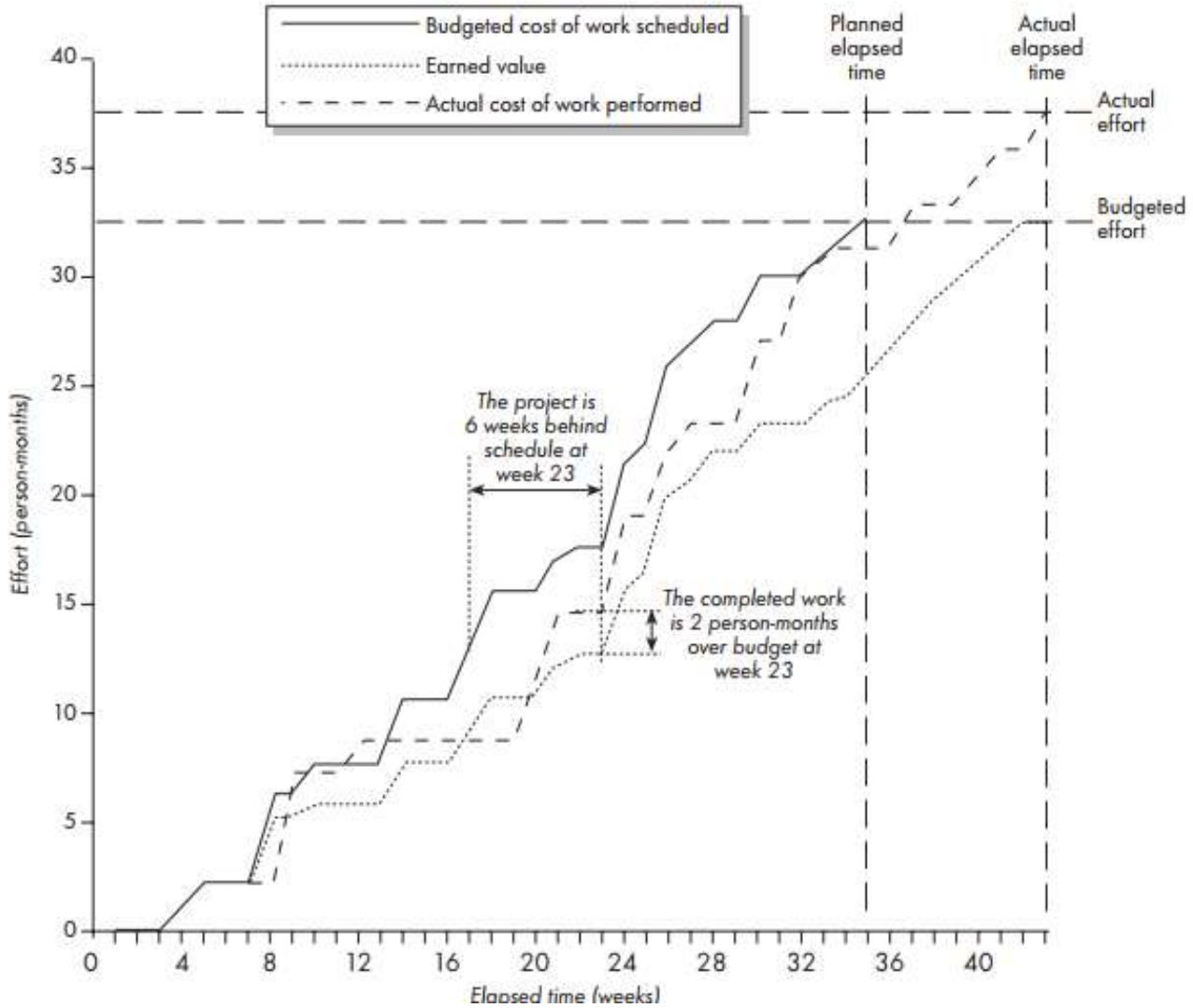
Earned value

- *Earned value* is the amount of work completed, measured according to the *budgeted* effort that the work was supposed to consume.
- It is also called the *budgeted cost of work performed*.

Earned value charts

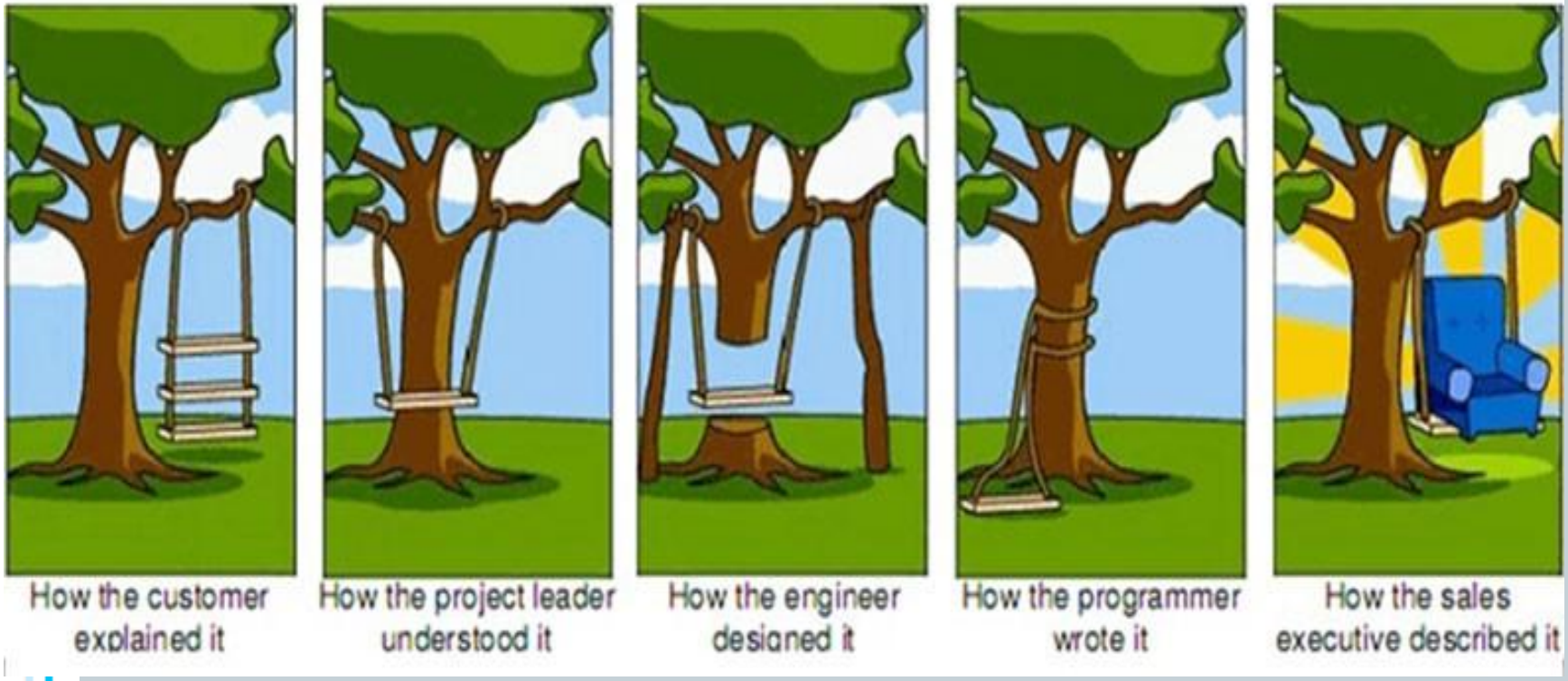
- An earned value chart has three curves:
  - The budgeted cost of the work scheduled(planned value)
  - The earned value.
  - The actual cost of the work performed so far.

Example of an earned value chart

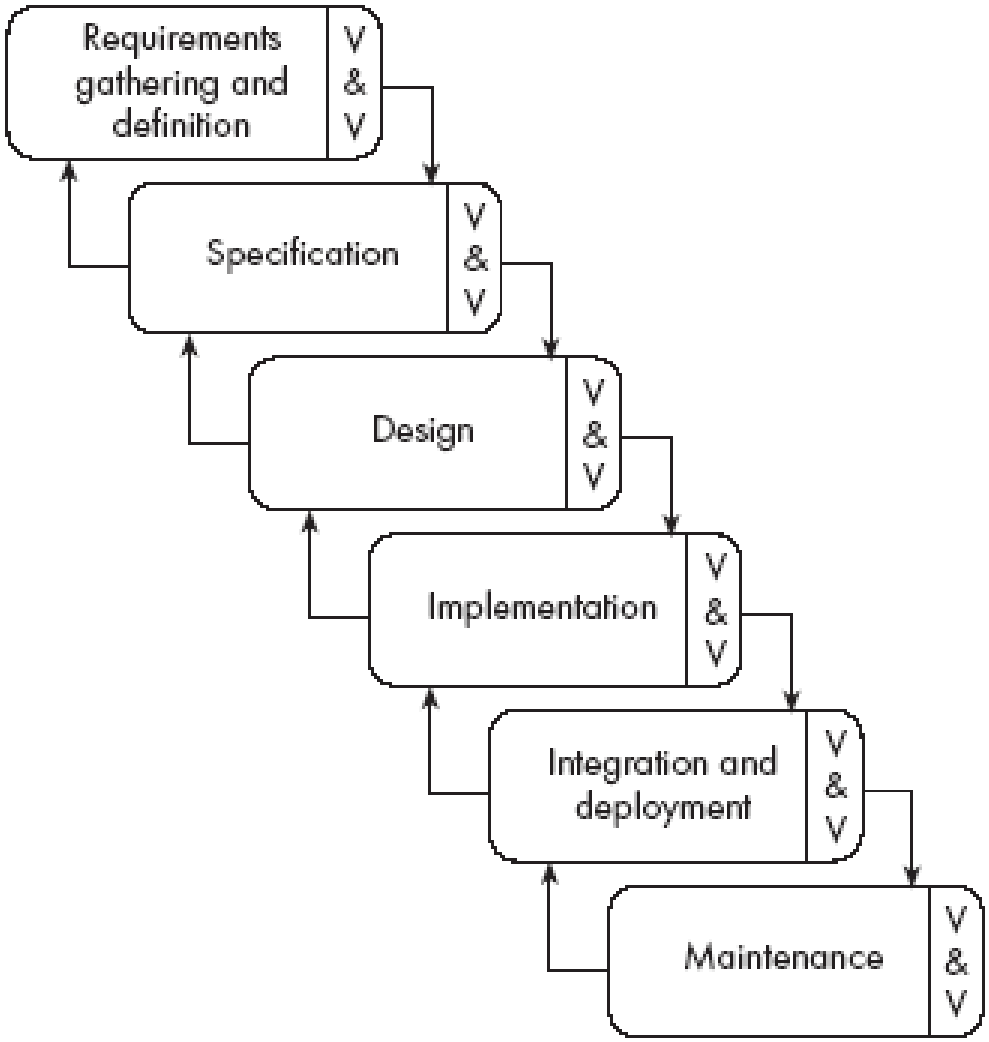


CS3004D Software Engineering

Software Crisis



The waterfall model



Software Process Models

- Software process models are general approaches for organizing a project into activities.
- Also termed as Software Life Cycle Model
  - Help the project manager and his or her team to decide:
    - What work should be done;
    - In what sequence to perform the work.
  - The models should be seen as *aids to thinking*, not rigid prescriptions of the way to do things.
  - Each project ends up with its own unique plan.

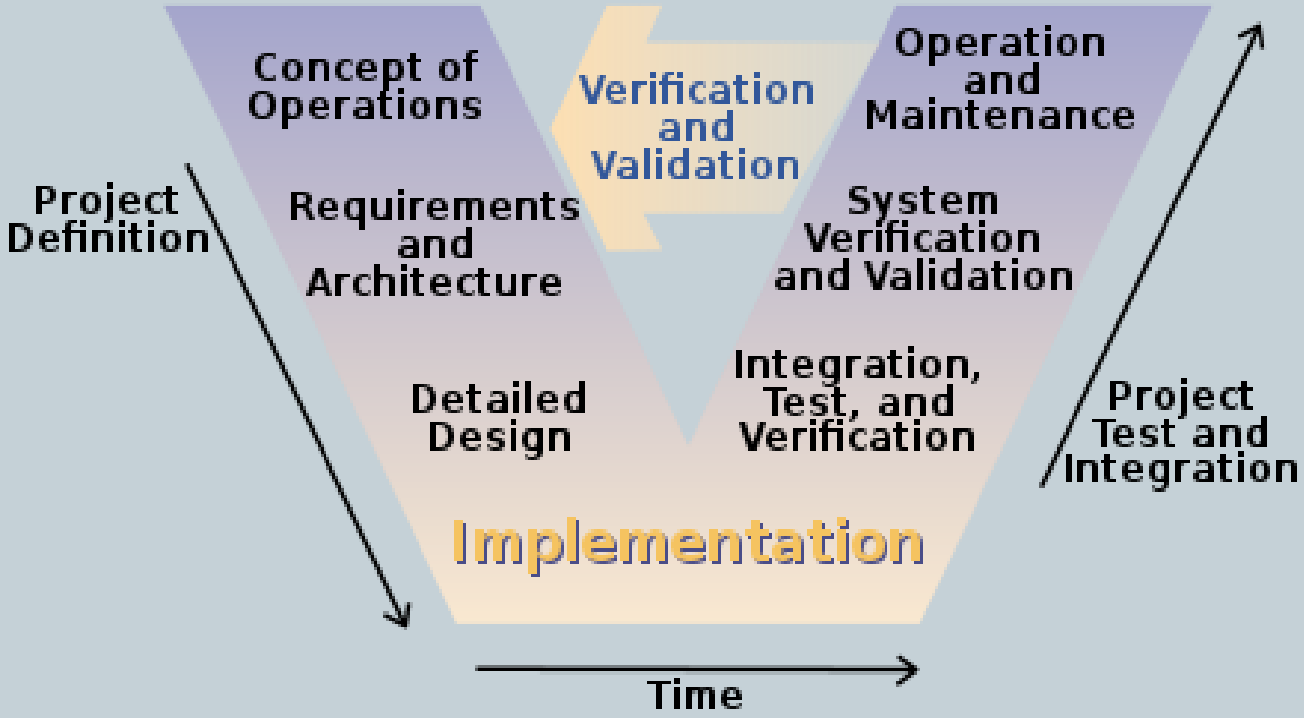
The waterfall model

- The classic way of looking at S.E. that accounts for the importance of requirements, design and quality assurance.
  - The model suggests that software engineers should work in a series of stages.
  - Before completing each stage, they should perform quality assurance (verification and validation).
  - The waterfall model also recognizes, to a limited extent, that you sometimes have to step back to earlier stages.

Limitations of the waterfall model

- The model implies that you should attempt to complete a given stage before moving on to the next stage
  - Does not account for the fact that requirements constantly change.
  - It also means that customers can not use anything until the entire system is complete.
- The model makes no allowances for prototyping.
- Have to wait till integration, and may give false impression on the progress of project
- It implies that you can get the requirements right by simply writing them down and reviewing them.
- The model implies that once the product is finished, everything else is maintenance.

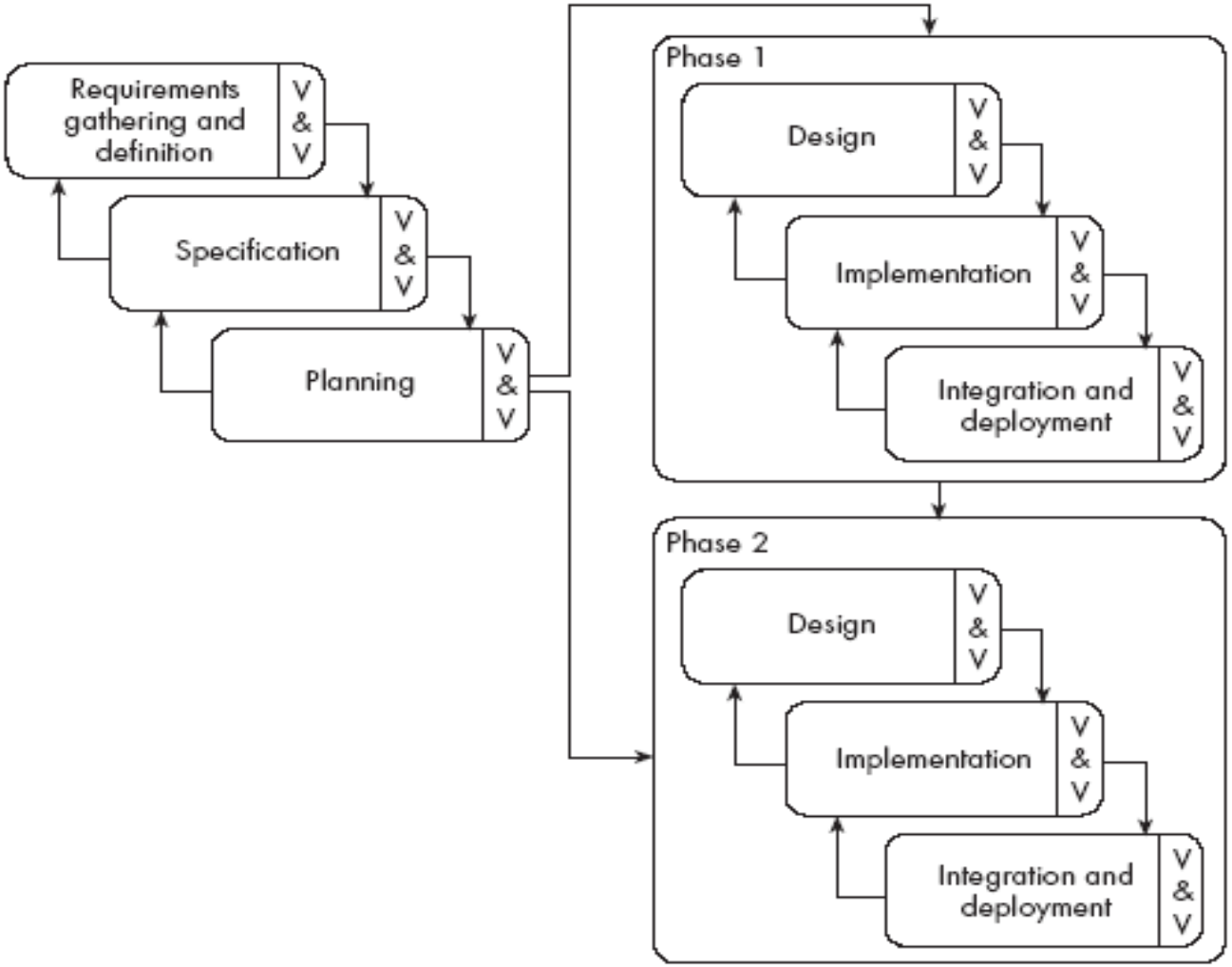
V Model


Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

V Model

- A variation of waterfall model
  - Emphasises Verification and Validation
  - Testing activities are planned in parallel with the development
  - Used in applications where reliability and safety are important

The phased-release model



The phased-release model

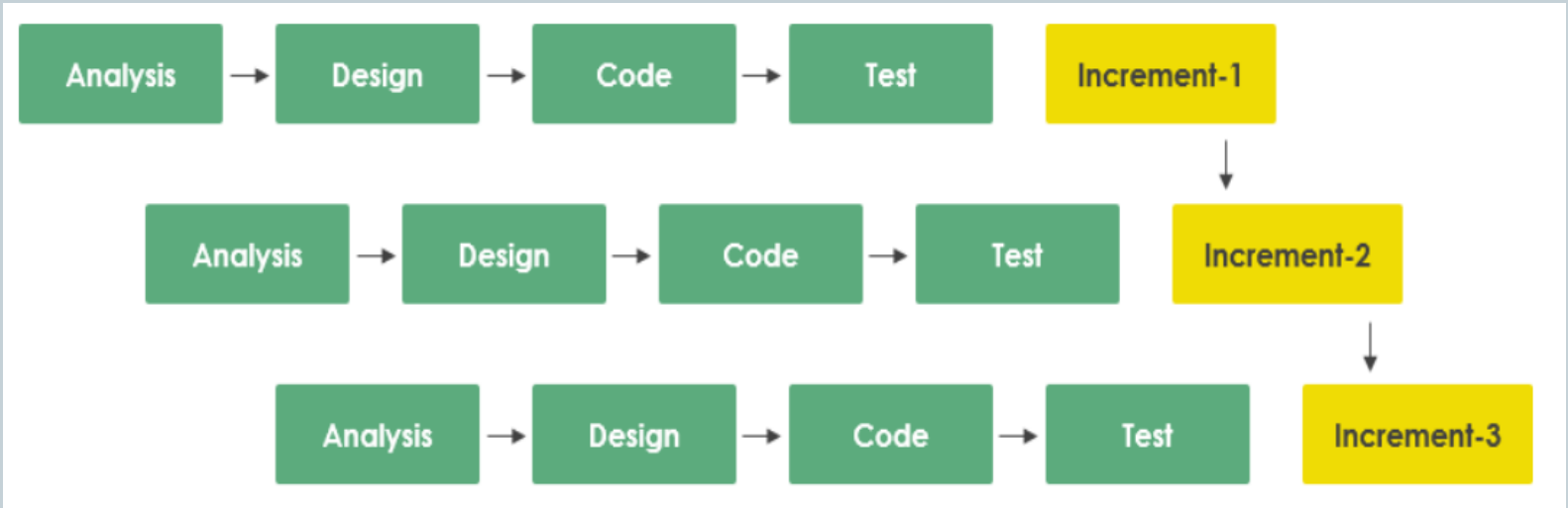
- It introduces the notion of *incremental* development.
  - After requirements gathering and planning, the project should be broken into separate subprojects, or *phases*.
  - Each phase can be released to customers when ready.
  - Parts of the system will be available earlier than when using a strict waterfall approach.
  - However, it continues to suggest that all requirements be finalized at the start of development.

Prototyping

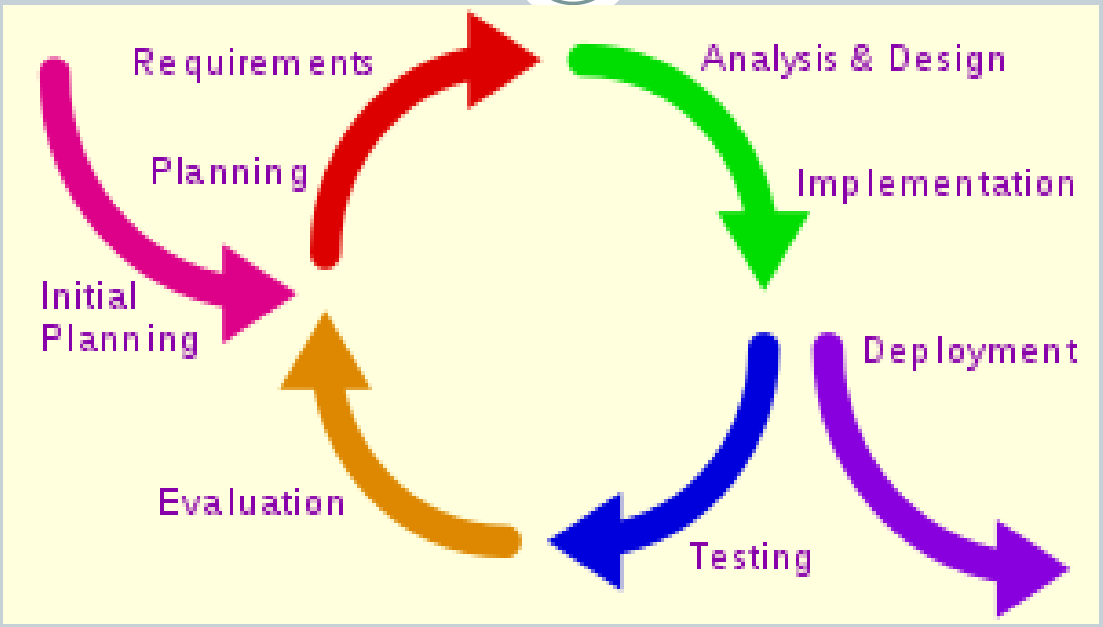
- Before the development starts a prototype needs to be built.
- A prototype is a toy implementation of the system.
- The software designer and implementer can get valuable feedback from the users early in the project

Incremental

- The product is designed, implemented and tested incrementally
- A little more is added each time until the product is finished

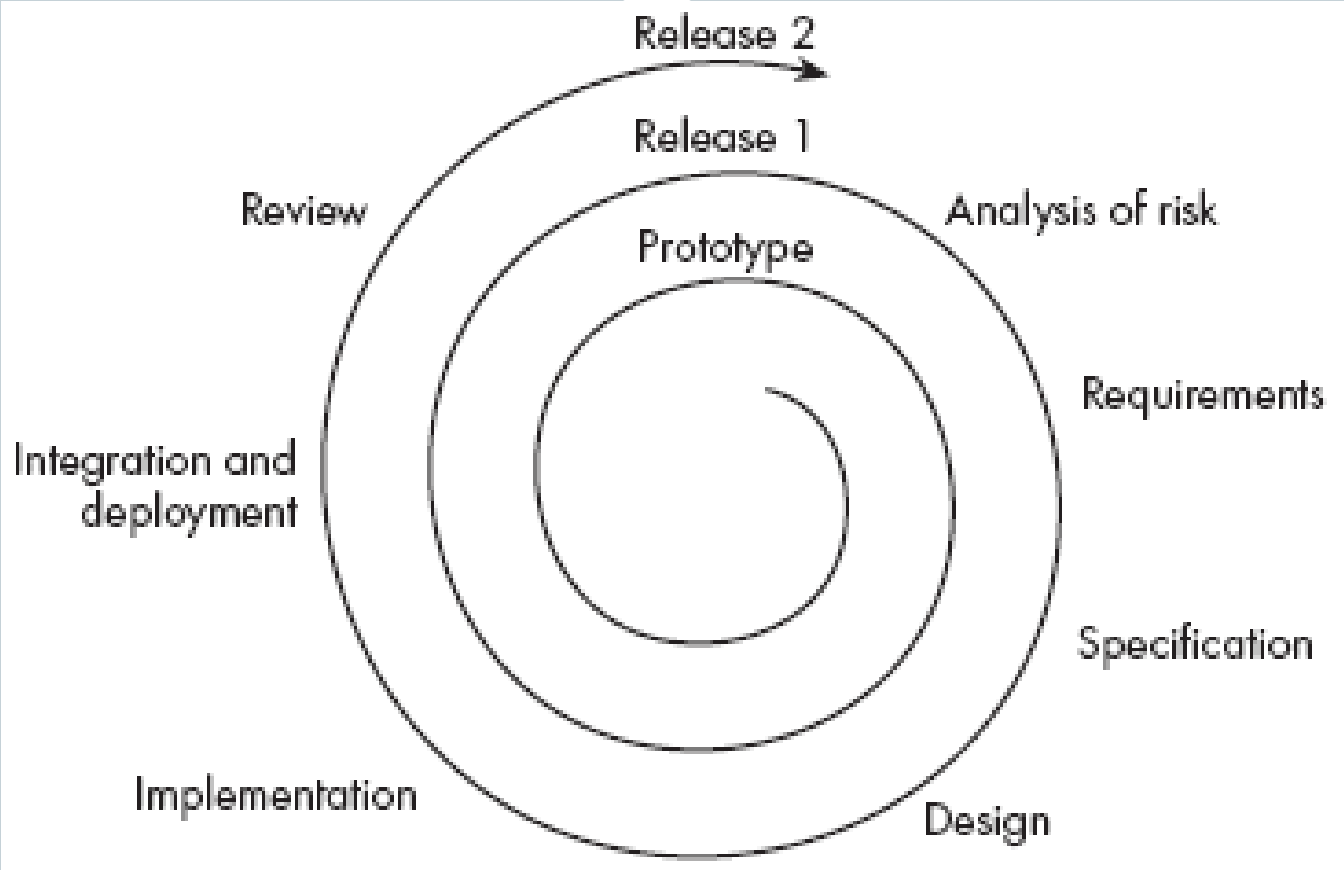


Iterative Approach



- Begins by specifying and implementing just part of the software, which can then be reviewed and prioritized in order to identify further requirements

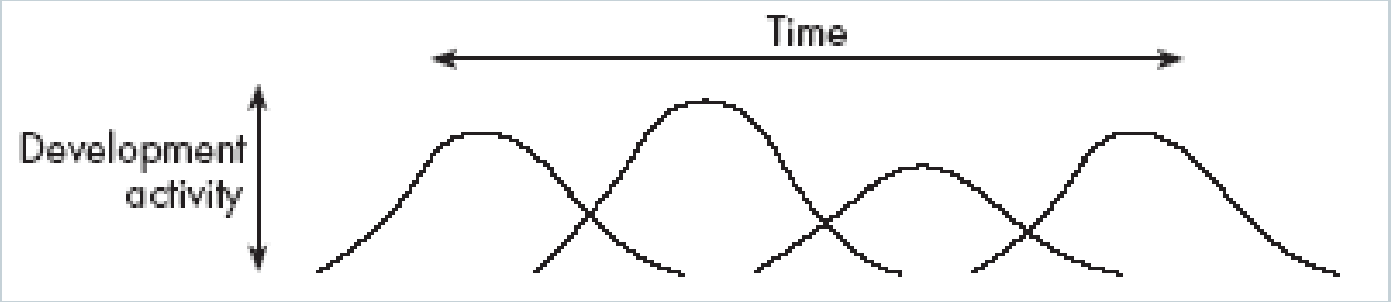
The spiral model



The spiral model

- It explicitly embraces prototyping and an *iterative* approach to software development.
  - Start by developing a small prototype.
  - Followed by a mini-waterfall process, primarily to gather requirements.
  - Then, the first prototype is reviewed.
  - In subsequent loops, the project team performs further requirements, design, implementation and review.
  - The first thing to do before embarking on each new loop is risk analysis.
  - Maintenance is simply a type of on-going development.

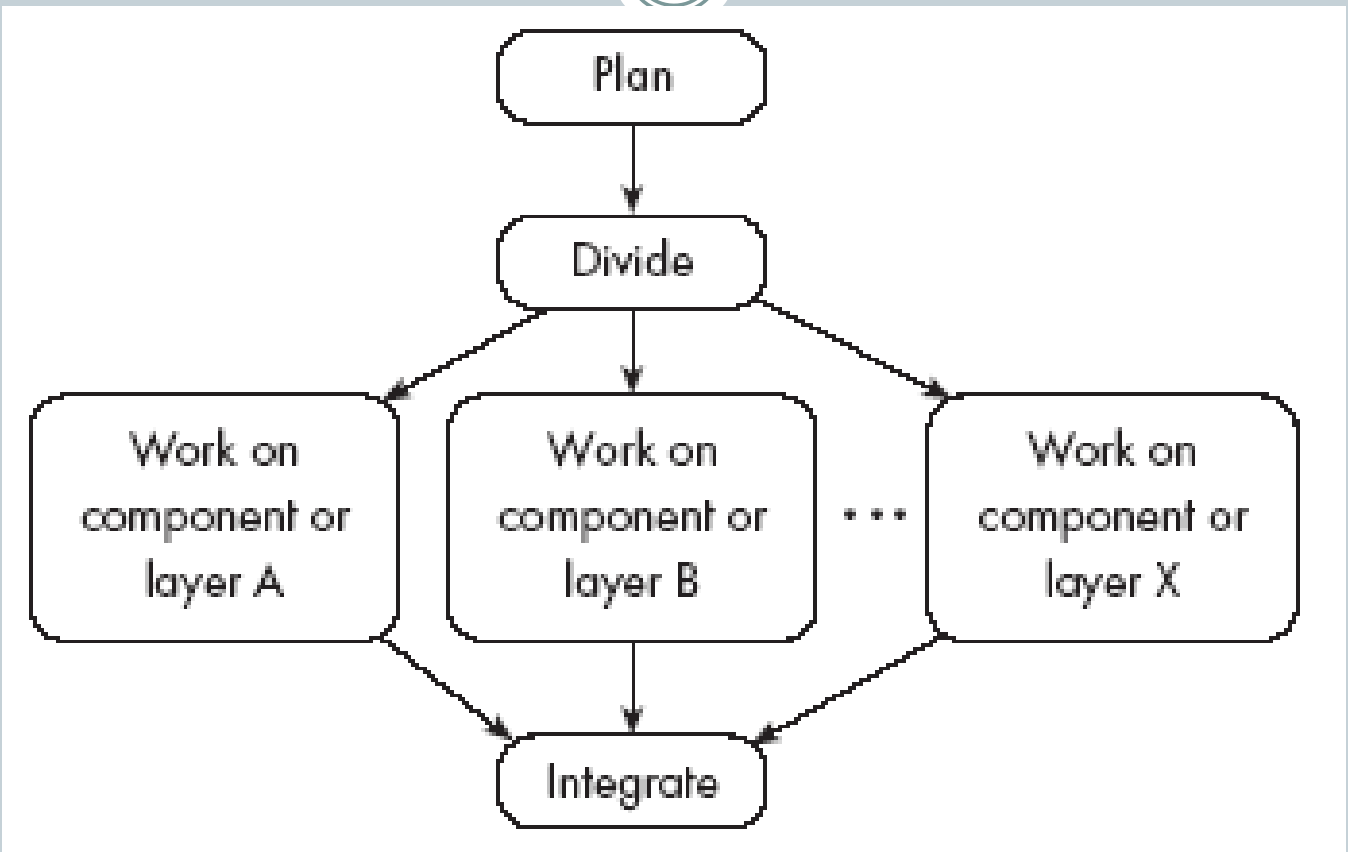
The evolutionary model



The evolutionary model

- It shows software development as a series of hills, each representing a separate loop of the spiral.
  - Shows that loops, or releases, tend to overlap each other.
  - Makes it clear that development work tends to reach a peak, at around the time of the deadline for completion.
  - Shows that each prototype or release can take
    - different amounts of time to deliver;
    - differing amounts of effort.

The concurrent engineering model



The concurrent engineering model

- It explicitly accounts for the divide and conquer principle.
  - Each team works on its own component, typically following a spiral or evolutionary approach.
  - There has to be some initial planning, and periodic integration.

- Open source models
  - software is distributed for free along with the source code, and interested people contribute improvements, without being paid by users
- Agile Model

Choosing a process model

When planning a particular project, you can **combine** the features of the models that apply best to your current project

Reengineering

- Periodically project managers should set aside some time to re-engineer part or all of the system
  - The extent of this work can vary considerably:
    - Cleaning up the code to make it more readable.
    - Completely replacing a layer.
    - *Re-factoring* part of the design.
  - In general, the objective of a re-engineering activity is to increase maintainability.