Please Add all vocab / key concepts to this quizlet. The quizlet can be run as a "Test" which is excellent to study from!
https://quizlet.com/250136641/cop4331-ucf-final-review-flash-cards/
Password is "nassif" no quotes all lowercase.
Anything in the text that you don't understand, provide a definition for it. (google / textbook)
Feel free to add to other chapters.
Italicize questions / comments
List of Turgut's Class directory:
http://www.cs.ucf.edu/~turgut/COURSES/

Chapter 1

Software Development:

- **Analysis** - to break a program into smaller, more manageable parts
- **Synthesis** - to recombine small parts into a program

Vocab:

- **Method** - A formal procedure to accomplish a goal
- **Tool** - An instrument or automated system for accomplishing something in a better way.
- **Procedure** - A combination of tools and techniques to produce a product.

- **Paradigm** - A philosophy or approach for building a product. (e.g OO versus Structured Approaches)
- **Computer Science -** Focused on computer hardware, compilers, operating systems, and programming languages.
- **Software Engineering -** A discipline that uses computer and software technologies as problem-solving tools.
- **Fault** - Occurs when a human makes a mistake, called an error, in performing some software activities.
- **Failure -** a departure from the system's required behavior.
- **Safety-Critical -** Something whose failure can pose a threat to human life or health.

What is good software? There are many views.

- **Transcendental View -** Quality is something we can recognize but not define.
- **User View** - Quality is fitness for the purpose.
- **Manufacturing / Process View -** quality is conformance to a specification.
- **Product View** - Quality is tied to inherent characteristics of the product and software metrics.
- **Value-Based View -** Quality is based on the amount a customer is willing to pay for it.
- **Customer** - the company, organization, or person who pays for the software system
- **Developer** - the company, organization, or person who is building the software system
- **User** - the person or people who will actually use the system
- **Stakeholders** - Participants in a software development project

List of all stakeholders:
Users
Developers
Clients: pay for the software to be developed
Customers: buy the software after it is developed

Users: use the system
Domain experts: familiar with the problem that the software must automate
Market Researchers: conduct surveys to determine future trends and potential customers
Lawyers or auditors: familiar with government, safety, or legal requirements
Software engineers
Shareholders

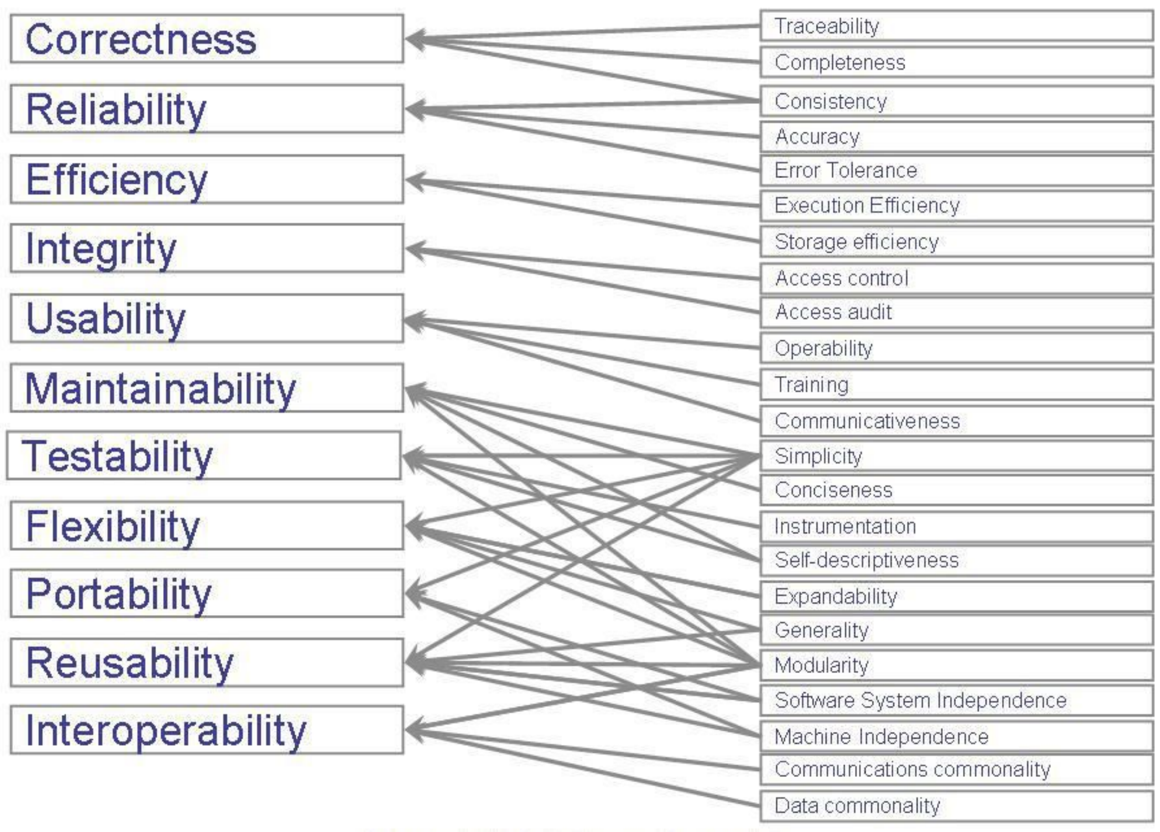
Short Answer: **<u>EVERYONE IS A STAKE HOLDER UNLESS THEY DONT RELATE TO THE SOFTWARE DEVELOPMENT PROJECT</u>**

Three ways of considering quality:

1. Quality of the Product
2. Quality of the Process that results from the product.
3. Quality of the Product in the context of the business environment in which it will be used.

External Characteristics - Functionality, number of failures, etc
Internal Characteristics - Number of faults

McCall's Quality Model:



Technical Terms:

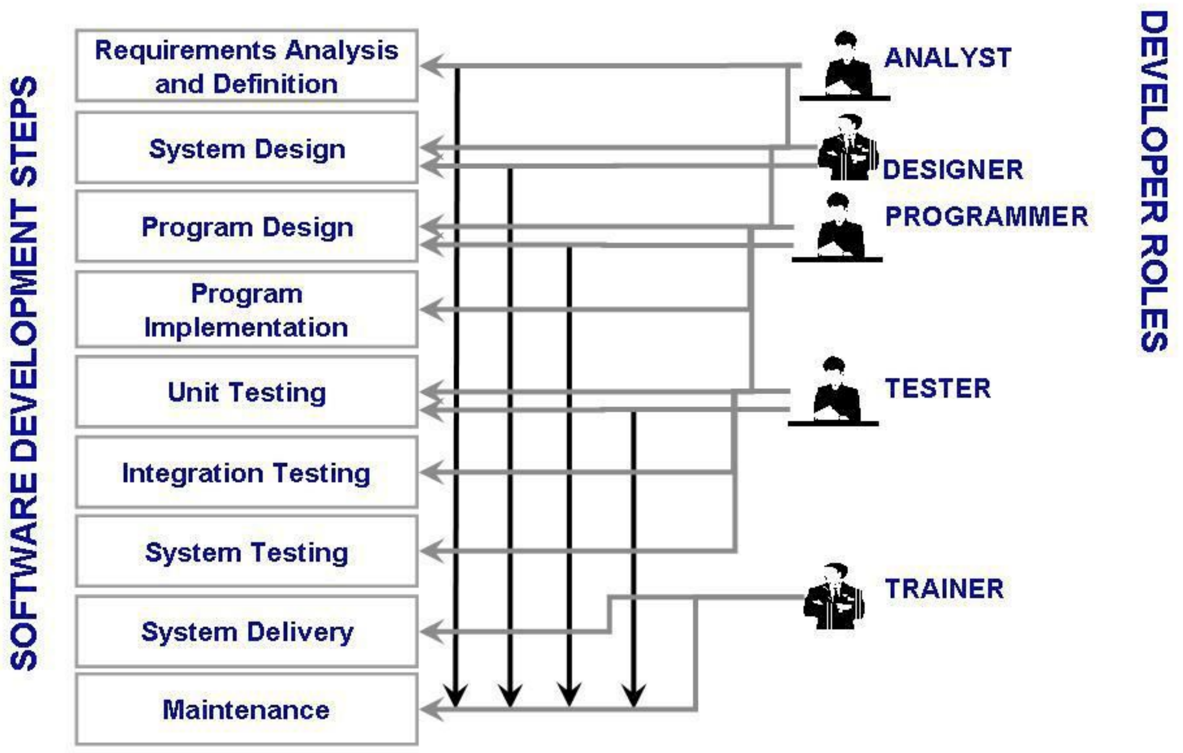
- **Activity -** An event initiated by a trigger.
- **Objects or Entities -** The elements involved in activities.
- **Relationship -** Defines the interaction between Activities and Entities.
- **System Boundaries -** Determine origins of input and destinations of output.

A system contains all of these. ^

Types of Systems:

- **Layered System -** A system of systems.
- **Requirement analysts**: work with the customers to identify and document the requirements
- **Designers**: generate a system-level description of what the system is supposed to do

- **Programmers**: write lines of code to implement the design
- **Testers**: catch faults
- **Trainers**: show users how to use the system
- **Maintenance team**: fix faults that show up later
- **Librarians**: prepare and store documents such as software requirements
- **Configuration management team**: maintain correspondence among various artifacts and software releases



Prototyping - building a smaller version of the system.

Architectural Design Types:

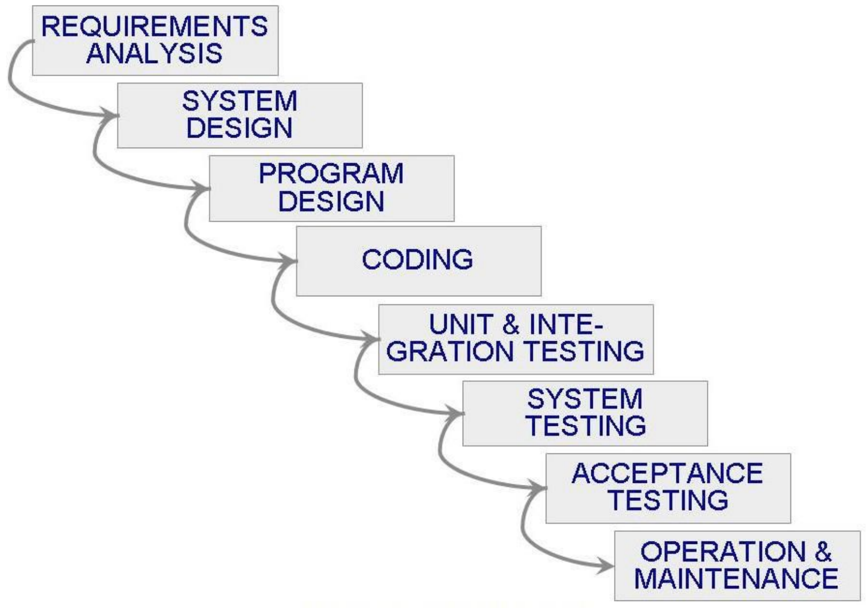
- Modular Decomposition
- Data-oriented Decomposition
- Event-driven Decomposition
- Outside-in Decomposition
- Object Oriented Decomposition

Chapter 2

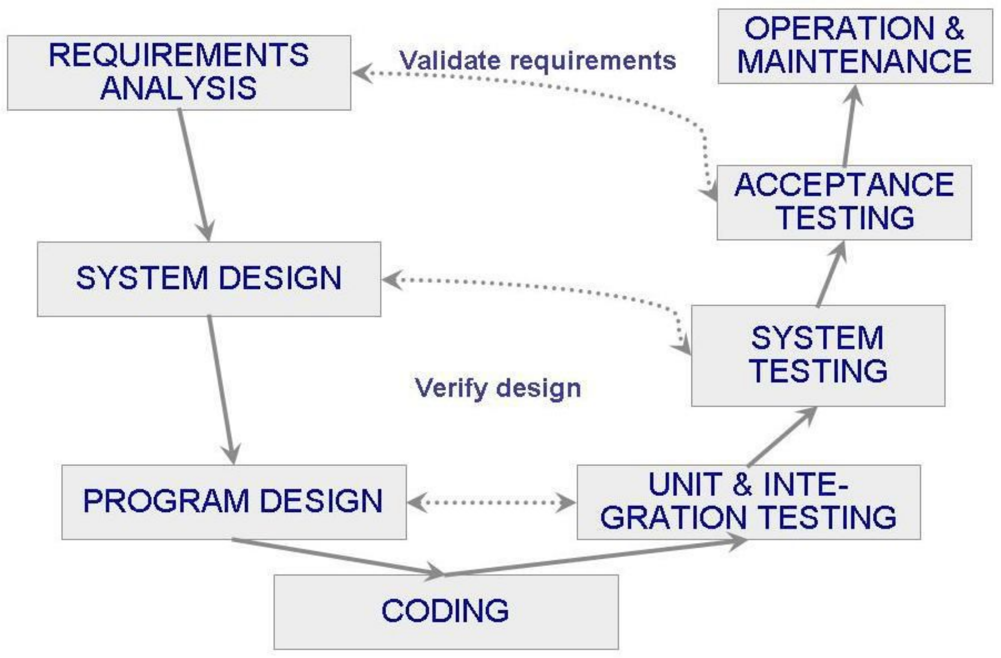
Process - a series of ordered tasks involving activities, constraints, and resources, that produce an intended output of some kind.

Why do we model a process?
To find inconsistencies, to form a common understanding, to meet functional requirements.

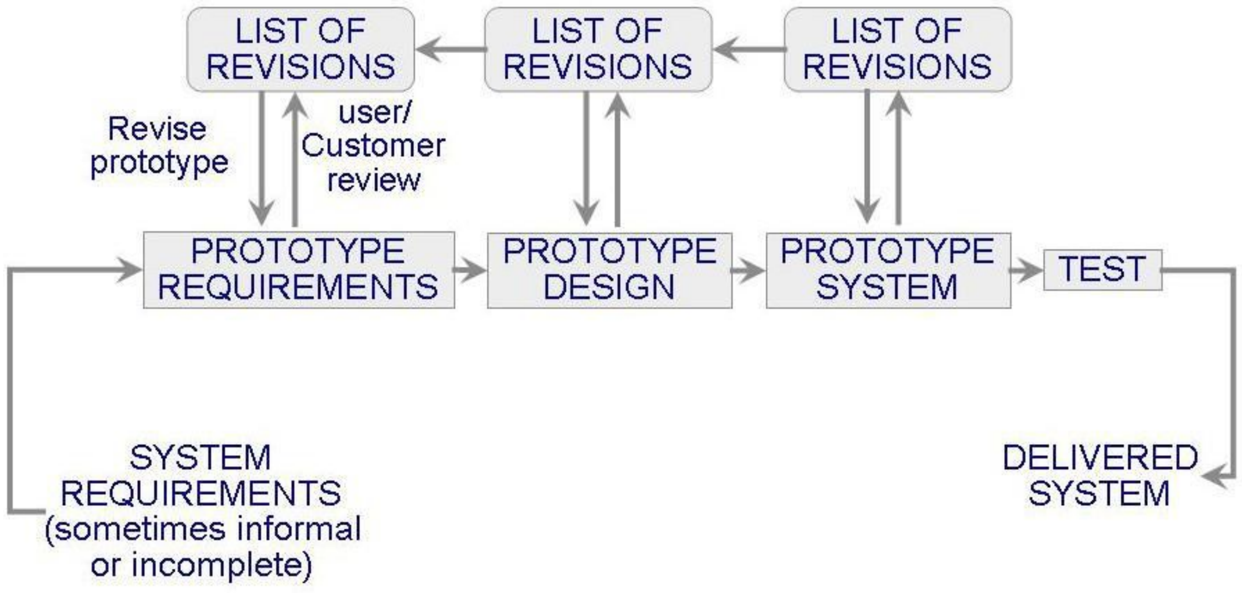
- **Waterfall model -** Well understood problems with little to no changes in requirements. High level view of process. Top Down Approach.



- **V model -** Variation of Waterfall model. Unit, integration, and acceptance testing test the procedural design, architectural design, and requirements.



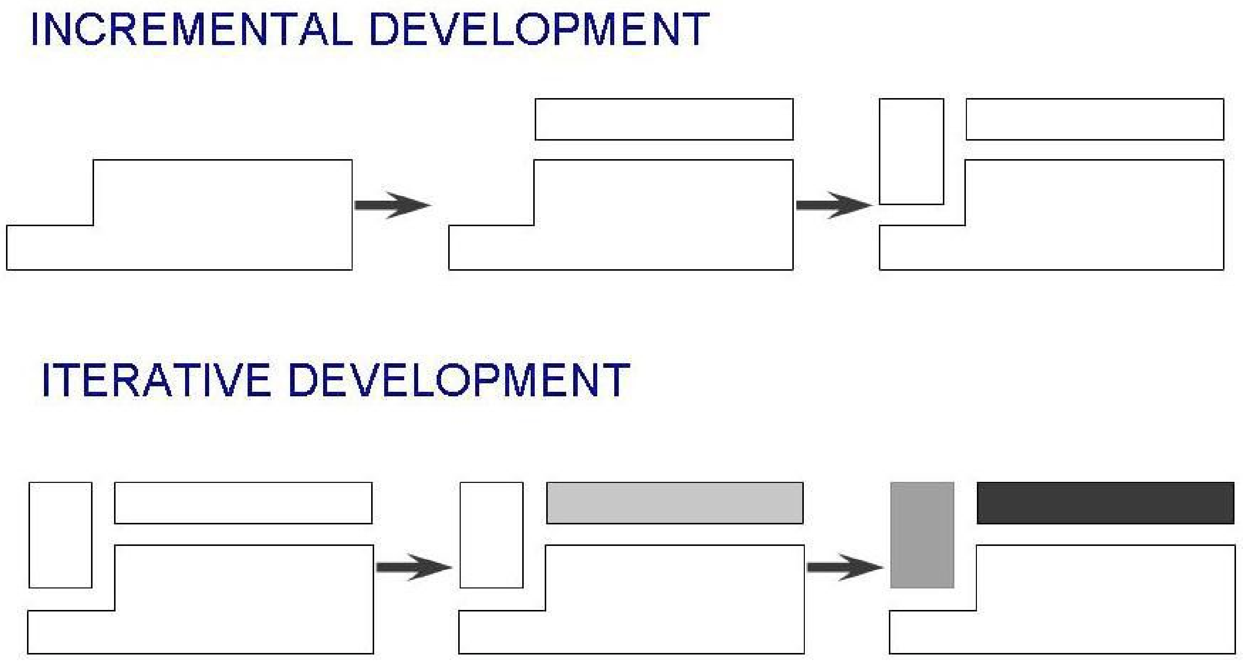
- **Prototyping model -** creating a smaller version of a software, repeated investigation of requirements and design.



- **Phased development: increments and iteration**

Incremental - starts with a small subsystem and adds functionality with each new release.

Iteration - starts with complete subsystem and changes functionality of each subsystem with each new release.



- **Spiral model** - A loop for software development starting in the center and revisiting requirements.

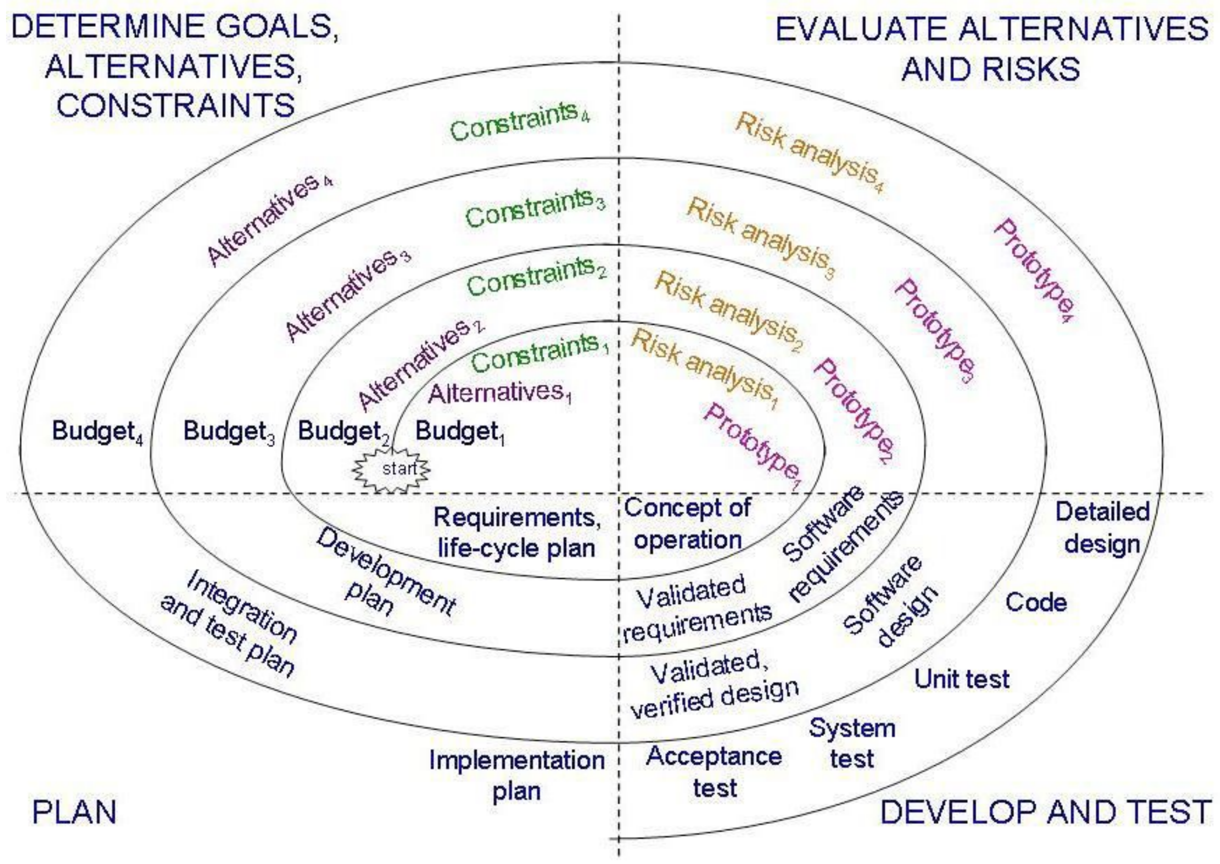


Figure 2.10 the spiral model.

- **Agile methods -** flexibly producing software quickly and capably. Values individuals over tools, prefer to invest time in working software than documentation, customer collaboration rather than contract negotiation, responds to change. Uses Scrum

Communication, Simplicity, Courage, and Feedback

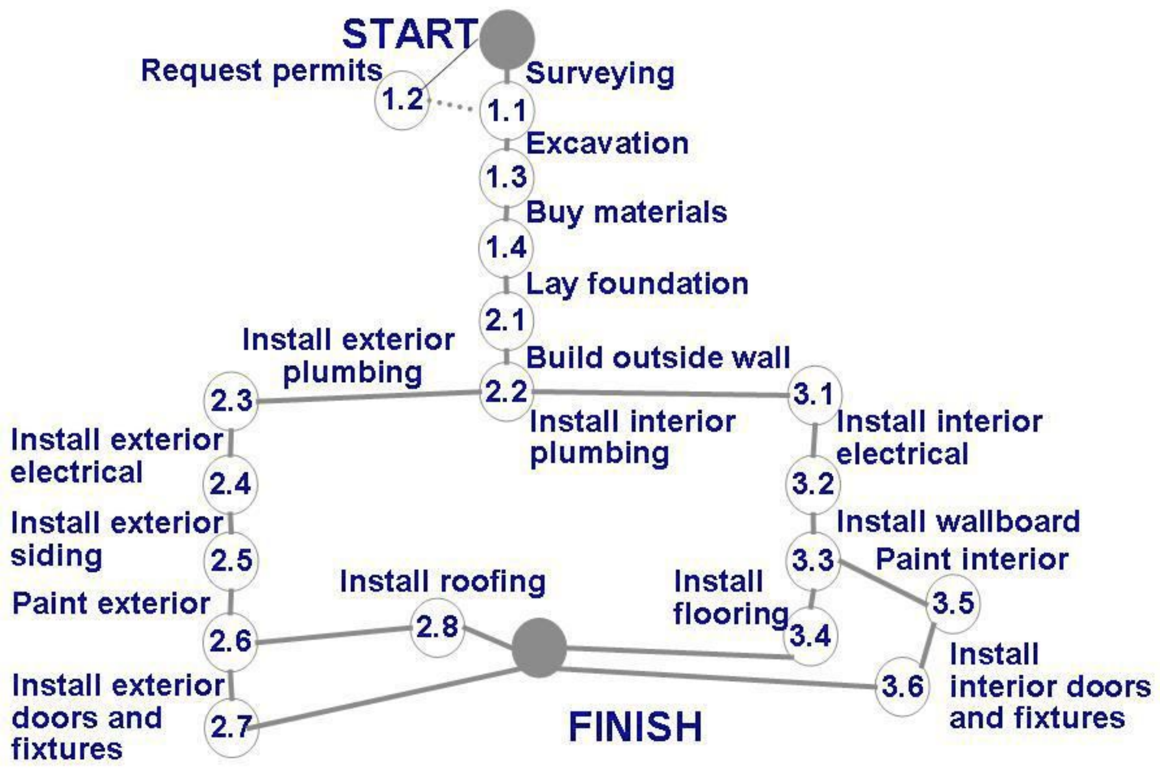
Seven types of elements in a process:
–Activity
–Sequence
–Process model
–Resource
–Control
–Policy
–Organization

Chapter 3

Project Deliverables:
–Documents
–Demonstrations of function
–Demonstrations of subsystems
–Demonstrations of accuracy
–Demonstrations of reliability, performance or security

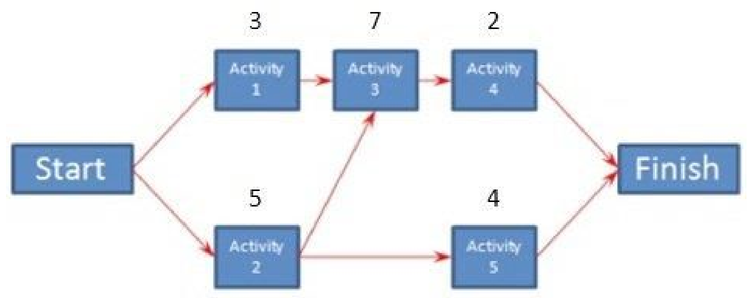
- **Activity -** takes place over a period of time; has objective criteria to determine completion
- **Milestone** - Completion of a major activity at a particular point in time.
- **Precursor** - an event that must happen first in order for an activity to start.
- **Duration -** Length of time needed to complete an activity.

Activity Graph

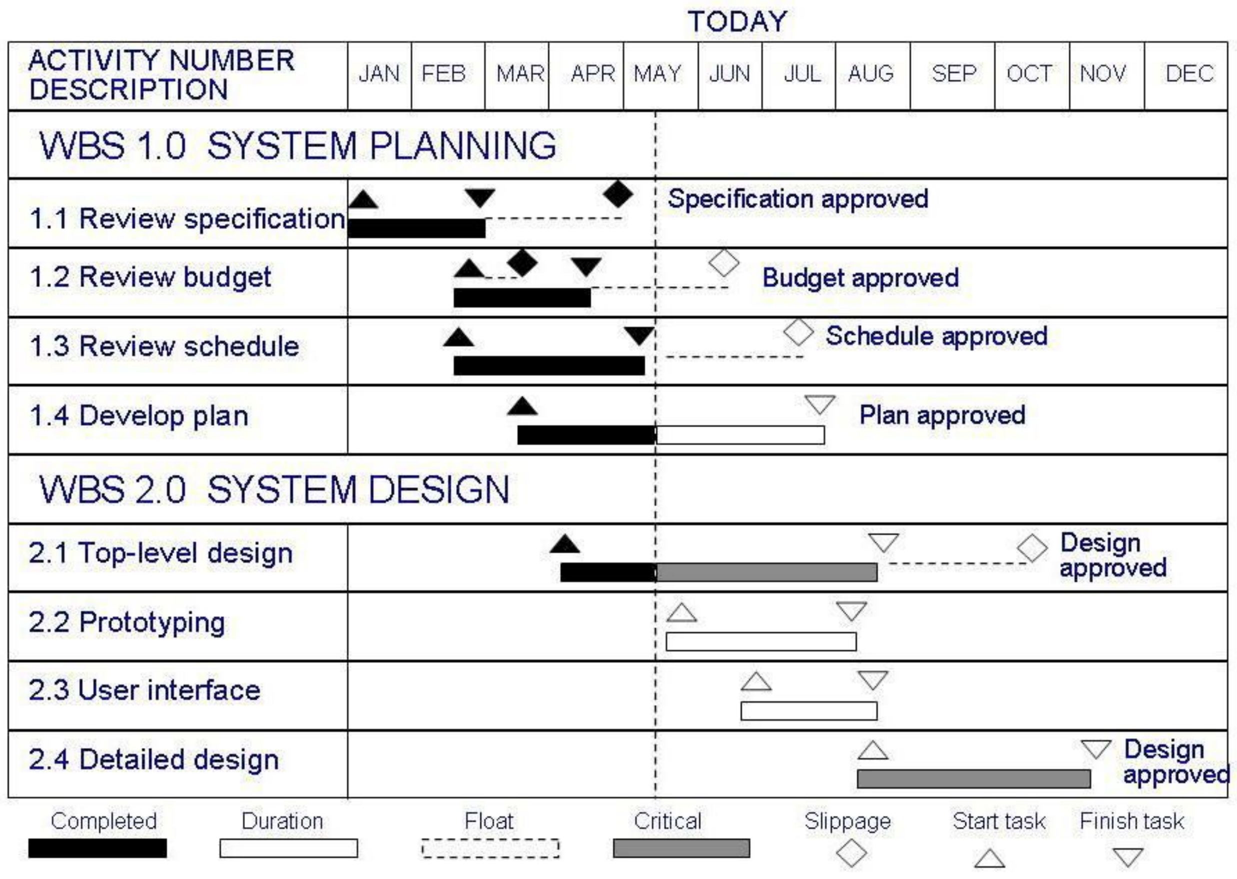


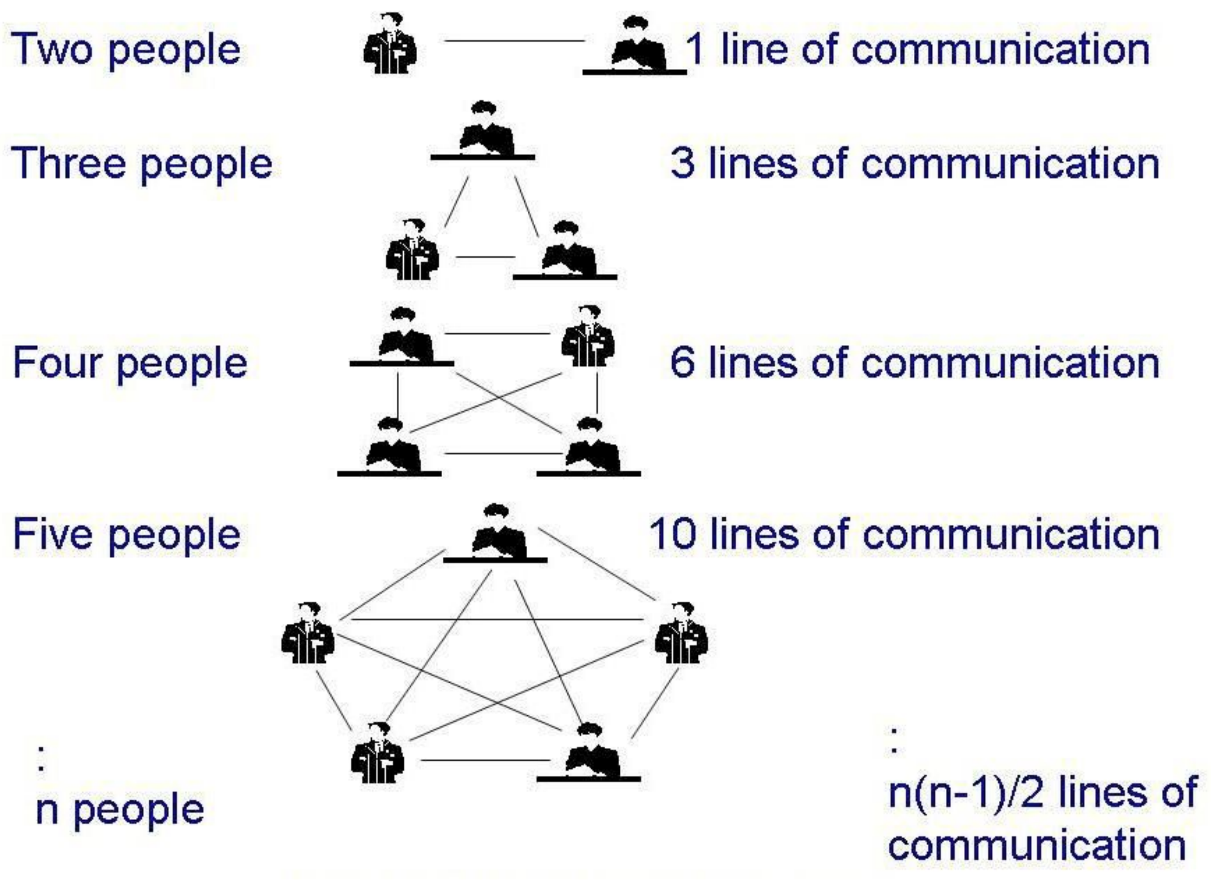
- **Real time:** estimated amount of time required for the activity to be completed
- **Available time:** amount of time available in the schedule for the activity's completion
- **Slack time:** The difference between the real time and the available time for an activity. How much time can be wasted.

How to find Critical Path - Start at the end, and select the closest item on the list. Continue until the beginning is reached. The definition of critical path is doing the most important things on the first - with no slack time.



Gantt Chart





Risk - An unwanted event that has negative consequences.
Risk impact - the loss associated with the event
Risk probability - the likelihood that the event will occur

1. Avoid Risk
2. Transfer Risk
3. Assume Risk

Life cycle objectives

Objectives: Why is the system being developed?
Milestones and schedules: What will be done by when?
Responsibilities: Who is responsible for a function?
Approach: How will the job be done, technically and managerially?
Resources: How much of each resource is needed?
Feasibility: Can this be done?

Chapter 4

Requirement - an expression of desired behavior.
A requirement deals with objects and their state, and the functions associated with them. Requirements focus on customer needs.

Requirements Analyst captures the requirements and often documents them.

Desirables:

- Loosely Coupled - not many dependencies between different modules (classes) of a program.
- Tightly High Cohesion - Focusing only on the specific purpose of one class - every function in that class pertains to it. Internal to modules. (Example: In a program for calculating pi, you don't want to count cows)

Tight requirements -> Waterfall or V Model
Unclear requirements -> Agile

Software inconsistencies are highlighted but not addressed until there is sufficient information to make an informed decision.

Eliciting Requirements

- Interviewing stakeholders
- Reviewing available documentations
- Observing the current system (if one exists)
- Apprenticing with users to learn about user's task in more details
- Interviewing user or stakeholders in groups

- Using domain specific strategies, such as Joint Application Design
- Brainstorming

Functional Requirement - describes required behavior of a system in terms of required activities.

Functional Requirements Defined

According to ReQtest a functional requirement simply tells a system to do something specific. On its own, this definition does not really suggest much, but let's take a real life example to provide more clarity. Functional requirements essentially describe what the product must do or the steps it is going to have to take to perform that action. For instance, every time a customer places an order online, a confirmation email is sent to them. As a general rule, functional requirements are usually written out as "shall" statements.

Types of functional requirements:

- Interface
- Business Requirements
- Regulatory/Compliance
- Security

Non-functional Requirement - describes some quality characteristic that the software must possess.

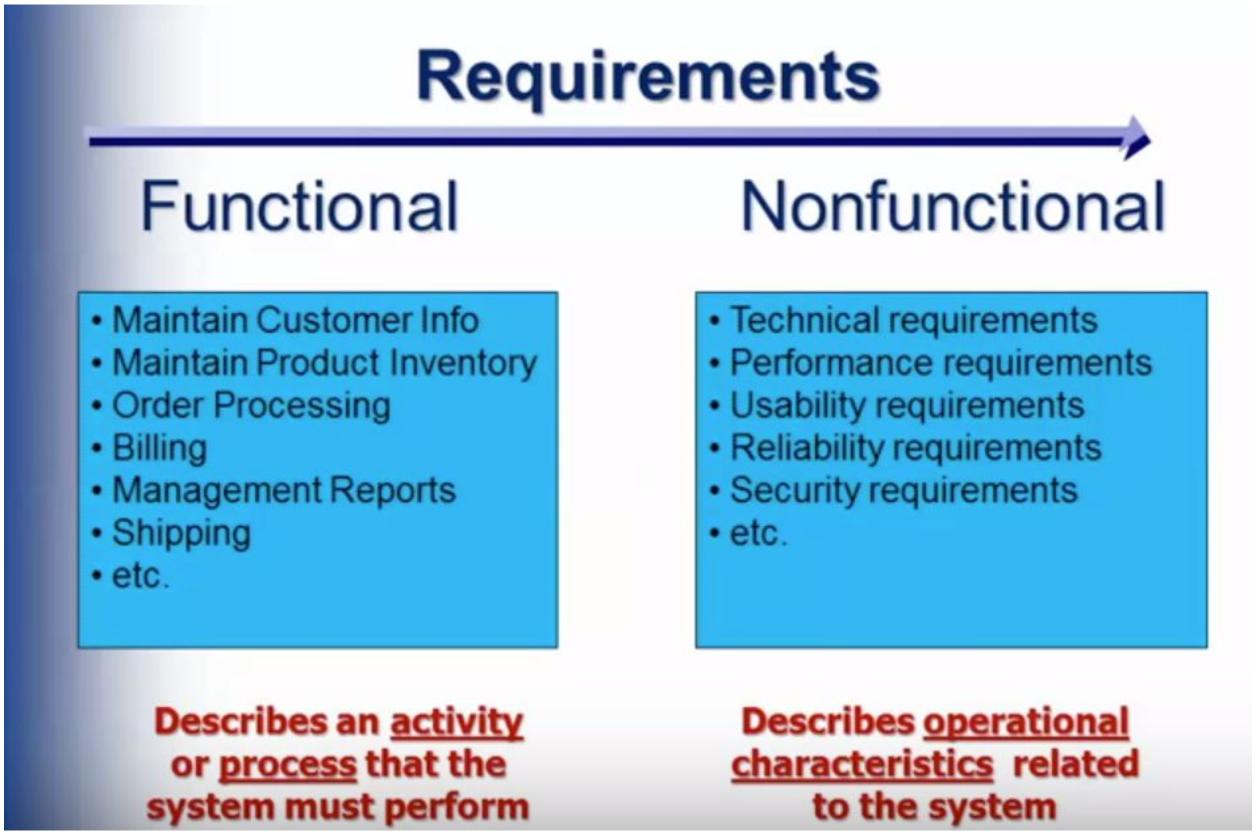
What is a Non Functional Requirement?

Now that we have a better understanding of what a functional requirement is, let's explore non-functional requirements. According to Stack Overflow, a non functional requirement requirement elaborates a performance characteristic of a particular system. Using the florist and the email example we looked at earlier, a non functional requirement would be the amount of time before an email confirmation is sent out to a customer after he places his order: In other words, *"The customer shall receive an email confirmation 5 minutes after his order has been placed online."*

Non functional requirements fall into many of these categories:

- Accessibility
- Capacity, current and forecast
- Documentation
- Efficiency
- Effectiveness
- Quality
- Reliability
- Response time

Functional Requirements are requirements that describe what they system must DO or what the output must BE. Things that are subjective, like timing, are non functional requirements.



Requirements

Functional

- Maintain Customer Info
- Maintain Product Inventory
- Order Processing
- Billing
- Management Reports
- Shipping
- etc.

Describes an activity or process that the system must perform

Nonfunctional

- Technical requirements
- Performance requirements
- Usability requirements
- Reliability requirements
- Security requirements
- etc.

Describes operational characteristics related to the system

Design Constraint - a design limitation such as choice of platform or interface components.
Process Constraint - a restriction on the techniques or resources to be used.
Requirements specification: restates the requirements as a specification of how the proposed system shall behave (aimed at technical people)

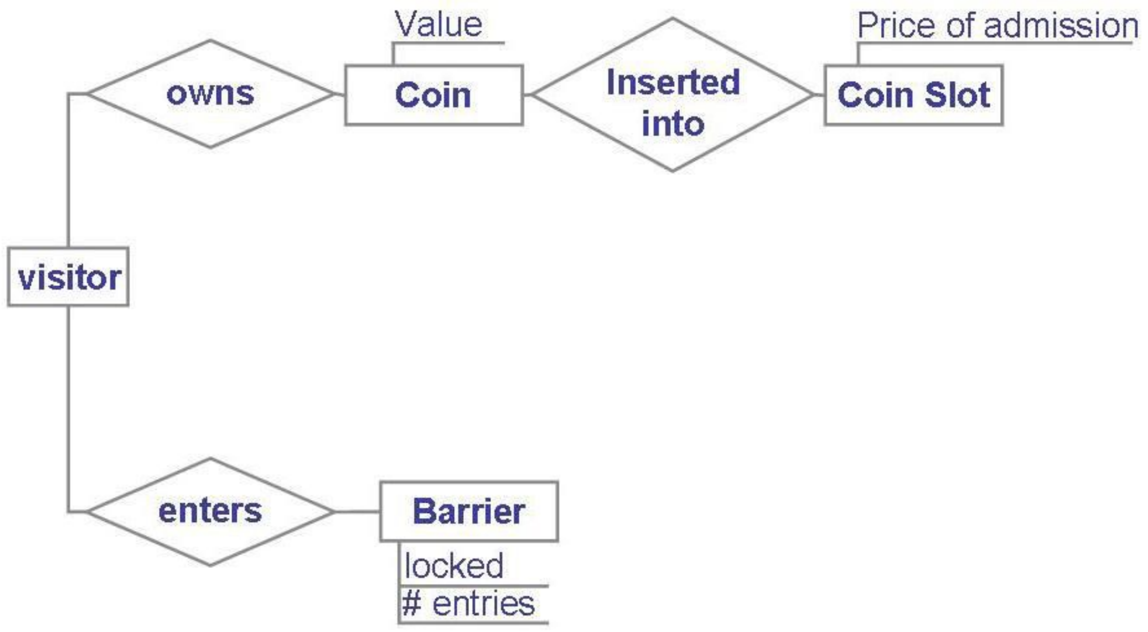
Requirements Definition + Technical Words = Requirements Specification

Requirements must be:

- Correct
- Consistent

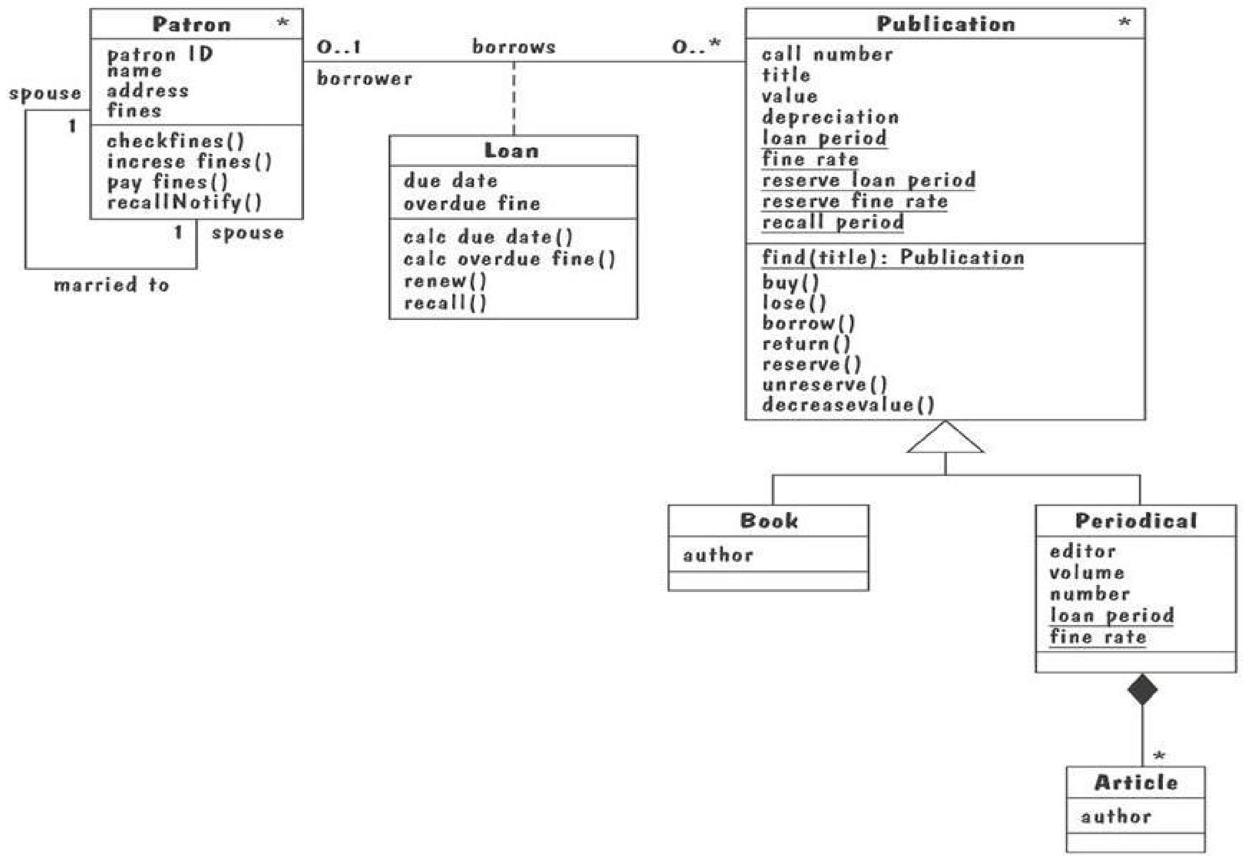
- Unambiguous
- Complete
- Feasible
- Relevant
- Testable
- Traceable

Entity-Relationship Diagrams:
Has three core constructs
–An entity: depicted as a rectangle, represents a collection of real-world objects that have common properties and behaviors
–A relationship: depicted as an edge between two entities, with diamond in the middle of the edge specifying the type of relationship
–An attribute: an annotation on an entity that describes data or properties associated with the entity



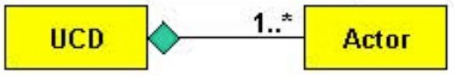
UML Diagrams

Uses methods to relate objects or entities.



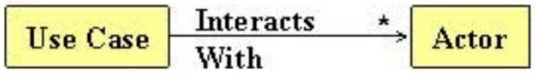
White Diamond - **Aggregation** - “has a”
Black Diamond - **Composition** - “uses a” or “contains”
Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). ... **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child).

to relate objects and classes.

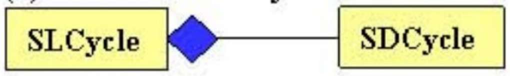


Composition

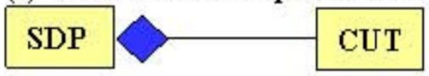
(a) Use Case to Actor Association



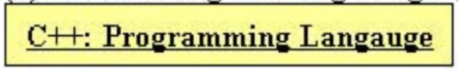
(b) Software Life Cycle to Software Development Cycle Composition



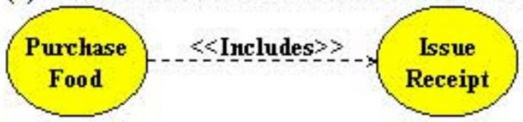
(c) Software Development Process (SDP) to Code & Unit Test (CUT) Composition



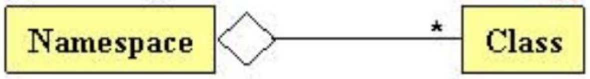
(d) C++ to Programming Language Is-An-Instance-Of



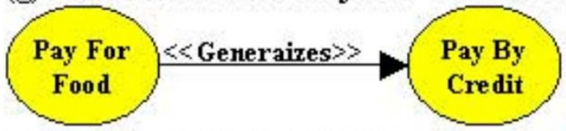
(e) In a restaurant: Purchase Food to Issue Receipt Includes



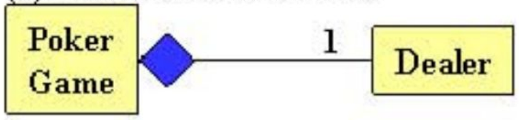
(f) Namespace to Class Definition Aggregation (-1 for wrong multiplicity)



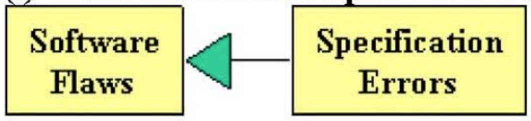
(g) In a restaurant: Pay-For-Food to Pay-by-Credit Generalizes (Specializes)



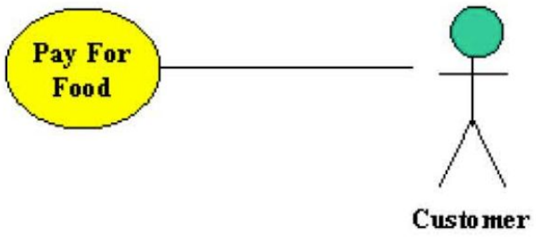
(h) Poker Game to Dealer Composition



(i) Software Flaws to Specification Errors Is-A-Subtype-Of

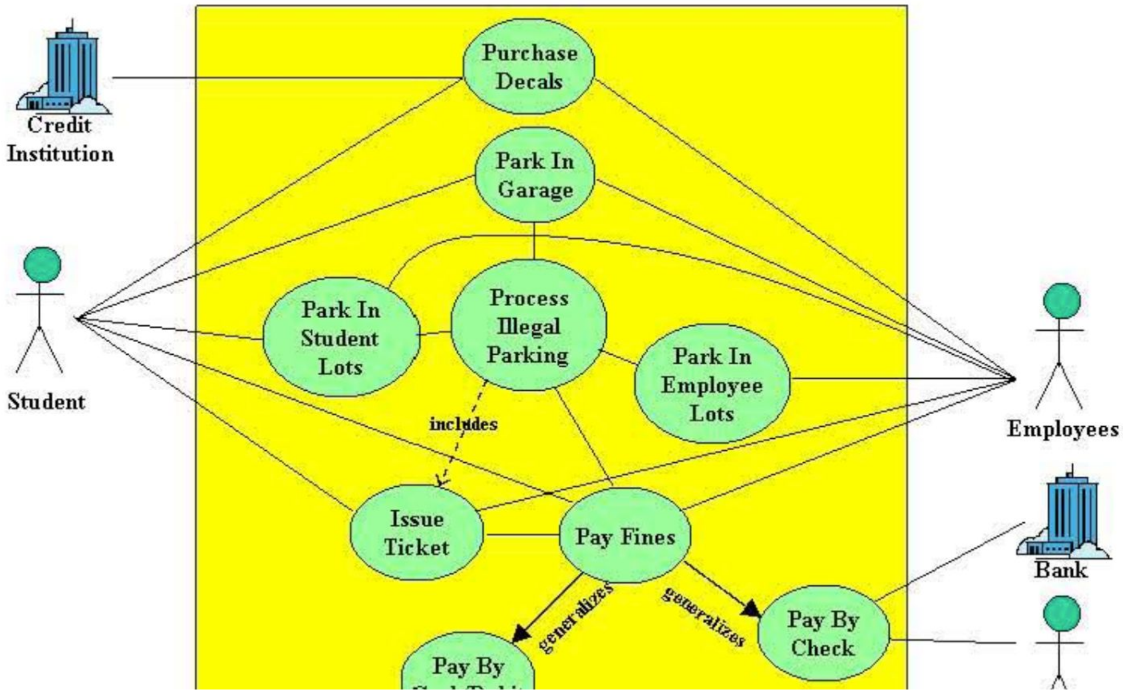


(j) In a restaurant: Pay-For-Food to Customer Associates (-1 for no stick figure)



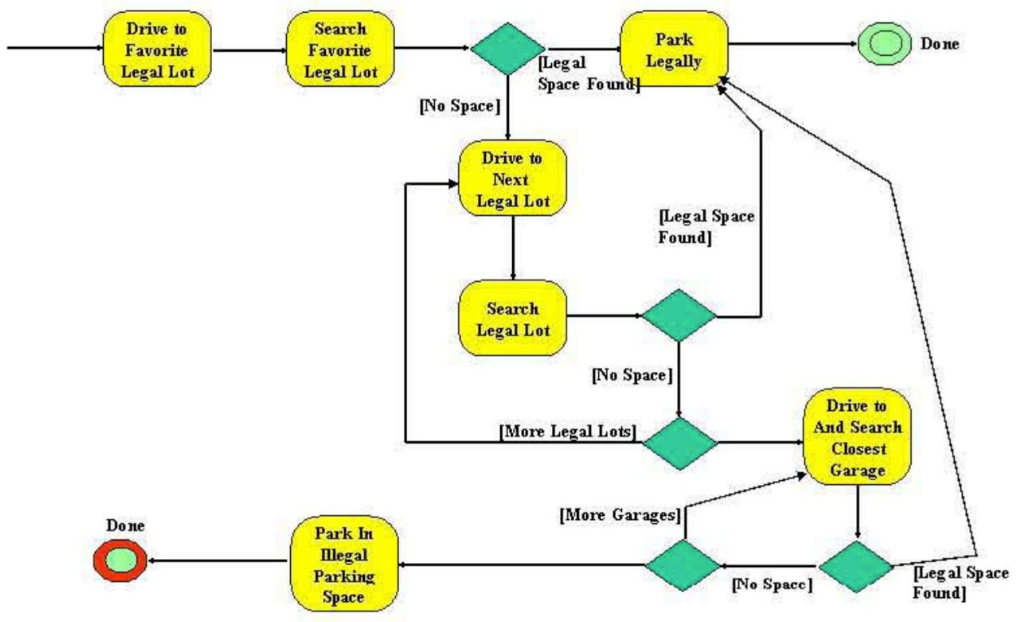
#6 (a) Use Case Diagram **POINTS = 12**

The Actors are important (four or five) 4 pts
The Use Cases are important (nine) 4pts
The relations are important (assoc., includes or uses, generalize or specialize) 4pts
Deduct for important omissions & wrong notation for relations.



Activity Diagram

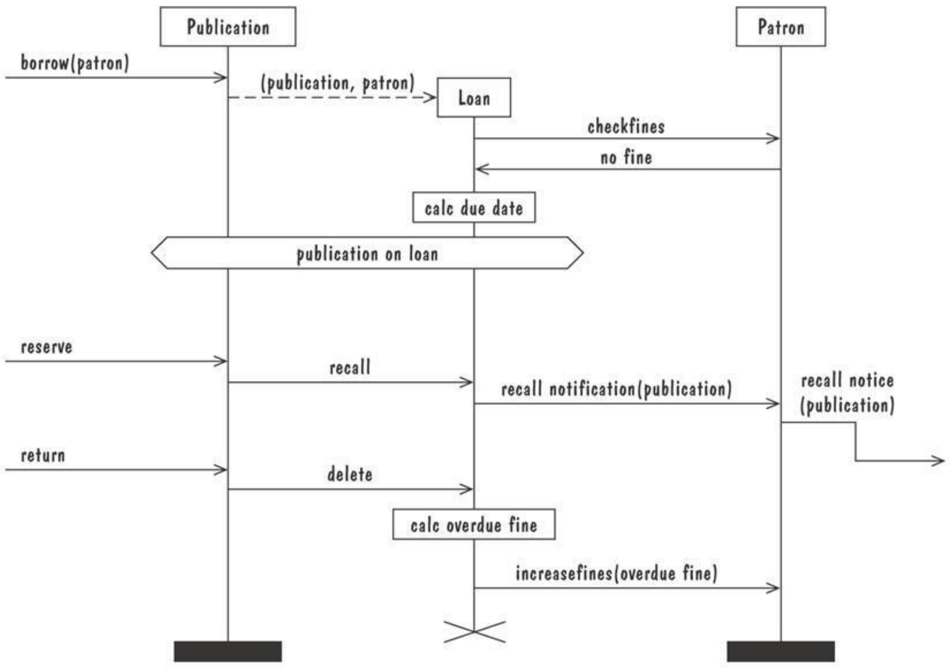
Proper use of notation – especially the conditional flow symbols – verb phrases (no nouns) (4 pts) Logical correctness of flow and level of detail (4 pts)



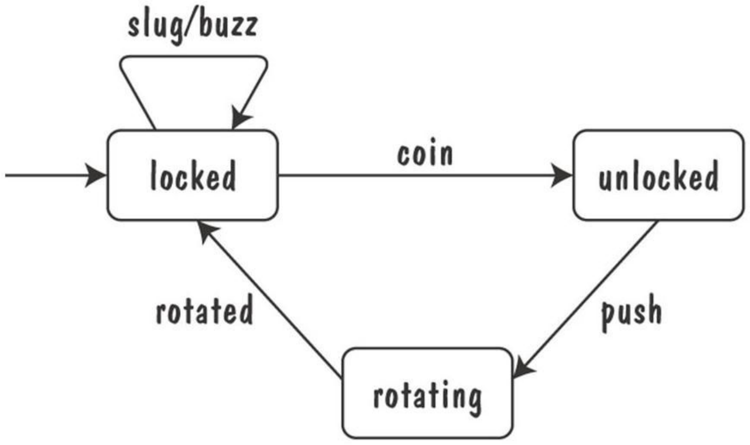
Sequence Charts are used in event traces using an enhanced event-trace notation. It facilitates creating and destroying entities, specifies actions and times, and composes traces.

Sequence Chart Notation:

- Vertical line - participating entity
- Message - arrow from sending entity to receiving entity
- Action - Labeled rectangles in execution line
- Conditions - Possible states in the entity’s evolution



State Machines are a graphical description of all dialog between a system and its environment represented by nodes and edges.
Node: represents a stable set of conditions that exist between event occurrences.
Edge (transition) represents a change in behavior or condition due to the occurence of an event.



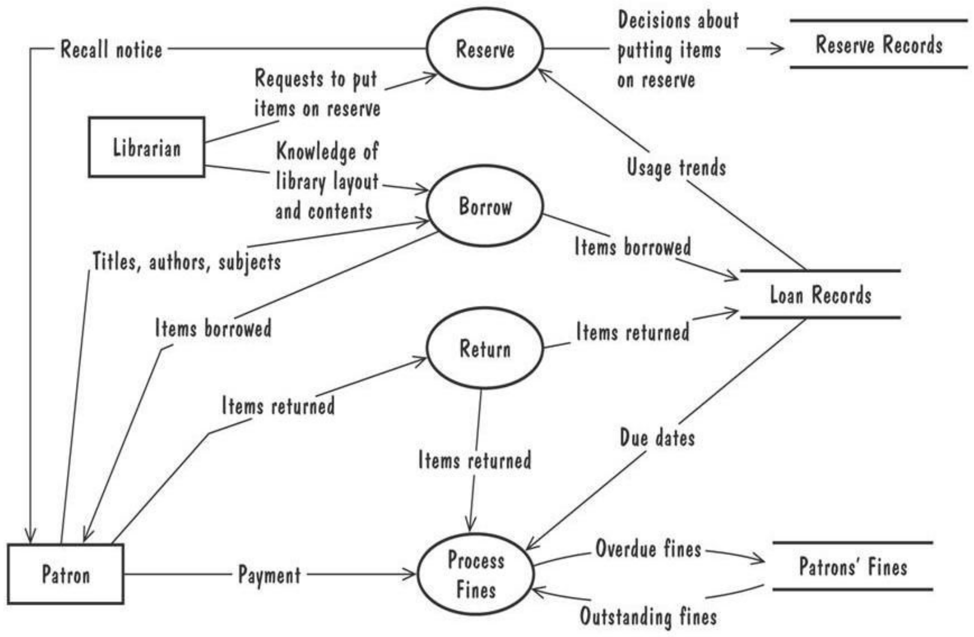
Data-Flow Diagrams (DFD) model the functionality and flow of data from one function to another.

Bubble - represents a process

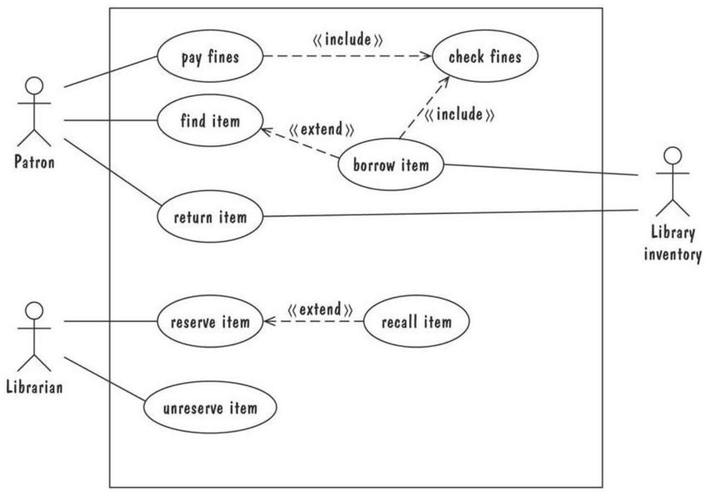
Arrow - represents data flow

Data Store - formal repository or database of information

Actors - entities that provide input data or receive output result.



DFD (Data Flow Diagram) Use Cases are used to specify the user views of the essential system behavior.



Include means the process has to have it, **extend** means the process might use it.

Formal methods (approach) are mathematically based specification and design techniques. Model requirements or software behavior as a collection of math functions or relations.

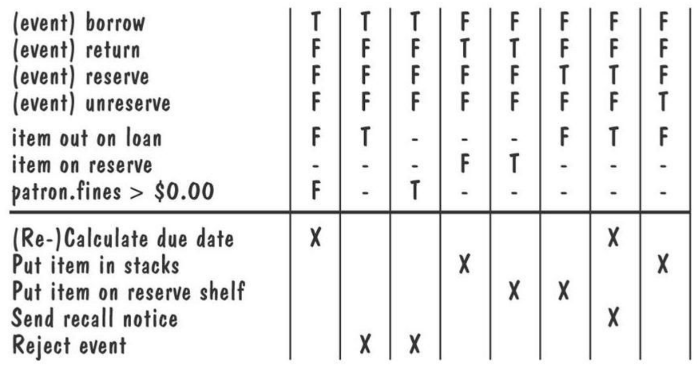
Functions - specify the state of the system's execution, and output

Relation - used whenever an input value maps more than one output value.

$$NetState(s,e)= \begin{cases} unlocked & s=locked \text{ AND } e=coin \\ rotating & s=unlocked \text{ AND } e=push \\ locked & (s=rotating \text{ AND } e=rotated) \\ & \text{OR } (s=locked \text{ AND } e=slug) \end{cases}$$

$$Output(s,e) = \begin{cases} buzz & s=locked \text{ AND } e=slug \\ \text{<none>} & Otherwise \end{cases}$$

Decision tables table representation of a functional specification that maps event and conditions to appropriate responses or actions.



(event) borrow	T	T	T	F	F	F	F	F
(event) return	F	F	F	T	T	F	F	F
(event) reserve	F	F	F	F	F	T	T	F
(event) unreserve	F	F	F	F	F	F	F	T
item out on loan	F	T	-	-	-	F	T	F
item on reserve	-	-	-	F	T	-	-	-
patron.fines > $0.00	F	-	T	-	-	-	-	-
(Re-)Calculate due date	X						X	
Put item in stacks				X				X
Put item on reserve shelf					X	X		
Send recall notice							X	
Reject event		X	X					

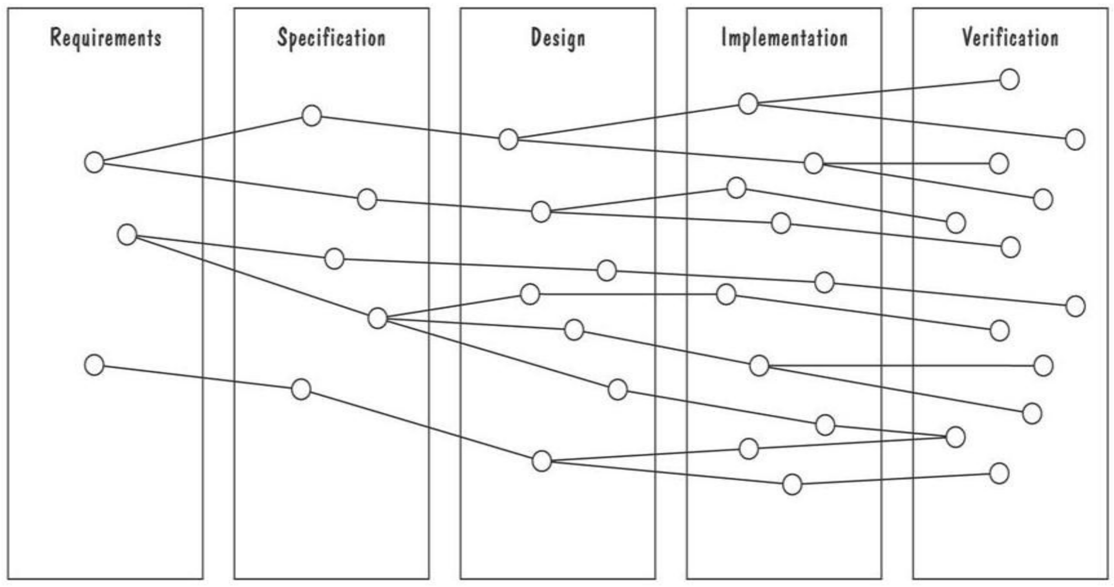
Prototyping is used to elicit the details of a proposed system and solicit feedback from users about the system.
Prototyping approaches:

Throwaway approach - a “quick-and-dirty” prototype not meant to be delivered but used to learn about the problem or solution.

Evolutionary approach - used to help answer questions and eventually build into the final product.

Prototyping vs Modeling
Prototyping - good for answering questions about the user interfaces
Modeling - quickly answer questions about constraints on the ordering of events.

Process management is a set of procedures that track the system requirements, design modules, code implementation of design, the tests that verify functionality, and documents that describe the system.

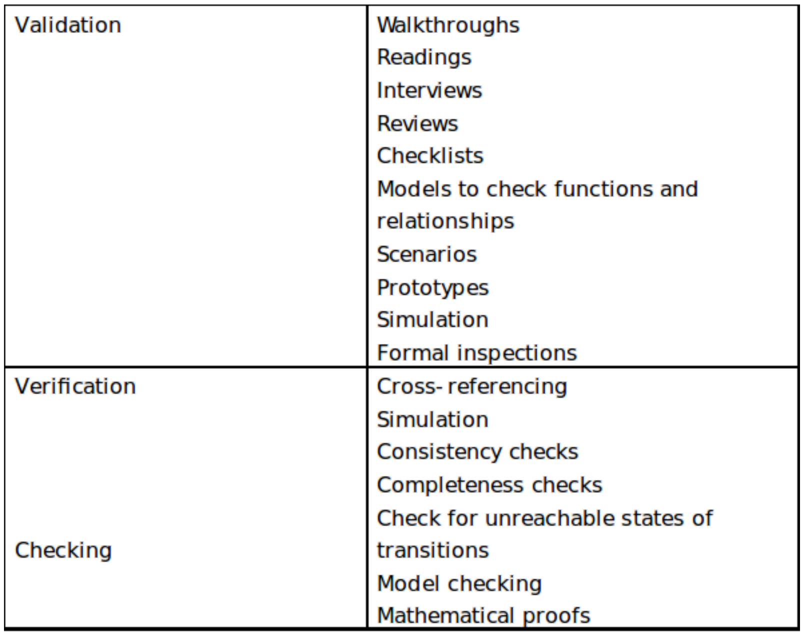


Validation checks that requirement definition accurately reflects the customer's needs.
Is it what the customer wants?

Verification check that one document or artifact conforms to another.
Is it correct?

Verification ensures that the system is built right while validation ensures the right system is built.

Validation and verification techniques:



Validation	Walkthroughs
	Readings
	Interviews
	Reviews
	Checklists
	Models to check functions and relationships
	Scenarios
	Prototypes
	Simulation
	Formal inspections
Verification	Cross- referencing
	Simulation
	Consistency checks
	Completeness checks
Checking	Check for unreachable states of transitions
	Model checking
	Mathematical proofs

Chapter 5 - Designing The Architecture

Design - creative process of figuring out how to implement all of a system's defined requirements. Software design is the process of implementing software solutions to one or more sets of problems based on a set of requirements.

- Early design decisions address the system's architecture
- Later design decisions address how to implement the individual units

Design Methods:

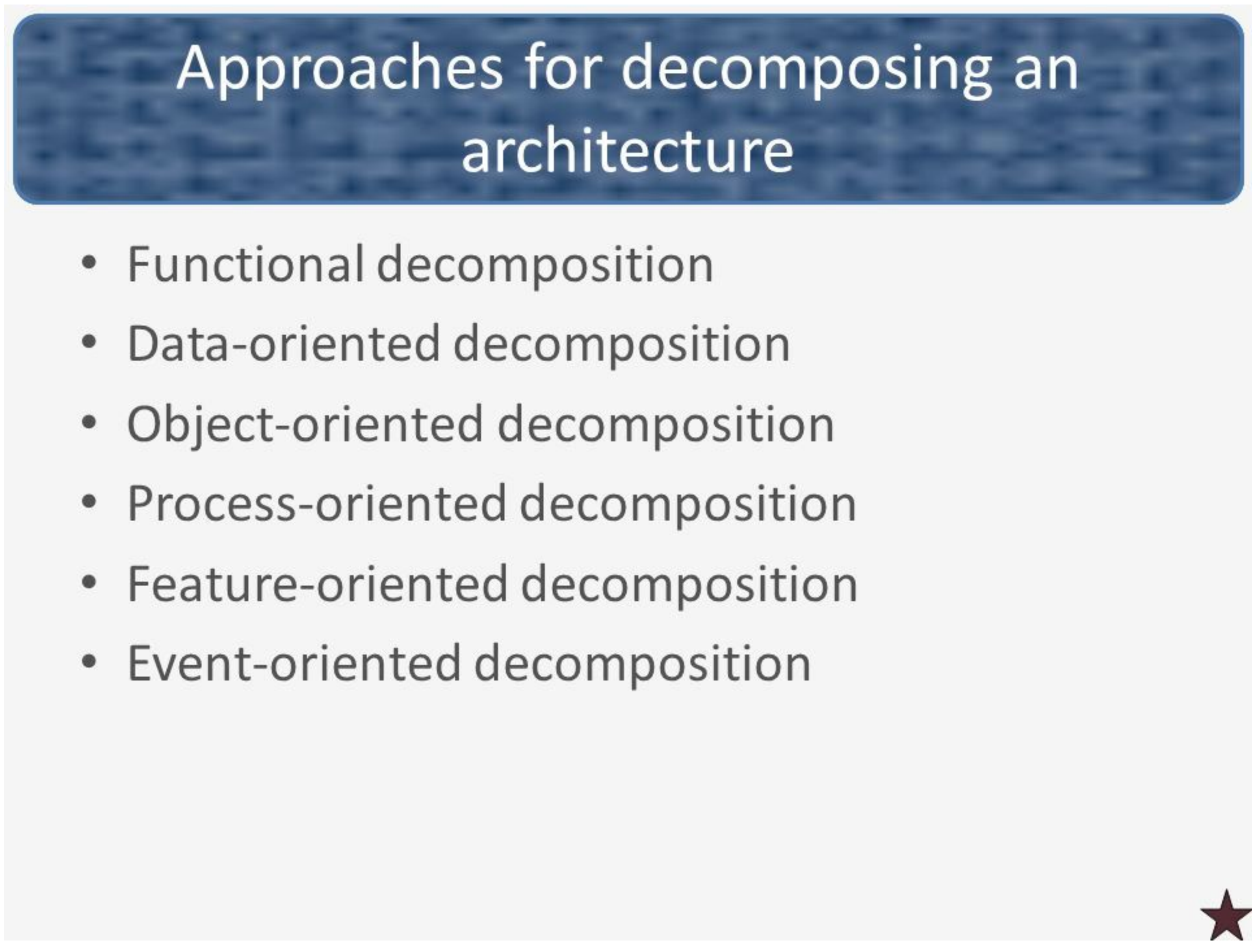
- Functional decomposition
- Feature-oriented decomposition
- Data-oriented decomposition
- Process-oriented decomposition
- Event-oriented decomposition
- Object-oriented design

Routine Design - process of reusing or adapting solutions from similar problems. (DRY vs WET: don't repeat yourself, vs write everything twice) (cloning / reference models) Often, a reference model doesn't exist for the problem.

Architectural Styles - generic solutions to accomplish desired functionality that have been proven over time.

- Tools for understanding options and evaluating chosen architecture:
    - ***Design patterns***: generic solutions for making lower-level design decisions
    - ***Design convention or idiom***: collection of design decisions and advice that, taken together, promotes certain design qualities
    - ***Innovative design:*** characterized by irregular bursts of progress that occur as we have flashes of insight, a novel solution
    - ***Design principle***: descriptive (not prescriptive) characteristics of good design

- **Agile Architecture**:
    - Based on:
        - Valuing individuals and interactions over processes and tools
        - Valuing working software over comprehensive documentation
        - Valuing customer collaboration over contract negotiation
        - Valuing response to change over following plans
    - Possible problems:
        - Complexity and change must be carefully managed
        - Programmers encouraged to write code as models are being produced
        - The need for constant refactoring

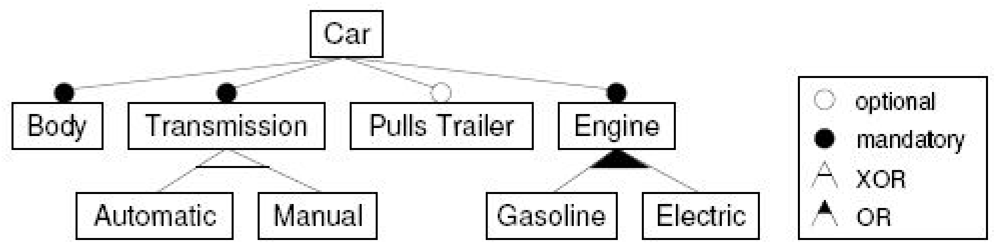


Approaches for decomposing an architecture

- Functional decomposition
- Data-oriented decomposition
- Object-oriented decomposition
- Process-oriented decomposition
- Feature-oriented decomposition
- Event-oriented decomposition

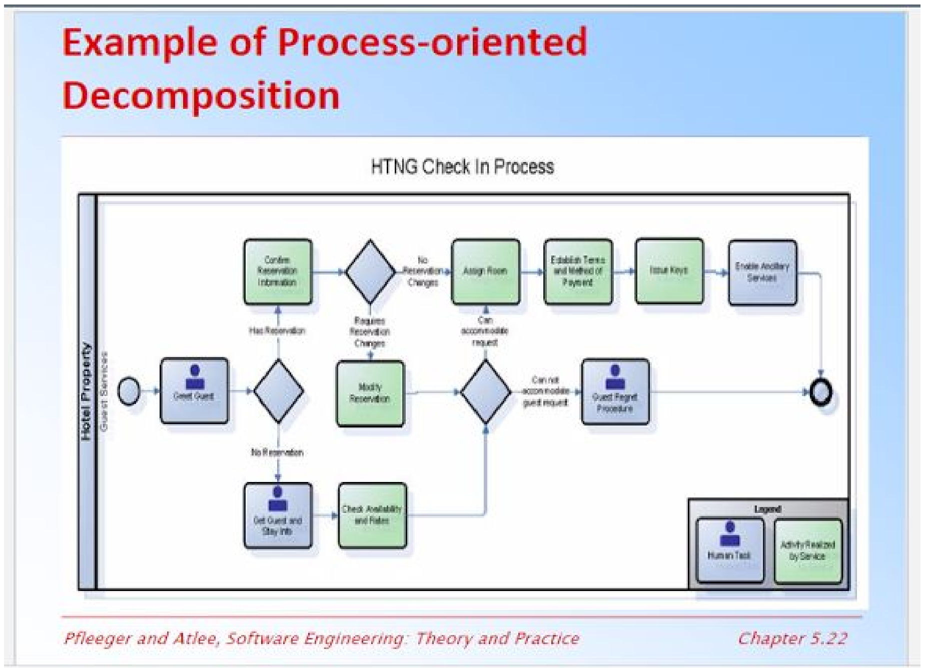
Functional Decomposition - partitions functions/requirements into modules, potentially dividing up the functionality and defining what modules call each other (how everything fits together). Example: A house is composed of walls, windows, and doors. Each of these components works together for the function of protection.

Feature-oriented Decomposition - assigns features to modules, and attempts to describe the system as a set of services and collections of features that interact with each other. Example: A car might have the attribute ‘red’. Other things can also be red but they aren’t cars.



Object-oriented Decomposition - assigns objects to modules and explain how the objects are related to one another, with each holding attributes and functions being actions/operations. Example: A dog class might include an animal class. Dog and animals are objects that can be related but don't have to be.

Process-oriented Decomposition - partitions the system into concurrent processes. High-level: identifies the system's main tasks, assigns tasks to runtime processes, explains how the tasks coordinate with each other. Low-level: describes the processes in more detail.



Example of Process-oriented Decomposition

Pfleeger and Atlee, Software Engineering: Theory and Practice *Chapter 5.22*

Event-oriented Decomposition - focuses on the events that the system must handle and assigns responsibility for events to different modules. High-level: catalogues system's expected input events. Low-level: decompose the system into states and describe how events trigger state transformation.

- ***But why an architecture anyway?***
    - To decompose the system into manageable units; an architecture is the keeper of the vision.

Modular - when each activity of the system is done by one thing and one thing only (DRY - make an architecture to support such). Example: If you have a class based on turning on a car, you don't want to add functionality to count the digits of pi.

Well-Defined - interface is accurate and depicts the externally available behavior with no extraneous information.

Dependencies View - shows dependencies between software modules, useful for planning and assessing software change impact. Example: Class Diagrams show the dependencies between classes and can help pinpoint which classes are most likely to be affected by changes.

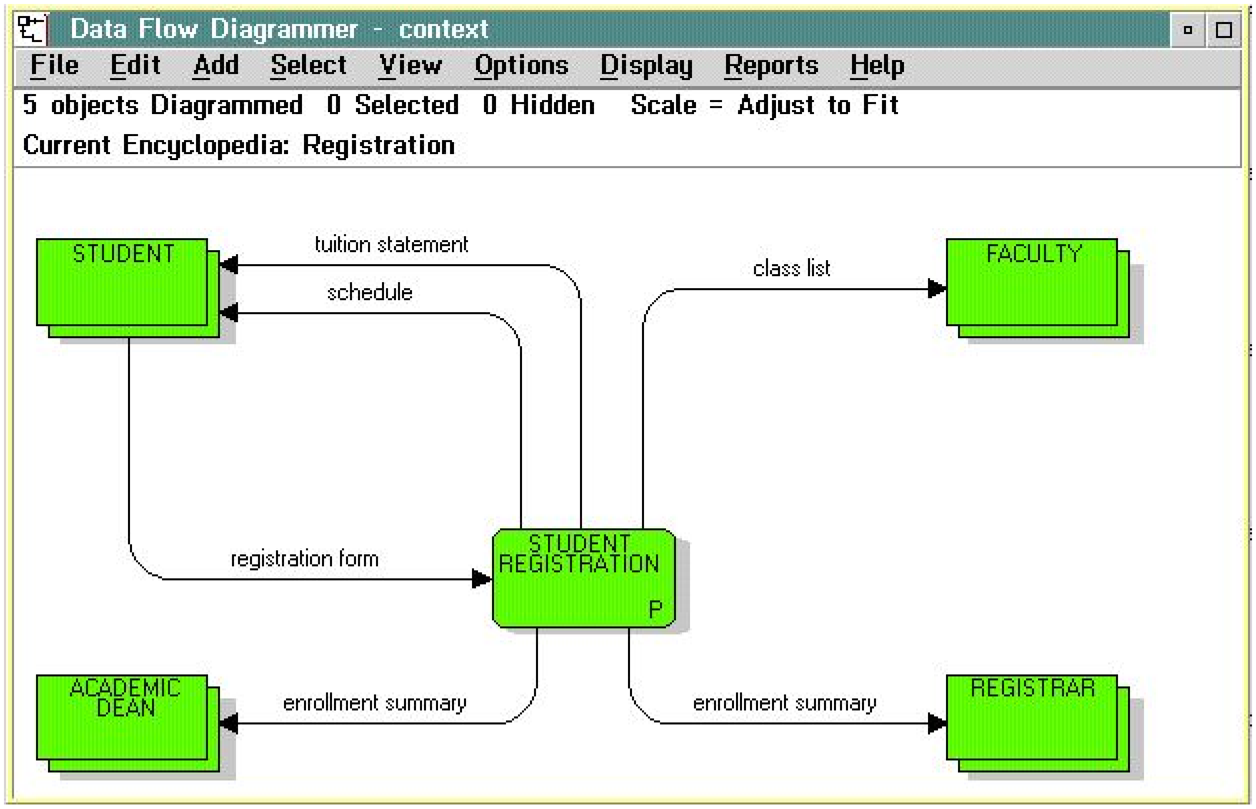
Decomposition View - portrays system as programmable units

Class Diagram - A type of UML Diagram used to express the static structure of an OO software design in terms of it's modular components and their relationships.

Generalization View - shows units that are related as generalizations or specializations of one another. (i.e a specific implementation of a Set: HashSet vs TreeSet) This is very similar to the layered model.

Work-Assignment View - decomposes the design into tasks that can be assigned to teams/groups, which helps to plan and allocate resources. Example: This document.

Data-oriented Decomposition - focuses on how data will be partitioned into modules, high-level design describes conceptual data structures.


Data Flow Diagrammer - context
File Edit Add Select View Options Display Reports Help
5 objects Diagrammed 0 Selected 0 Hidden Scale = Adjust to Fit
Current Encyclopedia: Registration
STUDENT
tuition statement
schedule
class list
FACULTY
registration form
STUDENT REGISTRATION
P
ACADEMIC DEAN
enrollment summary
enrollment summary
REGISTRAR

Quizlet vocab needs to be added starting here:

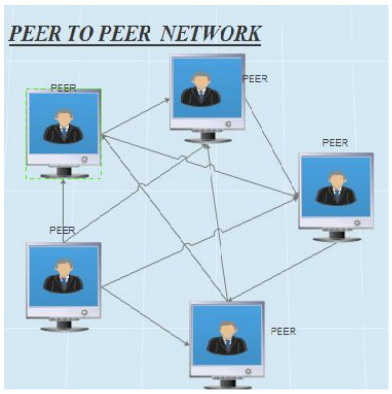
Architectural Styles & Strategies:

-Pipes-and-Filter (Hand off in the task relay race to the next task)
-Client-Server (Pretty self explanatory)
-Peer-to-Peer (Everyone is a server)
-Publish-Subscribe (Listen for when things happen)
-Repositories (Think data stores)

Pipes & Filter - streams of data (pipes) and transformations of the data from one process to another (filter). Example: in web, minifying file size is important, one way to build js/css resources is to use GulpJS to compile/minify/concatenate files in a pipe/filter manner, piping the content of the file, doing something and then doing the next thing with the previous output.
Use the *Pipes and Filters* architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (*Filters*) that are connected by channels (*Pipes*). Literally imagine a kitchen sink. Water flows from ground into the pipe, and it is "filtered" (changed).

Client-Server - server supplies content & services, client access them. Example: Uploading code to github. Github = server, you = client.

Peer-To-Peer (P2P) - each component can act as both a client and a server to other peer components. Example: Skype acts as a peer to peer because one person can lead the group, but they can also follow a group.



Publish-Subscribe - Components interact by broadcasting/reacting to events. Example: Fred the famous youtuber posts a new videos and millions of followers get notifications about said video.

Repositories - central data store & collection of components that can operate on the data store to retrieve and update information.

Combine Architectural Styles: they can work well together, and be combined in several ways.
Design is iterative: propose design decisions, assess, make changes, and propose more.
A cost-benefit analysis if often used for estimating and comparing cost and benefits of proposed changes.
People in review: system architect(s), program designers, tester, maintainer, recorder, moderator, and other interested devs not involved.

Quality Attributes

Modifiability - Design must be easy to change.
Performance - response time, throughput, and load. How quick it responds to requests, how many requests per minute, and how many users before performance starts to suffer.
Immunity - ability to thwart an attempted attack.
Resilience - ability to recover quickly and easily from an attack.
Usability - Ease in which a user is able to operate the system.

Fault - result of human error
Failure - departure from required behavior.

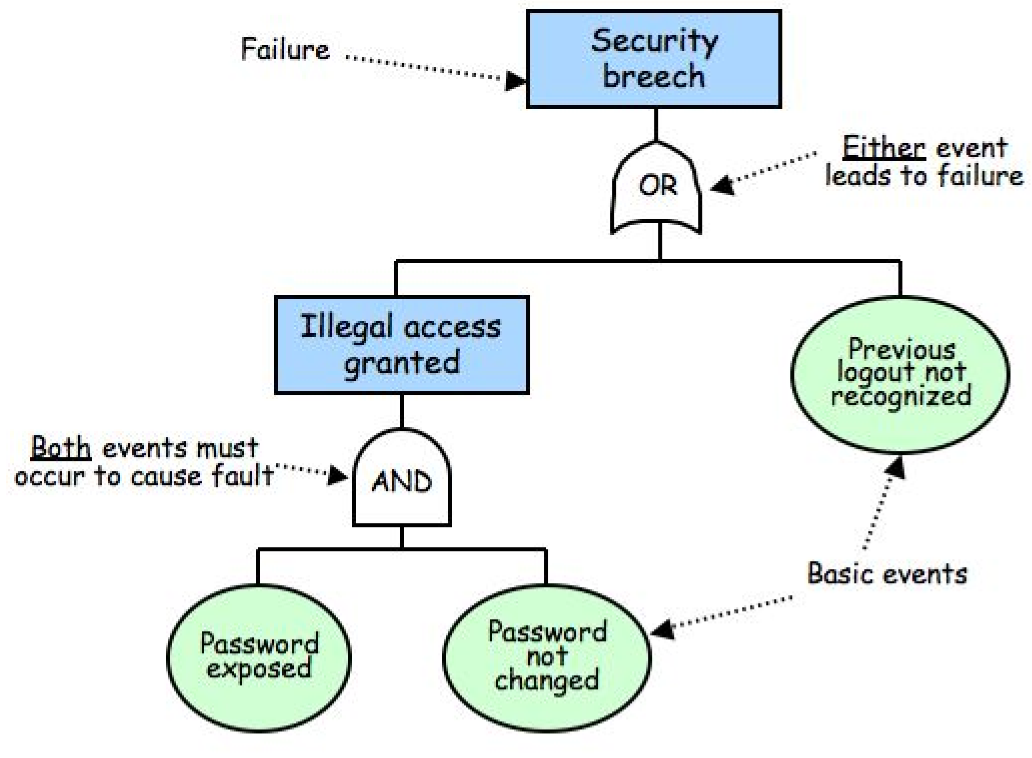
Fault-Tree Analysis - traces faults backwards through a design in order to determine which faults to correct/avoid/tolerate.
Data-flow Graph - depicts transfer of data from one process to another.
Control-Flow Graph - depicts possible transfer of control among modules.

Validation - making sure design satisfies requirements.
Verification - ensuring the design adheres to good design principles.

Active design review - critique/use the design doc. In ways developers would.
Passive Review Process - reading documentation and looking for problems.

Product Line - related products that reuse elements common to each other
Product Family - Related products in a product line, where they were developed together.
Core Asset Base - reusable code/assets.

Fault tree analysis?



Chapter 6

Refactoring - Objective: to simplify complicated solutions or to optimize the design. Code *refactoring* is the process of restructuring existing computer code—changing the factoring—without changing its external behavior.

Design Principles - Guidelines for decomposing a system's required functionality and behavior into modules

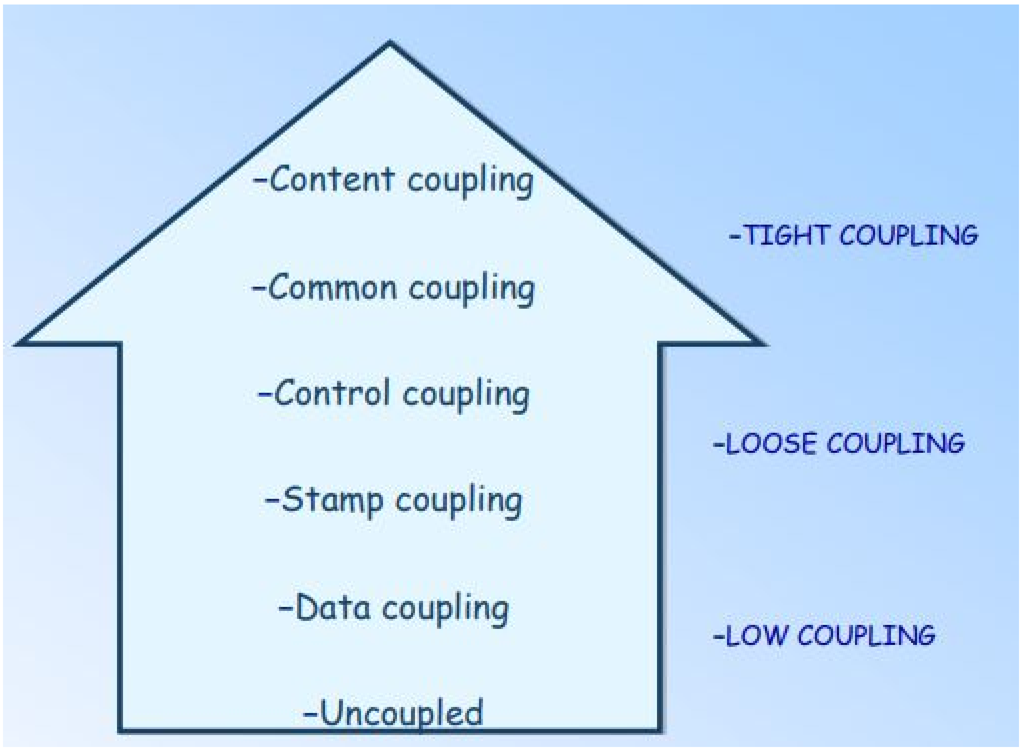
Principles identify the criteria for decomposing a system and deciding what information to provide (and what to conceal) in the resulting modules.

- Six Dominant Principles
  - Modularity
  - Interfaces
  - Information hiding
  - Incremental development
  - Abstraction
  - Generality

Modularity - The principle of keeping separate the various unrelated aspects of a system, so that each aspect can be studied in isolation (also called separation of concerns). Think of this like a large cardboard box that contains many smaller cardboard boxes.

Coupling and Cohesion are the two concepts that measure module independence to determine how well a design separates concerns.

Coupling (Low coupling is good, high coupling is bad)



Two modules are **Tightly Coupled** when they depend a great deal on each other.

Loosely Coupled modules have some dependence, but their interconnections are weak

Uncoupled - Modules have no interconnections at all; They are completely unrelated.

Stamp Coupling - Occurs when complex data structures are passed between modules

Data Coupling - When data values, and not structured data, are passed.

Modules can be dependent from each other in these ways:

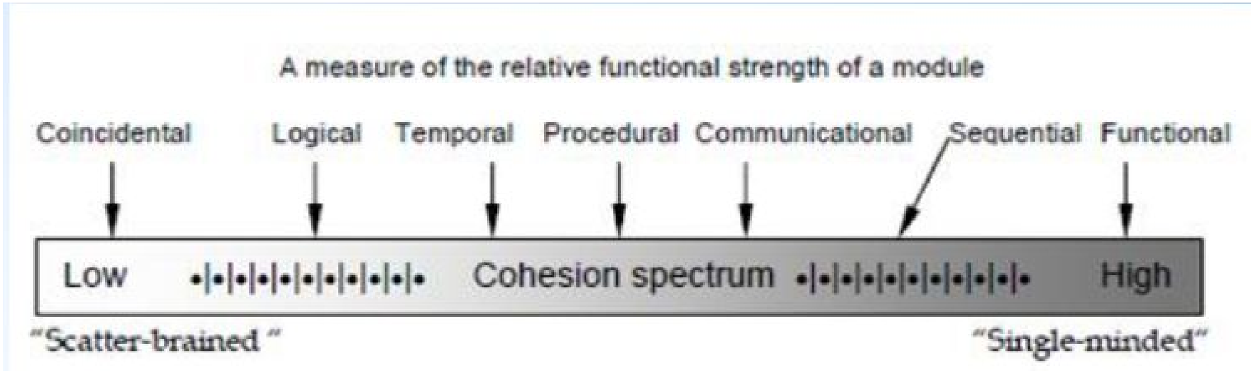
- The references made from one another.

- The amount of data passed from module to another.
- The amount of control one module has over the other.

Coupling along a spectrum of dependence.

Cohesion - Refers to the dependence within and among a module's internal elements.

Cohesion (low cohesion is bad to high cohesion is good)



- **Coincidental** (worst degree) - Parts are unrelated to one another.
- **Logical** - Parts are related only by the logic structure of code.
    - A logically cohesive module is one whose elements perform similar activities and in which the activities to be executed are chosen from outside the module.
- **Temporal** - Module's data and functions related because they are used the same time in an execution
- **Procedural** - Similar to temporal, and functions pertain to some related action or purpose. One step after another.
    - A procedurally cohesive module is one whose elements are involved in different activities, but the activities are sequential.
- **Communicational** - Operates on the same data set.
    - A communicationally cohesive module is one whose elements perform different functions, but each function references the same input information or output.

- **Functional** (ideal degree) - All elements essential to a single function are contained in one module, and all of the elements are essential to the performance of the function.
  - A functionally cohesive module is one in which all of the elements contribute to a single, well-defined task.
  - Performs one and only one problem related task.
- **Informational** - Adaption of functional cohesion to data abstraction and object-based design. All attributes, methods, and actions are strongly interdependent and essential to the object.

Interface - Defines what services the software unit provides to the rest of the system, and how other units can access those services.

- Must also define what unit require, in terms of services or assumptions, for it to work correctly.
- A software unit's interface describes what the unit requires of its environment, as well as what it provides to its environment.

Specification - describes the external visible properties of the software unit.

An interface specification should include:

- **Purpose** - document the functionality of each access function.
- **Preconditions** - assumptions
- **Protocols** - order in which functions are invoked
- **Postconditions** - visible effects
- **Quality attributes**

Information hiding - the segregation of design decisions in software units.

- Each software unit <u>encapsulates a separate design decision</u> that could be changed in the future.
- Only the interfaces and interface specifications are used to describe each software unit in terms of its externally properties.
- Using this principle, modules may exhibit different cohesions:

    - A module that hides a data representations may be informationally cohesive.
    - A module that hides an algorithm may be functionally cohesive.

In information hiding, resulting software units are loosely coupled.

In OO design, we decompose a system into objects and their abstract types.

- Each object hides its data representations from other objects.
- Only access is via a set of access functions that the object advertises in its interface.

Objects cannot be completely uncoupled from one another, because an object needs to know the identity of the other objects so that they can interact.

Uses relation - relates each software unit to the other software units on which it depends.

Uses graph - graph that help identify progressively larger subsets of our system that we can implement and test incrementally.

Uses graph for two designs:
- **Fan-in** refers to the number of units that use (call) a particular software unit.
- **Fan-out** refers to the number of units used (called) by particular software unit.

Sandwiching - technique that decomposes the cycle into two units, one with no dependencies.

Abstractions - model that omits some details so that it can focus on other details.

Object Oriented methodologies are the most popular design methodology.

Chapter 7: Maintaining Programming Standards

Programming Standards - Allows you to maintain correspondence between design and code components. Recall that it is important to match design with implementation.

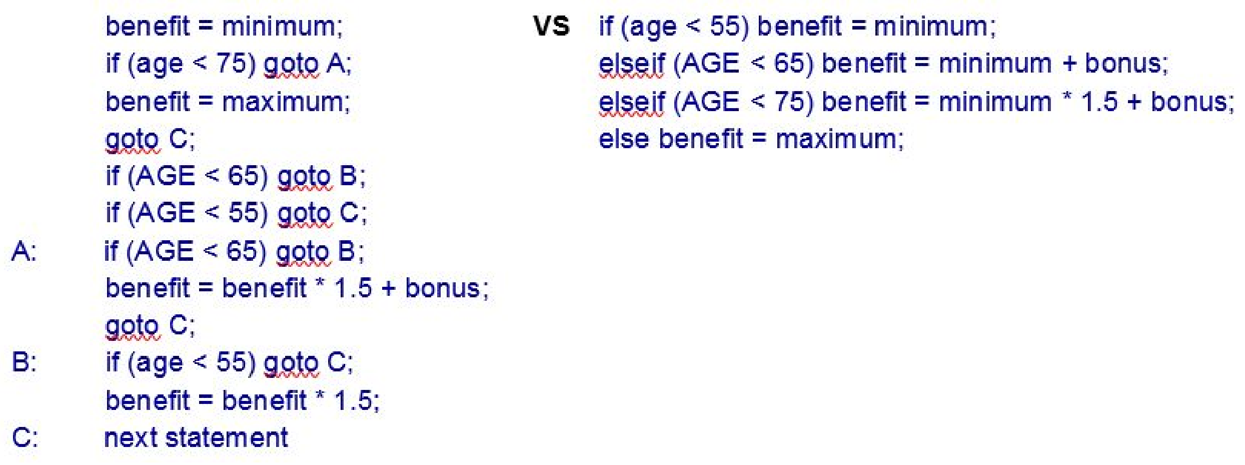
Use good programming characteristics:

- **Loosely Low Coupled** - not many dependencies between different modules (classes) of a program.
- **Tightly High Cohesion** - Focusing only on the specific purpose of one class - every function in that class pertains to it. Internal to modules. (Example: In a program for calculating pi, you don't want to count cows)
- **Well defined interfaces** - only code that relates to the class is included.

Standards(You): methods of code documentation
Standards(Others):
- Integrators, maintainers, testers

- Prologue documentation (page 375 textbook)
  What does this mean?
  I'm at a loss for this one, i don't have the textbook and we didn't cover it. I did find something similar to the following with a google search:
    - *The purpose of prologue is to provide both users and maintainers of the code with a better understanding of what the code does. Contains: Description, Author, Version, List of related classes*
      Ok! I get it. It seems to basically just be the top comment at every program - the description comment. Prologue means to start and documentation means to write, so maybe where the writing [comments] starts?
- Automated tools to identify dependencies

Programming Guidelines - Control Structures
Recall:
- Easy to read code
- Build program from modular blocks
- Not too specific, not too general
- Promote coupling with parameter names and comments
- Make dependencies visible



```
        benefit = minimum;
        if (age < 75) goto A;
        benefit = maximum;
        goto C;
        if (AGE < 65) goto B;
        if (AGE < 55) goto C;
A:      if (AGE < 65) goto B;
        benefit = benefit * 1.5 + bonus;
        goto C;
B:      if (age < 55) goto C;
        benefit = benefit * 1.5;
C:      next statement
```

VS

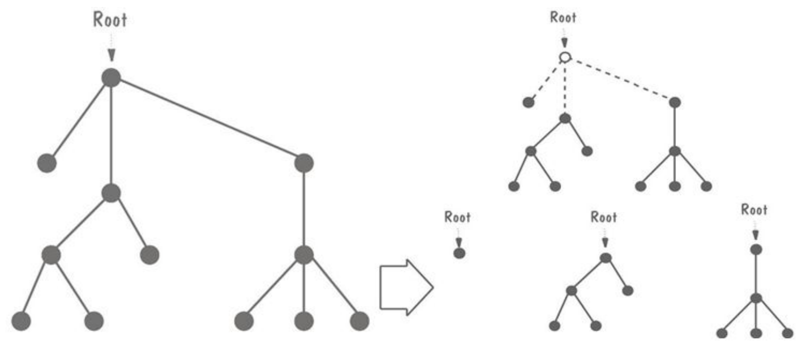
```
if (age < 55) benefit = minimum;
elseif (AGE < 65) benefit = minimum + bonus;
elseif (AGE < 75) benefit = minimum * 1.5 + bonus;
else benefit = maximum;
```

Algorithm - unambiguous specification of how to solve a class of problems with a general emphasis on performance(speed).

- Hidden Costs - there may be hidden costs to optimize, test, modify, or understand the code.
- Must know how the compiler optimizes (e.g., implementing a 3-dimensional array as a 1-dimensional array)
  ***note**: *this was on your slide deck but not my section's slides. Not sure if it's important for you guys thanks! Every info is welcome*

Data Structures - techniques using the structure of data to organize program. Keep it simple. Use data structure to determine program structure

Example: Rooted Tree (Recursive Data Structure)



Pseudocode - is an informal high-level description of the operating principle of a computer program or other algorithm.

Example of pseudocode evolution, skipping intermediate steps:

(Initial)

```
COMPONENT PARSE_LINE
        Read nest eighty characters.
                IF this is a continuation of the previous line,
                        Call CONTINUE
                ELSE determine command type
                ENDIF
        CASE of COMMAND_TYPE
                COMMAND_TYPE is paragraph: Call PARAGRAPH
                COMMAND_TYPE is indent : Call INDENT
                COMMAND_TYPE is skip line: Call SKIP_LINE
                COMMAND_TYPE is margin : Call MARGIN
                COMMAND_TYPE is new page : Call PAGE
                COMMAND_TYPE is double space : Call DOUBLE_SPACE
                COMMAND_TYPE is single space : Call SINGLE_SPACE
                COMMAND_TYPE is break : Call BREAK
        COMMAND_TYPE is anything else: Call ERROR
    ENDCASE
```

(Final)

```
INITIAL:
    Get parameter for indent, skip_line, margin.
    Set left margin to parameter for indent.
    Set temporary line pointer to left margin for all but paragraph; for paragraph, set to paragraph
      indent.
LINE_BREAKS:
    If not (DOUBLE_SPACE or SINGLE_SPACE), break line, flush line buffer and set line pointer to
      temporary line pointer
    If 0 lines left on page, eject page and print page header.
INDIVIDUAL CASES:
    INDENT, BREAK: do nothing.
    SKIP_LINE: skip parameter lines or eject
    PARAGRAPH: advance 1 line; if < 2 lines or page, eject.
    MARGIN: right_margin = parameter.
    DOUBLE_SPACE: interline_space = 2.
    SINGLE_SPACE: interline_space = 1;
    PAGE: eject page, print page header
```

Consumer Reuse - reuse components initially developed for other projects.
One example of consumer reuse might be calling a library to perform log calculations within a chemistry class.

Four key characteristics to check components to reuse:

1. Does component perform function or provide data needed?
2. Less modification than building component from scratch?
3. Component well documented?
4. Complete record of component's test/revision history?

Producer Reuse - create components designed to be reused in future applications.
One example might be to create a transistor that can be used in multiple electronics projects.

- Components are general

- Separate dependencies (isolate sections likely to change)
- General and well-defined component interface
- Include information on faults found and fixed
- Use clear naming conventions
- Document data structures and algorithms
- Keep communication and error-handling sections separate and easy to modify

Internal Documentation - documentation written in the code.
- **Header Comment Block**
    - What component is called
    - Who wrote component
    - Where component fits in general system design
    - When component was written/revised
    - Why the component exists
    - How component uses data structures, algorithms, and control
- Meaningful naming conventions
- Other program comments
- Formatting to enhance understanding
- Document data (data dictionary)

External Documentation
- Describe problem, algorithm, and data (data flow diagrams)

Extreme Programming - As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted. Usually occurs in two week cycles and may employ pair programming. Has two types of participants:
- Customers - define features using stories, describe detailed tests and assign priorities
- Programmers - implement stories

What is a “story”?
**note: Yeah i dislike this term.*

User stories - one of the primary development artifacts for Scrum and Extreme Programming (XP) project teams. A user story is a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it

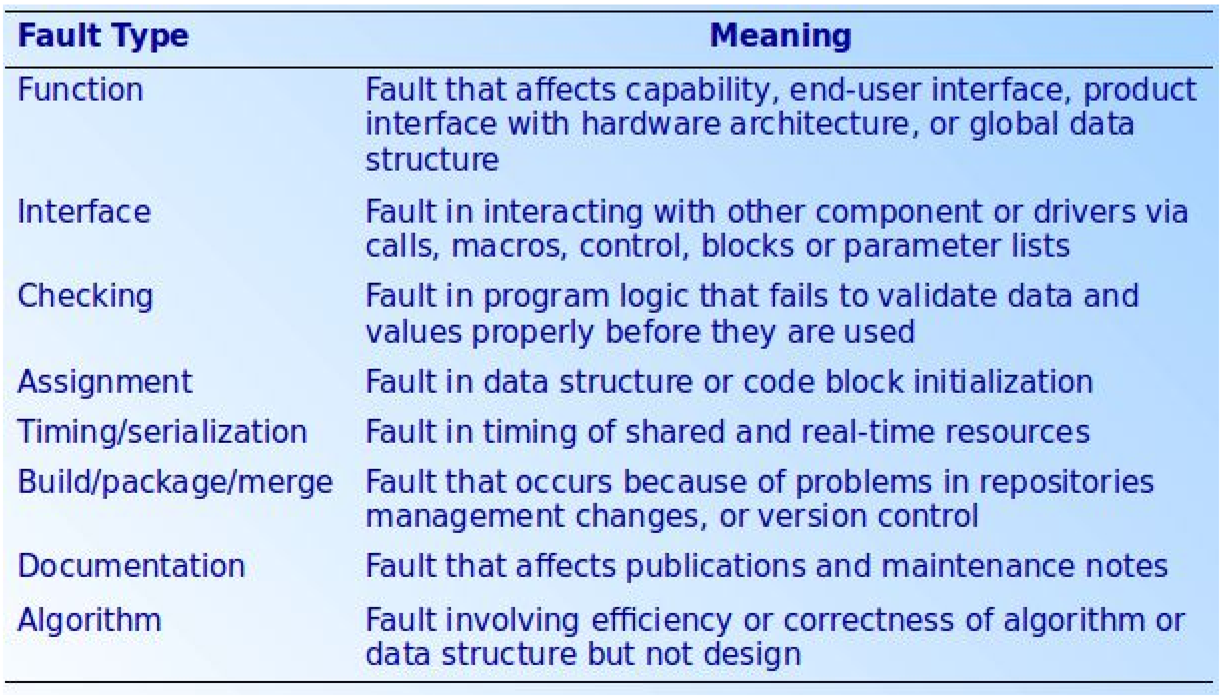
Pair Programming - Involves a driver(pilot) controlling the computer writing code while a “navigator” reviews the driver’s code and provides feedback. Airplane analogies work well here.

Documentation and code reviews - assist in planning, helps describe key abstractions, define system boundaries, and assist in communication among team members.

Chapter 8: Testing the Programs

Testing is used to discover faults and is considered successful when the fault is discovered.
Fault Identification is the process of determining what fault caused the failure
Fault Correction is the process of making changes to the system so that the faults are removed.

Types of Faults: Algorithmic, Computation/Precision, Documentation, Capacity/Boundary, Timing/Coordination, Performance, Standard and Procedure



Fault Type	Meaning
Function	Fault that affects capability, end-user interface, product interface with hardware architecture, or global data structure
Interface	Fault in interacting with other component or drivers via calls, macros, control, blocks or parameter lists
Checking	Fault in program logic that fails to validate data and values properly before they are used
Assignment	Fault in data structure or code block initialization
Timing/serialization	Fault in timing of shared and real-time resources
Build/package/merge	Fault that occurs because of problems in repositories management changes, or version control
Documentation	Fault that affects publications and maintenance notes
Algorithm	Fault involving efficiency or correctness of algorithm or data structure but not design

Egoless Programming: programs are viewed as components of a larger system.

Who performs testing? Usually an independent test team to avoid conflicts, improve objectivity, and allowing testing and coding to be done concurrently.

What factors affect testing choice philosophy? Number of logical paths, nature of input data, amount of computation, complexity of algorithm.

<u>Unit Testing</u>

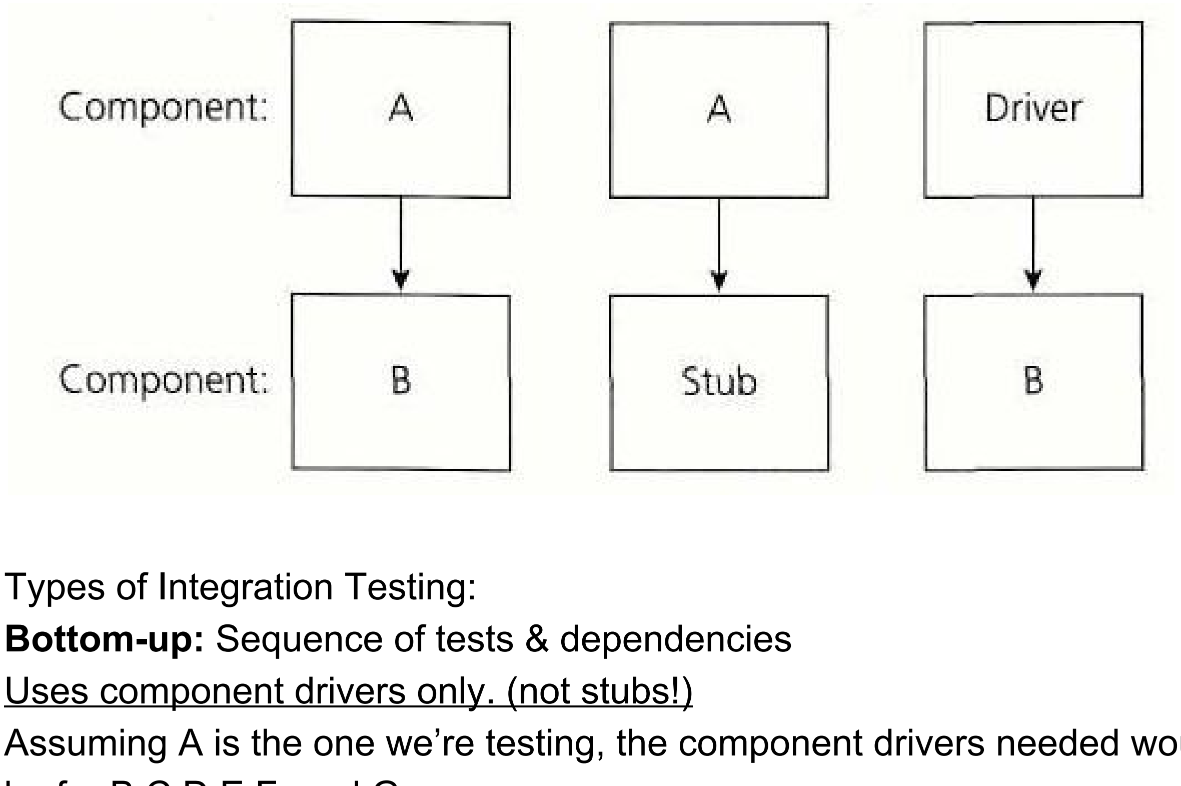
There are two steps **Unit Testing**, Code Walkthrough and Code Inspection.
The preparation rate, not the team size, usually determines inspection effectiveness. Effectiveness is based on familiarity of the product.

<u>Integration Testing</u>

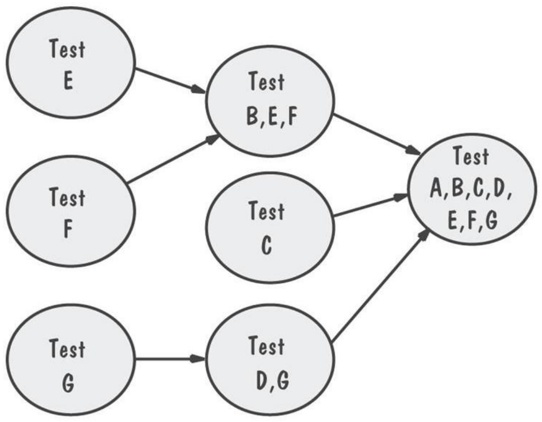
Component Driver: a routine that calls a particular component and passes a test case to it. Using testing data to run a particular component.

A **Driver** is a piece of software that drives (invokes) the Unit being tested. A driver creates necessary 'Inputs' required for the Unit and then invokes the Unit.

Stub: Simulate the activity of a missing component. A method **stub** or simply **stub** in **software** development is a piece of code used to stand in for some other **programming** functionality. A 'Stub' is a piece of software that works similar to a unit which is referenced by the Unit being tested, but it is much simpler than the actual unit. A Stub works as a 'Stand-in' for the subordinate unit and provides the minimum required behavior for that unit. A Stub is a dummy procedure, module or unit that stands in for an unfinished portion of a system.



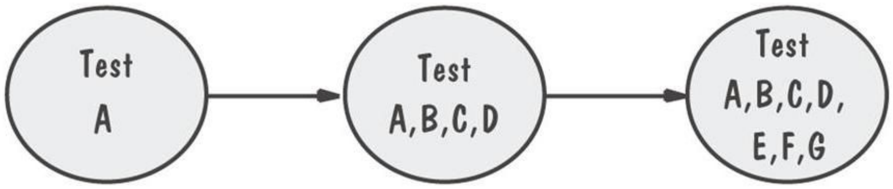
Types of Integration Testing:
Bottom-up: Sequence of tests & dependencies
<u>Uses component drivers only. (not stubs!)</u>
Assuming A is the one we're testing, the component drivers needed would be for B,C,D,E,F, and G.



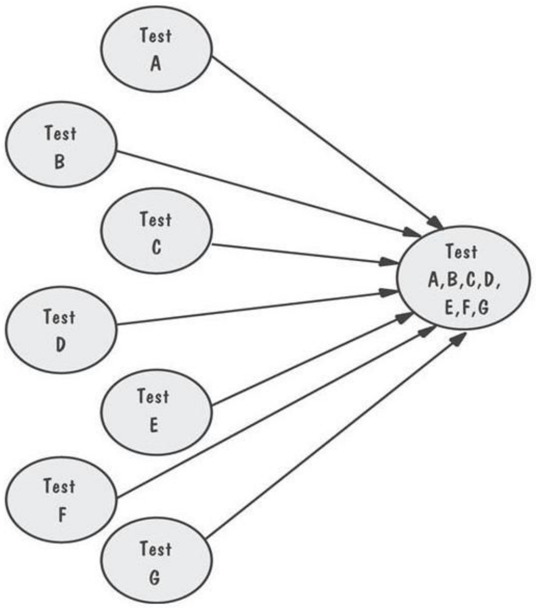
Top-down: Only A is tested by itself

Uses stubs only. (Not drivers!)

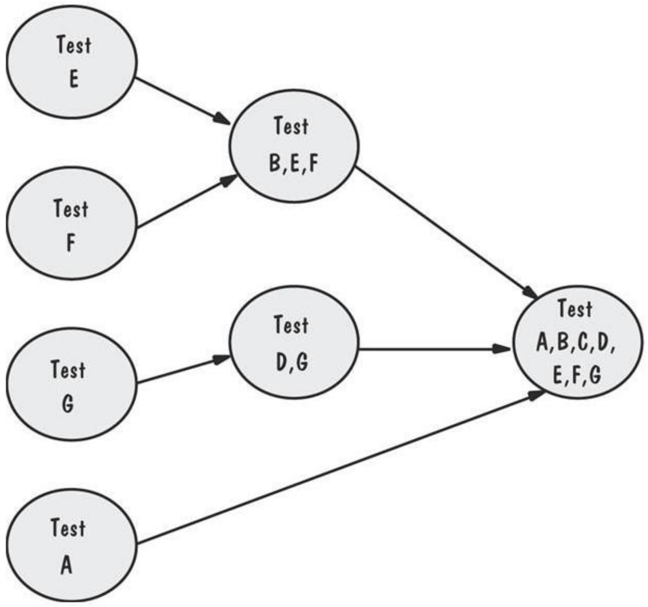
In this example, the stubs needed are B,C,D,E,F and G.



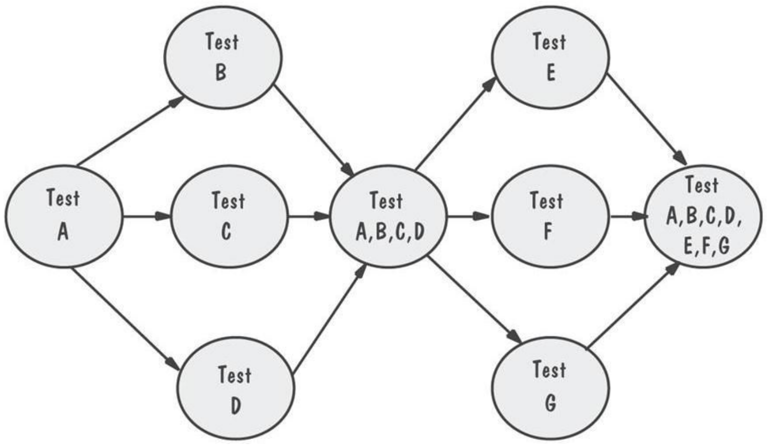
Big-bang: Uses both stubs and drivers to test the independent components because you have to test each individual component separately and also test all of them merged together.



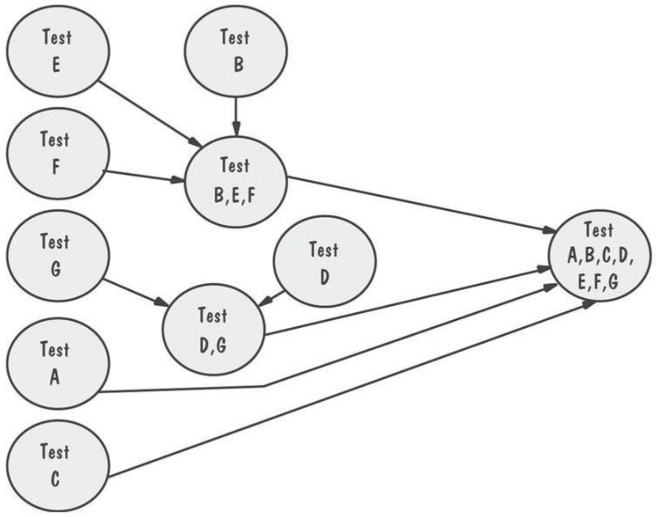
Sandwich: System of three layers - test, combine, test, combine
Sandwiches use both stubs and component drivers correct?



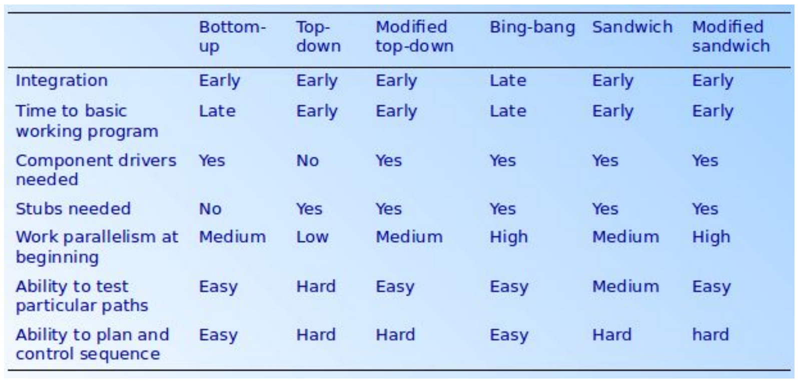
Modified top-down: Each level's components individually tested before merger. *Uses only stubs...no component drivers*



Modified sandwich: Allows upper-level components to be tested before merging them with others. A combination of bottom up and top down approaches. *Has both drivers and stubs? Yes, the Integration Strategies chart shows which strategies need drivers and which need stubs.*

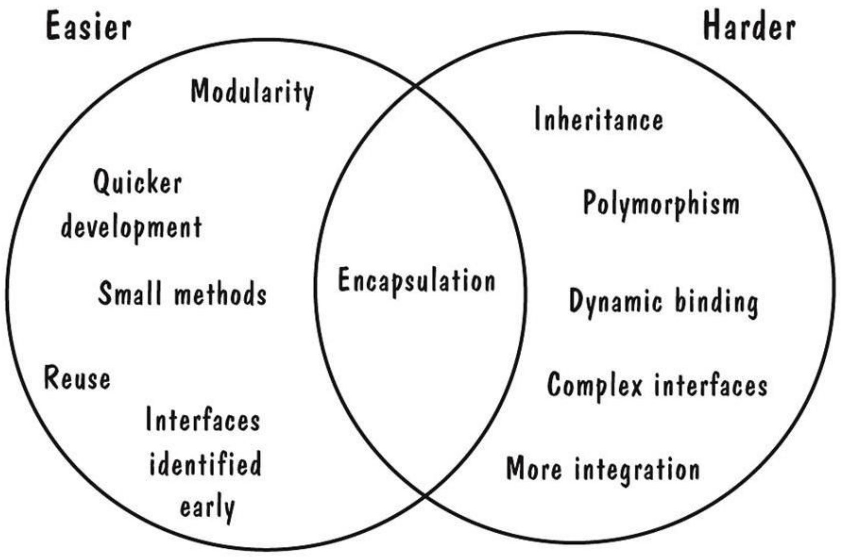


Comparison of Integration Strategies:



	Bottom-up	Top-down	Modified top-down	Bing-bang	Sandwich	Modified sandwich
Integration	Early	Early	Early	Late	Early	Early
Time to basic working program	Late	Early	Early	Late	Early	Early
Component drivers needed	Yes	No	Yes	Yes	Yes	Yes
Stubs needed	No	Yes	Yes	Yes	Yes	Yes
Work parallelism at beginning	Medium	Low	Medium	High	Medium	High
Ability to test particular paths	Easy	Hard	Easy	Easy	Medium	Easy
Ability to plan and control sequence	Easy	Hard	Hard	Easy	Hard	hard

<u>Object-Oriented Testing</u>
OO unit testing is less difficult, but integration testing is more extensive.



Test Planning Objectives: Establish test objectives, design test cases, write test cases, test test cases, execute tests, evaluate results

Test Plan - explains who does the testing, why the tests are performed, how tests are conducted, when tests are scheduled.

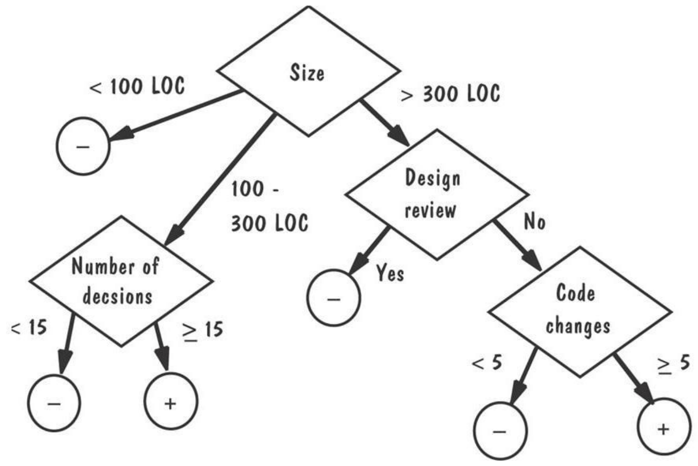
Formula for **Fault Seeding**

$$\frac{\text{detected seeded Faults}}{\text{total seeded faults}} = \frac{\text{detected nonseeded faults}}{\text{total nonseeded faults}}$$

Software Confidence

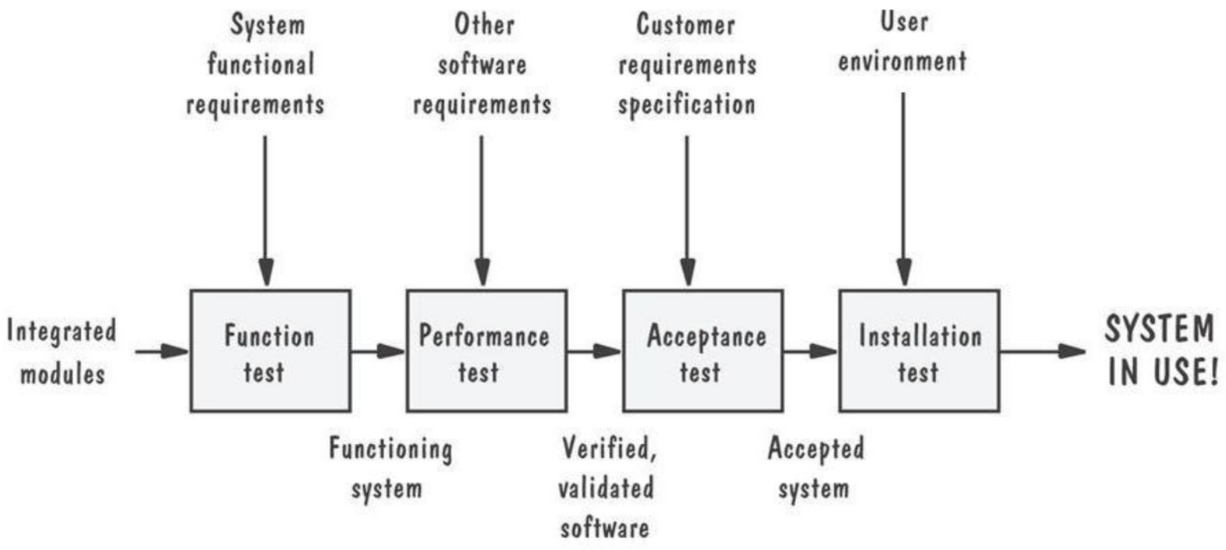
$$\begin{aligned} &= 1, && \text{if } n > N \\ &= S/(S - N + 1) && \text{if } n \le N \end{aligned}$$

Classification Tree Example:



Chapter 9: Testing the System

System Test - eliminates faults that lead to failures



Functional Testing - does the integrated system perform as promised by the requirements specification? (Do major components work correctly? For instance, making a flight reservation, reserving your seat, printing your boarding pass, checking your luggage, etc.)

- Similar to closed/**black box testing**
- *Black Box : functionality of the test objects. You can't see inside the program, you can only test it by giving the program inputs and tracking the outputs.*
- *White Box : structure of the test objects. You can see inside the code and test that the code runs as expected.*

Performance Testing - are the non-functional requirements met? (Security, speed, accuracy of computations, etc.)

- Performed AFTER functional testing
- Usually involves both hardware and software
- Designed and administered by test team (see below)
- Types of performance tests:
  - Stress test, volume test, configuration test, compatibility test, regression test, security test, timing test, environmental test, quality test, recovery test, maintenance test, documentation test, human factor (usage) test… see page 472-473 of the textbook

Once the functional and performance tests are completed, we're sure that:

- System operates correctly - the way the designers intended (**verified system**)
- System operates according to the requirements definition document (**validated system**)

These tests show the system works how the designers intended it to. But what about the customers?

Acceptance Testing - is the system what the customer expects? Will the customer <u>accept</u> the final product? (Customer can assure the system is what they requested)

- Performed after functional and performance tests
- Customers lead these tests
- Types of acceptance tests:
    - Benchmark test: used frequently when trying to compare similar software products
    - Pilot test: install on experimental basis, customer test drives the system
        - Alpha test: in-house customer test; within own organization
        - Beta test: customers pilot the system
    - Parallel testing: used when a new system replaces an old one; new system operates in parallel with old system as the user transitions to the new system

Installation Testing - does the system run at the actual environment or customer site(s)?

An **accepted system** is a system that operates the way the customers intended. A customer's intents are not always completely clear in the beginning, but the system becomes accepted once it meets all finalized requirements stated by customer.

Techniques used in system testing:

Build or integration plan

- Based on testing each of the subsystems independently
    - Roll-out in subsystem phases; sometimes called a **spin**
    - Define the "spins" (subsystems) to be tested
    - Describe how, where, when, and by whom the tests will be conducted

Regression testing

Regression means to return to a former or less developed state. Thus:

- Ensures the next version of the system performs like the older version. Example: Iphone 5,6,X...

    - Identifies new faults that may have been introduced as current ones are being corrected - often using already executed test cases which are re-executed to ensure existing functionalities work fine
- Regression testing steps
    - Inserting new code
    - Testing functions affected by new code
    - Testing essential functions of *m* to verify that they still work properly
    - Continue function testing *m* + 1

Configuration management (Github, Subversion, etc)

- Developing and testing different configurations
    - Versions (one for each platform) and releases (minor fixes)
    - Deltas and separate files
        - Designate one of the versions as the main version and all other versions will become a variation of the main version. Thus, we only need to store the differences as files (called **deltas**) rather than complete versions of all components.   This is significant because it saves storage and allows users to quickly revert if necessary.
    - Conditional compilation - by usage of parameters, allows the compiler to produce differences in the executables
    - Change control - changes approved by a configuration management team: eliminates problems and ensures components checked out

System testing consists of a test team, which is made up of:

- **Professional testers**: who organize and run the tests (often excludes any coders)
- **Analysts**: who create the requirements
- **System designers**: who understand the proposed solution
- **Configuration management specialists**: who help control fixes
- **Users**: who evaluate issues that arise

Software reliability - operating without failure under given conditions for a given time interval. Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment.

Software availability - operating successfully according to specifications at a given point in time. High availability software refers to the use of software to ensure that systems are running (available) most of the time.

Software maintainability - for a given condition of use, a maintenance activity can be carried out within stated time interval using provided procedures and resources

Different levels of failure severity:

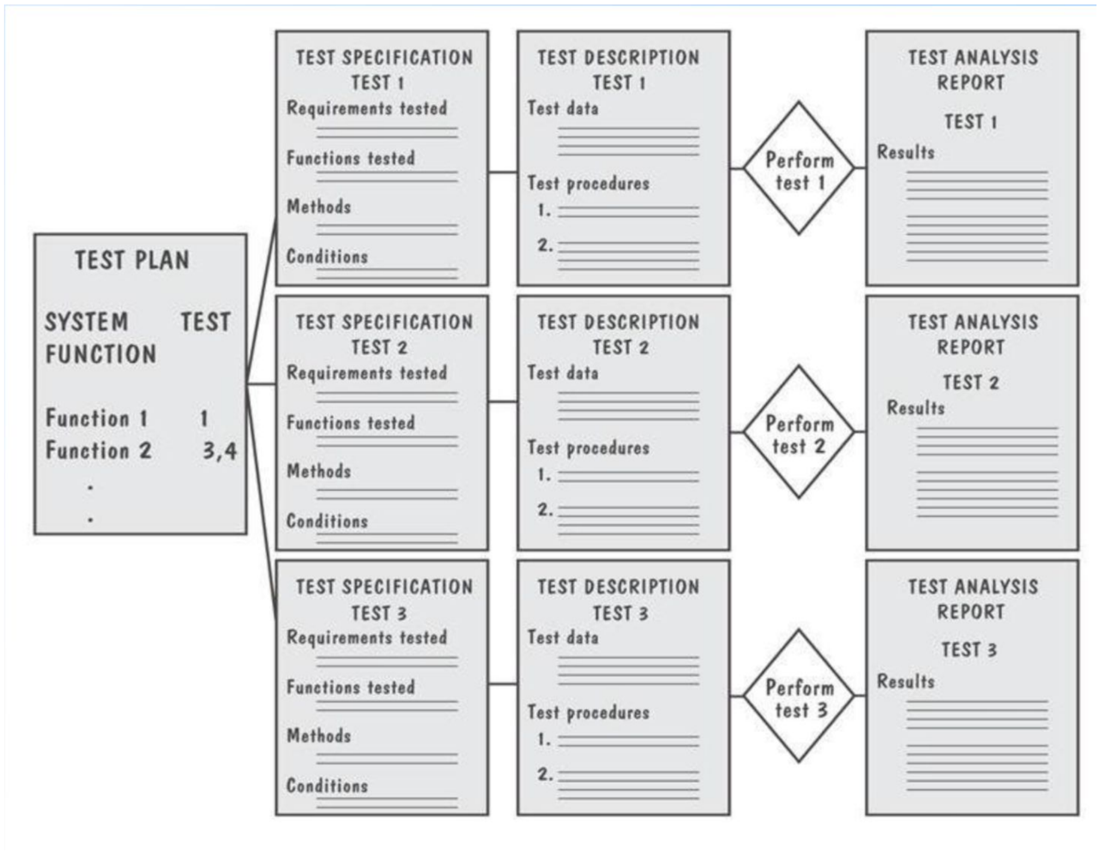
- **Catastrophic**: causes death or system loss
- **Critical**: causes severe injury or major system damage
- **Marginal**: causes minor injury or minor system damage
- **Minor**: causes no injury or system damage

Our inability to predict when the next failure will occur is because:

- Type 1 uncertainty: how the system will be used
- Type 2 uncertainty: lack of knowledge about the effect of fault removal

Test Documentation: used to overcome the complexity and difficulty of testing

- **Test plan**: describes system and plan for exercising all functions and characteristics
- **Test specification and evaluation**: details each test and defines criteria for evaluating each feature
- **Test description**: specifies test data and procedures for each test
- **Test analysis report**: results of each test



Test effectiveness: can be measured by dividing the number of faults found in a given test by the total number of faults found to date

Test efficiency: computed by dividing the number of faults found in testing by the effort needed to perform testing (like faults per staff hour)

Problem report forms:

- **Location**: where did the problem occur?
- **Timing**: when did it occur?
- **Symptom**: what was observed?
- **End result**: what were the consequences?
- **Mechanism**: how did it occur?
- **Cause**: why did it occur?
- **Severity**: how much was the user or business affected?
- **Cost**: how much did it cost?

Testing safety-critical systems - ultra-high reliability:

- **Design diversity**: use different kinds of designs, designers
- **Software safety cases**: make explicit the ways the software addresses possible problems
- **Cleanroom**: certifying software with respect to the specification. Example: ESD can break small electronic components, and a single piece of dust within the silicon can destroy a transistor.

Chapter 10

Delivering The System

Training (Types of People Who Use a System)
Users: exercise the main system functions
Operators: perform supplementary functions, create backup copies, define who has access

User Functions
Manipulating, Analyzing,Communicating Data
Simulating Activities

Operator Functions
Granting user, file access
Backups, Installing new software, recovery

Types of Training
User Training

- Introduces primary functions,
- Relates how functions are performed now with the existing system, versus how they will perform later with the new system

Operator Training

- Focuses on support functions and addresses how the system works rather than what the system does

Special Training Needs
- Infrequent vs frequent use
- New users
- Existing users brushing up on things missed
- Need for specialized training for both users and operators like generating special reports

Training Aids
- Documents
- Well defined icons and online help
- Demos and classes
- Expert users

Types of Documentation
- **User's Manual**
  - Begin w/ general purpose, and progress to detailed functional desc
  - System's capabilities & functions
  - System features, characteristics, advantages
- **Operator's Manual**
  - Hard/Software config
  - Methods of granting/denying access
- **General System Guide**
  - System details in terms the customer can understand
- **Tutorials**
  - Multimedia based step by step automated tutorials
- **Other documentation: programmer's guide**
  - Technical counterpart to user's manual for those who will maintain and enhance the system

        - An overview of how the software and hardware are configured
        - Software components detailed and their functions performed
        - System support functions
        - System enhancements

Failure Messages

- Name of code component executing when failure occured

Chapter 11

Maintenance : defined as any work done to change the system after it is in operation.

The Changing System: Lehman`s System Types :

- S-System: formally defined, derivable from a specification. (Matrix Multiplication)
- P-System: requirements based on approximate solution to a problem, but real-world remains stable. (Chess Program)
- E-System: embedded in the real world and changes as the world does. (Software to predict how economy functions but economy is not completely understood)

S = Stable
P = Problem
E = Economy

Changes During the System Life Cycle

- S-System: unchanged
- P-System: incremental change

        - An approximate solution
        - Changes as discrepancies and omissions are identified
- E-System: constant change

  General Notes(she did not mention much about ch 11 on review day):
- ❖ The more a system is linked to the real world, the more likely it will change and the more difficult it will be to maintain.
- ❖ Measuring maintainability is hard. Types of maintenance include:
    - ➢ Corrective: maintaining control over day-to-day functions
    - ➢ Adaptive: maintaining control over system modifications
    - ➢ Perfective: perfecting existing functions
    - ➢ Preventive: preventing system performance from degrading to unacceptable levels.
- ❖ Impact analysis builds and tracks links among the requirements, design, code, and test cases.
- ❖ **Software rejuvenation** involves:
    - ➢ Redocumentation: static analysis adds more information.
    - ➢ Restructuring: transform to improve code structure.
    - ➢ Reverse Engineering: recreate design and specification information from the code.
    - ➢ Reengineering: reverse engineer and then make changes to specification and design to complete logical model; then generate new system from revised specification and design.

<u>Acronyms:</u>

DFD: Data Flow Diagram

DRY: Don’t repeat yourself

WET: Write Everything Twice

References:

https://quizlet.com/235354759/cop-4331-ucf-chapter-5-and-6-flash-cards/

This quizlet isn't great, I give it 5/7 that's why I wanted to make a better one.

https://www.youtube.com/watch?v=gyXcu78bWis

Good explanation of functional v non functional reqs

Found this interesting quizlet:

https://quizlet.com/40184691/cop-4331-ch-5-flash-cards/

Post any helpful links here :)

https://www.koofers.com/files/exam-ma1an2lzrr/

https://quizlet.com/244690879/chapter-9-and-10-cop-4331c-ucf-fall-2017-flash-cards/

Chat Room:

1. *What questions do you expect to see on this exam?*
    - *UML Diagrams*
    - *Select which of these is a stakeholder*
    - *5 Architectural Styles*
    - *Composition & Aggregation*
    - *Stamp Coupling*
    - *Critical Path*
    - *Difference between composition and aggregation.*
2. *Nassif says pseudo code will be on this exam. What does he mean?*
3.
4.

5.
6.

<u>Practice Exam:</u>

1) What are the three ways of considering quality for a process?
2) Name the 5 Architectural Styles
3) What is the difference between composition and aggregation?
4) What is stamp coupling? How does it differ from data coupling?
5) What are the six design principles?

Practice Exam:
http://www.cs.ucf.edu/~workman/cop4331/