Software Engineering
Final Exam
Review

Note: This review supplements the previous reviews, since the final will be cumulative

Chapter 9

- Design: A problem solving process whose objective is to find and describe a way:
  - Implement the system's functional requirements
  - While respecting the constraints imposed by the non-functional requirements
    - Including the budget
  - And while adhering to the general principles of good quality
- Design Issues
  - Sub-problems of the overall design problem
  - Solutions: design options
- To make decisions, the software engineer uses knowledge of:
  - Requirements
  - Design so far
  - Technology
  - Software design principles
  - What has worked well in the past
- Component: Any piece of software or hardware that has a clear role
- Module: A component that is defined at the programming language level
- System: A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software, or both
- Top-down Design
  - First design the very high level structure of the system
  - Then gradually work down to detailed decisions about low-level constructs
  - Finally arrive at detailed decisions such as:
    - The format of particular data items
    - The individual algorithms that will be used
- Bottom-up Design
  - Make decisions about reusable low-level entities
  - Then decide how these will be put together to create high-level constructs
- A mix of top-down and bottom-up approaches are normally used
- Different Aspects of Design
  - Architecture Design: The division into subsystems and components
  - Class Design: The various features of classes
  - User Interface Design
  - Algorithm Design: The design of computational mechanisms
  - Protocol Design: The design of communications protocol
- Principles Leading to Good Design
  - Goals
    - Increase profit by reducing cost and increasing revenue
    - Ensuring that we actually conform with the requirements
    - Accelerating development
    - Increasing RUMER qualities
- Design Principles
  - Divide and Conquer

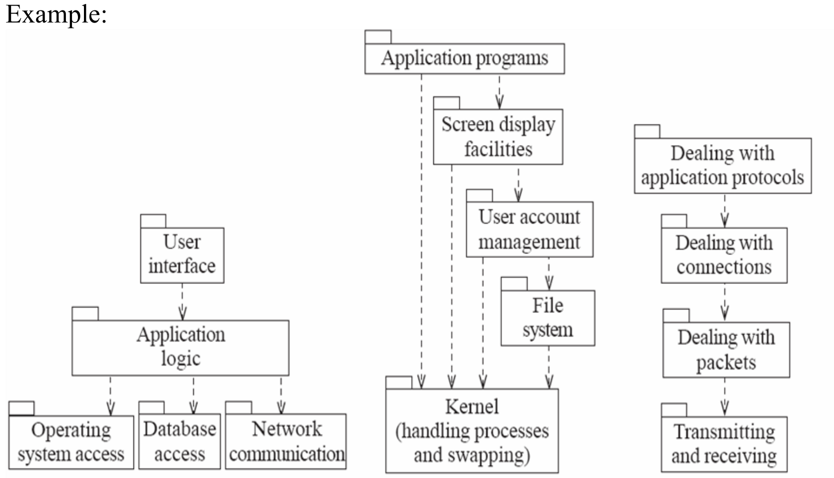
        - Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things
            - Separate people can work on each part
            - An individual software engineer can specialize
            - Each individual component is smaller, and therefore easier to understand
            - Parts can be replaced or changed without having to replace or extensively change other parts
        - Ways of Dividing a Software System
            - Distributed system: clients and servers
            - System can be divided up into subsystems
            - Subsystem can be divided up into one or more packages
            - A package is divided up into classes
            - A class is divided up into methods
    - Increase Cohesion Where Possible
        - A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things
        - Functional Cohesion
            - This is achieved when all code that computes a particular result is kept together – and everything else is kept out
                - i.e. When a module only performs a single computation, and returns a result, *without having side-effects*
        - Layer Cohesion
            - All the facilities for providing or accessing a set of related services are kept together, and everything else is kept out
            - The layers should form a hierarchy
            - Commonly organized into an API (Application Programmer Interface)
        - Communicational Cohesion
            - All the modules that access or manipulate certain data are kept together (i.e. in the same class) – and everything else is kept out
            - A class would have good communication cohesion if:
                - All the system's facilities for manipulating and storing its data are contained in this class
                - If the class does not do anything other than manage its data
        - Sequential Cohesion
            - Procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out
            - You should achieve sequential cohesion only after achieving the other types of cohesion
        - Procedural Cohesion
            - Keep together several procedures that are used one after the other
                - Even if one does not provide input to the next
                - Weaker than sequential cohesion
        - Temporal Cohesion
            - Operations that are performed during the same phase of the program are kept together, and everything else is kept out
                - For example, placing together the code used during system start-up or initialization
                - Weaker than procedural cohesion
        - Utility Cohesion
            - When related utilities which cannot logically be placed in other cohesive units are kept together
    - Reduce Coupling Where Possible

    - Coupling occurs when there are interdependencies between one module and another
    - Content Coupling
        - Occurs when one component surreptitiously modifies data that is internal to another component
        - Solution: Encapsulate all instance variables; declare them as private
    - Common Coupling
        - Occurs whenever you use a global variable
        - Solution: Use the Singleton pattern where applicable, or simply do not use global variables
    - Control Coupling
        - Occurs when one procedure calls another using a ‘flag’ or ‘command’ that explicitly controls what the second procedure does
        - Solution: Use polymorphic methods or a lookup table
    - Stamp Coupling
        - Occurs whenever one of your application classes is declared as the type of a method argument
        - Solutions: Use an interface as the argument type or pass simple variables (atomic types)
    - Data Coupling
        - Occurs whenever the types of method arguments are either primitive or else simple library classes
        - Solution: Do not give methods unnecessary arguments
        - There is a trade-off between data coupling and stamp coupling
            - Increasing one often decreases the other
    - Routine Call Coupling
        - Occurs when one routine (or method in an object-oriented system) calls another
        - Routine call coupling is always present in any system
    - Type Use Coupling
        - Occurs when a module uses a data type defined in another module
    - Inclusion or Import Coupling
        - Occurs when one component imports a package
            - As in Java
        - Or when one component includes another
            - As in C++
    - External Coupling
        - When a module has a dependency on such things as the operating system, shared libraries or the hardware
- Keep the Level of Abstraction as High as Possible
    - Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity
        - A good abstraction is said to provide information hiding
    - Classes are abstractions that contain procedural abstractions
        - Increased by defining all variables as private
        - The fewer public methods in a class, the better the abstraction
        - Superclasses and interfaces increase the level of abstraction
        - Attributes and associations are also abstractions
        - Methods are procedural abstractions
            - Better abstractions are achieved by giving methods fewer parameters
- Increase Reusability Where Possible
    - Design the various aspects of your system so that they can be used again in other contexts

- Reuse Existing Designs and Code Where Possible
  - Design with reuse is complementary to design for reusability
- Design for Flexibility
  - Actively anticipate changes that a design may have to undergo in the future, and prepare for them
    - Reduce coupling, increase cohesion
    - Create abstractions
    - Do not hard-code anything
    - Leave all options open
      - Do not restrict the options of people who have to modify the system later
    - Use reusable code and make code reusable
- Anticipate Obsolescence
  - Plan for changes in technology or environment so the software will continue to run or can be easily changed
- Design for Portability
  - Have the software run on as many platforms as possible
- Design for Testability
  - Take steps to make testing easier
- Design Defensively
  - Never trust how others will try to use a component you are designing
  - Design by Contract
    - A technique that allows you to design defensively in a systematic way
    - Key idea
      - Each method has an explicit contract with its callers
      - Each contract has
        - Preconditions
        - Postconditions
        - Invariants

- Techniques for Making Good Design Decisions
  - Using priorities and objectives to decide among alternatives
    - List and describe the alternatives for the design decision
    - List the advantages and disadvantages of each alternative with respect to objectives and priorities
    - Determine whether any of the alternatives prevents you from meeting one or more of the objectives
    - Choose the alternative that helps you to best meet your objectives
    - Adjust priorities for subsequent decision making
- Software Architecture
  - Software architecture is the process of designing the global organization of a software system, including:
    - Dividing software into subsystems
    - Deciding how these will interact
    - Determining their interfaces
  - Why you need to develop an architectural model:
    - To enable everyone to better understand the system
    - To allow people to work on individual pieces of the system in isolation
    - To prepare for extension of the system
    - To facilitate reuse and reusability
  - Contents of a good architectural model
    - A system’s architecture will often be expressed in terms of several different views
      - The logical breakdown into subsystems
      - The interfaces among the subsystems

            - The dynamics of the interaction among components at runtime
            - The data that will be shared among the subsystems
            - The components that will exist at runtime, and the machines or devices on which they will be located
    - Design stable architecture
        - To ensure the maintainability and reliability of a system, an architectural model must be designed to be stable
    - Developing an architectural model
        - Start by sketching an outline of the architecture
            - Based on the principle requirements and use cases
            - Determine the main components that will be needed
            - Choose among the various architectural patterns
            - Refine the architecture
            - Consider each use case and adjust the architecture to make it realizable
            - Mature the architecture
- Architecture Patterns
    - The notion of patterns can be applied to software architecture
    - The Multi-Layer Architectural Pattern
        - In a layered system, each layer communicates only with the layer immediately below it
            - Each layer has a well defined interface used by the layer immediately above
                - Lower layers: Services
            - A complex system can be built by superimposing layers at increasing levels of abstraction
        - Example:

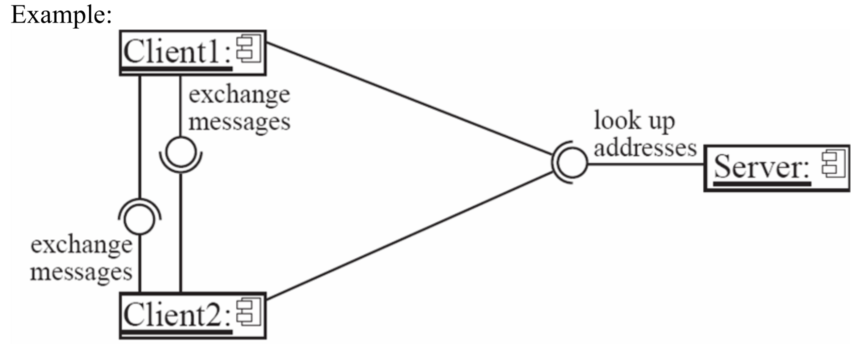


(a) Typical layers in an application program (b) Typical layers in an operating system (c) Simplified view of layers in a communication system

        - Design Principles
            - Divide and Conquer – layers can be independently designed
            - Increase Cohesion – uses layer cohesion
            - Reduce Coupling – Lower level layers do not know about higher level ones; higher level layers access lower level ones through well-defined API’s
            - Increase abstraction – You do not need to know how the lower level layers are implemented
            - Increase reusability – Lower level layers can often be designed generically

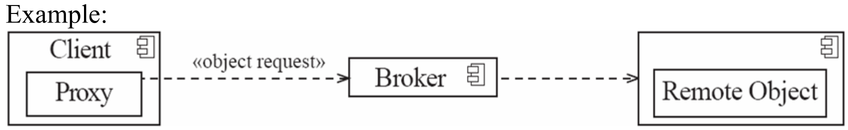
- Increase reuse – Reuse layers in other applications that provide needed services
- Increase flexibility – You can add new facilities to lower level layers, or replace higher level ones
- Anticipate Obsolescence – Isolating components makes the system obsolescent resistant
- Design for portability – All dependent facilities can be isolated into lower layers
- Design for testability – Layers can be tested independently
- Design defensively – API's are natural places to build in rigorous assertion-checking

o Client Server and other Distributed Architectural Patterns
  - At least one component has the role of server, waiting for and then handling connections
  - There is at least one component that has the role of client, initiating connections in order to obtain some service
  - Extension: Peer-to-Peer pattern
    - A system composed of various software components that are distributed over several hosts
  - Example:



  - Design Principles
    - Divide and Conquer – Client and Server systems
    - Increase Cohesion – The server can provide a cohesive service to clients
    - Reduce Coupling – One communication channel exchanging simple messages
    - Increase Abstraction – Separate distributed components are often good abstractions
    - Increase Reuse – Possible to find suitable frameworks to build good distributed systems
    - Design for Flexibility – Easily reconfigured
    - Design for Portability – Write clients for new platforms without having to port the server
    - Design for Testability – Test clients and servers independently
    - Design Defensively – You can put rigorous checks into the message handling code

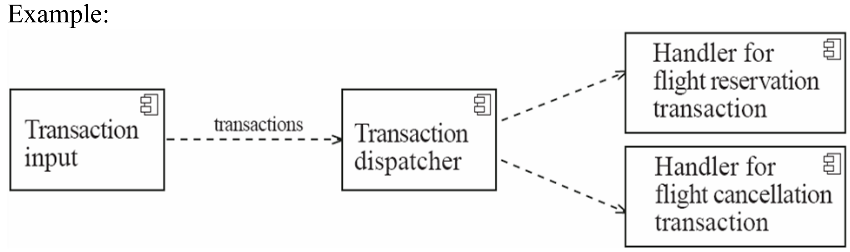
o Broker Architectural Pattern
  - Transparently distribute aspects of the software system to different nodes
  - Example:



  - Design Principles
    - Divide and Conquer – Remote objects can be independently designed

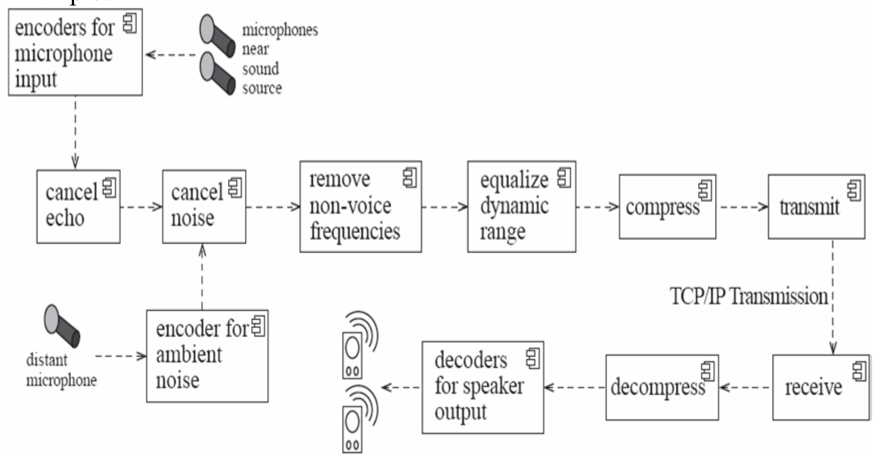
- Increase Reusability – Possible to design remote objects so that other systems can use them too
- Increase Reuse – You may be able to reuse remote objects that others have created
- Design for Flexibility – Brokers can be updated as required, or the proxy can communicate with a different remote object
- Design for Portability – Can write clients for new platforms while still accessing brokers and remote objects on other platforms
- Design Defensively – You can provide careful assertion checking in the remote objects

- Transaction-Processing Architectural Pattern
  - A process reads a series of inputs one by one
    - Each input describes a transaction
    - There is a transaction dispatcher component that decides what to do with each transaction
    - This dispatches a procedure call or message to one of a series of components that will handle the transaction
  - Example:



  - Design Principles
    - Divide and Conquer – Transaction handlers are suitable system divisions
    - Increase Cohesion – Transaction handlers are naturally cohesive units
    - Reduce Coupling – Separating the dispatcher from the handler tends to reduce coupling
    - Design for Flexibility – You can readily add new transaction handlers
    - Design Defensively – You can add assertion checking in each transaction handler and/or in the dispatcher
- The Pipe-and-Filter Architectural Pattern
  - A stream of data, in a relatively simple format, is passed through a series of processes
    - Each process transforms the data in some way
    - The data is constantly fed into the pipeline
    - The processes work concurrently
    - The architecture is very flexible

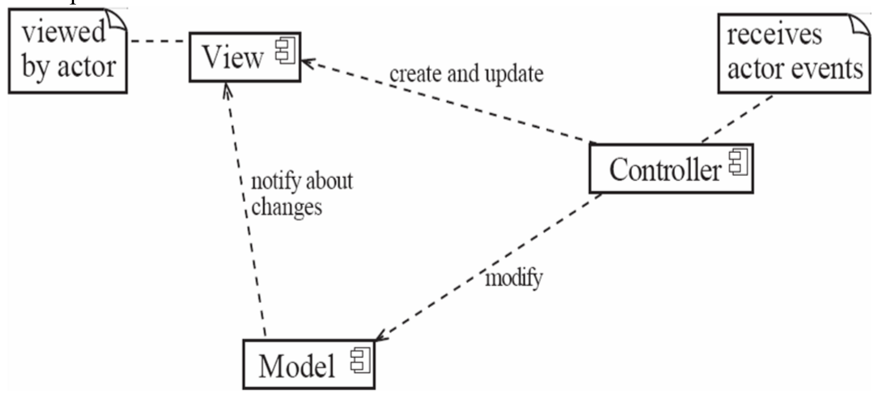
- Example:



- Design Principles
  - Divide and Conquer – Separate processes can be independently designed
  - Increase Cohesion – The processes have functional cohesion
  - Reduce Coupling – The processes have only one input and one output
  - Increase Abstraction – Pipeline components are often good abstractions
  - Increase Reusability – The processes can be used in many different contexts
  - Increase Reuse – It is often possible to find reusable components to insert into a pipeline
  - Design for Flexibility – There are several ways in which the system is flexible
  - Design for Testability – It is normally easy to test the individual processes
  - Design Defensively – You rigorously check the inputs of each component, or you can use design by contract

- The Model-View-Controller (MVC) Architectural Pattern
  - An architectural pattern used to help separate the user interface layer from other parts of the system
    - Model – Underlying classes whose instances are to be viewed and manipulated
    - View – Contains objects used to render the appearance of the data from the model in the user interface
    - Controller – Contains the objects that control and handle the user's interaction with the view and the model
    - The Observable design pattern is normally used to separate the model from the view

- Example:



- Design Principles
  - Divide and Conquer – The three components can be somewhat independently designed
  - Increase Cohesion – The components have stronger layer cohesion than if the view and controller were together in a single UI layer
  - Reduce Coupling – The communication channels between the three components are minimal
  - Increase Reuse – The view and the controller normally make extensive use of reusable components for various kinds of UI controls
  - Design for Flexibility – It is usually quite easy to change the UI by changing the view, the controller, or both
  - Design for Testability – You can test the application separate from the UI

Chapter 10

- Basic Definitions
  - Failure: Unacceptable behavior exhibited by a system
    - Frequency of failures measures the reliability
    - Design goal: achieve low failure rate, thus ensuring high reliability
    - A failure can result from a violation of an explicit or implicit requirement
  - Defect: A flaw in any aspect of the system that contributes, or may potentially contribute, to the occurrence of one or more failures
  - Error: A slip-up or inappropriate decision by a software developer that leads to the introduction of a defect
- Effective and Efficient Testing
  - To test effectively, you must use a strategy that uncovers as many defects as possible
  - To test efficiently, you must find the largest possible number of defects using the fewest possible tests
- Black-box Testing
  - Testers provide the system with inputs and observe the outputs
    - They can see none of
      - The source code
      - The internal data
      - Any of the design documentation describing the system's internals
- Glass-box Testing
  - Also called 'white-box' or 'structural' testing
  - Testers have access to the system design
    - They can
      - Examine the design documents

            - View the code
            - Observe at run time the steps taken by algorithms and their internal data
        - Individual programmers often informally employ glass-box testing to verify their own code
- Equivalence Classes
    - Inappropriate to test by brute force, using every possible input value
    - You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms
        - Such groups are called equivalence classes
    - Examples
        - Valid input number: (1 – 12)
            - Equivalence Classes are: [-inf..0], [1..12], [13..inf]
        - Valid input is one of ten strings representing a type of fuel
            - Equivalence Classes are
                - 10 classes, one for each string
                - A class representing all other strings
- Detecting Specific Categories of Defects
    - A tester must try to uncover any defects the other software engineers might have introduced
        - This means designing tests that explicitly try to catch a range of specific types of defects that commonly occur
- Defects in Ordinary Algorithms
    - Incorrect Logical Conditions
        - Defect:
            - The logical conditions that govern looping and if-then-else statements are wrongfully formatted
        - Testing Strategy:
            - Use equivalence class and boundary testing
            - Consider as an input each variable used in a rule or logical condition
    - Performing a Calculation in the Wrong Part of a Control Construct
        - Defect:
            - The program performs an action when it should not, or does not perform an action when it should
            - Typically caused by inappropriately excluding or including the action from a loop or an if construct
        - Testing Strategy:
            - Design tests that execute each loop zero times, exactly once, and more than once
            - Anything that could happen while looping is made to occur on the first, an intermediate, and the last iteration
    - Not Terminating a Loop or Recursion
        - Defect:
            - A loop or recursion does not always terminate, i.e. it is 'infinite'
        - Testing Strategies:
            - Analyze what causes a repetitive action to be stopped
            - Run test cases that you anticipate might not be handled correctly
    - Not Setting Up the Correct Preconditions for an Algorithm
        - Defect:
            - Preconditions state what must be true before the algorithm should be executed
            - A defect would exist if the program proceeds to do its work, even when the preconditions are not satisfied
        - Testing Strategy:

            - Run test cases in which each precondition is not satisfied
    - Not Handling Null Conditions
        - Defect:
            - A null condition is a situation where there are normally one or more data items to process, but sometimes there are none
            - It is a defect when a program behaves abnormally when a null condition is encountered
        - Testing Strategy:
            - Brainstorm to determine unusual conditions and run appropriate tests
    - Not Handling Singleton or Non-singleton Conditions
        - Defect:
            - A singleton condition occurs when there is normally more than one of something, but sometimes there is only one
            - A non-singleton is the inverse
            - Defects occur when the unusual case is not properly handled
        - Testing Strategy:
            - Brainstorm to determine unusual conditions and run appropriate tests
    - Off-by-one Errors
        - Defect:
            - A program inappropriately adds or subtracts one
            - Or loops one too many times or one too few times
            - This is a particularly common type of defect
        - Testing Strategy:
            - Develop tests in which you verify that the program:
                - Computes the correct numerical answer
                - Performs the correct number of iterations
    - Operator Precedence Errors
        - Defect:
            - An operator precedence error occurs when a programmer omits needed parentheses, or puts parentheses in the wrong place
            - Operator precedence errors are often extremely obvious
                - But can occasionally lie hidden until special conditions arise
            - E.g. If x * y + z should be x * (y + z) this would be hidden if z was normally zero
        - Testing Strategy:
            - In software that computes formulae, run tests that anticipate such defects
    - Use of Inappropriate Standard Algorithms
        - Defect:
            - An inappropriate standard algorithm is one that is unnecessarily inefficient or has some other property that is widely recognized as being bad
        - Testing Strategies:
            - The tester has to know properties of algorithms and design tests that will determine whether any undesirable algorithms have been implemented
        - Examples:
            - An inefficient sort algorithm
                - The most classical choice 'bad' choice of algorithm is sorting using a so-called 'bubble sort'
            - An inefficient search algorithm
                - Ensure that the search time does not increase unacceptably as the list gets longer

        - Check that the position of the searched item does not have a noticeable impact on search time
      - A non-stable sort
      - A search or sot that is case sensitive when it should not be, or vice versa
- Defects in Numerical Algorithms
  - Not using enough bits or digits
    - Defect:
      - A system does not use variables capable of representing the largest values that could be stored
      - When the capacity is exceeded, an unexpected exception is thrown, or the data stored is incorrect
    - Testing Strategies:
      - Test using very large numbers to ensure the system has a wide enough margin of error
  - Not using enough places after the decimal point or significant figures
    - Defects:
      - A floating point value might not have the capacity to store enough significant figures
      - A fixed point value might not store enough places after the decimal point
      - A typical manifestation is excessive rounding
    - Testing Strategies:
      - Perform calculations that involve many significant figures, and large differences in magnitude
      - Verify that the calculated results are correct
  - Ordering operations poorly so errors build up
    - Defect:
      - A large number does not store enough significant figures to be able to accurately represent the result
    - Testing Strategies:
      - Make sure the program works with inputs that have large positive and negative exponents
      - Have the program work with numbers that vary a lot in magnitude
        - Make sure computations are still accurately performed
  - Assuming a floating point value will be exactly equal to some other value
    - Defect:
      - If you perform an arithmetic calculation on a floating point value, then the result will very rarely be computed exactly
      - To test quality, you should always test if it is within a small range around that value
    - Testing Strategies:
      - Standard boundary testing should detect this type of defect
- Defects in Timing and Co-ordination
  - Deadlock and livelock
    - Defects:
      - A deadlock is a situation where two or more threads are stopped, waiting for each other to do something
        - The system is hung
      - Livelock is similar, but now the system can do some computations, but can never get out of some states
    - Testing Strategies:
      - Deadlocks and livelocks occur due to unusual combinations of conditions that are hard to anticipate or reproduce

            - It is often most effectual to use inspection to detect such defects, rather than testing alone
            - However, when testing
                - Vary the time consumption of different threads
                - Run a large number of threads concurrently
                - Deliberately deny resources to one or more threads
    - Critical races
        - Defects:
            - One thread experiences a failure because another thread interferes with the 'normal' sequence of events
        - Testing Strategies:
            - It is particularly hard to test for critical races using black box testing alone
            - One possible, although invasive, strategy is to deliberately slow down one of the threads
            - Use inspection
    - Semaphore and synchronization
        - Critical races can be prevented by locking data so that they cannot be accessed by other threads when they are not ready
            - One widely used locking mechanism is called a semaphore
            - In Java, the synchronized keyword can be used
                - It ensures that no other thread can access an object until the synchronized method terminates
- Defects in Handling Stress and Unusual Situations
    - Insufficient throughput or response time on minimal configurations
        - Defect:
            - On a minimal configuration, the system's throughput or response time fails to meet requirements
        - Testing Strategy:
            - Perform testing using minimally configured platforms
    - Incompatible with specific configurations of hardware or software
        - Defect:
            - The system fails if it is run using particular configurations of hardware, operating systems, and external libraries
        - Testing Strategy:
            - Extensively execute the system with all possible configurations that might be encountered by users
    - Defects in handling peak loads or missing resources
        - Defects:
            - The system does not gracefully handle resource shortage
            - Resources that might be in short supply include:
                - Memory, disk space or network bandwidth, permission
            - The program being tested should report the problem in a way the user will understand
        - Testing Strategies:
            - Devise a method of denying resources
            - Run a very large number of copies of the program being tested, all at the same time
    - Inappropriate management of resources
        - Defect:
            - A program uses certain resources but does not make them available when it no longer needs them
        - Testing Strategy:

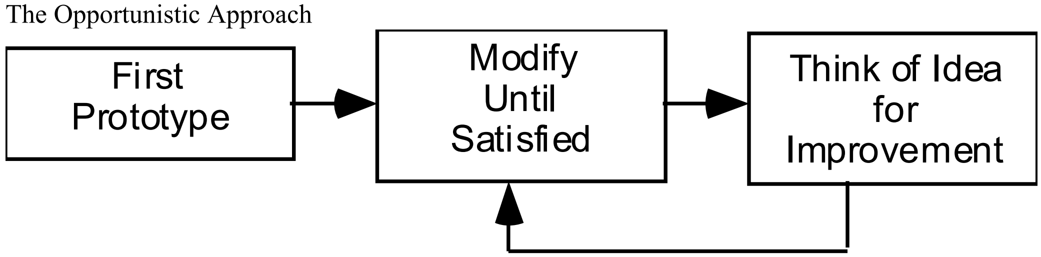
        - Run the program intensively in such a way that it uses many resources, relinquishes them and then uses them again repeatedly
  - Defects in the process of recovering from a crash
    - Defects:
      - Any system will undergo a sudden failure if its hardware fails, or if its power is turned off
      - It is a defect if the system is left in an unstable state and hence is unable to fully recover
      - It is also a defect if a system does not correctly deal with the crashes of related systems
    - Testing Strategies:
      - Kill a program at various times during execution
      - Try turning the power off, however operating systems themselves are often intolerant of doing that
- Documentation Defects
  - Defect:
    - The software has a defect if the user manual, reference manual, or on-line help:
      - Gives incorrect information
      - Fails to give information relevant to a problem
  - Testing Strategy:
    - Examine all the end-user documentation, making sure it is correct
    - Work through the use cases, making sure that each of them is adequately explained to the user
- Writing Formal Test Cases and Test Plans
  - A test case is an explicit set of instructions designed to detect a particular class of defect in a software system
  - A test plan is a document that contains a complete set of test cases for a system
      - Along with other information about the testing process
    - The test plan is one of the standard forms of documentation
    - The test plan should be written long before testing starts
- Strategies for Testing Large Systems
  - Big bang testing versus integration testing
    - In big bang testing, you take the entire system and test it as a unit
    - A better strategy in most cases is incremental testing
      - You test each individual subsystem in isolation
      - Continue testing as you add more and more subsystems to the final product
  - Top-down Testing
    - Start by testing just the user interface
    - The underlying functionality are simulated by stubs
    - Then you work downwards, integrating lower and lower layers
    - The big drawback to top-down testing is the cost of writing the stubs
  - Bottom-up Testing
    - Start by testing the very lowest levels of the software
    - You need drivers to test the lower layers of software
    - Drivers in bottom-up testing have a similar role to stubs in top-down testing, and are time-consuming to write
  - Sandwich Testing
    - A hybrid between bottom-up and top-down testing
    - Test the user interface in isolation, using stubs
    - Test the very lowest level functions, using drivers
    - When the complete system is integrated, only the middle layer remains on which to perform the final set of tests
- The Test-Fix-Test Cycle

- When a failure occurs during testing:
    - Failure is reported into a failure tracking system
    - Screened, assigned a priority
    - Low-level priorities are often put in a known-bugs list and released with the software to be fixed later
    - Someone is assigned to investigate the failure
    - That person tracks down the defect and fixes it
    - Finally a new version of the system is created, ready to be tested again
- The Ripple Effect
  - There is a high probability that the efforts to remove the defects may have actually added new defects
- Regression Testing
  - It tends to be far too expensive to re-run every single test case every time a change is made to the software
  - Hence only a subset of the previously-successful test cases is actually re-run
  - This is called regression testing
  - The "law of the conservation of bugs"
    - The number of bugs remaining in a large system is proportional to the number of bugs already fixed
- Inspections
  - An inspection is an activity in which one or more people systematically
    - Examine source code or documentation, looking for defects
    - Normally, inspection involves a meeting…
      - Although participants can also inspect alone at their desks

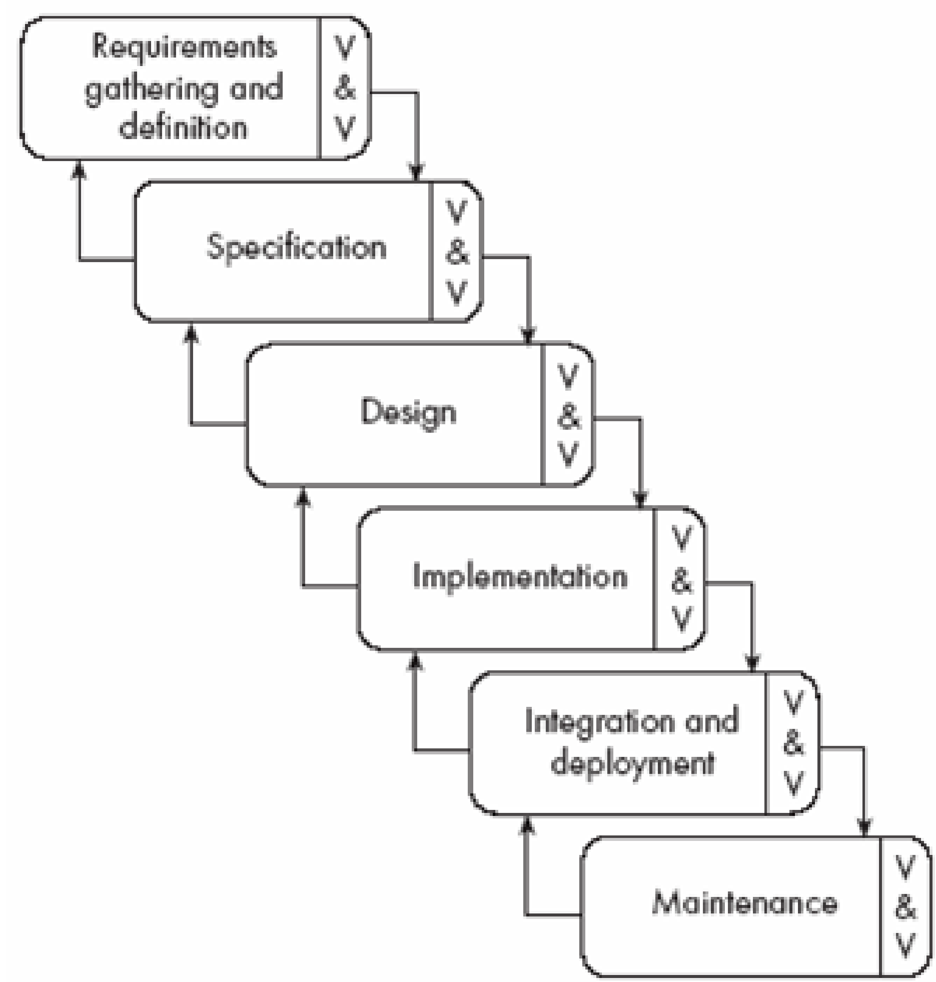
Chapter 11

- What is project management?
  - Project management encompasses all the activities needed to plan and execute a project:
    - Deciding what needs to be done
    - Estimating costs
    - Ensuring there are suitable people to undertake the project
    - Defining responsibilities
    - Scheduling
    - Making arrangements for work
    - Directing
    - Being a technical leader
    - Reviewing and approving decisions made by others
    - Building morale and supporting staff
    - Monitoring and controlling
    - Coordinating the work with managers of other projects
    - Reporting
    - Continually striving to improve the process
- Software Process Models
  - Software process models are general approaches for organizing a project into activities
    - 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 Opportunistic Approach

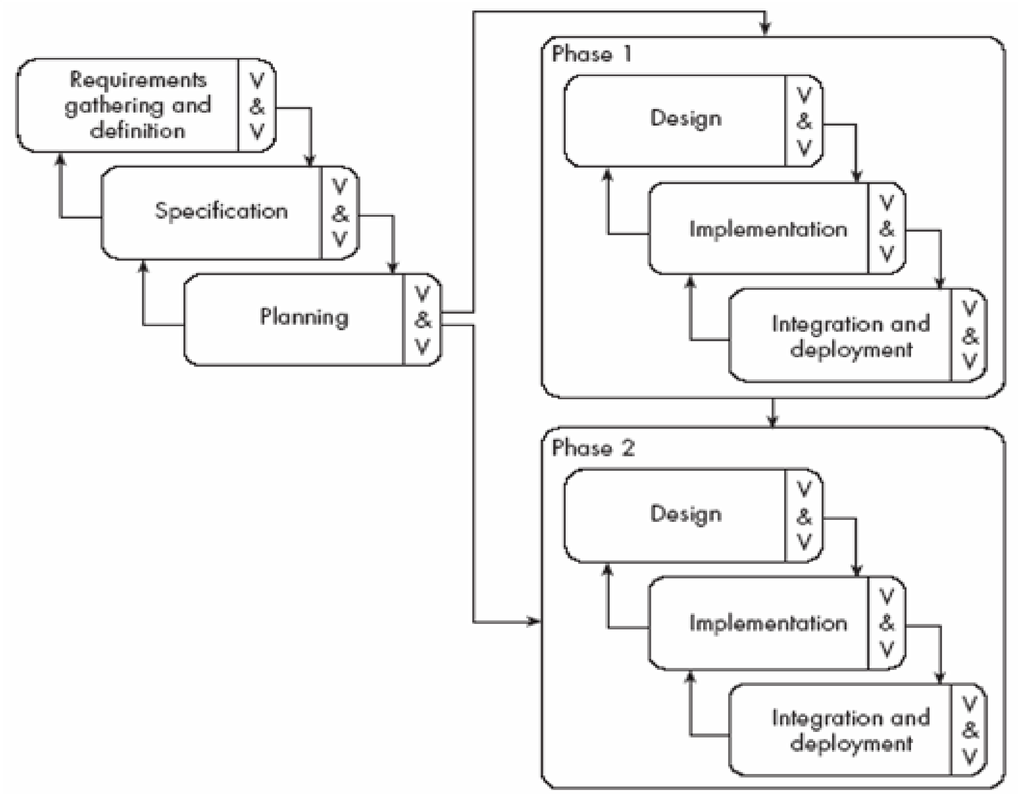


  - … is what occurs when an organization does not follow good engineering practices
    - It does not stress the importance of working out requirements and a design first
    - The design of software deteriorates if it is not well designed
    - No plans = no aim
    - No recognition of a need for systematic testing
- The Waterfall Model



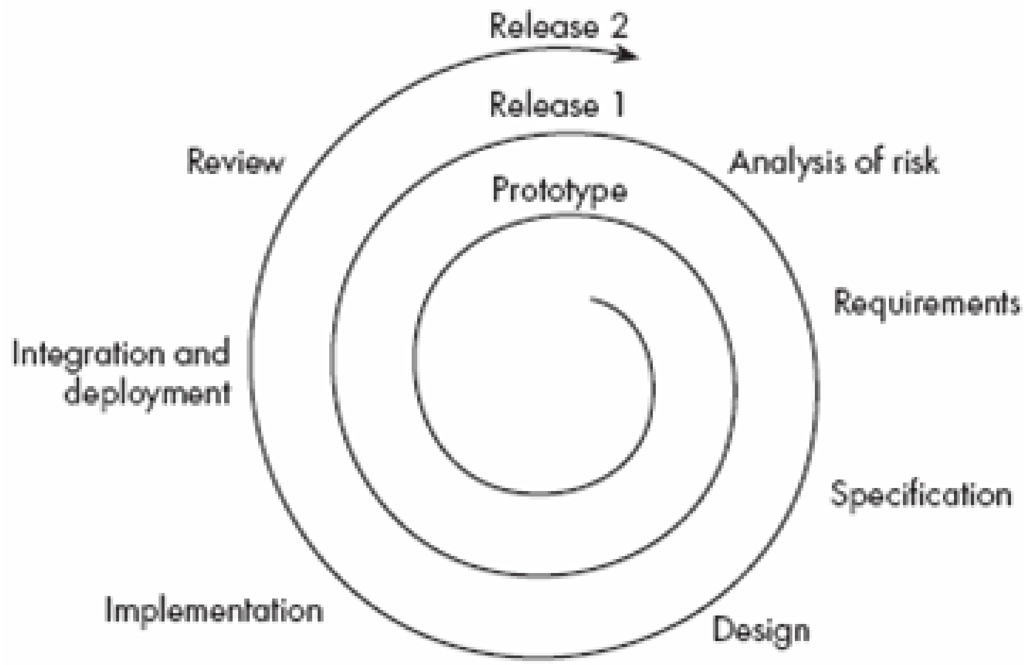
  - The classic way of looking at S.E. that accounts for the importance of requirements, design and quality assurance
    - The model suggests that software engineers should work in a series of stages
    - Before the complete each stage, they should complete quality assurance (verification and validation)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

        - Suggests you should complete a stage before moving on to the next
            - Doesn't account for requirements changing
            - Customers cannot use anything until the entire system is complete
        - Makes no allowances for prototyping
        - Implies you can get the requirements right by simply writing them down and reviewing them
        - Implies that once the product is finished, everything else is maintenance
- The Phased-Release Model

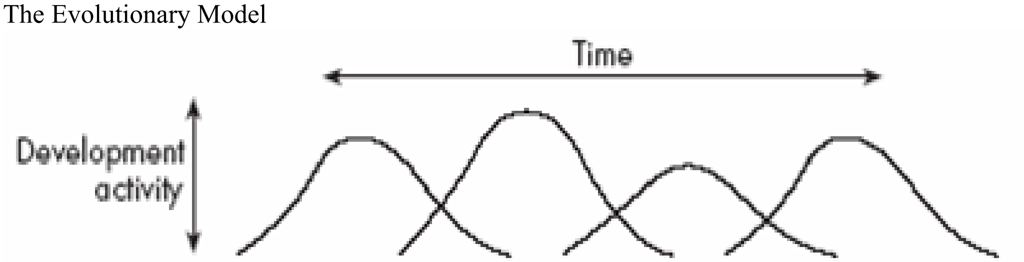


    - It introduces the notion of incremental development
        - After requirements gathering and planning, the project should be broken up into separate subprojects, or phases
        - Each phase can be released to customers when ready
        - However, continues to suggest that all requirements be finalized at the start of development

- The Spiral Model

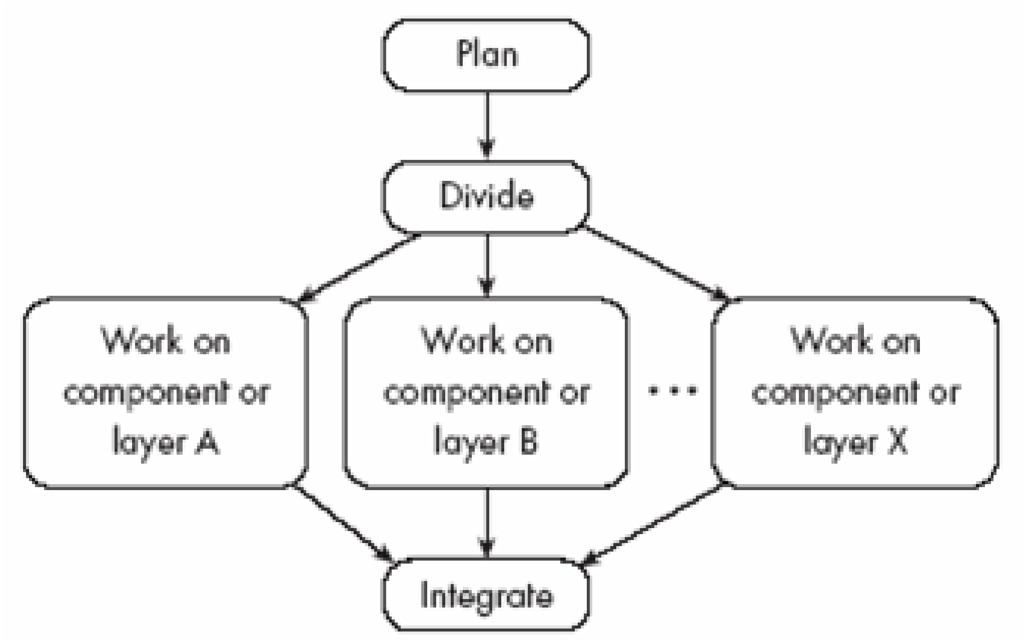


  - It explicitly embraces prototyping and an iterative approach to software development
    - Start by developing a small prototype
    - Followed by a mini-waterfall process, primarily to gather requirements
    - Then, the first prototype is reviewed
    - In subsequent loops, the project team performs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



  - It explicitly accounts for the divide and conquer principle
- Choosing a Process Model
  - From the Waterfall Model:
    - Incorporate the notion of stages
  - From the Phased-Release Model:
    - Incorporate the notion of doing some high-level analysis, and then dividing the project into releases
  - From the Spiral Model:
    - Incorporate prototyping and risk analysis
  - From the Evolutionary Model:
    - Incorporate the notion of varying amounts of time and work, with overlapping releases
  - From the Concurrent Engineering Model:
    - Incorporate the notion of breaking the system down into components and developing them in parallel