1. Study Guide
2. **Basic characteristics of software engineering**
3. **Types of software and its desired attributes**
   1. Types of Software
      1. Generic SW
         1. Stand-alone systems and sold on the open market to any customer
         2. Company that develops the SW controls the SW Specification
      2. Customized SW
         1. Commissioned by a customer
         2. Developed especially for this customer
         3. SW Specification is developed and controlled by the customer
   2. Desired Attributes
      1. Maintainability
         1. SW should be able to evolve in order to meet changing demands
      2. Dependability and Security
         1. Dependability includes Reliability, Safety, and Security
         2. Dependable = should not physical or economic damage as a result of failure
         3. Security = malicious users should not be able to access or damage the system
      3. Efficiency
         1. SW should not be wasteful of resources
      4. Acceptability
         1. SW must be acceptable to its target users
4. **Software engineering ethics**
   1. Confidentiality
      1. Respect confidentiality of employers and customers
   2. Competence
      1. Never misrepresent your level of competence
   3. Intellectual Property Rights
      1. Beware of local laws governing the use of intellectual property like patents and copyrights.
      2. Ensure intellectual property is protected
   4. Computer Misuse
      1. Never misuse technical skills to mess around or to harm people.
5. **Software process models and process activities**
   1. Process Models
      1. Waterfall Model
         1. Takes the fundamental process activities of specification, development, validation, and evolution and represents them as process phases such as:
            1. Requirements Definition
            2. System and Software Design (design the entire architecture)
            3. Implementation and Unit Testing
            4. Integration and System Testing (all the programs are integrated and tested)
            5. Operation and Maintenance (put into use in market and any bugs are corrected)
         2. It is Plan-driven
         3. Each phase depends on the previous phase
         4. Weakness: its inflexible to partition the project into distinct phases, commitments must be made at an early stage in the process.
         5. This should only be used when requirements are well understood and unlikely to change radically during development.
      2. Incremental Developments
         1. Interleaves the activities of specification, development, and validation. The systems are developed as a series of version (increments), with each version adding functionality to the previous version
         2. Is Agile-Driven
         3. Developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been develop.
         4. Specification, Development, and Validation activities are interleaved rather than separate.
         5. Advantages: Cheaper and easier to make changes, don't have to define all requirements in the beginning, rapid delivery, and easier to get customer feedback
         6. Disadvantages: It is not feasible to create documents for each release and System structure tends to degrade as new incremented are added
      3. Reuse-oriented Software Engineering
         1. Based on the existence of a significant number of reusable components. The SW development process focuses on integrating these components into a system rather than developing them from scratch.
         2. Phases
            1. Requirements Specification
            2. Component Analysis

Search for component to implement the specification

* + - * 1. Requirements Modification

Requirements modified to suit the component analyzed

* + - * 1. System Design with Reuse

System is designed or an existing framework is reused

* + - * 1. Development and Integration
        2. System Validation
  1. Process Activities
     1. SW Specification
        1. The process of understanding and defining what services are required from the system and identifying the constraint on the systems operation and development
        2. Functionality of the SW and its constraints must be defined
        3. Presented at 2 levels
           1. End-users and Customer
           2. Developers
        4. 4 Main Activities
           1. Feasibility study

Decides whether it is feasible to go ahead with a more detailed analysis

* + - * 1. Requirements elicitation and analysis

Other existing systems are inspected and discussions with potential users occur here

* + - * 1. Requirements specification

Translating the information gathered in previous stage into a document (1 is User Requirements for the customer and 1 is System Requirements for the developers)

* + - * 1. Requirements Validation

Requirements are checked for realism, consistency, and completeness.

* + 1. Software Design and Implementation
       1. The SW is designed and is implemented to meet its specification
       2. Design Inputs
          1. Platform Information
          2. Requirements Specification
          3. Data Description
       3. Design Activities
          1. Architectural Design

Design the overall structure of the system and the components it will use and their relationships

* + - * 1. Interface Design

Define the interface between components

* + - * 1. Component Design

Design how each component will operate

* + - * 1. Database Design

Design system data structures and how to represent these in a DB

* + - 1. Design Outputs
         1. System Architecture
         2. Database Specification
         3. Interface Specification
         4. Component Specification
    1. Software Validation
       1. SW is validated to ensure that it does what the customer wants
       2. Goal: To show that the system both conforms to specification and that it meets the customer's expectations
       3. Activities
          1. Component/Development Testing

Each component is tested independently without other system components by the people developing the system.

Components include functions, objects, etc.

* + - * 1. System Testing

Components are integrated and testing is done to find errors that result from interactions between components and component interface problems

* + - * 1. Acceptance Testing

System is tested with data supplied by the system customer

* + 1. Software Evolution
       1. SW must evolve to meet changing customer needs
       2. Activities
          1. Define System Requirements
          2. Assess Existing Systems
          3. Propose System Changes
          4. Modify Systems or Create New System

1. **Managing change in software process**
   1. Change is inevitable
   2. 2 Related approaches to reduce the cost of change
      1. Change Avoidance (by using Prototypes)
      2. Change Tolerance
      3. 2 Ways of coping with change and changing system requirements
         1. System Prototyping -> a version or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions
            1. This is Change Avoidance b/c it allows users to experiment with the system before delivery and therefore refine their requirements. (The number of requirement change proposals made after delivery is therefore likely to be reduced)
            2. Activities

Establish Prototype Objectives (creates Protoyping Plan)

Define Protoype Functionality (creates Outline Definition)

What to include and what to leave out

Develop Prototype (creates Executable Protoype)

Evaluate Prototype (creates Evaluation Report)

* + - * 1. Advantages

Helps validate requirements

Helps explore SW solutions and GUI design

* + - 1. Incremental Delivery ->System increments are given to customer for comment and experimentation
         1. This is Change Avoidance and Change Tolerance.
         2. Avoids premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at low cost
         3. Activities

Define Outline Requirements

Assign Requirements to Increments

Design System Architecture

Develop System Incremental

Validate Incremental

Integrate Incremental

Validate Systematic

Deploy Increment and maybe begin developing next increment

* + - * 1. Advantages

Customer can gain experience with system in increments and learn about more requirements for later increments

Because high-priority services are delivered first, these will be tested heavily at first and also by each increment

* + - * 1. Disadvantages

Difficult to identify common facilities that are needed by all increments

* + - 1. Spiral Model
         1. Activities/Stages

Object Setting

Determine Object, the potential alternatives, and the constraints on the system

Risk Assessment and Reduction

For each risk identified, do a detailed analysis.

Evaluate alternatives and resolve risks and reduce them

Development and Validation

A development model is chosen (i.e. Waterfall, Increment, etc.)

Develop and Verify product

Planning

Review project and decide whether to continue another iteration of the loop

Plan next phase

* + - * 1. THIS COMBINES CHANGE AVOIDANCE AND CHANGE TOLERANCE

1. **Agile methods vs. plan-driven development: characteristics, advantages and disadvantages**
   1. Differences
      1. Plan-driven
         1. Identifies separate stages and has each stage have associated outputs
         2. Formal documents are used to communicate between stages
         3. Must plan and schedule al of the process activities before starting on them
      2. Agile
         1. Considers Design and implementation to be central activities…the other activities are incorporated into these activities
   2. Agile
      1. Types of Development where Agile Development is Successful
         1. Small to medium sized projects
         2. Custom system development within an organization
            1. Where there is clear commitment from the customer
            2. Where there are not a lot external rules and regulations that affect SW
      2. Principles of Agile Methods
         1. Customer Involvement
            1. They should be closely involved throughout the process
         2. Incremental Delivery
         3. People not process
            1. Team member should be left to develop their own ways of working without prescriptive processes
         4. Embrace change
            1. Expect system requirements to change…so design the system to accommodate these changes
         5. Maintain simplicity
      3. Advantages
         1. Accommodates change
         2. Rapid development
      4. Disadvantages with the Principles of Agile Methods
         1. Success depends on a willing and committed customer
         2. Team member may not have suitable personalities for the intense involvement that is typical of agile methods
         3. Prioritizing changes can be extremely difficult
         4. Maintaining simplicity requires extra work
         5. Difficult for plan-oriented companies to move towards an incremental, and informal model
         6. It is difficult to keep the customer involved. Also team members must rely on understanding aspects of the system without consulting documentation (this is much worse for separated teams).
         7. Structure of system tends to degrade over time
         8. Difficult to use on large system development
   3. Plan-Driven
      1. Advantages
         1. Heavily documented
         2. Traceability
      2. Disadvantages
         1. Slow
2. **Extreme programming and Scrum approach**
   1. Extreme Programming
      1. Programmers work in pairs and develop test for each task before writing code
         1. All of the tests must be successful before code is integrated into the system
      2. Requirements are expressed as Scenarios (called User Stories)
      3. Release Cycle
         1. Select User Stories for this Release
         2. Break Down Stories to Tasks
         3. Plan Release
         4. Develop/Integrate/Test SW
         5. Release SW
         6. Evaluate System
         7. Go back to step a (Select User Stories)
      4. Practices
         1. Incremental development is supported through small, frequent releases…requirements are based on User Stories
         2. Customer involvement is supported through interaction between customer and development team
         3. People are supported through pair programming, collective ownership of system code, and sustainable development process that does NOT involve long working hours
         4. Change is embraced
         5. Maintaining simplicity by constant refactoring
      5. More XP Practices
         1. Incremental Planning
         2. Small Releases
         3. Simple Design
         4. Test-first development
         5. Refactoring
         6. Pair Programming
         7. Collective Ownership
         8. Continuous integration
         9. Sustainable pace
         10. On-site customer
      6. XP address the Incremental Issue of system degrading after many changes by constantly forcing the refactoring of code
      7. Testing in XP
         1. Key features of testing gin XP
            1. Test-first development
            2. Incremental test development from scenarios
            3. User involvement in the test development and validation

Customer creates acceptance tests for the stories that are to be implemented in the next release

* + - * 1. The use of automated testing frameworks
      1. Reasons why testing may NOT be thorough
         1. Programmers prefer programming to testing and therefore, may take shortcuts when writing tests
         2. Some tests can be very difficult to write incrementally
         3. It is difficult to judge the completeness of a set of tests
    1. Pair Programming
       1. Programmers work in pairs to develop SW
          1. They actually sit together to develop software
          2. However, the same pairs do not always program together…pairs are dynamically create so that all team members work with each other during the development process
       2. Advantages
          1. Promotes team collective responsibility…individuals are not held responsible for problems within the code
          2. Acts as an informal review process because each line of code is looked at by at least 2 people…it helps speed up code review
          3. It helps support refactoring
       3. People think pair programming is much slower than solo programming
          1. But it is comparable in time (when looking at students) because

Pairs discuss SW before development so they have less false starts and less rework

Errors are avoided thanks to the informal inspection 🡪 less time is spend on debugging

* + - * 1. The results do not hold though for experienced programmers….there is a significant loss of productivity
  1. Scrum Approach
     1. Does not use pair programming or test-first development
     2. 3 Phases in Scrum
        1. Outline Planning and Architecture Design Phase
           1. General objectives for the project and design the SW architecture
        2. Sprint Cycles (this can be repeated)
           1. Each cycle develops an increment of the system
           2. Cycles

Asses

Select

Develop

Review

* + - 1. Project Closure
         1. Completes the required documentation, assesses lessons learned from the project, and wraps up the project
    1. Key Characteristics
       1. Sprints are fixed length
          1. Usually 2-4 weeks
          2. Correspond to development or a release in XP
       2. Starting point for planning is the product backlog (list of work to be done on the project)
          1. During the Assess phase of the Sprint Cycle, this is reviewed, and priorities and risks are assigned.
          2. Customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each sprint
       3. Selection phase in Sprint Cycle involves all of the project team who work with the customer to select the features and functionality to be developed during the Sprint
       4. Once these are agreed, the team organizes themselves to develop SW
          1. Short daily meetings involving all team members are held to review progress and maybe reprioritize work
       5. At the end of the sprint cycle, the work done is reviewed and present to stakeholders. The next cycle then begins
    2. Advantages
       1. Project is broken down in a set of manageable and understandable chunks
       2. Unstable requirements do not hold up progress
       3. The whole team has visibility of everything and consequently team communication is improved
       4. Customer see on-time delivery of increments and gain feedback on how the product works
       5. Trust between customer and developers is established and positive culture is created in which everyone expects the project to succeed

1. **Types of software requirements**
   1. User vs. System Requirements
2. User Requirements: Statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate
   1. These tend to be general
   2. Readers are NOT usually concerned with how the system will be implemented
3. System Requirements: More detailed descriptions of the SW system’s functions, services, and operational constraints. This document (aka functional specification) should define exactly what is to be implemented.
   1. These tend to be more specific
   2. Readers need to know more precisely what the system will do
   3. Function vs. Non-Functional Requirements
      1. Functional:
         1. Statements of services the system should provide
         2. How the system should react to particular inputs
         3. How the system should behave in particular situation
         4. Maybe, state what the system should NOT do
         5. Describe what the system should do
         6. Define specific facilities to be provided by the system (these have been taken from the User Requirements document)
         7. Should be COMPLETE and CONSISTENT
            1. Completeness: All services required by the user should be define
            2. Consistency: Requirements should not have contradictory definitions
      2. Non-Functional:
         1. Constraints on the services or functions offered by the system
         2. Timing constraints
         3. Constraints on development process
         4. Constraints imposed by standards
         5. Requirements that are not directly concerned with the specific services delivered by the system to its users
         6. Usually deal with performance, security, availability, reliability, response time, and store occupancy requirements/constraints.
         7. It is difficult to relate components to Non-functional requirements (it is easier to relate component to functional requirements) because:
            1. Non-functional requirements may affect the overall architecture of a system rather than individual components (i.e. Performance)
            2. A single non-functional component, such as security requirement, may generate a number of related functional requirements
         8. Types of Non-Functional Requirements
            1. Product Requirements

Specify or constrain the behavior of the software (Ex. Performance requirements, reliability requirements, security requirements, and usability requirements)

* + - * 1. Organizational Requirements

Broad system requirements derived from policies and procedures in the customer’s and developer’s organization. (Ex. Operational process requirements, development process requirements, and environmental process requirements)

* + - * 1. External Requirements

All requirements that are derived from factors external to the system and its development process (Ex. Regulatory requirements, legislative requirements, and ethical requirements)

* + - 1. Metrics for Specifying Non-Functional Requirements
         1. You should write Non-Functional Requirements quantitatively so that they can objectively tested
         2. Metrics

Speed

Processed transactions/second, Response time, Screen refresh time

Size

Mbytes, # of ROM chips

Ease of Use

Training time, Number of help frames

Reliability

Mean time to failure, Probability of unavailability, rate of failure occurrence, and availability

Robustness

Time to restart after failure, % of events causing failure, probability of data corruption on failure

Portability

% of target dependent statements, # of target systems

Non-Functional requirements often conflict with other Non-Functional and Functional Requirements

1. **Approaches to requirements specification**
   1. Writing Requirements Specifications
      1. User
         1. Written in natural language with appropriate diagrams and tables
      2. System
         1. Natural Language sentences
            1. Written using numbered sentences each expressing one requirement
         2. Structured natural language
            1. Written in natural language on a standard form or template. Each field provides info about an aspect of the requirement
         3. Design description language
            1. Uses language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system.
         4. Graphical notations
            1. Graphical models like UML
         5. Mathematical specifications
            1. Based on mathematical concepts such as finite state machines
2. **Activities (phases, stages) of requirements engineering process**
   1. Requirements Engineering 4 Phases
      1. Feasibility Study
         1. Is system useful to the business
      2. Elicitation and Analysis
         1. Discovering requirements
      3. Specification
         1. Converting these requirements into some standard form
      4. Validation
         1. Checking that the requirements actually define the system that the customer wants
   2. Requirements engineering is an iterative process in which the activities are interleaved in a spiral…the output is the System Requirements Document.
      1. This spiral accommodates approaches to development where the requirements are developed to different levels of detail
      2. The # of iterations around the spiral can vary so it can be exited after some or all of the user requirements have been elicited
      3. Agile development can be used so that the requirements and the system implementation are developed together
3. **Types of software system models**
   1. System Modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. Models are used during the Requirements Engineering process to help derive the requirements for the system, during the design process to describe the system to engineers implementing the system and after implementation to document the system’s structure and operation.
   2. Different Perspectives from which to create different models to represent the system:
      1. External/Context Perspective
         1. You model the context or environment of the system
      2. Interaction Perspective
         1. You model the interactions between a system and its environment or between the components of a system
      3. Structural Perspective
         1. You model the organization of a system or the structure of the data that is processed by the system
      4. Behavioral Perspective
         1. Where you model the dynamic behavior of the system and how it responds to events
   3. 5 Diagrams from UML that could represent the essentials of a system
      1. Activity Diagrams: show the activities involved in a process or in data processing
      2. Use Case Diagrams: show the interactions between a system and its environment
      3. Sequence Diagrams: Show interactions between actors and the system and between system component
      4. Class Diagrams: show the object classes in the system and the associations between these classes.
         1. State Diagrams: Show how the system reacts to internal and external events.
   4. External/Context Models
      1. Context models normally show that the environment includes several other automated systems
      2. Context models do not show the types of relationships between the systems in the environment and the system that is being specified
      3. These models are often used with other models such as Business Process Models (which are Activity Diagrams).
         1. Activity Diagrams are intended to show that activities that make up a system process and the flow of control from one activity to another. These Diagrams also show which systems in the system and/or environment take care of certain activities
   5. Interactional Models
      1. Models User Interaction and System-to-System Interaction and Component Interaction.
      2. 2 Approaches to Interaction Modeling
         1. 1) Use Case Modeling
            1. Used to model interactions between a system and external actors (users or other systems)
         2. 2) Sequence Diagrams
            1. Used to model interactions between system components, although external agents may also be included
      3. Use Case Modeling (Section 5.2.1)
         1. Used to support Requirements Elicitation
         2. They give a fairly simple overview of an interaction so you have to provide more detail to understand what is involved. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram.
      4. Sequence Diagrams (Section 5.2.2)
         1. Used to model the interactions between the actors and objects in a system and the interactions between the object themselves.
         2. Shows the sequence of interactions that take place during a particular use case or use case instance.
         3. The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these. Interactions between objects are indicated by annotated arrows. The rectangles on the dotted lines indicate the lifeline of the objects concerned. The annotations of the arrows indicate the calls to the objects, their parameters, and their return values.
         4. Unless you are using sequence diagrams for code generation or detailed documentation, you don’t have to include every interaction in these diagrams.
            1. If you develop system models early in the development process to support requirements engineering and high-level design, there will be many interactions which depend on implementation decisions.
   6. Structural Models
      1. These display the organization of a system in terms of the components that make up that system and their relationships.
      2. Class Diagrams
         1. Used when developing an Object-Orient system model to show that classes in the system and the associations between these classes.
         2. An association is a link between classes that indicates that there is a relationship between these classes. These are shown in diagrams with simple lines and number that show 1-1 or 1-n or m-n relationships.
         3. Can also be used to describe databases
         4. Sometimes these diagrams can omit class details such as attributes and methods
            1. If they are included, the name of the object class is at the top of the box, the attributes in the middle (this must include attribute names and optionally their types), and operations in the lower section
   7. Behavioral Models
      1. Are models of the dynamic behavior of the system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.
      2. 2 Types of Stimuli
         1. Data
            1. Some data arrives that has to be processed by the system
         2. Events
            1. Some event happens that triggers system processing. Events may have associated data but this is not always the case.
      3. Data Driven Modeling (Section 5.4.1)
         1. Shows the sequence of actions involved in processing input data and generating an associate output. They show the entire sequence of actions that take place from an input being processed to the corresponding output, which is the system’s response.
         2. UML Sequence Diagrams can also be used here but you must structure them so that messages are only sent from the left to the right.
      4. Event Driven Modeling (Section 5.4.2)
         1. Shows how a system responds to external and internal events
         2. Based on the assumption that a system has a finite number of states and that events may cause a transition from one state to another.
         3. State Diagrams show system states and events that cause transitions from one state to another. They do NOT show the flow of data within the system.
         4. Each state in the diagram must include a brief description of the actions taken in that state.
         5. In addition to the state diagram, you have to provide more detail about both the stimuli and the system states. (You can use tabular descriptions)
4. **Model-driven software engineering approach**
   1. Is an approach to SW Development where models rather than program are the principal outputs of the development process. The programs that execute on hardware/software platform are then generated automatically from the models.
   2. Model Driven Architecture focuses on the design and implementation stages of software development whereas Model Driven Engineering is concerned with all aspects of the SW Engineering Process.
   3. Arguments for and against Model Driven Engineering
      1. For
         1. Allows engineers to think about system at a high level of abstraction, without concern for the details of their implementation.
         2. This reduces the likelihood of errors, speeds up the design and implementation process, and allows for the creation of reusable, platform-independent application models
      2. Against
         1. Models are a good way of facilitating discussions about SW design…It does NOT always follow that the abstractions that supported by the model are the right abstractions for the implementation.
         2. Arguments for platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system’s lifetime. For this class of system, we know that implementation is not the major problem-requirements engineering, security and dependability, integration with legacy systems, and testing are more significant
   4. Model Driven Architecture (Section 5.5.1)
      1. Model-focused approach of SW Design and Implementation that uses a subset of UML models to describe a system.
      2. 3 Types of Abstract System Model should be produced
         1. 1) Computation independent model (CIM)
            1. Models the important domain abstractions used in the system
         2. 2) Platform Independent Model (PIM)
            1. Models the operation of the system without reference to its implementation
         3. 3) Platform Specific Models (PSM)
            1. Transformations of the PIM with a separate PSM for each application platform. There may be layers of PSM, with each layer adding some platform-specific detail.
      3. Transformations between these models may be defined and pplied automatically by software tools
      4. CIM to PIM translation is till at the research prototype stage
      5. PIM to PSM is mature and several communication tools are available that provide translators from PIMs to common platforms such as Java and J2EE.
   5. Executable UML (Section 5.5.2)
      1. Completely automated transformation of models to code
      2. To create an executable subset of UML, the number of model types has therefore been dramatically reduced to 3 key model types:
         1. Domain models identify the principal concerns in the system. These are defined using UML class diagrams that include objects, attributes, and associations.
         2. Class models, in which classes are define, along with their attributes and operations
         3. State models, in which a state diagram is associated with each class and is used to describe the lifecycle of the class
5. **Fundamental characteristics of architectural design**
   1. Architectural Design is concerned with understanding how a system should be organized and designing the overall structure of that system.
      1. It is the 1st stage in the SW Design Process
      2. The output of this process is an architectural model that describes how the system ir organized as a set of communicating components.
   2. You can design SW Architectures at 2 Levels of Abstraction:
      1. Architecture in the Small
         1. Concerned with the architecture of individual programs.
         2. Concerned with the way that an individual program is decomposed into components
      2. Architecture in the Large
         1. Concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.
   3. 3 Advantages of Explicitly Designing and Documenting System Architecture
      1. Stakeholder Communication
         1. The architecture is a high-level presentation of the system that can be used as a focus for discussion
      2. System Analysis
         1. Making the system architecture explicit at an early stage in the development requires some analysis. These decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, etc.
      3. Large-scale Reuse
         1. System architecture is often the same for systems with similar components
   4. Because of the close relationship between Non-Functional Requirements and Software Architecture. The particular Architecural Style/Pattern and Stucture that you choose for a system should depend on the Non-Functional Requirements:
      1. Performance
         1. If this is a critical requirement, architecture should localize critical operations within a small number of components on the same computer
      2. Security
         1. If this is a critical requirement, a layered structure for the architecture should be used
      3. Safety
         1. If this is a critical requirements, all safety-related operations should be held in one component
      4. Availability
         1. If this is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system
      5. Maintainability
         1. If this is a critical requirement, the architecture should be designed with fine-grained, self-contained components that can easily change
   5. There should be 4 fundamental views
      1. 1) Logical View
         1. Show key abstractions in the system as objects or object classes
      2. 2) Process View
         1. Shows how at run-time they system is composed of interacting processes
      3. 3) Development View
         1. Shows how the SW is decomposed for development…shows the breakdown of the SW into components that are implement by a single developer or development team
      4. 4) Physical View
         1. Shows the system hardware and how software components are distributed across processors in the system.
6. **Types of architectural patterns; their advantages and disadvantages**
7. These are stylized, abstract description of good practice/methods, which has been tried and tested in different systems and environments.
8. Layered Architecture (Section 6.3.1)
   1. Used to achieve separation and independence.
   2. Supports incremental development
   3. Info
      1. Description
         1. Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system
      2. When Used
         1. When incorporating new functionality on top of existing systems
         2. When development is spread across teams each assigned a layer
         3. When there is a need for multi-level security
      3. Advantages
         1. Can add/remove layers as long the interface is the same
      4. Disadvantages
         1. Is difficult
         2. Layers may need to skip layers in order to interact with other layers
9. Repository Architecture (Section 6.3.2)
   1. Shows how a set of interacting components can share data
   2. Info
      1. Description
         1. All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository
      2. When Used
         1. When you have a system in which large volumes of info is gathered and needs to be stored for a long time
         2. You it also when the inclusion of data in the repository triggers an action or tool
      3. Advantages
         1. Components can be independent
         2. Changes made to one component can be propogated to all components
         3. All data can be managed consistently
      4. Disadvantages
         1. Repository is a single point of failure
         2. Can be inefficient in organizing all communication through the repository
10. Client-Server Architecture (Section 6.3.3)
    1. Concerned with the static structure of the system and does not show its run-time organization.
    2. Organized as a set of services and associated servers, and clients that access and use the services
    3. Major Components
       1. A set of servers that offer services to other components
       2. A set of clients that call of the services offered by servers
       3. A network that allows the clients to access these services
       4. Info
          1. Description
             1. The functionality of the sytem is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them
          2. When Used
             1. Used when data in a shared database has to be accessed from a range of locations
          3. Advantages
             1. Servers can be distributed across a network
             2. General functionality can be available to all clients and does not need to be implement by all services
          4. Disadvantages
             1. Each service is a single point of failure so susceptible to denial of service attacks or server failure
             2. Performance may be unpredictable due to the network
11. Pipe and Filter Architecture (Section 6.3.4)
    1. Model of Run-Time organization of a system where functional transformations process their inputs and produce outputs
    2. Data flows from one to another and is transformed as it moves through the sequence
    3. Info
       1. Description
          1. Processing of the data in a system is organized so that each processing component (filter) I discrete and carries one type of data transformation. The data flows (pipe) from one component to another for processing
       2. When used
          1. Commonly used in data processing applications
       3. Advantages
          1. Easy to understand and supports transformation reuse
          2. Can easily evolve
       4. Disadvantages
          1. Format for data transfer must be agreed upon
          2. Overhead exists because each process must parse it input and unparsed its output
12. **Types of application systems architectures and their characteristics**
    1. 2 Types of Application Architecture
       1. Transaction processing applications
          1. Database centered applications that process user requests for information and update the info in a database
       2. Language Processing Systems
          1. System in which the user’s intentions are expressed in a formal language. The system processes this language into an internal format and then interprets this internal representation.
             1. Ex. Compilers
    2. Transaction Processing Systems (Section 6.4.1)
       1. Designed to process user requests for information from a database, or requests to update a database.
       2. Changes will only be record if they finish successfully
          1. Ex. Until all steps have been finished by users at ATM like enter card, pin, money wanted, etc. the database won’t record the deposit or withdrawal.
       3. Can be organized as PIPE AND FILTER ARCHITECTURES
    3. Information System (Section 6.4.2)
       1. Can be organized as Client Server Architecture
    4. Language Processing System (Section 6.4.3)
       1. Translate a natural or artificial language into another representation of that language, and for, programming languages, may also execute the resulting code.
       2. Can be used with a Pipe and Filter Architecture or Repository Architecture
13. **Stages of object-oriented design**
    1. Stages
       1. System Context and Interactions
       2. Architectural Design
       3. Object Class Identification
       4. Design Models
       5. Interface Specification
    2. System Context and Interactions
       1. Need to develop and understanding of the relationships between the SW that is being designed and its external environment.
          1. This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment
          2. Understanding of the context also lets you establish the boundaries of the system also lets you establish the boundaries of the system which will help you decide what features are implemented in the system being designed and what features are in other associated systems.
          3. System context models and interaction models provide complementary views of the relationships between a system and its environment
             1. System Context Model

A structural model that demonstrates the other systems in the environment of the system being developed

May be represented using associations

Associations simply show that there are some relationships between the entities involved in the association

* + - * 1. Interaction Model

Dynamic Model that shows how the system interacts with its environment as it is used

Should be abstract and avoid too much detail

You can use an Use Case Model

Each possible interaction is named in an ellipse and the external entity involved in the interaction is represented by a stick figure

Each Use Case should be described in structured natural language that includes

System (name of system)

Use Case (name of use case)

Actors

Dat (summarizes what is supposed to happen)

Stimulus

Response

Comments

* 1. Architectural Design
     1. In this step, you identify the major components of your system and their interactions and then many organize the components using an architectural pattern such as Layered or Client-Server model
  2. Object Class Identification
     1. Purpose here is to identify the objects in your system
     2. How to do this…3 approaches
        1. Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs
        2. Use tangible entities (things) in the application domain such as aircraft, roles (doctor, etc.), events such as requests, interactions such as meetings, locations such as offices, and so on.
        3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations
     3. You can use your knowledge of the application domain to identify the objects
     4. After finding the objects, you can refine the design by trying to incorporate them into an inheritance hierarchy.
  3. Design Models
     1. These models show the objects or object classes in a system. They show the associations and relationships between these entities.
     2. These models form a bridge between system requirements and the implementation of a system.
     3. When you use the UML to develop a design, you will normally develop 2 kinds of design model:
        1. Structural Models
           1. Describe the static structure of the system using object classes and their relationships
           2. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.
        2. Dynamic Models
           1. Describe the dynamic structure of the system and show the interactions between the system objects
           2. Interactions that may be documented include the sequence of service requests made by objects and the state changes that are triggered by these object interactions
     4. 3 Models are useful for adding detail to use case and architectural models
        1. Subsystem Models
           1. Show logical groupings of objects into coherent subsystems
           2. Shows how the system is organized into groups of related objects
           3. These are Static (Structural) Models
        2. Sequence Models
           1. Show the sequence of object interactions
           2. These are represented using a UML sequence or a collaboration diagram
           3. When documenting a design, you should produce a sequence model for each significant interaction. (There should be a Sequence model for each Use Case Model)
           4. Used to model the combined behavior of a group of object
           5. Sequence Models are Dynamic Models
        3. State Machine Models
           1. Show how individual objects change their state in response to events
           2. Represented in the UML using state diagrams
           3. Summarize the behavior of an object or a subsystem in response to messages and events.
           4. State Machine Models are Dynamic Models
  4. Interface Specification
     1. Specification of interfaces between components in the design
     2. This is needed so that objects and subsystems can be designed in parallel
     3. Interface design is concerned with specifying the detail of the interface to an object or to a group of objects.
     4. Should not reveal details of data representation in interface design

1. **Design patterns**
   1. A pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. It is a well-tried solution to a common problem
   2. Patterns include the following sections
      1. Pattern name
      2. Description
      3. Problem Description
      4. Solution Description
      5. Consequences
   3. Gang of Four define 4 essential elements
      1. A name that is meaningful to the pattern
      2. A description of the problem area that explains when the pattern may be applied
      3. A solution description
      4. A statement of the consequences
2. **Software reuse issues; configuration management; open source development**
   1. Implementation issues include
      1. Reuse
      2. Configuration management
      3. Host-Target Development
   2. Reuse
      1. SW Reuse is possible at different levels
         1. Abstraction Level
            1. Instead of reusing software directly, you use knowledge of successful abstractions in the design of your software. Examples are Design and Architectural Patterns
         2. Object Level
            1. Directly reuse objects from a library rather than writing the code yourself.
            2. You must find libraries and discover if the objects and functions deliver the functionality you need
         3. Component Level
            1. Reuse components (are collections of objects that cooperate with each other to provide related functions and services). You often have to adapt and extend the component by adding some code of your own
         4. System Level
            1. Reuse entire application systems
            2. Usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a SW product line) or by using the system’s own configuration interface.
      2. Advantages
         1. Develop new systems more quickly
         2. Cheaper
         3. Lower risk
      3. Costs associated with Reuse
         1. Costs of time spent looking for SW to reuse and assessing whether or not is meets your needs. You may have to tests the SW to make sure that it will work in your environment
         2. Costs of buying the reusable SW
         3. Costs of adapting and configuring the reusable SW components or systems to felect the requirements of the system that you are developing
         4. Costs of integrating reusable SW elements with each other and with new code that you have developed
   3. Configuration Management
      1. Ex. Of why this is important
         1. When people are developing SW, you have to make sure that team members don’t interfere with each other’s’ work. If two people are working on the same component, their changes must be coordinated.
         2. You must also ensure that everyone can access the most up-to-date versions of SW components
         3. When something goes wrong, you should be able to recover to a previous version of the system
      2. Is the process of managing a changing software system
      3. Goal is to support the system integration process so that all developers can access the project code and documents ina controlled way, find out what changes have been made, and compile and link components to create a system.
      4. 3 fundamental configuration management activities
         1. Version management
            1. Support is provided to keep track of the different versions of SW components. T
         2. System Integration
            1. Support is provided to help developers define what versions of components are used to create each version of the system. This description is then used to build a system automatically by compiling and linking the required components
            2. Ex is Unix/Linux Make file
         3. Problem Tracking
            1. Support is provided to allow users to report bugs and other problems, and to allow developers to see who is working on these problems
   4. Open-Source Development
      1. An approach to development in which the source code of a software system is published and volunteers are invited to participate in the development process.
      2. Usually cheap to acquire open source SW
      3. 2 Main open sources issues
         1. Should the product that is being developed make use of open source components?
         2. Should an open source approach be used for the software’s development?
      4. Open Source Licensing
         1. Legally, the developer of the code still owns to code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
         2. 3 General Models
            1. GNU General Public License (GPL)

“Reciprocal”

If you use open source software that is licensed under the GPL license, then you must make that SW open source also.

* + - * 1. GNU Lesser General Public License (LGPL)

You can write components that link to open source code without having to publish the source code of these components. But if you change the licensed component, then you must publish this as open source

* + - * 1. Berkeley Standard Distribution License (BSD)

Non-reciprocal

Not obliged to republish any changes to modifications made to open source code. You can include the code in proprietary systems that are sold. If you use open source components, you must acknowledge the original creator of the code

* + - * 1. Companies that manage projects that use open source code should:

Establish a system for maintaining info about open source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used.

Be aware of the different types of licenses and understand how a component is licensed before it is used

Educate people about open source

Have auditing system in place so that terms of a license are not broken in a team

Participate in the open source community

1. **Various classifications of testing; phases of development testing**
   1. Goal of Testing is to show that a program does what it is intended to do and to discover program defects before it is put into use.
   2. Testing Processes have 2 Goals
      1. Demonstrate (to the developer ane customer) that the SW meets its requirements.
         1. For Customer SW, this means that there should be at least on etest for every requirement
         2. For Generic SW, this means that there should be tests for all system features and a combination of these services
         3. This goal leads to Validation Testing
      2. To discover situations in which the behavior of the SW is incorrect, undesirable, or does not conform to its specification. These are consequences of SW defects.
         1. This goal leads to Defect Testing
   3. Validation Testing
      1. Where you expect the system to perform correctly using a given set of test cases that reflect the system’s expected use.
   4. Defect Testing
      1. Test cases are designed to expose defects
      2. Test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used
   5. Testing cannot demonstrate that SW is free of defects or that is will behave as specified in every circumstance
      1. “Testing can only show the presence of errors, not their absence”
   6. Testing is part of a broader process of SW Verification and Validation.
      1. Validation
         1. Are we building the right product?
         2. Is more general than Verification
         3. Ensures that the SW meets the customer’s expectations.
      2. Verification
         1. Are we building the product right?
         2. Checks that the SW meets its stated functional and non-functional requirements.
   7. V&V processes are concerned with checking that SW being developed meets its specification and delivers the functionality expected by the customer.
      1. Goal of V&V
         1. To establish confidence that the SW system is “fit for purpose”. The level of required confidence depends on:
            1. SW Purpose

More critical SW -> more important is that it be reliable

* + - * 1. User expectations

At first users have low expectations because of their experience with buggy SW. Later they develop higher expectations and thus expect the SW to become more reliable

* + - * 1. Marketing Environment

If the SW is cheap, then users will tolerate bugs

If it is a competitive environment then early release of the system is appropriate (if even with bugs) so that it can be the 1st system in the market

* 1. Static V&V Techniques
     1. Don’t need to execute the SW to verify it
     2. V&V may also involve inspections and reviews to analyze and check the system requirements, design models, source code, and proposed system tests.
        1. 3 Advantages of Inspection over Testing
           1. During testing, errors can hide other errors. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error.
           2. Incomplete version of a system can be inspected without additional costs.
           3. Inspection can also consider broader quality attributes of a program, such as compliance with standards, portability, and maintainability. Can look for inefficiencies, inappropriate algorithms, and poor programming style.
        2. Disadvantages
           1. Not good for discovering defects that arise because of unexpected interactions between different parts of the program, timing problems, or problems with system performance.
           2. In small teams, it can be difficult and expensive to put together a separate inspection team
  2. Phases/Process
     1. Design Test Cases
        1. Output: Test Cases
     2. Prepare Test Data
        1. Input: Test Cases
        2. Output: Test Data
     3. Run program with Test Data
        1. Input: Test Data
        2. Output: Test Results
     4. Compare Results to Test Cases
        1. Input: Test Results and Test Cases
        2. Output: test Reports
  3. 3 Stages of Testing
     1. Development Testing
        1. System is tested during development to discover bugs and defects
     2. Release Testing
        1. Separate testing team tests a complete version of the system before it is released to users
     3. User Testing
        1. Users and potential users of a system test the system in their own environment
  4. **Development Testing**
     1. Includes all testing activities that are carried out by the team developing the system
        1. Tester of the SW is usually the programmer who developed that SW, although this is not always the case.
        2. For more critical systems, this can be more formal with a separate testing group within the development team.
     2. Testing may be carried out at 3 levels of granularity
        1. Unit Testing
           1. Individual program units or object classes are tested. Focus on testing the functionality of objects or methods
        2. Component Testing
           1. Several individual unites are integrated to create composite components. Focuses on testing component interfaces
        3. System Testing
           1. Some or all of the components are integrated and the system is test as a whole. Focuses on testing component interfaces
     3. Development Testing is primarily a defect testing process (aim is to discover bugs in the SW).
     4. Unit Testing (Section 8.1.1)
        1. Process of testing program components, such as methods or object classes. Individual functions or methods are the simplest type of component.
        2. Test should call these routines with different input parameters.
        3. When testing object classes, tests should provide coverage of all of the features of the object:
           1. Test all operations with the object
           2. Set an check the value of all attributes associated with the object
           3. Put the object into all possible states. You should simulate all events that cause a state change.
        4. Whenever possible, try to automate Unit Testing
           1. Frameworks like JUnit write and run your unit tests.
           2. An automated test has 3 parts

A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.

A call part, where you call the object or method to be tested

An assertion part, where you compare the result of the call, with the expected result. If the assertion evaluates to true, the test is successful and fails otherwise.

* + - 1. You the unit you are testing depends on another object (like a database) you can make and create a mock object.
      2. Types of test cases
         1. Partition Testing

Identify groups of inputs that have common characteristics and should be processed in the same way. You should chose test from within each of these groups

* + - * 1. Guideline-based Testing

Use testing guidelines to choose test cases. Guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

* + - 1. Partition Testing
         1. Input data and output results of a program often fall into a number of different classes with common characteristics. Examples of classes are positive numbers, negative numbers, and so on. Programs normally behave in a comparable way for all members of a class -> Because of this sometimes these classes are called Equivalence Partitions or Domains.

Once you identify a set of partitions, you choose test cases from each of these partitions.

A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition. (Reason for this is that programmers tend to consider typical values of input when developing a system).

* + - 1. Guideline Testing
         1. Encapsulate knowledge of what kinds of test cases are effective for discovering errors. For example, when testing programs with array, sequences, lists ,etc try the following:

Test eh SW with sequences that have only a single value

Use different sequences of different sizes in different tests

Derive tests so that the first, middle, and last elements of the sequence are accessed.

* + - * 1. Other guidelines include

Choose inputs that force the system to generate all error messages

Design inputs that cause input buffers to overflow

Repeat the same input or series of input numerous times

Force invalid outputs to be generated

Force computation results to be too large or too small.

* + 1. Component Testing (Section 8.1.3)
       1. Focus is on testing that the component interfaces behave according to specifications
       2. Types of interfaces
          1. Parameter interfaces

Where data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.

* + - * 1. Shared memory interfaces

Interfaces in which a block of memory is shared between components. Data is palced in the memory by one subsystem and retrieved from there by other sub-systems.

Used in embedded sytems

* + - * 1. Procedural interfaces

One component encapsulates a set of procedures that can be called by other components

* + - * 1. Message passing interfaces

One component requests a service from another component by passing a message to it

* + - 1. Types of interface errors
         1. Interface misuse

A calling component calls some other component and makes an error in the use of its interface. Parameters may be of the wrong type or there are too many parameters.

* + - * 1. Interface misunderstanding

Calling component misunderstands the specification of the interface of the call component and makes assumptions about its behavior.

* + - * 1. Timing errors

Producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-data information because the producer has not updated.

* + - 1. Guidelines for Interface Testing
         1. Examine the code to be tested and explicitly list each call to an external component. Design a set of tests in which the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.
         2. Where pointers are passed across an interface, always test the interface with null pointer parameters
         3. Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings
         4. Use stress testing in message passing systems. Generate many more messages than are likely to occur in practice
         5. Where several components interact through shared memory, design tests that vary the order in which these components are activiated.
    1. System Testing (Section 8.1.4)
       1. Involves integrating components to create a version of the system and then testing the integrated system.
       2. System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
       3. Differences with Component Testing
          1. In System Testing, components that have been separately developed and off the shelf system may be integrated with newly developed components. The complete system is then tested
          2. Component developed by different team members or groups may be integrated at this stage. System testing is collective rather than an individual process
       4. Some elements of system functionality only become obvious when you put the components together.
       5. System Testing should focus on testing the interactions between the component and objects that make up a system
       6. Use Case Testing is effective here because of the need to test interactions
          1. Use Cases forces interactions to occur

1. **Test-driven development**
   1. Is an approach to program development in which you interleave testing and code development
   2. You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code you have developed passes its test.
   3. It was part of Agile Method but it can also be used in Plan-Driven development
   4. TDD Process
      1. Start by identifying the increment of functionality that is required. Usually small and implementable in few lines of code
      2. Write a test for this functionality and implement this as an automated test.
      3. Runt the test, along with other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail. This is deliberate as it shows that the test adds something to the test set.
      4. You then implement the functionality and re-run the test. This may include refactoring existing code to improve it and add new code to what’s already there.
      5. Once all tests run successfully, move on to implementing the next chunk of functionality
   5. Using frameworks like JUnit you can every test every time that you add functionality.
   6. Advantages
      1. Helps programmers clarify their ideas of what a code segment is actually supposed to do.
      2. To write a test, you need to understand what is intended, as this understanding makes it easier to write the required code.
      3. Code coverage
         1. Every code segment that you write should have at least one associated test. Therefore you can be confident that all of the code in the system has actually been executed.
      4. Regression Testing
         1. A test suite is developed incrementally as a program is developed. You can always run regression tests to check that changes to the program have not introduced new bugs.
      5. Simplified debugging
         1. When a test fails, it should be obvious where the problem lies. You do not need to use debugging tools to locate the problem.
      6. System documentation
         1. Tests themselves act as a form of documentation that describe what the code should be doing.
   7. Negatives
      1. If you are reusing large code components or legacy systems then you need to write tests for these systems as a whole.
      2. TDD may be ineffective in multi-threaded systems.
      3. Still need to provide a separate Validation Testing process
2. **Release and unit testing**
   1. Release Testing
      1. Is the process of testing a particular release of a system that is intended for use outside of the development team
      2. 2 Distinction between Release Testing and System Testing
         1. A separate team that has not been involved in the system development should be responsible for release testing
         2. System Testing by the development team should focus on discovering bugs in the system (Defect Testing). The objective of Release Testing is to check that the system meets its requirements and is good enough for external use (Validation Testing)
      3. Goal of Release Testing
         1. Convince the supplier of the system that it is good enough for use.
         2. Release Testing therefore, has to show that the system delivers its specified functionality, performance, and dependability and that it does not fail during normal use.
      4. Requirements-Based Testing (Section 8.3.1)
         1. Requirements should be testable…A requirement should be written so that a test can be written for it
         2. Is a systematic approach to test case design where you consider each requirement and derive a set of tests for it.
         3. Is Validation Testing…not Defect Testing – You are trying to demonstrate that the system has properly implemented its requirements
      5. Scenario Testing (Section 8.3.2)
         1. Is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system.
         2. A scenario is a story that describes one way in which the system might be used. They should be realistic and real system use should be able to relate to them.
            1. A release tester would go through this story acting as the user and note how the system behaves in response to different inputs.
      6. Performance Testing (Section 8.3.3)
         1. Once a system has been completely integrated, it is possible to test for emergent properties, such as performance and reliability.
         2. Performance tests have to be designed to ensure that they system can process its intended load.
         3. Is concerned with demonstrating that the system meets its requirements and discovering problems and defects in the system.
         4. To test whether performance requirements are being achieved, you may have to construct an operational file
            1. Operational File

Is a set of tests that reflect the actual mix of work that will be handled by the system. Ex. 90% of transaction in a system are of type A, 5% of type B, and the remainder types C, D, and E.

* + - 1. An effective way to discover defects is to design tests around the limits of the system.
         1. In Performance Testing, this means stressing the system by making demands that are outside the design limits of the software.
         2. This type of testing has 2 functions

It test failure behavior of the system (i.e. does the system corrupt data upon failure)

It stresses the system and may cause defects to come to light that would not normally be discovered.

* + - 1. This testing is important in distributed system which experience severe degradation when they are heavily loaded.
         1. The network gets swamped with coordination data that the different processes must exchange. The processes become slower and slower as they wait for required data from other processes.
  1. User Testing
     1. Is where users of customers provide input and advice on system testing
        1. This may involve formally testing a system that has been commissioned from an external supplier, or could be an informal process where users experiment with a new software product to see if they like lit and that it does what they need.
     2. Is essential because influences from the user’s working environment have a major effect on the reliability, performance, usability, and robustness of the system. It is impossible for the developer to replicate the system’s working environment.
     3. 3 Types of User Testing
        1. Alpha Testing
           1. Users of the SW work with development team to test the SW at the developers site
           2. Often used when developing SW products that are sold as shrink-wrapped systems.
        2. Beta Testing
           1. A release of the SW is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
           2. Takes place when an early, sometimes unfinished, release of a software system is made available to customers and users for evaluation.
           3. Can be publicly available or selectively available
           4. Used for SW that are used in many different environments
           5. Helps with marketing
        3. Acceptance Testing
           1. Customer test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment
           2. Takes place after Release Testing.
           3. Involves a customer formally testing a system to decide whether or not it should be accepted from the developer.
           4. 6 stages in the Acceptance Testing Process

Define acceptance criteria

Should take place early in the process before the contract for the system is signed

Plan acceptance testing

Deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule.

Should define risks to the testing process, such as system crashes and inadequate performance, and discuss how these risks can be mitigated

Derive acceptance tests

Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable.

Acceptance tests should aim to test both the functional and non-functional characteristics of the system.

Run acceptance tests

The agreed acceptance tests are executed on the system.

Negotiate test results

It is very unlikely that all of the define acceptance tests will pass and that there will be no problems with the system. IF this is the case, the testing is finished and the system can be handed over. If there are problems, the developer and the customer have to negotiate to decide if the system is good enough to be put into use. They must also agree on the developer’s response to identified problems

Reject/Accept System

Involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If rejected, then further development is required.

* + - * 1. There is no separate Acceptance Testing process in XP programming since the user is PART of the development team and has been making tests the entire time.

1. **Software evolution and its dynamics**
   1. SW Evolution
      1. Evolution may be an informal process between the system users and developers. In other companies, it is a formalized process with structured documentation produced at each stage in the process
      2. System change proposals are the driver for system evolution in all organizations.
         1. They come from existing requirements that have not been implemented, request for new requirements, bug reports, and new ideas for SW improvement.
         2. The process of change identification and system evolution are cyclic and continue throughout the lifetime of a system
      3. Software Evolution Process
         1. Change Requests
         2. Impact Analysis
         3. Release Planning (covers all proposed changes below… and decides which one to implement)
            1. Fault Repair
            2. Platform Adaptation
            3. System Enhancement
         4. Change Implementation (an iteration of development)
            1. Proposed Changes
            2. Requirements Analysis
            3. Requirements Updating
            4. Software Development (including testing)
         5. System Release
      4. Change Requests sometimes relate to system problems that have to be tackled urgently. These urgent changes can arise for 3 reasons:
         1. If a serious system fault occurs that has to be repaired to allow normal operation
         2. If changes to the system operating environment have unexpected effects that disrupt normal operation
         3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system.
      5. Urgent Change Requests can be problematic because it means that you may not be able to follow the formal change analysis process of modifying the requirements and design. So the danger comes in to play when you make changes to code and never update the documentation. They are also problematic because you use a quick solution to a problem and not the best solution.
      6. Evolution pretty much involves continuing the agile development process.
      7. 2 Problems that arise when there is a handover from a development team to a separate team responsible for evolution
         1. Where the development team has used an agile approach but the evolution team is unfamiliar with the agile methods and prefers a plan-based approach.
            1. The team may expect detailed documentation to support evolution but this is not commonly produced in agile processes
            2. Where a plan-based approach has been used for development but the evolution team prefers to use agile methods. In this case, the evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.
   2. Evolution Dynamics
      1. Is the study of system change.
      2. Lehman’s Laws
         1. Continuing Change
            1. A program that is used in a real-world environment must change or else it will become increasing less useful.
         2. Increasing complexity
            1. As more change occurs, the structure of the system tends to become more complex.
         3. Large Program Evolution
            1. Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release
         4. Organizational Stability
            1. Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
         5. Conservation of familiarity
            1. Over the lifetime of a system, the incremental change in each release is approximately constant
            2. The more functionality that there is in an increment, the more faults that there will be.
         6. Continuing Growth
            1. Functionality offered by system has to continually increase to maintain user satisfaction
         7. Declining Quality
            1. Quality of systems will decline unless they are modified to reflect changes in their operational environment
         8. Feedback System
            1. Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback system to achieve significant product improvement
2. **Maintenance; types and characteristic activities**
   1. Is the general process of changing a system after it has been delivered.
   2. This term is usually applied to custom software in which separate development groups are involved before and after the delivery.
   3. Changes made to software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or accommodate new requirements.
   4. 3 Types of SW Maintenance
      1. Fault Repairs
         1. Coding errors are usually relatively cheap to correct
         2. Design errors are more expensive as they involve rewriting several program components.
         3. Requirement errors are the most expensive to repair because of the extensive system redesign with may be necessary.
      2. Environmental Adaptations
         1. Is required when some aspect of the system’s environment such as the hardware, the platform operating system, or other support software changes. The application system must be modified to adapt it to cope with these environmental changes
      3. Functionality addition
         1. Is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the SW is often much greater than for the other types of maintenance.
   5. More of the maintenance budget is spent on implementing new requirements than on fixing bugs.
   6. Maintenance Effort Distribution
      1. 65% = Functionality Addition or Modification
      2. 17% = Fault Repair
      3. 18% = Environmental Adaptation
   7. It is cost effective to spend time in design and development to make the system more maintainable and easier to modify in the future.
      1. How to reduce Maintenance Cost
         1. Use precise specifications
         2. Use of Object Oriented Development
         3. Configuration Management
      2. This is the idea behind Agile Methods
   8. It is more expensive to add functionality after a system is in operation than it is to implement the same functionality during development. The reasons for this are:
      1. Team Stability
         1. After a sytem has been delivered, it is normal for the development team to be broken up and for people to work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. They need to spend time understanding the existing system before implementing changes to it.
      2. Poor development practice
         1. The contract to maintain a system is usually separate from the system development contract. The maintenance contract may be given to a different company rather than the original system developer. This factor, along with the lack of team stability, means that there is no incentive for a development team to write maintainable software. If a development team can cut corners to save effort during development it is worthwhile for them to do so, even if this means that the software is more difficult to change in the future
      3. Staff skills
         1. Maintenance staffs are often relatively inexperienced and unfamiliar with the application domain. Maintenance has a poor image among software engineers. Also, the system maybe implemented in obsolete languages that the maintenance team do not know.
      4. Program age and structure
         1. As changes are made to programs, their structure tends to degrade. Consequently, they become harder to understand and change.
      5. \*\*NOTE
      6. The first 3 issues above deal with how organization still views development and maintenance as separate processes with no incentive to develop code that is easier to maintain.
      7. Activities
   9. Maintenance Prediction (Section 9.3.1)
      1. You should try to predict what system changes might be proposed and what parts of the system are likely to be the most difficult to maintain.
      2. Stages
         1. Predicting System Changes
            1. How many change requests can be expected?
            2. What parts of the system are most likely to be affected by change requests?
         2. Predicting Maintainability
            1. What parts of the system will be the most expensive to maintain?
         3. Predicting Maintenance Costs
            1. What will be the lifetime maintenance costs of this system?
            2. What will be the costs of maintaining this system over the next year?
      3. To evaluate the relationships between a system and its environment, you should assess:
         1. The number and complexity of system interfaces
            1. The more interfaces and the more complex these interfaces, the more likely changes will be required as new requirements are proposed
         2. The number of inherently volatile system requirements
            1. Requirements that reflect organizational policies and procedures are likely to be more volatile
         3. The business processes in which the system is used
            1. As business processes evolve, they generate system change requests. The more processes that use a system, the more the demands for system change
      4. The more complex a system, the more difficult it is to maintain
      5. To reduce maintenance costs, you should try to replace complex system components with simpler alternatives
      6. After a system has been put into services, you can use process data to help predict maintainability. Examples of process metrics that can be used for assessing maintainability are:
         1. Number of requests for corrective maintenance
            1. an increase in bug and failure reports
         2. Average time required for impact analysis
            1. this reflects the number of program components that are affected by the change request. If time increases, it implies more and more components are affected and maintainability is decreasing
         3. Average time taken to implement a change request
            1. This is the amount of time that you need to modify the system and its documentation, after you have assessed which components are affected
         4. Number of outstanding change requests
            1. An increase in this number over time may imply a decline in maintainability.
   10. Software Reengineering (Section 9.3.2)
       1. To make legacy systems easier to maintain, you can reengineer them to improve their structure and understandability.
       2. Reengineering may involve:
          1. documenting the system
          2. refactoring the system architecture
          3. translating programs to a modern programming language
          4. Modifying and updating the structure and values of the system’s data.
       3. 2 Important benefits from reengineering rather than replacement
          1. Reduced Risk
             1. There is a high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.
          2. Reduced Cost
             1. The cost of reengineering may be significantly less than the cost of developing new software.
       4. Activities of the Reengineering Process
          1. Source Code Translation
             1. Program is converted from an old programming language to a more modern version of the same language or to a different language.
             2. SW Tools may be used here
          2. Reverse engineering
             1. Program is analyzed and information extracted from it. This helps to document its organization and functionality
             2. Usually is automated
          3. Program Structure Improvement
             1. The control structure of the program is analyzed and modified to make it easier to read and understand.
          4. Program Modularization
             1. Related parts of the program are grouped together and, where appropriate, redundancy is removed. This may also involve Architectural Refactoring
          5. Data Reengineering
             1. The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure.
   11. Preventative Maintenance by Refactoring (Section 9.3.3)
       1. Refactoring is the process of making improvements to a program to slow down degradation through change.
          1. It means modifying a program to improve its structure, to reduce its complexity, or to make it easier to understand.
       2. Refactoring can be seen as “Preventative maintenance” that reduces problems of future change
       3. Reengineering takes place after a system has been maintained for some time and maintenance costs are increasing. Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases costs and difficulties of maintaining a system.
       4. Refactoring is an inherent part of agile methods
       5. Situation in which programs can be improved
          1. Duplicate Code
             1. Fix this by replacing repeated code with a call to a function
          2. Long methods
             1. Fix this by separating the methods out into smaller methods
          3. Switch (Case) Statements
             1. These often involve duplication around multiple parts of the program
             2. This can be fixed through polymorphism
          4. Data Clumping
             1. This occurs when the same group of data items (fields in classes, parameters in methods) reoccurs in several places in a program.
             2. These can be fixed by having an object encapsulate all of the data
          5. Speculative Generality
             1. Occurs when developers include generality in a program in case it is required in future
             2. This can often be fixed by simply removing it.
3. **Management of legacy systems**
   1. Many companies have many legacy systems in which they must decide whether to proceed with maintain or not.
   2. 4 Strategic Options for Realistic Assessments of Legacy Systems
   3. Scrap the system completely
      1. Should be chosen when the system is not making an effective contribution to business processes.
      2. Leave the system unchanged and continue with regular maintenance
         1. Should be chosen when the system is still required but it is fairly stable and the system users make relatively few change requests
      3. Reengineer the system to improve its maintainability
         1. Should be chosen when the system quality has been degraded by change and where a new change to the system is till be proposed.
      4. Replace all or part of the system with a new system
         1. Should be chosen when factors, such as new hardware, mean that the old system cannot continue in operation or where off-the-shelf system would allow the new system to be developed at a reasonable cost.
   4. When you access a legacy system, you must do so from a business perspective and technical perspective
      1. Business Perspective
         1. Decide if the business really needs the system.
      2. Technical Perspective
         1. Assess the quality of the application software and the system’s support software and hardware.
      3. Then use a combination of the business value and system quality to inform your decision on what to do with legacy system.
      4. 4 Clusters of Systems
         1. Low Quality, Low Business Value
            1. Keeping these systems will be expensive and the rate of return to the business will be small. These should be scrapped.
         2. Low quality, High Business Value
            1. These systems cannot be scrapped since they have value.
            2. Their low quality means that it is expensive to maintain them. These system should be reengineered to improve their quality. They may be replaced, if suitable off-the-shelf system is available.
         3. High Quality, Low Business Value
            1. Don’t contribute much to the business but they may not be very expensive to maintain.
            2. It is not worth replacing these systems so noramml system maintenance may be continued if expensive changes are not required if expensive changes are not required and the system hardware remains in use. If expensive changes are required, the software should be scrapped
         4. High Quality, High Business Value
            1. These system have to be kept in operation.
            2. Their high quality means you don’t need to invest much in transformation or system replacement.
            3. Normal system maintenance should be used.
      5. To asses business value of a system you must discuss the following info with stakeholders:
         1. The use of the system
            1. How much is it used?
            2. Does it serve an importan function?
         2. The business processes that are supported
         3. The system dependability
            1. Is the system dependable and reliable?
         4. Thee system outputs
            1. Is whatever the system outputs/perform important and used by customers?
      6. To asses value of a system from the technical perspective you must consider the application system and its environment
         1. Environment Assessment Factors
            1. Supplier Stability

Is the supplier still in existence?

Is the supplier likely to still exist?

* + - * 1. Failure Rate

Does the hardware have a high rate of reported failures?

Does the support software crash and force system restarts?

* + - * 1. Age

How old is the hardware and software?

* + - * 1. Performance

Is the performance of the system adequate?

* + - * 1. Support Requirements

What local support is required by the hardware and software?

* + - * 1. Maintenance Costs

What are the costs of hardware maintenance and support software licenses?

* + - * 1. Interoperability

Are there problems interfacing the system to other systems?

* + - 1. Application Assessment Factors
         1. Understandability

How difficult is it to understand the code?

How complex are the control structures used?

* + - * 1. Documentation

What system documentation is available?

Is it consistent, complete, and current?

* + - * 1. Data

Is there an explicit model for the system?

To what extent is data duplicated across files?

* + - * 1. Performances

Is the performance of the application adequate?

* + - * 1. Programming Language

Are modern compilers available for the programming language used to develop the system?

* + - * 1. Configuration Management

Are all version of all parts of the system managed by a configuration management system?

* + - * 1. Test Data

Does test data for the system exist?

* + - * 1. Personnel Skills

Are there people available who have the skills to maintain the application?

* + 1. Data that may be helpful in quality assessment are:
       1. The number of system change requests
          1. The higher this number is, the lower the quality of the system since each change damages the system structure
       2. The number of user interfaces
          1. More interfaces means that there is a higher chance that there are redundancies in them
       3. The volume of data used by the system
          1. High volume of data means that it is more likely that there will be data inconsistencies that reduce system quality.

1. **Dependability and its dimensions**
   1. When designing a dependable system, you must consider:
      1. Hardware Failure
         1. System hardware may fail because of mistakes in its design because components fail as a result of manufacturing errors, or because components have reached the end of their natural life
      2. Software Failure
         1. System software may fail because of mistakes in it specification, design, or implementation
      3. Operational Failure
         1. Human users may fail to use or operate the system correctly.
2. **Characteristics of the four major dimensions of dependability**
   1. Dependability of a computer system is a property of the system that reflects its trustworthiness.
   2. Trustworthiness means the degree of confidence a user has that the system will operate as they expect, and that the system will not “fail” in normal use.
   3. Principal Dependability Properties
      1. Availability
         1. The ability of the system to deliver services when requested
         2. The probability that it will be up and running and able to deliver useful services to users at any given time
      2. Reliability
         1. The ability of the system to deliver servies as specified
         2. The probability that the system will correctly deliver services as expected by the user
      3. Safety
         1. The ability of the system to operate without catastrophic failure
         2. How likely it is that the system will cause damage to people or its environment
      4. Security
         1. The ability of the system to protect itself against accidental or deliberate intrusion
         2. How likely it is that the system can resist accidental or deliberate intrusions
   4. Other dependability properties
      1. Repairability
         1. Can the system be repaired quickly if it fails?
      2. Maintainability
         1. Can the system adapt to new requirements?
      3. Survivability
         1. Can the system continue to deliver service whilst under attack and while it is disabled
      4. Error tolerance
         1. Can the system detect errors and either fix them automatically or request the user to reinput their data?
   5. Availability and Reliability
      1. These are 2 related properties
      2. Availability = Probability that the system will be up and running to deliver these services to users on request.
      3. Reliability = Probability that the system’s services will be delivered as defined in the system specification
      4. More precise definitions
         1. Reliability =Probability of failure-free operation over a specified time, in a given-environment, for a specific purpose
         2. Availability = Probability that a system, at a point in time, will be operational and able to deliver the requested services.
      5. A common cause of perceived unreliability is that the system specification does not match the expectations of the system users
      6. Availability not only depends on the number of system crashes, but also on the time need to repair faults that have cause the failure.
      7. Reliability Terminology
         1. Human error or mistakes
            1. Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works expect when the transmission time is between 23:0 and midnight (24:00)
         2. System Fault
            1. A characteristic of a system that can lead to a system error. The fault is the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23:00.
         3. System Error
            1. An erroneous system state that can lead to system behavior that is unexpected by system users. The value of the transmission time is set incorrectly (to 24:xx rather than 00:xx) when the faulty code is executed
         4. System Failure
            1. An event that occurs at some point when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid.
      8. When an input or sequence of events causes faulty code in a system to be executed, an erroneous state is created that may lead to a software failure.
      9. A program reliability depends on the number of system inputs that are members of the set of inputs that lead to an erroneous output
         1. The practical reliability of a program depends on the number of inputs causing erroneous outputs (failures) during normal use of the system by most users.
      10. System faults don’t always lead to system errors and system errors do not always lead to system failures. The reasons are:
          1. Not all code in a program is executed
             1. The code that includes a fault may never be executed because of the way that the software is used
          2. Errors are transient
             1. A state variable may have an incorrect value caused by the execution of faulty code. However, before this is accessed and causes a system failure, some other system input may be processed that resets that state to a valid value.
          3. The system may include fault detection and protection mechanisms
             1. These ensure that the erroneous behavior is discovered and corrected before the system services are affected
          4. Users adapt their behavior to avoid using inputs that they know cause program failures
      11. 3 Approaches to improve reliability
          1. Fault Avoidance
             1. Use techniques that either minimizes the possibility of human errors and/or that trap mistakes before they result in the introduction of system faults.
          2. Fault Detection and Removal
             1. Use of Verification and Validation techniques that increase the chances that faults will be detected and removed before the system is used.
             2. Ex. Systematic testing and debugging
          3. Fault Tolerance
             1. Techniques that ensure that faults in system do not result in system errors or that system errors do not result in system failures. Incorporation of self-checking facilities in a system and the use of redundant system modules.
   6. Safety
      1. Safety-critical systems are system where it is essential that system operation is always safe
      2. 2 Classes of Safety Critical SW
         1. Primary Safety Critical SW
            1. SW that is embedded as a controller in a system
            2. Malfunctioning of such SW can cause a hardware malfunction, which results in human injury or environmental damage.
         2. Secondary Safety-Critical SW
            1. SW that can indirectly result in a injury
            2. Ex: Computer-aided engineering design system whose malfunctioning might result in a design fault in the object being designed
      3. 4 Reasons why Reliable System may NOT be Safe:
         1. We can never be 100% certain that a system is fault-free and fault-tolerant. Undetected faults can be dormant for a long time and software failures can occur after many years of reliable operation
         2. The specification may be incomplete in that is does not describe the required behavior of the system in some critical situations
            1. A high percentage of system malfunctions are the result of specification rather than design errors.
         3. Hardware malfunction may cause the system to behave in an unpredictable way, and present the software with an unanticipated environment. When components are close to physical failure, they may behave erratically and generate signals that are outside the ranges that can be handled by the SW.
         4. The system operators may generate inputs that are not individually incorrect but which, in some situations, can lead to a system malfunction.
      4. Safety Terminology
         1. Accident/Mishap
            1. An unplanned event or sequence of events which results in human death or injury, damage to property, or to the environment.
            2. Ex. Overdose of insulin
         2. Hazard
            1. A condition with the potential for causing or contributing to an accident.
            2. Ex. Failure of the sensor that measures blood glucose
         3. Damage
            1. A measure of the loss resulting from an accident/mishap.
         4. Hazard Severity
            1. An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results.
         5. Hazard Probability
            1. Probability of the events occurring which create a hazard
         6. Risk
            1. Measure of the probability that the system will cause an accident.
            2. Assessed by considering the Hazard Probability, the Hazard Severity, and the probability that the hazard will lead to an accident.
      5. The key to assuring safety is to ensure either that accidents do not occur or that the consequences of an accident are minimal. This can be achieved in 3 complementary ways:
         1. Hazard Avoidance
            1. System is designed to avoid hazards
            2. Ex. A cutting system that requires an operator to use 2 hands to press separate buttons simultaneously avoids the hazard of the operator’s hands being in the blade pathway.
         2. Hazard Detection and Removal
            1. The system is designed so that hazards are detected and removed before they result in an accident
            2. Ex. A chemical plant system may detect excessive pressure and open a relief valve to reduce these pressures before an explosion occurs
         3. Damage limitation
            1. System may include protection features that minimize the damage that may result from an accident.
            2. Ex. An aircraft engine normally includes automatic fire extinguishers. If a fire occurs, it can often be controlled before it poses a threat to the aircraft.
      6. Accidents most often occur when several things go wrong at the same time
      7. Accidents are an inevitable part of using complex systems
   7. Security
      1. Security = Attribute that reflects the ability of the system to protect itself from external attacks (Accidental or Deliberate)
      2. Security Terminology
         1. Asset
            1. Something of value which has to be protected. The asset may be the SW system itself or data used by that system
            2. Ex. Records of each patient in a database
         2. Exposure
            1. Possible loss or harm to a computing system. This can be loss or damage to data, or can be a loss of time and effort if recovery is necessary after a security breach.
            2. Ex. Potential financial loss from future patients who do not seek treatment because they do not trust the clinic to maintain their data.
         3. Vulnerability
            1. A weakness in a computer-based system that may be exploited to cause loss or harm
            2. Ex. A weak password system which makes it easy for users to set passwords. User ids that are the same as names
         4. Attack
            1. An exploitation of a system’s vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage
            2. Ex. An impersonation of an authorized user
         5. Threats
            1. Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subject to an attack.
            2. Ex. An unauthorized user will gain access to the system by guessing the credentials (login name and pw) of an authorized user
         6. Control
            1. A protective measure that reduces a system’s vulnerability. Encryption is an example of a control that reduces a vulnerability of a weak access control system.
            2. Ex. A password checking system that disallows user passwords that are proper names or words that are normally included in a dictionary.
      3. 3 Main Types of Threats in a Networked System
         1. Threats to confidentiality of the system and its data
         2. Threats to the integrity of the system and its data (i.e. damage or corrupt data)
         3. Threats to the availability of the system and its data
      4. 3 Controls you might use to enhance security
         1. Vulnerability Avoidance
            1. Controls that are intended to ensure that attacks are unsuccessful
            2. Ex. Military System are not connected to public networks so that external access is impossible
         2. Attack Detection and Neutralization
            1. Controls that are intended to detect and repel attacks. Includes functionality that monitors the system’s operation and checks for unusual patterns of activity.
         3. Exposure limitation and recovery
            1. Controls that support recovery from problems
            2. Ex. Automated backup strategies and information “mirroring” to insurance policies that cover the costs associated with a successful attack on the system
3. **Characteristics of risk-driven requirements specification**
   1. Dependability and security requirements specify how the system should protect itself from internal faults, stop system failures causing damage to the environment, stop accidents or attacks from the system’s environment damaging the system, and facilitate recovery in the event of failure.
   2. A risk-driven approach is an approach that has been widely used by safety- and security-critical system developers. It focuses on those events that could cause the most damage or that are likely to occur frequently. Events that have only minor consequences or that are extremely rare may be ignored.
      1. Safety-critical
         1. Risk are associated with hazards that can result in accidents
      2. Security-critical
         1. Risk come from insider and outsider attacks on a system that are intended to exploit vulnerabilities
   3. Stages in the process
      1. Risk Identification
         1. Potential risk are identified
         2. Risks are dependent on the environment in which the system is to be used.
      2. Risk Analysis and Classification
         1. Risks are considered separately
         2. Potentially serious risks and those that are not implausible are further analyzed
         3. Risks may be eliminated because they are unlikely to arise or because they cannot be detected by the software
      3. Risk Decomposition
         1. Each risk is analyzed to discover potential root causes of the risk
         2. Root causes are the reasons why the system may fail
            1. They may be hardware or software or inherent vulnerabilities
      4. Risk Reduction
         1. Proposals for ways in which the identified risks may be reduced or eliminated are made
   4. For large system, Risk Analysis may be structured into phases:
      1. Preliminary Risk Analysis
         1. Major risks from the environment are identified
         2. These are independent from the technology being used for system development
         3. Goal is to develop an initial set of security and dependability requirements for the system
      2. Life-Cycle Risk Analysis
         1. Takes place during the system development
         2. Concerned mostly with risks that arise from system design decisions
      3. Operational Risk Analysis
         1. Concerned with the system user interface and risks from operator errors
      4. These phases are necessary because it is impossible to make all dependability and security decisions without complete info about the system implementation. Dependability and security requirements are particularly affected by technology choices and design decisions. Security requirements may have to be modified because they conflict with the security features that are provided by an off-the-shelf system.
4. **Reliability specification**
   1. Is a measurable system attribute
   2. 2 Types of Reliability Requirements
      1. Non-functional Requirements
         1. Define the number of failures that are acceptable during normal use of the system, or the time in which the system is unavailable for use.
         2. These are quantitative reliability requirements
      2. Function Requirements
         1. Define system and software functions that avoid, detect, or tolerate faults in the software and so ensure that these faults do not lead to system failure
   3. To achieve some required level of reliability, the function and design requirements of the system should specify the faults to be detected and the action that should be taken to ensure that these faults do not lead to system failures
   4. Process
      1. Risk Identification
         1. Identify the types of system failures that may lead to economic losses of some kind
      2. Risk Analysis
         1. Involves estimating the costs and consequences of different types of software failure and selecting high-consequence failures for farther analysis
      3. Risk Decomposition
         1. Do a root cause analysis of serious and probable system failures
      4. Risk Reduction
         1. Should generate quantitative reliability specifications that set out the acceptable probabilities of the different types of failures.
         2. These should take into account the costs of failures. You may use different probabilities for different system services. You may also generate function reliability requirements
   5. Types of System Failure
      1. Loss of Services
      2. Incorrect Service Delivery
      3. System/Data Corruption
   6. Reliability Metrics (Section 12.3.1)
      1. Reliability can be specified as a probability that a system failure will occur when a system is in use
      2. Metrics
         1. Probability of Failure On Demand (POFOD)
            1. You define the probability that a demand for service from a system will result in a system failure
            2. This should be used in situations where a failure on demand can lead to a serious system failure
         2. Rate of Occurrence of Failures (ROCOF)
            1. Sets out the probable number of system failures that are likely to be observed relative to a certain time period (e.g. hour) or to the number of system executions.
            2. The reciprocal of this is the Mean Time to Failure (MTTF)

Average number of time units between observed system failures

* + - * 1. This should be used in situations where demands on systems are made regularly rather than intermittently
      1. Availability (AVAIL)
         1. Reflects a systems ability to deliver services when requested
         2. Is the probability that a system will be operational when a demand is made for service
         3. Also depends on the time required to get the system back into operation
         4. Ex. AVAIL of 0.9999 mean that, on average, the system will be available for 99.99% of the operating time
    1. To access the reliability of a system, you must capture the following data:
       1. The number of system failures given a number of requests for system services (used for POFOD)
       2. The time or number of transactions between system failures plus the total elapsed time or total number of transactions (used for ROCOC and MTTF)
       3. The repair or restart time after a system failure that leads to loss of service (Use for Availability)
    2. Time units that can be used include calendar, processor, or discrete (e.g., number of transactions) time
  1. Non-Functional Reliability Requirements (Section 12.3.2)
     1. These requirements are quantitative specifications of the required reliability and availability of a system, calculated using one of the metrics described in the section above (12.3.1)
     2. Advantages in deriving quantitative reliability specifications
        1. Process of deciding what required level of the reliability helps to clarify what stakeholders really need. It helps stakeholders understand that there are different types of system failure, and it makes clear to them that high levels of reliability are very expensive to achieve
        2. It provides a basis for assessing when to step testing a system

1. **Safety and security specification**
   1. Security

* It is impractical to specify the specification of security requirements quantitatively
  + These are also usually “shall not” requirements
* Security is more challenging than safety because:
  + When considering safety, you can assume that the environment in which the system is installed is not hostile (nothing is trying to perform an attack)
  + When system failures occur that pose a risk to safety, you look for the errors or omissions that have caused the failure. When deliberate attacks cause system failures, finding the root cause may be more difficult as the attacker may try to conceal the cause of the failure
  + It is usually acceptable to shut down a system or to degrade system services to avoid a safety-related failure. However, attacks on a system may be so-called denial of service attacks. Shutting down that system means that the attack was successful.
  + Safety-related events are not generated by an intelligent adversary.
* Security requirements have to be more extensive than safety requirements
* 10 Types of Security Requirements that may be include in System Specification
  + Identification requirements specify whether or not a system should identify its users before interacting with them
  + Authentication requirements specify how users are identified
  + Authorization requirements specify the privileges and access permissions of identified users
  + Immunity Requirements specify how the system should protect itself against viruses, worms, and similar threats.
  + Integrity requirements specify how data corruption can be avoided
  + Intrusion Detection requirements specify what mechanisms should be used to detect attacks on the system
  + Non-repudiation requirements specify that a party in a transaction cannot deny its involvement in that transaction
  + Privacy requirements specify how data privacy is to be maintained
  + Security Auditing requirements specify how system use can be audited and checked
  + System Maintenance Security requirements specify how an application can prevent authorized changes from accidently defeating its security mechanisms
* As discussed in section 12.1, Risk Analysis and Assessment has 3 stages
  + Preliminary Risk Analysis
    - Decision on the detailed system requirements, the system design, or the implementation technology have not been made
    - The aim of this assessment process is to derive security requirements for the system as a whole
  + Life-Cycle Risk Analysis
    - This assessment takes place during the system development life cycle after design choices have been made
    - Additional security requirements take account of the technologies used in building the system and system design and implementation decisions
  + Operational Risk Analysis
    - This risk assessment considers the risks posed by malicious attacks on the operational system by users, with or without insider knowledge of the system
* Process Stages
  + Asset Identification
    - System assets that may require protection are identified. (Part of Risk Identification)
  + Asset Value Assessment
    - Where you estimate the value of the identified assets (Part of Risk Analysis)
  + Exposure Assessment
    - Where you assess the potential losses associated with each asset (Part of Risk Analysis)
  + Threat Identification
    - Where you identify the threats to system assets (Part of Risk Analysis)
  + Attack Assessment
    - Where you decompose each threat into attacks that might be made on the system and the possible ways in which these attacks may occur (Part of Risk Decomposition)
  + Control Identification
    - Where you propose the controls that might be put in place to protect an asset. (Part of Risk Reduction)
  + Feasibility Assessment
    - Where you assess the technical feasibility and the costs of the proposed controls. (Part of Risk Reduction)
  + Security Requirements Definition
    1. Where knowledge of the exposure, threats, and control assessments is used to derive system security requirements. (Part of Risk Reduction)
  1. Safety
     1. Safety-critical systems are systems in which failures may affect the environment of the system and cause injury or death to the people in the environment.
     2. Goal: To identify requirements that will minimize the probability that such system failure occur
     3. Safety requirements are primarily protection requirements and are not concerned with normal system operation.
     4. In deriving these requirements you need to find a balance between safety and functionality and avoid overprotection
     5. Safety specification is usually focused on the hazards that may arise in a given situation, and the events that can lead to these hazards (Hazard = something that can result in death or injury)
     6. Process
        1. Risk Identification
           1. Hazard identification process that identifies hazards that may threaten the system
        2. Risk Analysis
           1. Process of hazard assessment to decide which hazards are the most dangerous and/or most likely to occur. These should be prioritized when deriving safety requirements
        3. Risk Decomposition
           1. Discovering events that can lead to the occurrence of a hazard.
           2. AKA Hazard Analysis
        4. Risk Reduction
           1. Identification of safety requirements
           2. Concerned with ensuring the hazard does not arise or lead to an accident or that if an accident does occur, the associated damage is minimized
     7. **Hazard Identification (Section 12.2.1)**
        1. Different Classes of hazards such as physical, electrical, biological, radiation, service failure, and so on.
           1. Each of these classes of hazards can then be analyzed to discover specific hazards that could occur
        2. Experienced engineers, working with domain experts and professional safety advisers, identify hazards from previous experience and from an analysis of the application domain.
     8. **Hazard Assessment (Section 12.2.2)**
        1. This stage focuses on understanding the probability that a hazard will occur and the consequences if an accident or incident associated with that hazard should occur
           1. You need to make this analysis to understand whether a hazard is a serious threat to the system or to the environment
           2. Analysis also provides a basis for deciding on how to manage the risk associated with the hazard
        2. For each hazard, a statement of acceptability is created
           1. This is expressed in terms of risk, where risk takes into account the likelihood of an accident and its consequences
           2. 3 Risk Categories

Intolerable Risks (High costs for company)

Risks that threaten human life

System must be designed so that such hazards either cannot arise or, that if they do, features in the system will ensure that they are detected before they cause an accident

As Low As Reasonably Practical (ALARP) Risks

Risks that haves less serious consequences OR that are serious but have very low probability of occurring

System should be designed so that the probability of an accident arising because of a hazard is minimized, subject to other considerations such as cost and delivery

Acceptable Risks (Low costs for company)

Risks whose associated accidents normally result in minor damage

System designers should take all possible steps to reduce ‘acceptable’ risks, so long as these do not increase costs, delivery time, or other non-functional system attributes

* + 1. **Hazard Analysis (Section 12.2.3)**
       1. Is the process of discovering the root causes of hazards in a safety-critical system
          1. Find out what events or combination of events could cause a system failure that results in a hazard
       2. 2 Approaches to finding causes
          1. Top-down (Deductive)

Tend to be easier

Start with the hazard and work from that to the possible system failure

* + - * 1. Bottom-up (Inductive)

Start with a proposed system failure and identify what hazards might result from that failure

* + - 1. Fault Tree analysis is an approach to hazard analysis
         1. Here, you start with the hazards that have been identified
         2. For each hazard, you then work backwards to discover the possible causes of that hazard
         3. You put the hazard at the root of the tree and identify the system states that can lead to that hazard
         4. For each of these states, you then identify further system states that can lead to them
         5. You continue this decomposition until you reach the root causes of the risk
         6. Hazards that only arise from a combination of root causes are usually less likely to lead to an accident than hazards with a single root cause
    1. **Risk Reduction (Section 12.2.4)**
       1. Once potential risks and their root causes have been identified, you are then able to derive safety requirements that manage and ensure that incidents or accidents do not occur.
       2. 3 Strategies
          1. Hazard Avoidance

System is designed so that the hazard cannot occur

* + - * 1. Hazard Detection and Removal

System is designed so that hazards are detected and removed before they result in an accident

* + - * 1. Damage Limitation

System is designed so that the consequences of accident are minimized

* + - * 1. Intolerable hazards may be handled by minimizing their probability and adding a protection system that provides a safety backup

1. **Formal methods of specification**
   1. Formal methods are mathematically-based approaches to software development where you define a formal model of the software
   2. To create a model, you translate system user requirements, which are expressed in natural language, diagrams, and tables, into a mathematical language which has formally defined semantics
   3. Formal specifications are essential for a verification of the design and implementation of software. They are also the most precise way of specifying systems, and so reduce the scope for misunderstanding.
   4. These specifications are usually developed as part of a plan-based SW process, where the system is completely specified before development.
   5. Formal Specification in a Plan-based SW Process
      1. User Requirements Definition
      2. System Requirements Specification
      3. Architectural Design
      4. Formal Specification
      5. High-Level Design
   6. Advantages of developing a formal specification are:
      1. As you develop a formal specification in detail, you develop a deep and detailed understanding of the system requirements
      2. As the specification is expressed in a language with formally define semantics, you can analyze it automatically to discover inconsistencies and incompleteness
      3. If you use a methods such as the B method, you can transform the formal specification into a program through a sequence of correctness-preserving transformations
      4. Program testing costs may be reduced because you have verified the program against its specification
   7. Arguments against using Formal Specification
      1. Problem owners and domain experts cannot understand a formal specification so they cannot check that it accurately represent their requirements
      2. It is fairly easy to quantify the costs of creating a formal specification, but more difficult to estimate the possible cost savings that will result from its use
      3. Most SW engineers have not been trained to use formal specification languages
      4. It is difficult to scale current approaches to formal specification up to very large system
      5. Formal specification is not compatible with agile methods of development