Software Engineering:

- Establishment and use of sound engineering principles to obtain economical software that is reliable and works efficiently on real machines
- A concerted effort should be made to understand the problem before a software solution is developed
- Design becomes a pivotal activity
- Software should exhibit high quality
- Software should be maintainable
- The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is the application of engineering to software
- 2 types of software systems: software products and custom software systems
- Project-based software engineering:
    - The starting point for software development is a set of software requirements that are owned by an external client and which set out what they want a software system to do to support their business processes
    - Software is developed by a software company that designs and implements a system that delivers functionality to meet the requirements
    - Customers may change their requirements at any time in response to business changes, the contractor must change the software to reflect these requirements changes
    - Custom software usually has a long life and it must be supported over that lifetime
- Software products:
    - Generic to software systems that provide functionality that is useful to a range of customers
    - Many different types of products are available from large-scale business systems through personal products to simple mobile phones apps and games
    - Software product engineering methods and techniques have evolved from software engineering techniques that support the development of one-off, custom software systems
    - Custom software systems are still crucial for large businesses, governments, and public bodies - developed in dedicated software projects
- Product software engineering:
    - A software development company is responsible for deciding on the development timescale, what features to include, and when the product should change
    - Rapid delivery of software products is essential to capture the market for that type of product
    - The starting point for product development is a business opportunity that is identified by individuals or a company - they develop a software product to take advantage of this opportunity and sell this to customers
    - The company that identified the opportunity to design and implement a set of software features that realize the opportunity and that will be useful to customers
- Product vision:

        - The starting point for software product development is a ‘product vision’
        - Simple statements that define the essence of the product to be developed
        - Product vision should answer three fundamental questions:
            - What is the product to be developed?
            - Who are the target customers and users?
            - Why should customers buy this product?
        - Moore’s Vision template:
            - FOR (target customer)
            - WHO (statement of need or opportunity)
            - THE (product name) IS A (product category)
            - That (key benefit, compelling reason to buy)
            - UNLIKE (primary competitive advantage)
            - OUR PRODUCT (statement of primary differentiation)
- Software product management:
    - Software product management is a business activity that focuses on the software products developed and sold by the business
    - Product managers take overall responsibility for the product and are involved in planning, development, and product marketing
    - Product managers are the interface between the organization, its customers, and the software development team
        - Involved at all stages of a product’s lifetime from the initial conception through to withdrawal of the product from the market
    - Product managers must look outward to customers and potential customers rather than focus on the software being developed
- Product management concerns:
    - Business needs - PMs have to ensure that the software being produced meets the business goals of the software development company
    - Technology constraints - PMs must take developers aware of technical issues that are important to customers
    - Customer experience - PMs should be in regular contact with customers and potential customers to understand what they are looking for in a product, the types of users and their backgrounds, and the ways that the product may be used
- Legacy software system
    - Software must be adapted to meet the needs of new computing environments or technology
    - Software must be enhanced to implement new business requirements
    - Software must be extended to make it interoperable with other more modern systems or databases
    - Software must be re-architected to make it viable within a network environment
- Software execution models
    - Stand-alone - the software executes entirely on the customer’s computers
    - Hybrid - part of the software’s functionality is implemented on the customer's computer but some features are implemented on the product developer's servers

- Software service - all of the product's characteristics are implemented on the developer's servers and the customer accesses these through a browser or a mobile app

Agile Software Engineering:

- Software products must be brought to the market quickly and rapidly so rapid software development and delivery are essential
- Almost all software products are now developed using agile
- Agile software engineering focuses on delivering functionality quickly and responding to changing product specifications and minimizing development overhead
- There are a large number of agile methods that have been developed
    - No best technique or method
    - It depends on who is using the technique, the type of development team, and the type of product being developed
- Agile methods:
    - Project-driven development evolved to support the engineering of large, long-lifetime systems where teams may be geographically dispersed and work on the software for several years
        - This approach is based on controlled and rigorous software development processes that include detailed project planning, requirements specification and analysis, and system modeling
        - Project-driven development involves significant overheads and documentation and it does not support the rapid growth and delivery of software
    - Agile methods were developed in the late 90s to address this issue
        - Agile focuses on the software rather than its documentation develop software in a series of increments and aims to reduce process bureaucracy as much as possible
    - Incremental development:
        - All agile methods are based on incremental development and delivery
        - Product development focuses on the software features where a feature does something for the software user
        - With incremental development, you prioritize features that are most important to implement first
            - Only define details of the feature being implemented in an increment
            - The feature is then implemented and delivered
        - Users or surrogate users can try it out and provide feedback to the development team, then you can go on and implement the next feature of the system
    - Scrum:
        - Software company managers need information that will help them understand how much it costs to develop a software product, how long it will take, and when the product can be brought to market

        - Plan-driven development provides this information through long-term development plans that identify deliverables - items the team will deliver and when these will be delivered
        - Plans always change so anything apart from short-term plans is unreliable
        - Scrum is an agile method that provides a framework for agile project organization and planning, it does not mandate any specific technical practices
    - Scrum and Sprints:
        - In scrum, the software is developed in sprints, usually 2-4 week periods in which software features are developed and delivered
        - During a sprint, the team has daily meetings to review progress and to update the list of incomplete work items
        - Sprints should produce a shippable product increment, which means that the developed software should be complete and ready to deploy
    - Key Scrum practices:
        - Product backlog - a to-do list of items to be implemented that is reviewed and updated before each sprint
        - Timeboxed sprints - fixed periods in which items from the product backlog are implemented
        - Self-organizing teams - teams that make their own decisions and work by discussing issues and making decisions by consensus
- Agile activities:
    - Scrum does not suggest the technical agile activities that should be used, however, two practices should always be used in a sprint
    - Test automation - as far as possible, product testing should be automated, should develop a suite of executable tests that can be run at any time
    - Continuous integration - components that change should be immediately integrated with other components to create a system, the system should then be tested for unanticipated component interaction problems
- Extreme Programming:
    - Most influential work that has changed software development culture
    - Coined by Kent Beck in 1998 as the approach was developed by pushing recognized good practices, such as iterative development to ‘extreme’ levels
    - Focused on 12 new development techniques geared to rapid, incremental software development, change, and delivery

Requirements Engineering:

- Process of establishing the needs of the stakeholder that are solved by software
    - Setting constraints
- Set of activities concerned with identifying and communicating the purpose of a software-intensive system and the context in which it will be used
- Acts as the bridge between the real-world needs of users, customers, and other constituencies affected by a software system and the capabilities and opportunities afforded by software-intensive technologies
- Inception: a set of questions that establish

        - Basic understanding of the problem
        - People who want a solution
        - Nature of the solution that is desired
        - Effectiveness of preliminary communication and collaboration between the customer and developer
- Elicitation: elicit requirements from all stakeholders
- Elaboration: create an analysis model that identifies data, function, and behavioral requirements
- Negotiation: agree on a deliverable system that is realistic for developers and customers
- Specification: can be any one or more of the following:
    - Written document
    - Set of models
    - Formal mathematical
    - Collection of user scenarios
    - Prototype
- Validation: review mechanism that looks for
    - Errors in content or interpretation
    - Areas where clarification may be required
    - Missing information
    - Inconsistencies
    - Conflicting or unrealistic requirements
- Requirements management
    - Functional requirements:
        - Defines the required behavior of the system between outputs and inputs
        - Describes how a product must behave and its features/functions
    - Non-functional requirements:
        - Quality attribute, performance attribute, security attribute, or general system constraint

Features and Acceptance Criteria:

- Represent a chunk of functionality that delivers considerable business value and fulfills a stakeholder need, is a collection of user stories and should be done within 2-3 months max
- Description: the context of the feature, how users are going to use it, focuses on what the need is rather than how to implement said feature
- Feature acceptance criteria: conditions to mark the feature as done

User Story Template: As a <type of user>, I want to <do something>, so that <some value is created>.

Software engineering is more than writing code:

- Its problem solving: Analysis (understanding the nature of the problem and breaking the problem into pieces) and Synthesis (putting the pieces together into a large structure)
    - Creating a solution
    - Engineering a system based on the solution
- Modeling
- Knowledge acquisition

- Rationale management
- Techniques: formal procedures for producing results using some well-defined notation
- Methodologies: a collection of techniques applied across software development and unified by a philosophical approach
- Tools: instruments or automated systems to accomplish a technique
    - CASE: Computer-Aided Software Engineering
- Dealing with complexity:
    - Notations (UML, OCL)
    - Requirements for engineering, analysis, and design
        - OOSE, SA/SD, scenario-based design, formal specifications
    - Testing
        - Vertical and horizontal testing
- Dealing with change:
    - Rationale management
        - Knowledge management
    - Release management
        - Big bang vs continuous integration
    - Software lifecycle
        - Linear models
        - Iterative models
        - Activity vs entity based views
- Application of these views
- Abstraction: complex systems are hard to understand
    - 7 +- 2 phenomena -> our short term memory cannot store more than 7 +- 2 pieces at the same time due to brain limitation
    - Chunking:
        - Group collection of objects to reduce complexity
        - Area code, common prefix, office-part
    - Allows us to ignore unessential details
    - Thought process where ideas are distanced from objects (abstraction as activity)
    - The resulting idea of a thought process where an idea has been distanced from an object (abstraction as an entity)
    - Models can express ideas
- Model: an abstraction of a system that enables us to answer questions about the system
    - A system that no longer exists
    - Existing system
    - Future systems to be built
- Why use models to describe software systems?
    - Object model: what is the structure of the system? (class diagram)
    - Functional model: what are the functions of the system? (use case diagram)
    - Dynamic model: how does the system react to external events? (activity and sequence diagrams)
    - System model: object model + functional model + dynamic model
    - Task model:

        - PERT chart: what are the dependencies between tasks?
        - Schedule: how can this be done within the time limit?
        - Organization chart: what are the roles in the project?
- Technique to deal with complexity: Decomposition
    - A technique used to master complexity
    - Two major types of decomposition
        - Functional decomposition
        - Object-oriented decomposition
    - Functional decomposition
        - The system is decomposed into modules
        - Each module is a significant function in the application domain
        - Modules can be decomposed into smaller modules
        - Functionality is spread all over the system
        - The maintainer must understand the whole system to make a single change to the system
        - Consequence:
            - The source code is hard to understand
            - Source code is complex and impossible to maintain
            - The user interface is often awkward and non-intuitive
    - Object-oriented decomposition
        - The system is decomposed into classes or objects
        - Each class is a major entity in the application domain
        - Classes can be decomposed into smaller classes
    - Object-oriented vs functional decomposition
- Object-Oriented Development:
    - How do we think of the elements we are putting into the software
    - How we approach requirements analysis, design, and implementation
- OO Methodology: build a model of an application and then add details to it during design and do the same from design to implementation
    - Stages:
        - System conception - sponsors/users
            - Output - problem statement - rarely complete or correct
        - Analysis:
            - Analyst constructs models to restate requirements
            - The analyst must understand the problem
            - Produces concise, precise abstraction of what the system must do (not how)
                - No implementation decisions
            - 2 parts:
                - Domain model - real-world objects
                - Application model - parts of the application system that are visible to the user
            - System design: system architecture - performance characteristics

        - Class design: adds details to the analysis model by the system design strategy
            - Uses the same OO concepts and notation
            - Focus: data structures and algorithms
        - Implementation
            - Translates classes and relationships from the class design into a particular programming language
            - Should be straightforward with all the decisions already made
- Hierarchy:
    - So far we have abstractions that lead us to classes and objects
    - Another way to deal with complexity is to provide relationships between these chunks
    - One of the most important relationships is the hierarchy
        - 2 types: “part-of” and “is-kind-of”
- 3 ways to deal with complexity: abstraction, decomposition, and hierarchy
- Object-oriented decomposition is good but unfortunately, depending on the purpose of the system, different objects can be found
- How can we do it?
    - Start with a description of the functionality of a system then proceed to a description of its structure
- Ordering of development activities: software lifecycle
- Models must be falsifiable
    - Any model is a theory in software engineering
        - We build models and try to find counterexamples by:
            - Requirements validation, user interface testing, review of design, source code testing, system testing, etc…
    - Testing: the act of disproving a model
- Concepts and phenomena:
    - Phenomenon: an object in the world of a domain as you perceive it
    - Concept: describes the common properties of phenomena
    - A concept is a 3-tuple:
        - Name: distinguishes the concept from other concepts
        - Purpose: properties that determine if a phenomenon is a member of a concept
        - Members: a set of phenomena that are part of the concept
    - Abstraction is the classification of phenomena into concepts
    - Modeling is the development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details
- Systems: a system is an organized set of communicating parts
    - Natural system: a system whose ultimate purpose is not known
    - Engineered system: a system that is designed and built by engineers for a specific purpose
    - Parts of a system can be considered as systems again (subsystems)

- Model-driven development
    - Build a platform-independent model of the functionality and behavior of an application
        - Describe the model in modeling notation (UML)
        - Convert the model into a platform-specific model
    - Generate executable from platform-specific model
        - Advantages:
            - Code can be generated from the model (“mostly”)
            - Portability and interoperability
            - Model-Driven Architecture effort:
            - Object Management Group
- Application vs Solution Domain
    - An application domain (analysis)
        - The environment in which the system is operating
    - Solution domain (design and implementation)
        - Technologies used to build the system
    - Both domains contain abstractions that we can use for the construction of the system model

UML: what is it?

- Unified modeling language
    - The nonproprietary standard for modeling software systems
    - Converge of notations used in object-oriented methods
        - OMT
        - Booch
        - OOSE
- You can model 80% of most problems by using about 20% UML
- **Use case diagrams:** describe the functional behavior of the system as seen by the user
- **Class diagrams:** describe the static structure of the system including objects, attributes, associations
- **Sequence diagrams:** describe the dynamic behavior between objects of the system
- **Statechart diagrams:** describe the dynamic behavior of an individual object
- **Activity diagrams:** describe the dynamic behavior of a system within a particular workflow
- Core conventions:
    - All UML diagrams denote graphs of nodes and edges
        - Nodes are entities drawn as rectangles or ovals
        - Rectangles denote classes or instances
        - Ovals denote functions
    - Names of classes are not underlined
    - Names of instances are underlined
    - An edge between two nodes denotes a relationship between the corresponding entities
- UML provides a wide variety of notations for modeling many aspects of software systems

- Today we only concentrate on a few notations:
    - Functional model: use case diagram
    - Object model: class diagram
    - Dynamic model: sequence diagrams, statechart, activity diagram
- Class diagrams:
    - Outline: recall: system modeling = functional modeling + object modeling + dynamic modeling
    - Activities during object modeling:
        - The main goal is to find the important abstractions
        - Steps:
            - Class identification: based on the fundamental assumption that we can find abstractions
            - Find the attributes
            - Find the methods
            - Find the associations between classes
        - Order of steps
            - Goal: get the desired abstractions
            - Order of steps is secondary, only a heuristic
        - What happens if we find the wrong abstractions
            - We iterate and revise the model
    - Class identification: crucial to object-oriented modeling
        - Helps identify the important entities of a system
        - Basic assumptions:
            - We can find the classes for a new software system (forward engineering)
            - We can identify the classes in an existing system (reverse engineering)
        - Approaches:
            - Application domain approach
                - Ask domain experts to identify relevant abstractions
            - Syntactic approach
                - Start with use cases
                - Analyze the text to identify the objects
                - Extract participating objects from the flow of events
            - Design patterns approach
                - Use reusable design patterns
            - Component-based approach
                - Identify existing solution cases
        - One problem: definition of the system boundary:
            - Which abstractions are outside, which abstractions are inside the system boundary
                - Actors are outside the system
                - classes/objects are inside the system

        - Another problem: classes/objects are not just found by taking a picture of a scene or domain
            - The application domain must be analyzed
            - Depending on the purpose of the system different things might be found
                - How can we identify the purpose of a system?
                - Scenarios and use cases -> functional model
- Object-oriented development and UML/class diagrams:
    - Object-oriented development:
        - A class: template or blueprint for which to define an object
        - An object: is an instance of a class, includes data and operations that operate on the object
        - Why develop this way?
            - Enforces good design
            - Easier to change
            - Increases reuse
            - Increases understandability because we model real-world entities
        - Key Concepts:
            - Data over function
            - Information hiding - principle
            - Encapsulation - technique
                - Hide the inner details of how a class works
            - Inheritance
        - Hide the data behind the well-defined interface
        - Object-oriented process: we are creating the classes and the relationship between them in
            - **Requirements analysis**
                - Develop an object-oriented model of the application domain, objects in that model reflect the entities and operations associated with the problem to be solved
            - **Design**
                - Develop an object-oriented model of a software system to implement the identified requirements
            - **Implementation**
                - Implement the software design using an object-oriented programming language such as Java
        - Object-oriented requirements analysis: focus on the objects of the real-world objects
            - obtain/prepare a textual description of the problem
            - Underline nouns -> classes
            - Underline adjectives -> attributes
            - Underline active verbs -> operations
    - Class diagrams: used to represent requirements, class models
        - Model domain concepts

        - Classes do not represent software classes, only environment/domain entities
        - Used to represent the structure of designs
            - Most common use
        - A concept can be represented by a single UML class or a set of UML classes
        - Concepts share the same properties expressed as attributes and relationships between classes
        - A design pattern is a concept
- Static vs Non-Static:
    - Non Static attributes and operations - all objects of the class get their own copy - limited to the lifetime of the object
    - Static attributes and operations - all objects of the class share the same copy - not limited to the lifetime of any object
    - Use underline to show static attributes/operations
- Different kinds of relationships exist:
    - Associations are structural relationships between objects
        - Can be uni or bi-directional
        - Aggregations represent a whole-and-part relationship
        - Composition is a stronger form of aggregation
    - Dependency or objects of one class work briefly with objects of another
    - Generalizations represent inheritance
- Creation tips:
    - Understand the problem
    - Choose good class names
    - Concentrate on the what
    - Start with a simple diagram
    - Refine until you feel it is complete
- View of the analyst
    - Analyst is interested
        - In application classes: associations between classes are relationships between abstractions in the application domain
        - Operations and attributes of the application classes
    - The analyst uses inheritance in the model to reflect the taxonomies in the application domain
        - Taxonomy: as is-a-hierarchy of abstractions in an application domain
    - The analyst is not interested
        - In the exact signature of operations
        - In solution domain classes
- View of the designer
    - The designer focuses on the solution to the problem, that is, the solution domain

        - Associations between classes are now references between classes in the application or solution domain
        - A critical design task is the specification of interfaces:
            - The designer describes the interface of classes and the interface of subsystems
            - Subsystems originate from modules
                - The module is a collection of classes
                - The subsystem is a collection of classes with an interface
        - Subsystems are modeled in UML with a package
- Goals of the designer
    - The most important design goals for the designer design usability and design reusability
    - Design usability: interfaces are usable from as many classes as possible within the system
    - Design reusability: interfaces are designed in a way, that other software systems can also reuse them
        - Class libraries
        - Frameworks
        - Design patterns
- View of the implementer
    - Class implementor
        - Must realize the interface of a class in a programming language
        - Interested in appropriate data structures and algorithms
    - Class extender
        - Interested in how to extend a class to solve a new problem or to adapt to a change in the application domain
    - Class user
        - Class user is interested in the signatures of the class operations and conditions, under which they can be invoked
        - The class user is not interested in the implementation of the class
- Developers have different views on class diagrams
    - According to the development activity, a developer plays different roles
        - Analyst
        - System designer
        - Object designer
        - Implementer
    - Each of these roles has a different view of the class diagram
- Why do we distinguish different users of class diagrams?
    - Models often don’t distinguish between application classes and solution classes
        - Modeling languages like UML allow the use of both types of classes in the same model
            - Address book, array
        - No solution classes in the analysis model

        - Many systems do not distinguish between the specification and the implementation of a class
            - Object-oriented programming languages allow the simultaneous use of specification and implementation of a class
            - We distinguish between the analysis model and the object design model - the analysis design model does not contain any implementation specification
    - Analysis model vs. object design model:
        - The analysis model is constructed during the analysis phase
            - Main stakeholders: end-user, customer, analyst
            - Class diagrams contain only application domain classes
        - The object design model is created during the object design phase
            - Main stakeholders: class specifiers, class implementers, class users, and class extenders
            - Class diagrams contain application domain as well as solution domain classes
        - The analysis model is the basis for communication between analysts, application domain experts, and end users
        - The object design model is the basis for communication between designers and implementers
    - Who does not use class diagrams?
        - Client and end user are usually not interested in class diagrams
            - Clients focus more on project management issues
            - End users are more interested in the functionality of the system
- Modeling in action:
    - If it is a face
        - What are its attributes?
    - Or is it a mask?
    - Investigate the functional model
        - Who is using it? -> actors
            - Art collector
            - Bankrobber
            - Carnival participant
        - How is it used -> event flow
    - Napkin design of a mask to be used at the Venetian Carnival
- Object vs Class:
    - Object (instance): exactly one thing
        - This lecture is on object modeling
    - A class describes a group of objects with similar properties
        - Game, tournament, mechanic, car, database
    - Object diagram: a graphical notation for modeling objects, classes, and their relationships
        - Class diagram: template for describing many instances of data. Useful for taxonomies, patterns, and schemata

        - Instance diagram: a particular set of objects relating to each other. Useful for discussing scenarios, test cases, and examples
- Purpose of an object diagram
    - The use of object diagrams is fairly limited, mainly to show examples of data structures
    - During the analysis phase of a project, you might create a class diagram to describe the structure of a system and then create a set of object diagrams as test cases to verify the accuracy and completeness of the class diagram
    - Before you create a class diagram, you might create an object diagram to discover facts about specific model elements and their links, or to illustrate specific examples of the classifier that are required
- **Summary:**
    - **System modeling:** functional modeling + object modeling + dynamic modeling
    - **Functional modeling:** from scenarios to use cases to objects
    - **Objects modeling is the central activity:** class identification is a major activity of object modeling where easy syntactic rules assist to find classes and objects
    - **Class diagrams are the "center of the universe" for the object-oriented developer** where the end-user focuses more on the functional model and usability
    - **Analysts, designers, and implementers have different modeling needs**
    - **There are three types of implementers with different roles during** class user, class implementor, and class extender
- Activity diagram: a flowchart diagram that is used to describe the dynamic aspect of the system.
    - Flow chart represents the flow of activities from one activity to another activity
        - Activities can be described as the operation of the system
        - The flow of control in the activity diagram is drawn from one operation to another
        - Flow can be sequential, branched, or concurrent
    - Purpose: activity diagrams are used for visualizing the dynamic nature of a system
        - It does not show any message flow from one activity to another
        - Can be described as to
            - Draw the activity flow of a system
            - Describe the sequence from one activity to another
            - Describe the parallel, branched, and concurrent flow of the system
    - Before drawing an activity diagram, we must have a clear understanding of the elements used in the activity diagram
        - First, we should identify the following elements:
            - Activities
            - Association
            - Conditions
            - Constraints

        - Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow, this mental layout is then transformed into an activity diagram
- Activity diagram components:
    - Initial node: a filled circle is the starting point of the diagram
    - Final node: a filled circle with a border is the ending point, an activity diagram can have zero or more activity final state
    - Activity: rounded rectangle represents activities/actions that occur. An activity is not necessarily a program, it may be a manual thing also
    - flow/edge: arrows in the diagram, no label is necessary
    - Fork: a black bar with one flow going into it and several leaving it. This denotes the beginning of parallel activities
    - Join a block bar with several flows entering it and one leaving it, this denotes the end of parallel activities
    - Merge a diamond with several flows entering and one leaving, the implication is that all incoming flows reach this point until processing continues
    - Sub-activity indicator: rake in the bottom corner of an activity, indicates that the activity is described by a more finely detailed activity diagram
    - Difference between a join and a merge
        - A join is different from a merge in that the join synchronizes two inflows and produces a single outflow. The outflow from a join cannot execute until all inflows have been received
        - A merge passes any control and flows straight through it. If two or more inflows are received by a merge symbol, the action pointed to by its outflows is executed two or more times
    - Decision: diamond with one flow entering and several leaving, flow leaving includes conditions as yes/no state
    - Flow final: a circle with X through it, indicates that the processes stop at this point
    - Swim lane: a partition in the activity diagram using a dashed line, called a swim lane, swim lanes may be horizontal or vertical

Component diagram:

- Introduction: UML component diagrams describe software components and their dependencies on each other
    - A component is an autonomous unit within a system
    - Components can be used to define software systems of arbitrary size and complexity
    - UML component diagrams enable to model of the high-level software components and the interfaces to those components
    - Important for component-based development (CBD)
    - Components and subsystems can be flexibly reused and replaced
    - A dependency exists between two elements if changes to the definition of one element may cause changes to the other
    - Component diagrams are often referred to as “wiring diagrams”

- The wiring of components can be represented on diagrams using components and dependencies between them
- A UML diagram classification:
    - Static: use case diagram, class diagram
    - Dynamic: state diagram, activity diagram, sequence diagram, collaboration diagram
    - Implementation: component diagram, deployment diagram
- UML component diagrams are:
    - Implementation diagrams: they describe the different elements required for implementing a system
- Other classifications:
    - Behavior diagrams:
        - A type of diagram that depicts the behavior of a system
            - Includes activity, state machine, use case diagrams, interaction diagrams
    - Interaction diagrams:
        - A subset of behavior diagrams that emphasize object interactions includes collaboration, activity, and sequence diagrams
    - Structure diagrams:
        - A type of diagram that depicts the elements of a specification that are irrespective of time
            - Includes class, composite structure, component, and deployment
- UML component diagrams are STRUCTURE DIAGRAMS
- Components: a modular unit with well-defined interfaces that is replaceable within its environment
    - An autonomous unit within a system:
        - Has one or more provided and required interfaces
        - Its internals are hidden and inaccessible
        - A component is encapsulated
        - Its dependencies are designed such that it can be treated as independently as possible
- Component notation: shown as a rectangle with
    - A keyword <<component>>
    - Optionally, in the right-hand corner, a component icon can be displayed
        - A component icon is a rectangle with two smaller rectangles jutting out from the left-hand side
        - This symbol is a visual stereotype
    - Component name
    - Components can be labeled with a stereotype, there are several standard stereotypes
    - A component can have:
        - Interfaces: an interface represents a declaration of a set of operations and obligations

        - Usage dependencies: a relationship in which one element requires another element for its full implementation
        - Ports: represents an interaction point between a component and its environment
        - Connectors: connect two components
            - Connect the external contract of a component to the internal structure
- Interface: component defines its behavior in terms of provided and required interfaces
    - An interface
        - Is the definition of a collection of one or more operations
        - Provides only the operations but not the implementation
        - Implementation is normally provided by a class/component
        - In complex systems, the physical implementation is provided by a group of classes rather than a single class
    - May be shown using a rectangle symbol with a keyword <<interface>> preceding the name
    - For displaying the full signature, the interfacing rectangle can be expanded to show details
    - Can be provided or required
    - A provided interface:
        - Characterizes services that the component offers to its environment
        - Is modeled using a ball, labeled with the name, attached by a solid line to the component
    - A required interface:
        - Characterizes services that the component expects from its environment
        - Is modeled using a socket, labeled with the name, attached by a solid line to the component
    - Where two components/classes provide and require the same interface, these two notations may be combined
    - The ball-and-socket notation hint at the interface in question serves to mediate interactions between the two components
    - If an interface is shown using the rectangle symbol, we can use an alternative notation, using dependency arrows
    - In a system context where multiple components require or provide a particular interface, a notation abstraction can be used that combines joining the interfaces
    - A component: specifies a CONTRACT of the services that it provides to its clients and that it requires from other components in terms of its provided and required interfaces
        - Can be replaced
        - The system can be extended

        - Dependencies: components can be connected by usage dependencies
        - Usage dependency:
            - A usage dependency is a relationship in which one element requires another element for its full implementation
            - Is a dependency in which the client requires the presence of the supplier
            - Is shown as a dashed arrow with a <<use>> keyword
            - The arrowhead point forms the dependent component of the one of which it is dependent
        - Port: specifies a distinct interaction point
            - Between that component and its environment
            - Between that component and its internal parts
        - Is shown as a small square symbol
        - Ports can be named, and the name is placed near the square symbol
        - Is associated with the interfaces that specify the nature of the interactions that may occur over a port
        - Ports can support unidirectional communication or bi-directional communication
            - If there are multiple interfaces associated with a port, these interfaces may be listed with the interface icon, separated by commas
        - All interactions of a component with its environment are achieved through a port
        - The internals are fully isolated from the environment
        - This allows such a component to be used in any context that satisfies the constraints specified by its ports
- External view:
    - Components have an external view and an internal view
    - An external view shows publicly visible properties and operations
    - An external view of a component using interface symbols sticking out of the component box
    - The interface can be listed in the component box
- Internal view:
    - An internal or white box view of a component is where the realizing class/components are nested within the component shape
    - Realization is a relationship between two sets of model elements
        - One represents a specification
        - The other represents an implementation of the latter
    - The internal class that realizes the behavior of a component may be displayed in an additional compartment
    - Compartments can also be used to display ports, connectors, or implementation artifacts
    - An artifact is the specification of a physical piece of information
    - Components can be built recursively

- Assembly
    - Two kinds of connectors:
        - Delegation
        - Assembly
    - Assembly connector:
        - A connector between two components defines that one component provides the services that another component requires
        - He must only be defined from a required interface to a provided interface
        - An assembly connector is notated by a “ball-and-socket” connection
        - This notation allows for succinct graphical wiring of components
    - Semantics:
        - The semantics for an assembly connector are signals that travel along an instance of a connector originating in a required port and delivered to a provided port
        - Interfaces provided and required must be compatible
        - Interface compatibility between provided and required ports that are connected enables an existing component in a system to be replaced
        - Multiple connections directed from a single required interface to provided interfaces indicate that the instance that will handle the signal will be determined at the execution time
- Delegation:
    - A delegation connector links the external contract of a component to the internal realization
    - Represents the forwarding of signals
    - Must only be defined between user interfaces or ports of the same kind
    - The target interface must support a signature compatible with a subset of operations of the source interface
    - A port may delegate to a set of ports on subordinate components
    - The Union of the target interfaces must be signature compatible with the source interface
    - Semantics:
        - Is the declaration that behavior that is available on a component instance is not realized by that component itself, but by another instance that has compatible capabilities
        - Is used to model the hierarchical decomposition
        - Message and signal flow will occur between the connected ports
- Deployment diagrams:
    - A strong link between component diagrams and deployment diagrams
    - Deployment diagrams
        - Show the physical relationship between hardware and software in a system

        - Hardware elements:
            - Computers
            - Embedded processors
            - Devices such as sensors
        - Are used to show the nodes where software components reside in the run-time system
    - Contains nodes and connections
    - A node usually represents a piece of hardware in the system
    - A connection depicts the communication path used by the hardware to communicate
    - Usually indicates the method such as TCP/IP
    - Contain artifact
    - An artifact:
        - Is the specification of a physical piece of information
        - Ex. source files, binary executable files, table in a database system
        - An artifact defined by the user represents a concrete element in the physical world
    - An artifact manifests one or more model elements
    - A <<manifestation>> is the concrete physical of one or more model elements by an artifact
    - This model element is often a component
    - A manifestation is notated as a dashed line with an open arrow-head labeled with the keyword <<manifest>>

Dynamic Modeling:

- We distinguish between two types of operations:
    - Activity: operations that take time to complete
        - Associated with states
    - Action: instantaneous operation
        - Associated with events
- A state chart diagram relates events and states from one class
- An object model with several classes with interesting behavior has a set of state diagrams
- Notation is based on work by Harel
- Added are a few object-oriented modifications
- State: an abstraction of the attributes of a class
    - The state is the aggregation of several attributes of a class
    - A state is an equivalence class of all those attribute values and links that do need to be distinguished
    - State has duration
- Dynamic modeling of user interfaces
    - Statechart diagrams can be used for the design of user interfaces
    - States: name of screens
    - Actions or activities are shown as bullets under the screen name

- Practical tips for dynamic modeling
    - Construct dynamic models only for classes with significant dynamic behavior
        - Avoid analysis by paralysis
    - Consider only relevant attributes
        - Use abstraction if necessary
    - Look at the granularity of the application when deciding on actions and activities
    - Reduce notational clutter
        - Try to put actions into superstate boxes
- States: passive or active (what an object is doing)
    - Pseudostates
        - Filled circle - start
        - Filled circle within a circle - end
        - Open diamond - a decision
        - X within a circle - error exit
    - You can show the internal behavior of a state
        - Advanced state behavior section/book
        - Internal transitions/book
- Transitions: causes the change of states from the current state to the target state
    - trigger[guard]/action
        - A trigger is an event that may cause a transition from one state to another
        - Guard is a boolean condition that permits or blocks the transition
        - Action is an uninterruptible activity that executes while the transition takes place
- Class diagram - design:
    - Add data types after:
        - Add inputs/outputs for operations (signature)
        - Add implementation details
- Sequence diagram vs activity diagram:
    - A sequence diagram simply depicts an interaction between objects in a sequential order i.e. the order in which these interactions take place. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used to document and understand requirements for new and existing systems
    - An activity diagram is a flowchart diagram used to describe the dynamic aspect of the system. The flowchart represents the flow of activities from one activity to another activity. The activities can be described as the operation of a system. The flow of control in the activity diagram is drawn from one operation to another. This flow can be sequential, branched, or concurrent.
    - BOOK: activity diagrams give a process view of a system, and sequence diagrams give a logical view
- How do you find classes?
    - We have already established several sources for class identification
        - Application domain analysis: we find classes by talking to the client and identifying abstractions by observing the end user

        - General world knowledge and intuition
        - Textual analysis of event flow in use cases
    - Today we identify classes from dynamic models
    - Two good heuristics:
        - Actions and activities in state chart diagrams are candidates for public operations in classes
        - Activity lines in sequence diagrams are candidates for objects
- Dynamic modeling with UML:
    - Two UML diagram types for dynamic modeling:
        - Interaction diagrams describe the dynamic behavior between objects
        - Statechart diagrams describe the dynamic behavior of a single object
    - Two types of interaction diagrams:
        - Sequence diagram:
            - Describes the dynamic behavior of several objects over time
            - Interaction diagrams that detail how operations are carried out, capture the interaction between objects in the context of a collaboration. Sequence diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent the time what messages are sent and when
            - Collaboration diagram (communication diagram)
                - Shows how objects interact to perform the behavior of a particular use case or a part of a use case
            - Is a graphical description of the objects participating in a use case using a DAG (directed acyclic graph) notation
            - Heuristic for finding participating objects:
                - An event/msg always has a sender and a receiver
                - Find them for each event -> These are the objects participating in the use case
            - Sequence diagrams are derived from use cases
            - The structure of the sequence diagram helps us to determine how decentralized the system is
            - We distinguish two structures for sequence diagrams
                - Fork diagrams and stair diagrams
                - Fork diagram:
                    - Dynamic behavior is placed in a single object, usually a control object
                        - It knows all the other objects and often uses them for direct questions and commands
                - Stair diagram:
                    - Dynamic behavior is distributed, each object delegates responsibility to other objects
                        - Each object knows only a few of the other objects and knows which objects can help with a specific behavior

        - Fork or stair?
            - Object-oriented supporters claim that the stair structure is better
            - Modeling advice:
                - Choose the stair - a decentralized control structure if:
                    - The operations have a strong connection
                    - The operations will always be performed in the same order
                - Choose the fork - a centralized control structure if
                    - The operations can change the order
                    - New operations are expected to be added as a result of new requirements
- UML state chart diagram
    - State chart diagram: a state machine that describes the response of an object of a given class to the receipt of outside stimuli (events)
    - Activity diagram: a special type of state chart diagram, where all states are action states (Moore Automaton)

Object Oriented design:

- Name a principle of the OO approach:
    - Inheritance
    - Polymorphism
    - Encapsulation
    - Abstraction
    - Reusable code
- Encapsulation: you can feed a cat but you can't directly change how hungry the cat is
- Abstraction: cell phones are complex but using them is simple
- Inheritance: a private teacher is a type of teacher and any teacher is a type of person
- Polymorphism: triangle, circle, and rectangle, are all in the same collection
- OO methodology: build a model of an application then add details to it during the design
    - Do the same from design to implementation
    - Stages:
        - System conception - sponsors/users
            - Output - problem statement - rarely complete or correct
        - Analysis:
            - Analyst constructs models to restate requirements
            - The analyst must understand the problem
            - Produces concise, precise abstraction of what the system must do (not how) meaning no implementation decisions
            - 2 parts: domain model - real-world objects
                - Application model - parts of the application system that are available to the user
        - System design: system architecture or performance characteristics
        - Class design:

            - The class designer adds details to the analysis model per the system design strategy
            - Uses the same OO concepts and notation
            - Focus: data structures and algorithms
        - Implementation:
            - Translates the classes and relationships from the class design into a particular programming language
            - Should be straightforward - all the decisions already made
- Hierarchy: so far we have abstractions that lead us to classes and objects otherwise known as ‘chunks’
    - Another way to deal with complexity is to provide relationships between these chunks
    - One of the most important relationships is a hierarchy
    - Two special hierarchies:
        - “Part of” hierarchy (aggregation)
        - “Is-kind-of” hierarchy (taxonomy)

Design: to form a plan or scheme of, to arrange or conceive in the mind for later execution

- Software design is an iterative process through which requirements are translated into a blueprint for constructing the software
- Sets the stage for construction (implementation)
- *What almost every engineer wants to do - creativity rules where the requirements, business needs, and technical considerations all come together to formulate the product/system*
- Process of design:
    - During the design process, the software requirements specifications are transformed into design models
    - Models describe the details of data structures, system architecture, interface, and components
    - Each design product is reviewed for quality before moving to the next phase of software development
    - At the end of the design process, a design model and specification document are produced
    - This document is composed of the design models that describe the data, architecture, interfaces, and components
- Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other, the environment, and the principles guiding its design and evolution.
    - Where a system is a collection of components organized to accomplish a specific function or set of functions. The term system encompasses individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest. A system exists to fulfill one or more missions in its environment
    - The environment or context determines the setting and circumstances of developmental, operational, political, and other influences upon that system

        - A mission is a use case or operation for which a system is intended by one or more stakeholders to meet some set of objectives
        - A stakeholder is an individual, team, or organization with interests in, or concerns relative to a system
- Design specification models:
    - Data design: transforms class models into design class realizations and the necessary data structures required to implement the software
    - Architectural design: defines the relationships among the major structural elements of the software, the design patterns that can be used to achieve the requirements that have been defined for the system, and the constraints that affect how the architectural patterns can be applied
    - Interface design: describes how the software elements communicate with each other, with other systems, and with human users
    - procedural/component-level design: transforms structural elements of the software architecture into a procedural description of software components, information obtained from the class models and behavioral models serve as the basis for component design
- Design fundamental concepts:
    - Abstraction:
        - Data abstraction
        - Procedural abstraction
    - Architecture
    - Patterns:
        - A pattern is a common solution to a common problem in a given context, while architectural styles can be viewed as patterns describing the high-level organization of software (their macro architecture), other design patterns can be used to describe details at a lower, more local level (microarchitecture)
        - Creational patterns
        - Structural patterns
        - Behavioral patterns
        - Design pattern: enables a designer to determine whether the pattern:
            - This applies to the current work
            - Can be reused
            - Can serve as a guide for developing a similar, but functionally or structurally different pattern
    - Modularity
        - Easier to change
        - Easier to build
        - Easier to maintain
        - What is a module?
            - Has an inside and an outside - separated by a boundary
            - In computer software, a module is an extension to a main program dedicated to a specific functionality, in programming, a module is a

section of code that is added in as a whole or is designed for easy reusability
        - Can be a
            - Function
            - Class
            - Package
            - Microservice
- Information hiding:
    - The principle of information hiding says that a good split of modules is when modules communicate with one another with only the information necessary to achieve the s/w function
    - So information hiding enforces access constraints for both
        - Procedural detail with a module
        - The local data structure used by that module
    - Data hiding is a criterion for modular design
    - How to know what modules to create
    - Benefits:
        - Reduces the likelihood of side effects
        - Limits the global impact of local design decisions
        - Emphasizes communication through controlled interfaces
        - Discourages the use of global data
        - Leads to encapsulation - an attribute of high-quality design
        - Results in higher quality software
- Functional requirements
    - Cohesion - the degree to which a module performs one and only one function
        - High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes
        - Functional cohesion: every essential element for a single computation is contained in the module, functional cohesion performs the task and functions, it is an ideal situation
        - Sequential cohesion: an element outputs some data that becomes the input for another element, i.e., data flow between the parts, it occurs naturally in functional programming languages
        - Communicational cohesion: two elements operate on the same input data or contribute towards the same output data
        - Procedural cohesion: elements of procedural cohesion ensure the order of execution, actions are still weakly connected and unlikely to be reusable
        - Temporal cohesion: elements are related by their timing involved, In a module connected with temporal cohesion all the tasks must be executed in the same period, this cohesion contains the code

for initializing all the parts of the system, and lots of different activities occur all at unit time
        - Logical cohesion: the elements are logically related and not functionally
        - Coincidental cohesion: elements are not related, elements have no conceptual relationship other than the han location in source code, it is accidental and the worst form of cohesion
    - Coupling - the degree to which a module is “connected” to other modules in a system
        - Data coupling: if the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be coupled. In data coupling, the components are independent of each other and communicate through data, module communicates don’t contain tramp data
        - Stamp coupling: the complete data structure is passed from one module to another module, therefore it involves stamp data, which may be necessary due to efficiency factors - this choice was made by the insightful designer, not the lazy programmer
        - Control coupling: if the modules communicate by passing control information, they are said to be control coupled, it can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality
        - External coupling: modules depend on other modules, external to the software being developed or to a particular type of hardware
        - Common coupling: modules have shared data such as global data structures, and changes in global data mean tracing back to all modules which access that data to evaluate the effect of change, so it has disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability
        - Concept coupling: one module can modify the data of another module, or control flow is passed from one module to the other module, this is the worst form of coupling and should be avoided
    - Refinement
    - Refactoring
- Should strive for HIGH COHESION AND LOW COUPLING

Design Guidelines:

1. A design should exhibit an architecture that has been created using recognizable architectural styles or patterns, is composed of components that exhibit good design characteristics, and can be implemented in an evolutionary fashion
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems
3. A design should contain distinct representations of data, architecture, interfaces, and components

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns
5. A design should lead to components that exhibit independent functional characteristics
6. A design should lead to interfaces that reduce the complexity of connections between components and the external environment
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis
8. A design should be represented using a notation that effectively communicates its meaning

Design Principles:

1. The design process should not suffer from “tunnel vision”!
2. The design should be traceable to the analysis model!
3. The design should not reinvent the wheel!
4. The design should “minimize the intellectual distance” between the software and the problem as it exists in the real world!
5. The design should exhibit uniformity and integration!
6. The design should be structured to accommodate change!
7. The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered!
8. Design is not coding, coding is not design!
9. The design should be assessed for quality as it is being created, not after the creation!
10. The design should be reviewed to minimize conceptual errors!

Class Diagram - Design

- Add data types after :
- Add inputs/outputs for operations (signature)
- Add implementation details

Design Patterns:

- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
- Design challenges:
    - Designing software for reuse is hard. One must find:
        - A good problem decomposition, and the right software
        - A design with flexibility, modularity, and elegance
    - Designs often emerge from trial and error
    - Successful designs do exist
        - Two designs are rarely identical
        - They exhibit some recurring characteristics
    - Can designs be described, codified, or standardized?
        - This would short-circuit the trial-and-error phase
        - Produce “better” software faster
- A design pattern is a solution to a common software problem in a context
    - It describes a recurring software structure
    - Is abstract from a programming language

- Identifies classes and their roles in the solution to a problem
- Patterns are not code or designs; must be instantiated/applied
- Benefits of using patterns:
    - Patterns are a common design vocabulary
        - Allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation
        - Embodies a culture; domain-specific patterns increase design speed
    - Patterns capture design expertise and allow that expertise to be communicated
        - Promotes design reuse and avoids mistakes
    - Improve documentation (less is needed) and understandability (patterns are described well once)
- Gang of four (GoF) patterns
    - Creational patterns (abstracting the object-instantiation process)
        - Factory method
        - Builder
        - Abstract factory
        - Prototype
        - Singleton
    - Structural patterns (how objects/classes can be combined to form larger structures)
        - Adapter
        - Decorator
        - Proxy
        - Bridge
        - Facade
        - Composite
        - Flyweight
    - Behavioral patterns (communication between objects)
        - Command
        - Mediator
        - Strategy
        - Template method
        - Interpreter
        - Observer
        - Chain of responsibility
        - Iterator
        - State
        - Visitor
- How to describe a pattern:
    - A software design pattern has 4 essential parts
        - The pattern name is a handle we can use to describe a design problem, its solutions, and its consequences in a word or two,

finding good names has been one of the hardest parts of developing our catalog
        - The problem describes when to apply the pattern and explains the problem and its context
        - The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution does not describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations
        - The consequences are the results and trade-offs of applying the pattern, though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern
- Pattern-based design
    - A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system
    - The requirements model describes the problem set, establishes a context, and identifies the system of forces that hold sway
- What is a strategy pattern?
    - The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable, strategy lets the algorithm vary independently from clients that use it
    - Strategy: an algorithm separated from the object that uses it, and encapsulated as its object
        - Each strategy implements one behavior, one implementation of how to solve the same problem
        - Separates algorithm for behavior from an object that wants to act
        - Allows changing an object's behavior dynamically without extending/changing the object itself
- What is the observer pattern?
    - Defines a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically
    - Event-driven software rules
        - What is an event?
            - It's a kind of broadcast message that can be "heard" by any object that chooses to listen
            - A typical event message from an object is "I changed"
            - This enables objects who depend on that object to update themselves

        - Android depends heavily on events
            - Every sensor on the phone sends events
        - User interfaces run almost entirely on events
            - Every mouse motion, click, etc…
        - Java doesn't have events
        - Simulating events with callbacks
            - Classic software techniques (OS, games, OOD)
            - If the map control wants to receive events from the location manager, it "registers" with it
            - Later, when the location changes, the location mgr "calls back" map control (and all others who registered)
            - This is how we implement events with methods
    - Key points about observer pattern
        - The observer pattern defines a one-to-many relationship between objects
        - When the state of one object changes, all of its dependents are notified
        - Observers are loosely coupled in that the observable knows nothing about them, other than that they implement the observer interface
        - Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects
        - Java has several implementations of the observer pattern, including general-purpose Java. util.Observable
- Decorator pattern
    - Allows adding new behaviors to an object without modifying the class
    - Open/Closed Principle (OCP)
        - Classes should be open to extension but closed to modification
        - Extension:
            - composition of abstract types, not concrete classes
                - Constructors and setters also take those abstract types
                - Composer delegates to the composed
            - Inheritance from those abstract classes and interfaces
        - Modification
            - Editing - allowed for fixing bugs, but not adding behavior
            - Danger - propagation of changes to numerous dependents

            - Costly invalidation of test cases
        - The closure is creating openings for extension
            - There is no ‘close’ operation but can use version control
- Adapter, factory, and more types of patterns
    - Making the concrete abstract
    - Factory pattern: Motivation
        - Correctly making objects is complex
        - Especially making a collection of related objects
            - Parts of a car
            - Look-and-feel of a window: canvas, scrollbar, etc…
        - The correct making of objects is not easily centralized in one place
            - Often do it all over code wherever an object is needed
        - DP angle is that “new” names and makes a concrete class; should be referring to abstract
            - The concrete class has to be named, but we can at least hide that from most of the system
            - The encapsulating class will violate OCP, but no others will
    - The simple factory is good, but not OCP
    - Adapter: Motivation
        - Real-world examples - database connectors, chat systems
        - Adapter:
            - Uses composition and delegation
            - The adapter (wrapper) is not a subclass of the wrapped object (as we are adapting)
    - OCP for observer
        - The display constructor takes the subject, and the subject register method takes an observer
        - Subject delegates to the observer
        - Open to adding observers without modification
    - OCP for decorator
        - Open to the addition of new drinks and condiments without modification to support all combinations

Test 2 Review:

- Abstract classes and interfaces are plentiful in Java code, and even in the Java development kit itself. Each code element serves a fundamental purpose:
    - Interfaces are a kind of code contract, which much be implemented by a concrete class
    - Abstract classes are similar to normal classes, with the difference that they can include abstract methods, which are methods without a body. Abstract classes cannot be instantiated

- Concrete Inheritance vs Interface class vs abstract class
    - Java is an object-oriented programming language that provides several mechanisms to create new classes based on existing classes
        - Inheritance, which allows a class to inherit properties and behaviors from another class
        - Abstract class, which is a class that can't be instantiated and is meant to be extended by other classes
        - Interfaces that declare a set of behaviors a class can implement
        - Classes can implement more than one interface, but they extend only one abstract class
        - Another difference is that interfaces can be implemented by classes or extended by interfaces, but classes can be only extended

Software Quality:

- Defined as: an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it
- Elements of software quality assurance (SQA):
    - Standards
    - Reviews and audits
    - Testing
    - error/defect collection and analysis
    - Charge management
    - Education
    - Vendor management
    - Security management
    - Safety
    - Risk management
- Role of SQA Group:
    - Prepares an SQA plan for a project
        - Plan identifies:
            - Evaluations to be performed
            - Audits and reviews to be performed
            - Standards that apply to the project
            - Procedures for error reporting and tracking
            - Documents to be produced by the SQA group
            - Amount of feedback provided to the software project team
    - Participates in the development of the project's software process description
        - The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards, and other parts of the software project plan
    - Reviews software engineering activities to verify compliance with the defined software process
        - Identifies, documents, and tracks deviations from the process and verifies that corrections have been made

    - Audits designated software work products to verify compliance with those defined as part of the software process
        - Reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made
        - Periodically reports the results of its work to the project manager
    - Ensures that deviations in software work and work products are documented and handled according to a documented procedure
    - Records any noncompliance and reports to senior management
        - Noncompliance items are tracked until they are resolved

SQA goals:

- Requirements quality: correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow
- Design quality: every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements
- Code quality: source code and related work products must conform to local coding standards and exhibit characteristics that will facilitate maintainability
- Quality control effectiveness: software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result

Software reliability:

- A simple measure of reliability is mean-time-between-failure where MTBF or mean-time-between-failure is equal to MTTF or mean-time-to-failure plus MTTR or mean-time-to-repair
- Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as Availability = [MTTF/(MTTF + MTTR)] * 100%

Software safety:

- Software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail
- If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards

ISO 9001:2008 Standard:

- ISO 9001:2008 is the quality assurance standard that applies to software engineering
- The standard contains 20 requirements that must be present for an effective quality assurance system
- Requirements delineated by ISO 9001:2008 address topics such as:
    - Management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques

Git and GitHub:

- What is a version control system?
    - A way to manage files and directories
    - Tracks change over time

    - Recall previous versions
    - Source control is a subset of the version control system
- Distributed version control
    - No central server
    - Every developer is a client, the server and the repository
- Git
    - Created by linus torvalds in April 2005
    - Replacement for BitKeeper to manage Linux kernel changes
    - A command line version control program
    - Uses checksums to ensure data integrity
    - Distributed version control (like BitKeeper)
    - Cross-platform (including Windows)
    - Open source, free
- Git distributed version control:
    - “If you’re not distributed, you’re not worth using” - linus torvalds
    - No need to connect to the central server
    - Can work without an internet connection
    - No single failure point
    - Developers can work independently and merge their work later
    - Every copy of a git repository can serve either as the server or as the client (and has complete history)
    - Git tracks changes, not versions
    - Bunch of little change sets floating around
- Is Git for me?
    - People primarily working with source code
    - Anyone wanting to track edits (especially changes to text files)
        - Review the history of changes
        - Anyone wanting to share, merge changes
    - Anyone not afraid of command line tools
- Most popular languages used with Git
    - Html, CSS, javascript, python, asp, scala, shell scripts, PHP, ruby, ruby on rails, Perl, java, c, c++, c#, objective c, Haskell, CoffeeScript, ActionScript
- What is a repository?
    - “Repo” = repository
    - Usually used to organize a single project
    - Repos can contain folders and files, images, videos, spreadsheets, and datasets - anything your project needs
- Two-tree architecture on other version control systems
    - The repository checks out working then working commits to the repository
- Git uses a three-tree architecture
    - Repo checks out the staging index, then the staging index checks out working, then working adds to the staging index and the staging index commits to the repository
- What is GitHub?

    - A platform to host git code repositories
    - Launched in 2008
    - Most popular Git host
    - Allows users to collaborate on projects from anywhere
    - GitHub makes git social
    - Free to start
- Important to remember
    - Sometimes developers choose to place repo on GitHub as a centralized place where everyone commits changes, but it doesn't have to be on GitHub
- How to copy (clone) files from the remote repo to the local machine
    - git clone URL <new_dir_name>
- How do I link my local repo to a remote repo?
    - git remote add <alias> <URL>
    - Which remotes am I linked to?
        - git remote
- Pushing to a remote repo
    - git push local_branch_alias branch_name
- Fetching from a remote repo
    - git fetch remote_repo_name
    - Fetch in no way changes your working dir or any commits that you've made
        - Fetch before you work
        - Fetch before you push
        - Fetch often
    - git merge must be done to merge fetched changes into a local branch
- git commit -a
    - Allows one to add to the staging index and commit at the same time
    - Grabs everything in a working directory
    - Files not tracked or being deleted are not included
    - git log --oneline
        - Gets first line and checksum of all commits in the current branch
- git diff
    - When using the checksum of older commits, will show you all changes compared to those in your working directory
- Renaming and deleting branches
    - git branch -m/--move old_name new_name
    - git branch -d branch_name
    - git branch -D branch_name
- Tagging - git tag can tag specific points in history as being important, such as releases versions
    - Two types of tags:
        - Lightweight - a pointer to a specific comment - an SHA stored in a file
            - git tag tag_name

        - Annotated - a full object stored in the Git database - SHA, tagger name, email, data, message and can be signed and verified with GNU Privacy Guard (GPG)
            - git tag -a tag_name -m “message”
    - How do I see tags?
        - git show tag_name
- Git workflow:
    - Initialize the project in a directory using git init
    - Add a file using a text editor to the directory
    - Add every change that has been made to the directory using git add
    - Commit the change to the repo using git commit -m “important message here”
    - After initializing a new git repo:
        - Commit the changes with a message 3
        - Add the changes 2
        - Make the changes 1
- A note about commit messages
    - Tell what it does in the present tense
    - Single line summary followed by blank space followed by a more complete description
    - Keep lines to less than 72 characters
    - A ticket or bug number helps
- How to see what was done?
    - git log
- The HEAD pointer
    - Points to a specific commit in the repo
    - As new commits are made, the pointer changes
    - HEAD always points to the “tip” of the currently checked-out branch in the repo
        - Not the working directory or staging index
    - Last state of the repo or what was checked out initially
    - HEAD points to the parent of the next commit
        - Where writing the next commit takes place
- Which files were changed and where do they sit in the three trees?
    - git status - allows one to see where files are in the three-tree scheme
- What changed in the working directory?
    - git diff - compares changes to files between the repo and working directory
- Deleting files from the repo
    - git rm filename.txt
        - Moves deleted file change to a staging area
        - It is not enough to delete the file in your working directory, you must commit the change
- Moving/renaming files
    - git mv filename1.txt filename2.txt
- 75% of the time, you’ll only use these commands
    - git init

    - git status
    - git log
    - git add
    - git commit
    - git diff
    - git rm
    - git mv
- What if I want to undo changes made to the working directory?
    - git checkout something
    - git checkout will grab the file from the repo
    - git checkout –file1.txt
- What if I want to undo changes added to the staging area?
    - git reset HEAD filename.txt
- What if I want to undo changes committed to the repo?
    - git commit --amend -m “message”
        - Allows one to amend a change to the last commit
        - Anything in the staging area will be amended to the last commit
- Which files are in a repo?
    - git ls-tree tree-ish
        - Tree-ish is a way to reference a repo; full SHA, part SHA, HEAD, others
- Branching:
    - Allows one to try new ideas
    - If an idea doesn’t work, throw away the branch, don’t have to undo many changes to the master branch
    - If it does work, merge ideas into the master branch
    - There is only one working directory
- In which branch am I?
    - git branch
- How do I create a new branch?
    - git branch new_branch_name
- How do I switch to a new branch?
    - git checkout new_branch_name
- Comparing branches: git diff first_branch … second_branch
- How do I merge a branch?
    - From the branch into which you want to merge another branch
    - git merge branch_to_merge
    - “Fast-forward” merge occurs when the HEAD of the master branch is seen when looking back
    - “Recursive” merge occurs by looking back and combining ancestors to resolve the merge
- Merge conflicts
    - What if there are two changes to the same line in two different commits?
    - Resolving merge conflicts
        - git will notate the conflict in the files

        - Solutions:
            - Abort the merge using git merge -abort
            - Manually fix the conflict
            - Use a merge tool
- Graphing merge history
    - git log --graph --oneline --all --decorate
- Tips to reduce merge pain
    - Merge often
    - Keep commits small/focused
    - Bring changes occurring to master into your branch frequently (“tracking”)

What is Git?

- git is a DVCS tool
- A major difference between git and other VCS: if files have not changed, Git doesn’t store the file again - just a link to the previous identical file
- Features
    - Nearly every operation is local
    - Git has three main states that your file can reside in committed, modified, and staged
    - Folk and pull model
- Modified
    - This means that you have changed the file but have not committed it to your database yet
- Staged
    - means mark a modified file which will go into your next commit
        - If it’s modified and added to the staging area, it is staged
        - Otherwise, it's modified
- Committed
    - This means that the data is safely stored in your local database
    - So Git can trace all those changes

What is a push?

- When you have your project at a point that you want to share, you have to push it upstream
- Push the local change to the origin repo. The -u tells Git to remember the parameters so that next time we can simply run git push

What is a pull?

- Fetch from and integrate with another repository or a local branch
- Fetch the specified remote’s copy of the current branch and immediately merge it into the local copy

Create a repo on GitHub

- Create a repo on the GitHub website and push your commit

Build the environment

- Folk the repository on the GitHub website
- Clone your folk
- Configure the remote

First successful bugfix and merge

- Create the branch and switch to this branch
- Make change at your branch
- Switch to the master repo and merge the code
- Push your change back to GitHub

Software Configuration Management:

- The process of controlling the evolution of a software system
- Simplifies sharing code and other documents
- Ability to revert to an older version
- Coherently integrate contributions from team members
- Notify interested parties about new modifications
- Track software issues
- Create an auditing trail
- SCM system allows us to answer questions
    - When was this software module last changed?
    - Who made changes in this line of code?
    - What is the difference between the current version and last week's?
    - How many lines of code did we change in this release?
    - Which files are changed most frequently?
- SCM simplifies collaboration and increases productivity
- Can all be done with old-fashioned email, file system, etc…
- A software configuration is a collection of work products created during a project's lifetime
- Software configuration item or SCI is any piece of knowledge created by a person
- A snapshot of a software configuration or a configuration item represents the version of that item at a specific time
- A commit refers to submitting a software configuration to the project database
- A build is a version of a program code and a process by which program code is converted into a stand-alone form that can be run on a computer
- Repository vs Workspace:
    - The project database is usually on a remote server or mirrored on several servers
    - SCM provides a set of tools to interact with the project database
- Version graph and branching
    - Each graph node represents a version of the software configuration shared with the team
        - The private workspace of one team member was publicized in the shared repository and becomes a part of "team memory"
    - Think of branches as separate folders, each with its content and history
    - The project snapshot at the tip of a branch represents the latest version
- Example working scenarios
    - Undoing mistakes in new work
    - Supporting multi-pronged product evolution
        - Parallel versions coexist at all times, but will eventually merge

        - Developing product lines
            - Coexisting parallel versions that will never merge
                - But they still need synchronization so that versions for different markets are at the same “level”
        - Working with a small team of peers
            - All developers can write to the project database
        - Working with a managed team
            - Only the configuration manager can write to the project database
    - Undoing commits
        - We can undo a commit; undo a merge; or undo a checkout
    - Multi-pronged product evolution
        - Parallel versions coexist at times, but will eventually merge
    - Working with peers: centralized workflow
        - Two developers clone from the hub and both make changes
        - The first developer to push his changes back up can do so with no problems
        - The second developer must merge in the first one’s work before pushing changes up, to not overwrite the first developer’s changes
    - Working with a managed team
        - The configuration manager pushes the current version to the central project repository
        - A contributor clones that repository and makes changes
        - The contributor pushes the changed version to his public repository
        - Contributor notifies the configuration manager requesting to pull changes
        - The configuration manager adds the contributor’s repository as a remote and merges locally
        - The configuration manager pushes merged changes to the central project repository
    - Rebasing existing branches
        - Rebasing = moving a branch to a different baseline commit
            - Appears as if it forked off from a different version
            - All commits in the branch are replayed on a new base

Estimation:

- Importance of estimations
    - During the planning phase of a project, a first guess about cost and time is necessary
    - Estimating is often the basis for the decision to start a project
    - Estimations are the foundation for project planning and further actions
        - Estimating is one of the core tasks of project management, still considered black magic
- Challenges
    - Incomplete knowledge about:
        - Project scope and changes
        - Prospective resources and staffing
        - Technical and organizational environment

        - Infrastructure
        - Feasibility of functional requirements
    - Comparability of projects in case of new or changing technologies, staff, methodologies
    - Learning curve problem
    - Different expectations toward project manager
- Problems with estimations
    - Estimation results are almost always too high and have to be adjusted in a structured and careful manner
    - Reviews by experts are always necessary
    - New technologies can make new parameters necessary
    - Depending on the situation, multiple methods are to be used in combination
- Guiding principles
    - Documentation of assumptions about
        - Estimation methodology
        - Project scope, staffing, technology
    - Definition of estimation accuracy
    - Increasing accuracy with project phases
        - Ex. better estimation for implementation phase after object design is finished
    - Reviews by experienced colleagues
- Components of an estimation
    - Cost
        - Personnel (in-person days or valued in personnel cost)
            - Person day: the effort of one person per working day
        - Material (PCs, software, tools, etc.…)
        - Extra costs (travel expenses etc…)
    - Development time
        - Project duration
        - Dependencies
    - Infrastructure
        - Rooms, technical infrastructure, especially in offshore scenarios
- Estimating development time
    - Development time is often estimated by formula
        - Duration = effort/people
    - Problem with formula, because:
        - A larger project team increases communication complexity which usually reduces productivity
    - Therefore it is not possible to reduce duration arbitrarily by adding more people to a project
    - In the lectures on organization and scheduling we take a more detailed look at this issue
- Estimating personnel cost

        - Personnel type: team leader, application, domain expert, analyst, designer, programmer, tester…
        - Cost rate: cost per person per day
        - Two alternatives for cost rate:
            - Single cost rate for all types (no differentiation necessary)
            - Assign different cost rates to different personnel types based on experience, qualification, and skills
        - Personnel cost: person days * cost rate
- Estimating effort:
    - The most difficult part of project planning
        - Many planning tasks especially project schedules, depend on determination and effort
    - Basic principle:
        - Select an estimation model or build one first
        - Evaluate known information: size and project data, resources, software process, system components
        - Feed this information as parametric input data into the model
        - Model converts the input into estimates: effort, schedule, performance, cycle time
- Top-down and bottom-down estimation
    - Two common approaches for estimations
        - Top-down approach
            - Estimate effort for the whole project
            - A breakdown into different project phases and work products
            - Normally used in the planning phase when little information is available on how to solve a problem
            - Based on experiences from similar projects
            - Not appropriate for project controlling
            - Risk add-ons usual
        - Bottom-up approach
            - Start with effort estimates for tasks on the lowest possible level
            - Aggregate the estimates until top activities are reached
            - Normally used after activities are broken down the task level and estimates for the tasks are available
            - The result can be used for project controlling (detailed level)
            - Smaller risk add-ons
        - Often a mixed approach with recurring estimation cycles is used
- Estimation techniques
    - Expert estimates
    - Lines of code
    - Function point analysis
    - COCOMO I
    - COCOMO II
- Expert estimates

        - Guesses from experienced people
        - No better than the participants
        - Suitable for atypical projects
        - Result justification difficult
        - Important when no detailed estimation can be done due to lacking information about the scope
- Lines of code
    - The traditional way of estimating application size
    - Advantage: easy to do
    - Disadvantages:
        - Focus on the developer's point of view
        - No standard definition for "line of code"
        - "You get what you measure": if the number of lines of code is the primary measure of productivity, programmers ignore opportunities for reuse
        - Multi-language environments: hard to compare mixed-language projects with single-language projects
    - "The use of lines of code metrics for productivity should be regarded as professional malpractice"
- Function point analysis
    - Developed by allen albrecht, IBM research, 1979
    - Technique to determine size of software projects
        - Size is measured from a functional point of view
        - Estimates are based on functional requirements
    - Albrecht originally used the technique to predict effort
        - Size is usually the primary driver of development effort
    - Independent of
        - Implementation of language and technology
        - Development methodology
        - The capability of the project team
    - A top-down approach based on function types
        - Three steps: plan the count, perform the count, estimate the effort
- Steps in function point analysis:
    - Plan the count
        - Type the count: development, enhancement, application
        - Identify the counting boundary
        - Identify sources for counting information: software, documentation, and/or expert
    - Perform the count
        - Count data access functions
        - Count transaction functions
    - Estimate the effort
        - Compute the unadjusted function points UFP
        - Compute the value-added factor VAF
        - Compute the adjusted function points FA

        - Compute the performance factor
        - Calculate the effort in person days
- Function types
    - Data function types
        - # of internal logical files ILF
        - # of external interface files EIF
    - Transaction function types
        - # of external input EI
        - # of external output EO
        - # of external queries EQ
    - Calculate the UFP or unadjusted function points:
        - UFP = a * EI + b * EO + c * EQ + d * ILF + e * EIF
        - Where a-f are weight factors
- General system complexity factors
    - The unadjusted function points are adjusted with general system complexity GSC factors where each of the GSC factors gets a value from 0-5
- Calculate the effort
    - After the GSC factors are determined, compute the value-added factor (VAF):
    - Function points = unadjusted function point * value-added factor
    - FP = UFP * VAF
    - Performance factor
        - PF = number of function points that can be completed per day
    - Effort = FP/PF
- Advantages of function point analysis
    - Independent of implementation language and technology
    - Estimates are based on the design specification
        - Usually known before implementation tasks are known
    - Users without technical knowledge can be integrated into the estimation process
        - Incorporation of experiences from different organizations
    - Easy to learn
        - Limited time effort
- Disadvantages of function point analysis
    - A complete description of functions necessary
        - Often not the case in early project stages -> especially in iterative software processes
    - Only the complexity of specification is estimated
        - Implementation is often more relevant for the estimation
    - High uncertainty in calculating function points:
        - Weight factors are usually deducted from past experiences
    - Does not measure the performance of people
- COCOMO or constructive cost model
    - Developed by barry boehm in 1981
    - Also called COCOMO I or basic COCOMO

    - The top-down approach to estimate the cost, effort, and schedule of software projects, based on the size and complexity of projects
    - Assumptions
        - Derivability of effort by comparing finished projects (“COCOMO database”)
        - System requirements do not change during development
        - Exclusion of some efforts (for example administration, training, rollout, integration)
- Calculation of effort
    - Estimate the number of instructions
        - KDSI = “Kilo Delivered Source Instructions”
    - Determine project complexity parameters: A, B
        - Regression analysis, matching project data to the equation
    - 3 levels of difficulty that characterize projects
        - Simple project - organic mode
        - Semi-complex project - semi-detached mode
        - Complex project - embedded mode
    - Calculate effort
        - Effort = A * KDSI^b
    - Also called basic COCOMO
    - The effort is counted in person months: 152 productive hours
    - A and B are constants based on the complexity of the project
    - Calculation of development time
        - Basic formula: T = C * effort ^ d
        - Where t is the time to develop in months
        - C and D are constants based on the complexity of the project
        - The effort is the effort in person months
- Other COCOMO models
    - Intermediate COCOMO
        - 15 cost drivers yielding a multiplicative correction factor
        - Basic COCOMO is based on a value of 1.00 for each of the cost drivers
    - Detailed COCOMO
        - Multipliers depend on phase: Requirements; System design; detailed design; code and unit test; integrate and test; maintenance
- Steps in intermediate COCOMO
    - Basic COCOMO steps:
        - Estimate the number of instructions
        - Determine project complexity parameters: A, B
        - Determine the level of difficulty that characterizes the project
    - New step:
        - Determine cost drivers
            - 15 cost drivers c1, c1…c15
        - Calculate effort
            - Effort = A * KSIB * c1 * … c15

        - The effort is measured in PM
        - A and B are constants based on the complexity of the project
- Intermediate COCOMO: 15 cost drivers
    - Product Attributes
        - Required reliability
        - Database size
        - Product Complexity
    - Computer attributes
        - Execution time constraint
        - Main storage constraint
        - Virtual storage volatility
        - Turnaround time
    - Personal attributes
        - Analyst capability
        - Applications experience
        - Programmer capability
        - Virtual machine experience
        - Language experience
    - Project attributes
        - Use of modern programming practices
        - Use of software tools
        - Required development schedule
    - Rated on a qualitative scale between “very low” and “extra high”
    - Associated values are multiplied by each other
- COCOMO II
    - Revision of COCOMO I in 1997
    - Provides three models of increasing detail
        - Application composition model
            - Estimates for prototypes based on gui builder tools and existing components
        - Early design model
            - Estimates before software architecture is defined
            - The system design phase, closest to the original COCOMO, uses function points as size estimation
        - Post architecture model
            - Estimates once the architecture is defined
            - For the actual development phase and maintenance; uses FPs or SLOC as a size measure
    - The estimator selects one of the three models based on the current state of the project
    - Targeted for iterative software lifecycle models
        - Boehm’s spiral model
        - COCOMO I assumed a waterfall model
            - 30% design; 30% coding; 40% integration and test

    - COCOMO II includes new cost drivers to deal with
        - Team experience
        - Developer skills
        - Distributed development
    - COCOMO II includes new equations for reuse
        - Enables build vs buy trade-offs
- COCOMO II added cost drivers
    - Development flexibility
    - Team cohesion
    - Developed for reuse
    - Precedent
    - Architecture and risk resolution
    - Personnel continuity
    - Documentation matches life cycle needs
    - Multi-site development
- Advantages of COCOMO
    - Appropriate for a quick, high-level estimation of project costs
    - Fair results with smaller projects in a well-known development environment
        - Assumes comparison with past projects is possible
    - Covers all development activities (from analysis to testing)
    - Intermediate COCOMO yields good results for projects on which the model is based
- Problems with COCOMO
    - Judgment requirement to determine the influencing factors and their values
    - Experience shows that estimation results can deviate from the actual effort by a factor of 4
    - Some important factors are not considered:
        - Skills of team members, travel, environmental factors, user interface quality, overhead cost
- Online availability of estimation tools
    - Basic and intermediate COCOMO I (JavaScript)
    - COCOMO II (Unix, Windows, and Java)
    - Function point calculator (Java)
- Summary:
    - Estimation is often the basis for the decision to start, plan, and manage a project
    - Estimating software projects is an extremely difficult project management function
    - If used properly, estimates can be a transparent way to discuss project effort and scope
    - However, few software organizations have established formal estimation processes
        - Existing estimation techniques have lots of possibilities to influence the results - must be used with care
- GSC factors in function point analysis:

        - Data communications: how many communication facilities aid in the transfer or exchange of information with the system?
        - Distributed data processing: how are distributed data and processing functions handled?
        - Performance: does the user require a specific response time or throughput?
        - Platform usage: how heavily used is the platform where the application will run?
        - Transaction rate: how frequently are transactions executed?
        - Online data entry: what percentage of the information is entered online?
        - End-user efficiency: is the application designed for end-users?
        - On-line update: how many ILFs are updated online?
        - Complex processing: does the application have extensive logical or mathematical processing?
        - Reusability: will the application meet one or the user's needs?
        - Installation ease: how difficult are the conversion and installation?
        - Operational ease: how automated are start-up, backup, and recovery procedures?
        - Multiple sites: will the application be installed at multiple sites for multiple organizations?
        - Adaptability and flexibility: is the application specifically designed to facilitate change?

Software project management:

- Many software projects fail due to faulty project management practices
- Goal of software project management
    - Enable a group of engineers to complete a software project
- Responsibility of the project managers
    - Write the project proposal
    - Estimate the cost of the project
    - Create and maintain the schedule
    - Staff the project
    - Monitor and control the project
        - Metrics
    - Software configuration management
    - Risk management
    - Present status
    - Deal with problems
- Project planning
    - Need to ensure the project can be brought in on time, within schedule, and with the necessary quality
    - Problems that may keep this from happening:
        - Changing requirements from the customer
        - Customer not satisfied
        - Team morale problems
        - Delays
        - Poor quality

        - Project failure
- Documentation
    - Project Plan
    - Software process documentation
    - Standards
    - Risk management plan
    - Contract
- Software cost estimation
    - Determine the size of the product/project
        - Work breakdown structure
    - Determine effort needed
    - Determine project duration and cost
- Goals of a software project manager
    - To finish on time
    - To finish under budget
    - To meet requirements
    - To keep the customers happy
    - To ensure a happy team
- Staffing issues
    - Employees quit
    - Employees get sick
    - Employees get pulled off onto other projects
    - Employees have multiple projects
- Software metrics
    - Quantitative measurements of a software product and project can help management understand software performance, quality, or productivity, and efficiency of software teams
    - Types
        - Formal code metrics
            - SLOC (source lines of code)
            - Code complexity
            - Instruction path length
        - Developer productivity metrics
            - Active days
            - Assignment scope
            - Efficiency
            - Code churn
        - Agile process metrics
            - Lead time
            - Cycle time
            - Velocity
        - Operational metrics
            - MTBF
            - MTTR

- Test metrics
    - Code coverage
    - Percent of automated tests
    - Defects in production

Software testing:

- The process of exercising a program with the specific intent of finding errors before delivery to the end user
- What testing shows:
    - Errors
    - Requirements conformance
    - Performance
    - An indication of quality
- Strategic approach
    - To perform effective testing, you should conduct effective technical reviews, many errors will be eliminated by doing this before testing commences
    - Testing begins at the component level and works “outward” toward the integration of the entire computer-based system
    - Different testing techniques are appropriate for different software engineering approaches and at different points in time
    - Testing is conducted by the developer of the software and (for large projects) an independent test group
    - Testing and debugging are different activities, but debugging must be accommodated in any testing strategy
- Verification and validation:
    - Verification refers to the set of tasks that ensure that software correctly implements a specific function
    - Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements
        - Verification: “Are we building the product right?”
        - Validation: “Are we building the right product?”
    - The distinction between the two terms is largely to do with the role of specifications. Validation is the process of checking whether the specification captures the customer’s needs. Verification is the process of checking that the software meets the specification
- Who tests the software?
    - Developer - understands the system but, will test “gently” and, is driven by “delivery”
    - Independent tester - must learn about the system, but, will attempt to break it and, is driven by quality
- Testing strategy:
    - System engineering, analysis modeling, design modeling, code generation, unit test, integration test, validation test, system test
    - We begin by “testing-in-the-small” and move toward “testing-in-the-large”
    - For conventional software

    - The module is our initial focus
    - Integration of modules follows
- For OO software
    - Our focus when “testing in the small” changes from an individual module to an OO class that encompasses attributes and operations and pimples communication and collaboration
- Strategic issues
    - Specify product requirements in a quantifiable manner long before testing commences
    - State testing objectives explicitly
    - Understand the users of the software and develop a profile for each user category
    - Develop a testing plan that emphasizes “rapid cycle testing”
    - Build “robust” software that is designed to test itself
    - Use effective technical reviews as a filter before testing
    - Conduct technical reviews to assess the test strategy and test cases themselves
    - Develop a continuous improvement approach for the testing process
- Unit testing
    - Software engineer creates module to be tested then produces results after module to be tested has been tested
    - Module to be tested - interface, local data structures, boundary conditions, independent paths, and error handling paths are all things to be tested by test cases
    - Environment:
        - Driver -> module -> stub; produces results
- Integration testing strategies
    - Options:
        - The “big bang” approach
        - An incremental construction strategy
- Top-down integration
    - The top module is tested with stubs
    - Stubs are replaced one at a time “depth first”
    - As new modules are integrated, some subset of tests is re-run
- Bottom-up integration
    - Drivers are replaced one at a time, “depth first”
    - Worker modules are grouped into builds and integrated
    - Cluster
- Sandwich testing
    - Top modules are tested with stubs
    - Worker modules are grouped into builds and integrated
    - Cluster
- Regression testing

        - Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
        - Whenever software is corrected, some aspect of the software configuration is changed
        - Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors
        - Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools
    - General testing criteria
        - Interface integrity - internal and external module interfaces are tested as each module or cluster is added to the software
        - Functional validity - test to uncover functional defects in the software
        - Information content - test for errors in local or global data structures
        - Performance - verify specified performance bounds are tested
- Object-Oriented testing
    - Begins by evaluating the correctness and consistency of the analysis and design models
    - Testing strategy changes
        - The concept of the ‘unit’ broadens due to encapsulation
        - Integration focuses on classes and their execution across a ‘thread’ or in the context of a usage scenario
        - Validation uses conventional black-box methods
        - Test case design draws on conventional methods but also encompasses special features
- Broadening the view of “testing”
    - It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs appear at the analysis, design, and code levels. Therefore, a problem in the definition of class attributes that are uncovered during the analysis will circumvent side effects that might occur if the problem were not discovered until the design or code
- OO testing strategy
    - Class testing is the equivalent of unit testing
        - Operations within the class are tested
        - The state behavior of the class is examined
    - Integration applied three different strategies
        - Thread-based testing - integrates the set of classes required to respond to one input or event
        - Use-based testing - integrates the set of classes required to respond to one use case
        - Cluster testing - integrates the set of classes required to demonstrate one collaboration
- High order testing
    - Verification testing - the focus is on software requirements

        - System testing - the focus is on system integration
        - alpha/beta testing - the focus is on customer usage
        - Recovery testing - forces the software to fail in a variety of ways and verifies that recovery is properly performed
        - Security testing - verifies that protection mechanisms built into a system will protect it from improper penetration
        - Stress testing - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
        - Performance testing - test the run-time performance of software within the context of an integrated system
- Debugging
    - Effort
        - Time required to diagnose the symptom and determine the cause
        - Time required to correct the error and conduct regression tests
    - Symptoms and causes
        - Symptom and cause may be geographically separated
        - Symptoms may disappear when another problem is fixed
        - The cause may be due to a combination of non-errors
        - The cause may be due to a system or compiler error
        - The cause may be due to assumptions that everyone believes
        - Symptoms may be intermittent
    - Consequences of bugs
        - More damage as you go down the list: mild, annoying, disturbing, serious, extreme, catastrophic, infectious
        - Bug categories: function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc...
    - Techniques
        - Brute force/testing
        - Backtracking
        - Induction
        - Deduction
    - Correcting the error
        - Is the cause of the bug reproduced in another part of the program? In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere
        - What “next bug” might be introduced by the fix I’m about to make? Before the correction is made, the source code should be evaluated to assess the coupling of logic and data structures
        - What could we have done to prevent this bug in the first place? This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs

- Think – before you act to correct
- Use tools to gain additional insight
- If you're at an impasse, get help from someone else
- Once you correct the bug, use regression testing to uncover any side effects

Testability

- Operability - it operates cleanly
- Observability - results of each test case are readily observed
- Controllability - the degree to which testing can be automated and optimized
- Decomposability - testing can be targeted
- Simplicity - reduce complex architecture and logic to simplify tests
- Stability - few changes are requested during testing
- Understandability - of the design
- What is a good test?
    - A good test has a high probability of finding an error
    - A good test is not redundant
    - A good test should be "best of breed"
    - A good test should neither be too simple nor too complex
- Internal and external views
    - Any engineered product can be tested in one of two ways:
        - Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
        - Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised
- Test case design
    - Objective: to uncover errors
    - Criteria: in a complete manner
    - Constraint: with a minimum of effort and time
- Exhaustive testing
    - There are infinite possible paths, if we execute one test per millisecond, the test would take years to complete
- Selective testing
    - Test a selected path and loop it less than 20 times
- Software testing
    - White-box methods and black-box methods are methods within strategies
    - White-box testing - the goal is to ensure that all statements and conditions have been executed at least once
    - Why cover?
        - Logic errors and incorrect assumptions are inversely proportional to a path's execution probability
        - We often believe that a path is not likely to be executed; in fact, reality is often counterintuitive

        - Typographical errors are random; untested paths will likely contain some
    - Basis path testing
        - First, we compute the cyclomatic complexity:
            - Number of simple decisions + 1
                - Or
            - Number of enclosed areas + 1
        - In this case, V(G) = 4
    - Deriving test cases
        - Using the design or code as a foundation, draw a corresponding flow graph
        - Determine the cyclomatic complexity of the resultant flow graph
        - Determine a basis set of linearly independent paths
        - Prepare test cases that will force the execution of each path in the basis set
    - Black-box testing
        - How is functional validity tested?
        - How are system behavior and performance tested?
        - What classes of input will make good test cases?
        - Is the system particularly sensitive to certain input values?
        - How are the boundaries of a data class isolated?
        - What data rates and data volume can the system tolerate?
        - What effect will specific combinations of data have on system operation?
    - Model-based testing
        - Analyze an existing behavioral model for the software or create one
            - Recall that a behavioral model indicates how the software will respond to external events or stimuli
        - Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state
            - The inputs will trigger events that will cause the transition to occur
        - Review the behavioral model and note the expected outputs as the software makes the transition from state to state
        - Execute the test cases
        - Compare actual and expected results and take corrective action as required
- Testing
    - Observations
        - It is impossible to completely test any nontrivial module or system
            - Practical limitations: complete testing is prohibitive in time and cost
            - Theoretical limitations: halting problem
        - “Testing can only show the presence of bugs, not their absence”
        - Testing is not free
        - Define your goals and priorities
    - Testing takes creativity

    - To develop an effective test, one must have:
        - Detailed understanding of the system
        - Application and solution domain knowledge
        - Knowledge of the testing techniques
        - Skill to apply these techniques
    - Testing is done best by independent testers
        - We often develop a certain mental attitude that the program should behave in a certain way when in fact it does not
        - Programmers often stick to the data set that makes the program work
        - A program often does not work when tried by somebody else
- Testing activities
    - Unit testing -> integration testing -> system testing -> acceptance testing
    - Object design doc -> system design doc -> req analysis doc -> client expectation
- Types of Testing
    - Unit testing
        - Individual component (class or subsystem)
        - Carried out by developers
        - Goal: to confirm that the component or subsystem is correctly coded and carries out the intended functionality
    - Integration testing
        - Groups of subsystems and eventually the entire system
        - Carried out by developers and testers
        - Goal: test the interfaces among the subsystems
    - System testing
        - The entire system
        - Carried out by testers
        - Goal: to determine if the system meets the requirements
    - Acceptance testing
        - Evaluates the system delivered by developers
        - Carried out by the client, may involve executing typical transaction-site site on a trial basis
        - Goal: to demonstrate that the system meets the requirements and is ready to use
- When should you write a test?
    - Traditionally after the source code is written
    - In XP before the source code is written
        - Test-driven development cycle
            - Add a test
            - Run the automated tests
                - See the new one fail
            - Write some code
            - Run the automated tests

            - See them succeed
        - Refactor code
- Unit testing
    - Static testing (at compile time)
        - Static analysis
        - Review
            - Walk-through
            - Code inspection
    - Dynamic testing (at run time)
        - Black-box testing
        - White-box testing
- Static analysis with eclipse
    - Compiler warnings and errors
        - Possibly uninitialized variable
        - Undocumented empty block
        - The assignment has no effect
    - Checkstyle
        - Check for code guideline violations
    - Find bugs
        - Check for code anomalies
    - Metrics
        - Check for structural anomalies
- Black-box testing
    - Focus on I/O behavior
        - If for any given input, we can predict the output, then the component passes the test
        - Requires test oracle
    - Goal: reduce the number of test cases by equivalence partitioning
        - Divide input conditions into equivalence classes
        - Choose test cases for each equivalence class
    - Test case selection
        - Input is valid across a range of values
            - The developer selects test cases from 3 equivalence classes
                - Below the range
                - Within the range
                - Above the range
        - Input is only valid if it is a member of a discrete set
            - The developer selects test cases from 2 equivalence classes:
                - Valid discrete values
                - Invalid discrete values
        - No rules, only guidelines
- White-box testing

        - Code coverage
        - Branch coverage
        - Condition coverage
        - Path coverage
    - Unit testing heuristics
        - Create unit tests when object design is completed
            - Black-box test: test the functional model
            - White-box test: test the dynamic model
        - Develop the test cases
            - Goal: find an effective number of test cases
        - Cross-check the test cases to eliminate duplicates
            - Don't waste your time
        - Desk-check your source code
            - Sometimes reduces testing time
        - Create test harness
            - Test drivers and test stubs are needed for integration testing
        - Describe the test oracle
            - Often the result of the first successfully executed test
        - Execute the test cases
            - Re-execute test whenever a change is made "regression testing"
        - Compare the results of the test with the test oracle
            - Automate this if possible
    - JUnit overview
        - A Java framework for writing and running unit tests
            - Test cases and fixtures
            - Test suites
            - Test runner
        - Written by kent beck and Erich Gamma
        - Written with "test-first" and pattern-based development in mind
            - Tests are written before the code
            - Allows for regression testing
            - Facilitates refactoring
        - JUnit is open source

Technical Debt:

- A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now
- Reasons to discuss technical debt
    - Helps educate technical staff about business decision making
    - Helps to educate business staff about technical decision making
    - Raises awareness/transparency of important issues that are often buried
    - Allows technical debt to be managed more explicitly
    - The analogy is quite rich and produces numerous insightful parallels
- Reasons to Take on technical debt

        - Shortening the time to market
        - Preserving startup capital
        - Delaying development expense
- Reasons Not to Take on technical debt
    - Debt can create a vicious cycle of
        - Being behind schedule -> taking on debt -> reduced productivity -> back to being behind schedule
- Business vs. technical viewpoints
    - Business staff tend to be quite optimistic about the benefit of taking on the debt and the costs of carrying the debt
    - Technical staff tends to be pessimistic about the benefit of taking on the debt and the costs of carrying the debt
    - The truth is normally somewhere in the middle
    - The technical debt metaphor gives these two groups a constructive way to approach some important discussions
- Short-term vs. long-term debt
    - Short-term debt
        - Financial debt
            - Used to finance temporary cash flow issues or to cover receivables
            - The expectation is normally that debt will be repaid in the short term
        - Technical debt
            - Skipping unit tests to get a release out the door
            - “We don’t have time to implement this the right way; just hack it in and we’ll fix it later after we ship”
            - Violating coding standards to upload a hotfix
    - Long-term debt
        - Financial debt
            - Used to finance longer-term investments
            - The expectation is normally that there is some strategic reason for taking on long-term debt
        - Technical debt
            - “We don’t think we’re going to need to support a second platform for at least 5 years, so our design supports only one platform”
- Intentional vs unintentional debt
    - Intentional debt
        - “If we don’t get this release done, there won’t be a second release”
        - “We have code written by a contractor that doesn’t follow our coding standards, we’ll clean that up later”
        - “We didn’t have time to write unit tests for all the code we wrote the last two months of the project. We’ll write those after we release”
    - Unintentional debt
        - A junior programmer writes bad code

        - A major design strategy turns out poorly
        - Your company acquires a company that has a lot of technical debt
        - A comprehensive refactoring project goes sideways
- Interest/debt service
    - Interest is a premium paid above and beyond the principal that is in essence a service charge for being allowed to take on the debt
    - Interest can be paid periodically or all at once when the debt is repaid
    - To qualify as “debt,” there must be some kind of interest - now or later or both
    - Consider the debtor's interest rate
        - Not all debt is equal
        - Pay off high-interest debt before you pay off low-interest debt
        - Some low-interest debt may be low enough that you never pay it off
- Summary of categories of technical debt
    - Unintentional debt: debt incurred unintentionally due to low-quality work
        - Honest mistakes: “I wish we’d known framework 2.1 would be so much better than framework 2.0”
        - Careless mistakes: “Design? What design? Design is for sissies”
    - Intentional debt: debt incurred intentionally
        - Short-term debt: short-term debt, usually incurred reactively, for tactical reasons
            - Focused short-term debt: individually identifiable shortcuts
            - Unfocused short-term debt: numerous tiny shortcuts
        - Long-term debt: long-term debt, usually incurred proactively, for strategic reasons
- Not all delayed work is debt
    - Many kinds of delayed or incomplete work are not debt:
        - Feature backlog, deferred features, cut features
    - If the work doesn’t create an interest payment of some kind, it isn’t debt
- Ins and outs of the technical debt analogy
    - Credit rating
        - Different consumers/businesses have different borrowing power based on their credit rating
        - Teams have different abilities to carry technical debt responsibly
            - A team with a lot of unintentional debt due to low-quality work will have less ability to take on debt for strategic reasons than a team with a lower unintentional debt
    - Acquired debt
        - Debt was taken on for important strategic reasons, but you weren’t there at the time
        - Debt was taken on for reasons that you or your team would never have bought into if you had been there at the time
        - Sometimes you acquire debt from another company in an acquisition
    - 90 days same as cash

        - Sometimes this option is available, no interest accrues if you pay the debt off quickly enough
- Low monthly payments
    - When debt is incurred, what threshold level of activity is needed to make the “minimum payment”
        - Roll up a new version on a website?
        - Send out DVDs?
        - Replace a chip in a computer?
        - Replace a box on an airplane?
    - The ability to pay off debt depends in part on the kind of software the debt applies to
- Retiring debt
    - Teams normally find “working off debt” to be motivating
    - Special case: end-of-life system
        - Unlike financial debt, when a system is retired, all its technical debt is retired with it
        - As a system approaches the end of life, cost justifying anything other than the most expedient solution becomes increasingly difficult
- Deciding to take on technical debt
    - Avoid binary “debt/no debt” decisions - the universe of options is rarely that limited
        - Especially look for “zero interest” options
        - Give yourself the option of never paying off the debt
    - Key questions to ask before deciding to take on debt
        - Do we have estimates for the debt and non-debt options?
        - Do we believe the estimates?
        - How much do we reduce our effort now by taking on specific technical debt?
        - How much will the quick and dirty option cost now?
        - How much will the clean option cost now?
        - How much will it cost later to replace the Q&D option with the clean option?
        - How much will the “interest payment” on the Q&D option be?
        - How will the interest payment be structured or when will we have to pay “interest”?
        - Are there any issues involved in paying back the debt?
        - Why do we believe that it is better to incur the effort later than to incur it now? What is expected to change that will make the payment more palatable in the future than it is now?
        - Can we track this specific debt?
        - Have we considered all options - especially have we considered any zero interest options that we can implement at low cost now?
        - Who in the business is going to own the debt?

        - How much debt is enough/too much?
            - There is no one right answer for business debt, and there is no one right answer for technical debt
            - Technical staff's attitude is sometimes extreme: "zero debt"
            - Business staff tends to have a higher tolerance for technical debt (but shorter memory)
            - Work is required to ensure the organization remembers its debt decisions
        - Indicators of undesirable debt
            - High-interest rate
            - High minimum payment
            - Required, ongoing payments
            - Debt is untrackable
            - Debt is not intentional
            - Low ROI
- Tracking technical debt
    - All "good debt" can be tracked
    - Log as defects
    - Include in product backlogs
    - Monitor project velocity
    - Monitor the amount of rework
    - Ways to measure debt
        - Total of debt in the product backlog
        - Maintenance budget
        - "Aged" customer work backlog
        - Considering measuring debt in money, not features, e.g. "50% of the R&D budget is nonproductive maintenance work"
- Paying down technical debt
    - Good and bad reasons to pay down debt
        - Good reasons are normally based on business needs/business benefit
        - Bad reasons tend to be very technical sounding
- Bad reasons to pay down technical debt
    - Business is rarely motivated to pay down technical debt because the design is "bad", "old", "crufty", etc…
    - Justification for reducing debt needs to be tied to business benefit
    - Some debt reduction can be planned into the normal workflow as part of doing a good job and doesn't have to be justified separately
- Approaches to paying down technical debt
    - Do a debt reduction iteration immediately following full product release - short-term debt
    - Amortize small debt payments into each iteration
    - Above a certain threshold, debt reduction has to be approved as a separate project
    - Individuals rotate through debt-reduction duty

        - Offshore resources are used to reduce debt
    - What can we do with technical debt?
        - Talk with technical staff about it
        - Communicate to business staff the implications of it
        - Measure the amount of it
        - Make explicit decisions about whether we should take on more of it
        - Get input from the business about whether the business believes we should have less or more of it
        - Strategize how to avoid it
        - Track it
        - Make explicit decisions about when and how to reduce it

STIGs, SCAP, and Data Metrics:

- What is a STIG?
    - Security technical implementation guide:
        - A compendium of DOD policies, security regulations, and best practices for securing an AI or IA-enabled device
        - A Guide for information security
        - Mandated in DODD 8500.1, DODI 8500.2
        - Endorsed by CJCSI 6510.01, AR 25-2, and AFI 33-202
    - Goals
        - Intrusion avoidance
        - Intrusion detection
        - Response and recovery
        - Security implementation guidance
- Customer challenges
    - Limited resources available to assess compliance with numerous requirements
        - Manual efforts
        - Impacts of different approaches for different technologies
    - Understanding what documents apply
    - Need for guidance in new technologies
    - Support for new releases of technologies used
    - Ability to extract information into other databases or products
- Maintenance challenges
    - High demand from the user community for new and updated security guidance
    - The rapid pace of new technology
    - Limited resources to develop guidance and tools to evaluate compliance
    - Development and maintenance of varying tools/techniques for supporting compliance check
        - Gold disk (windows)
        - Security readiness review scripts (Unix, some DB)
- General challenges
    - Secure product development
        - No master list of all requirements for products
        - Vendors do not know, in detail, what requirements they have to meet

        - Not knowing “when they are done”
    - IA compliance reporting
        - Determining compliance statistics
        - Inability to validate that all requirements are addressed in current checklists
        - Inconsistent reporting of findings and compliance status
    - Security guide development
        - High demand for new and updated security guidance
        - Duplication of requirements
        - Interpretation of DoD IA controls
        - Requirements not written in a measurable format
        - Inconsistency in documents from different sources
- SCAP: our way ahead
    - Security Content automation protocol (SCAP) is a collection of specifications
        - Specifications originally developed by the government are now being adopted as the industry standard
        - Supports a standards-based approach to develop and publish IA configuration guidance, assess assets, and report compliance
    - Benefits of SCAP
        - Enables vendor community to develop standardized guidance once for use by all communities
        - Allow more commercial assessment tools to utilize DoD configuration guidance
        - Requires less time to develop and publish additional guidance
    - Specifications
        - CVE - common vulnerabilities and exposures
            - Common naming of emerging vulnerabilities
        - CCE - common configuration enumeration
            - Common naming of configuration vulnerabilities
        - CPE - common platform enumeration
            - Language to describe operating systems/platforms
        - CVSS - common vulnerability scoring system
            - Scoring system to describe the severity of a vulnerability
        - XCCDF - extensible configuration checklist description format
            - XML definition of a checklist
        - OVAL - open vulnerability and assessment language
            - A common language for assessing the status of a vulnerability
        - OCIL - open checklist interactive language
            - A common language to express questions to be presented to a user and interpret responses
        - CCI - control correlation identifiers
            - A common identifier for policy-based requirements
            - Currently not under the SCAP umbrella
- Moving ahead for the future

    - Migrating our tools and processes to take advantage of SCAP's benefit
        - The policy auditor component of HBSS already supports assessing vulnerabilities using SCAP content
    - Developing security requirements guides that address overarching requirements for a technology
        - Promotes structure mapping of the STIGs to the new DoD control set
    - Expanding work with the operating system and software vendors to leverage content and standards
- Leveraging SCAP (XCCDF)
    - Publication of DoD content (STIGs) using the eXtensible Configuration checklist description format (XCCDF)
        - Provides a standardized look for STIGs
        - Supports customers' requests to extract data for import into another database
        - XCCDF benchmarks can be used by SCAP-capable tools to automate the assessment of vulnerabilities
            - Note that OVAL is required for the true automation of a check
- Control Correlation Identifiers (CCIs)
    - A decomposition of an IA control or an IA industry best practice into single, actionable statements
    - A foundational element of an IA policy or standard, written with a neutral position on an IA practice so as not to imply the specifics of the requirement
    - Not specific to a product or common platform enumeration
- The CCI list is a collection of CCI items, which express common IA practices or controls
- The CCI data specification is
    - Proposed to work in conjunction with the national institute of Standards and Technology (NIST) security content automation protocol (SCAP)
- Security requirements guide (SRGs)
    - Similar to what a STIG is today
        - Prose discussion of requirements
        - Planned SRGs include the operating system, application, network, policy
        - Initial focus on operating systems to support Unix development efforts
    - Will be expressed in XCCDF with the ability to produce a human-readable version
    - A method to convey additional technology-specific details about the common control identifiers (CCI) to product vendors
        - SRGs are not intended for use by security tool vendors for assessments
- SCAP content status
    - Windows STIG content
        - Windows XP, Vista, 2003, 2008 published using XCCDF with OVAL
            - Many of the automatable configuration checks have OVAL content available to support the assessment of the vulnerability using SCAP-compliant tools such as HBSS policy auditor
    - UNIX STIG content

        - UNIX technical interchange meeting occurred in May 2010
        - Adopting the approach of identifying SRG requirements to allow both in-house and vendor-supported assessment of requirements
    - Other STIG content
        - 20 different products supported with XCCDF benchmarks
            - OVAL content not available
    - IAVM content
        - Working with JFCC/NW-JTF GNO combined staff and NSA to determine the best approaches for publishing IAVM content
- Metrics
    - Ongoing metrics analysis to identify trends/challenges observed throughout DoD
        - Goal to generate recommendations/approaches for material and/or non-material solutions or process enhancements
    - Correlation and trending of vulnerability-related information difficult given varying standards and repositories
    - Leveraging industry-standard approaches for identifying vulnerabilities is critical
        - Supported by existing SCAP standards
    - Efforts are underway to advance the reporting formats used as part of SCAP
- Summary
    - Adoption of a standards-based approach using SCAP addresses many of the key challenges that DoD faces today
        - Increased speed in publishing guidance to the IA community
        - Increased reliability of results based on industry-standard checking mechanisms
        - Improved metrics and trending information relative to asset and vulnerability status

Definitions:

- Project:
    - A project has a duration and consists of functions, activities, and tasks
- Work package:
    - A description of the work to be accomplished in an activity or task
- Work product:
    - Any tangible item that results from a project function, activity, or task
- Project baseline:
    - A work product that has been formally reviewed and agreed upon
    - A project baseline can only be changed through a formal change procedure
- Project deliverable
    - A work product to be delivered to the consumer

Activities, tasks, and functions

- Activity: a major unit of work with precise dates that consist of smaller activities or tasks. It culminates in a project milestone.
- Task: the smallest unit of the work subject to management. Small enough for adequate planning and tracking. Large enough to avoid micromanagement
- Project function: an activity or set of activities that span the duration of the project

Tasks

- The smallest unit of management accountability
    - The atomic unit of planning and tracking
    - Tasks have a finite duration, need resources, and produce tangible results (documents, code)
- The description of a task is done in a work package
    - Name, description of work to be done
    - Preconditions for starting, duration, required resources
        - Other work packages that need to be completed before this task can be started
    - Work products to be produced, acceptance criteria for it
    - Risk involved
- Completion criteria
    - Includes the acceptance criteria for the work products produced by the task

Determining task sizes

- Finding the appropriate task size is problematic
    - Todo lists and templates from previous projects
    - During initial planning, a task is necessarily large
    - You may not know how to decompose the problem into tasks at first
    - Each software development activity identifies more tasks and modifies existing ones
- Tasks must be decomposed into sizes that allow monitoring
    - Depends on the nature of the work and how well the task is understood
    - A work package usually corresponds to a well-defined work assignment for one worker for a week or two
    - Work assignments are also called action items

Approaches to developing work breakdown structures

- Product component approach
    - Structure the work based on the work products
- Functional approach
    - Structure the work based on development activities and project functions
- Geographical area approach
    - Structure the work based on the geographical location
- Organizational approach
    - Structure the work based on the organizational structure

When to use what approach

- The teams are distributed over the continent
    - Geographical area approach
- The teams consist of experienced developers
    - Product component approach
- The project has mostly beginners or an inexperienced project manager
    - Functional approach
- The project is a continuation of a previously successful project, there are no changes in the requirements and no new technology enablers

        - Organizational approach
    - Whatever approach you choose, stick with it to prevent possible overlap in categories

Mixing different approaches is bad

- Consider the WBS for an activity, prepare a report
- Functional approach
    - Write draft report
    - Have the draft report reviewed
    - Write final report

How do you develop a good WBS?

- Top-down approach:
    - Start at the highest, top-level activities and systematically develop increasing levels of detail for all activities
- Bottom-up approach
    - Generating all activities you can think of that will have to be done and then group them into categories
- Which one you use depends on
    - How familiar you and your team are with the project
    - Whether similar projects have successfully been performed in the past
    - How many new methods and technologies will be used

The top-down WBS development

- Specify all activities required for the entire project to be finished
- Determine all tasks required to complete each activity
- If necessary, specify sub-activities required to complete each task
- Continue in this way until you have adequately detailed your project
- The top-down WBS approach is good if
    - You are familiar with the problem or your team
    - You have successfully managed a similar project in the past
    - You are not introducing new methodologies, methods, or tools

The brainstorming WBS development

- On a single list, write any activities you think will have to be performed for your project
- Brainstorming means you
    - Don’t worry about overlap or level of detail
    - Don’t discuss activity wordings or other details
    - Don’t make any judgments
    - Write everything down
- Then study the list and group activities into a few major categories with common characteristics
- If appropriate, the group identified activities into higher-level activities
- Consider each category you have created and use the top-down and use the top-down WBS development to determine any additional activities you may have overlooked

Displaying work breakdown structures

- Three different formats are usually used
- Organizational-chart format

        - Effectively portrays an overview of your project and the hierarchical relationships of different activities and tasks
    - Outline format
        - Sub Activities and tasks are indented
    - Bubble format
        - The bubble in the center represents your project
        - Lines from the center bubble lead to activities
        - Lines from activities lead to tasks

What is the best display format for WBS?

- Organization-chart format:
    - Often good for a “bird view” of the project
    - Less effective for displaying large numbers of activities
- Outline format
    - Easier to read and understand if WBS contains many activities
- Bubble format
    - Effective for supporting brainstorming
    - Not so good for displaying work breakdown structures to audiences who are not familiar with the project
- In large groups
    - Use bubble format to develop the WBS, then turn it into an organization chart or outline format
    - Display activities in the organization chart format
    - Display sub-activities and tasks in outline format

Heuristics for developing high-quality WBS

- Involve the people who will be doing the work in the development of the WBS
    - In particular, involve the developers
- Review and include information from work breakdown structures that were developed for similar projects
    - Use a project template if possible
- Use more than one WBS approach
    - Do project component and functional approaches simultaneously
    - This allows you to often identify overlooked activities
- Make assumptions regarding uncertain activities
    - Identify risky activities
    - These are often the activities whose times are hard to estimate

Choose a single WBS approach

- Develop the WBS with different approaches. This is good because it allows you to identify activities that you may overlook otherwise
- Choose a single WBS approach to be used in the SPMP and for your project
    - Nothing confuses people fast than trying to use two different work breakdown structures to describe the same project

How detailed should the WBS be?

- Sometimes the activities are not clear at all, especially in software projects because of
    - Unclear requirements and/or changing requirements

        - Dependency on technology enablers that are promised to appear after project kickoff
        - Simultaneous development of hardware and software
- Heuristic: a project plan, especially for an innovative software project, should not address details beyond 3 months
    - Even for the first 3 months project activities might not all be detailable, for example when the requirements are unclear or change or introduction of technology enablers is expected
- How should we describe a WBS for a longer project?

Doing a WBS for long-term projects

- Developing a WBS for a long project, longer than 3 months, use at least two phases
- Phase 1: Plan your WBS in detail
    - List all activities that take 2 weeks or less to complete
- Phase 2: Plan your WBS for these phases in less detail
    - List activities that take between 1 and 2 months
- At the end of phase 1, revise the activities planned for phase 2 and plan them on the two-week level for the next 3 months
    - Modify future activities based on the results of your work so far
- Revise the SPMP throughout the project
    - Remember: the SPMP is an evolving document just like the RAD and SDD

Phases and large projects

- Project-initiation phase
- Steady-state phase
    - Initial planning phase
- Project-termination phase

Summary:

- Different approaches to developing a WBS
    - Product approach
    - Functional approach
    - Geographical approach
    - Organizational approach
- Top-down and bottom-up WBS development
- Heuristics for developing good WBS
- WBS for large projects

Project Risk Management:

- What is project risk?
    - An event that, if it occurs, causes either a positive or negative impact on a project
    - Key attributes of risk
        - Uncertainty
        - Positive and negative
        - Cause and consequence
- Risk management
    - Risk management is concerned with identifying risks and drawing up plans to minimize their effect on a project

        - A risk is a probability that some adverse (or positive) circumstance will occur
            - Project risks affect schedule or resources
            - Product risks affect the quality or performance of the software being developed
            - Business risks affect the organization developing or procuring the software
- Risk management process
    - PMBOK definition
        - The systematic process of identifying, analyzing, and responding to project risk
    - Steps
        - Risk management planning
        - Risk identification
        - qualitative/quantitative risk analysis
        - Risk response planning
        - Risk monitoring and control
- Value from managing risks
    - Opportunity to move from “fire-fighting” to proactive decision-making on the project
    - Better chance of project success
    - Improved project schedule and cost performance
    - Stakeholders and team members better understand the nature of the project
    - Helps define the strengths and weaknesses of the project
- Why not risk management?
    - With so many benefits to managing risk, why is it often overlooked?
        - The organization is too busy with real problems to worry about potential ones
        - There is a perception that there is not too much that can go wrong
        - They have a fatalistic belief that not much can be done about risks
        - Shoot the messenger mentality, fear that disclosure of project risks will be seen as an indication of project weakness
- Won’t identified risks make the project look bad?
    - All projects have risks, denial does not make them go away, it just makes you unprepared for them if they occur
    - Risk in itself is not bad, it is how well the project plans for and reacts to risks that counts
    - Formal risk management is a cornerstone of good project management. Stakeholder visibility into project risks makes it easier to get additional resources and organizational support when risks do occur
- Risk management planning
    - Plan for the planning
        - Risk planning should be appropriate for the project
        - The questions you should ask
            - How risky is the project?

            - Is it a new technology or something your organization is familiar with?
            - Do you have past projects to reference?
            - What is the visibility of the project?
            - How important is the project?
- The risk management plan
    - What should it include?
        - How you will identify, quantify, or qualify risk
            - Methods and tools
        - Budget, yes budget
        - Who is doing what
        - How often
        - Risk categories, levels, and thresholds for action
        - Reporting requirements
        - Monitoring, tracking, and documenting strategies
- The risk management process
    - Risk identification
        - Identify project, product, and business risks
    - Risk analysis
        - Access the likelihood and consequences of these risks
    - Risk response planning
        - Draw up plans to avoid or minimize the effects of the risk
    - Risk monitoring
        - Monitor the risks throughout the project
- Identifying risk
    - A continuous, iterative process
    - What is it and what does it look like
    - The sooner the better
    - The more the merrier
    - A fact is not a risk, it’s an issue
    - Be specific
    - Don’t try to do everything at once
- Identification techniques
    - Brainstorming
    - Checklists
    - Interviewing
    - SWOT analysis
    - Delphi technique
    - Diagramming techniques
        - Cause and effect
        - Flow charts
        - Influence diagrams
- Software risks
    - Staff turnover: experienced staff will leave the project before it is finished

        - Management change: there will be a change in organizational management with different priorities
        - Hardware unavailability: hardware that is essential for the project will not be delivered on schedule
        - Requirements change: there will be a larger number of changes to the requirements than anticipated
        - Specification delays: specifications of essential interfaces are not available on schedule
        - Size underestimate: the size of the system has been underestimated
        - CASE tool underperformance: CASE tools that support the project do not perform as anticipated
        - Technology change: the underlying technology on which the system is built is superseded by new technology
        - Product competition: a competitive product is marketed before the system is completed
- Risk analysis
    - Assess the probability, seriousness, and urgency of each risk
    - Probability may be very low, low, moderate, high, or very high
    - Risk effects might be catastrophic, serious, tolerable, or insignificant
    - Urgency might be immediate, short-term, or long term
- Analyzing risk - qualitative
    - Subjective
    - Educated guess
    - High, medium, low
    - Red, yellow, green
    - 1-10
    - prioritized /ranked list of all identified risks
    - The first step in risk analysis!
- Analyzing risk - quantitative
    - Numerical/statistical analysis
    - Determines probability of occurrence and consequences of risks
    - Should be focused on the highest risks as determined by qualitative risk analysis and risk threshold
- Risk response planning
    - What are we going to do about it?
    - techniques/strategies
        - Avoidance - eliminate it
        - Transference - pawn it off
        - Mitigation - reduce the probability or impact of it
        - Acceptance - do nothing
    - The strategy should be commensurate with the risk
        - Don’t spend more money preventing the risk rather than the impact the risk would have if it occurs
    - Risk response plan/risk response register

- Risk monitoring and control
  - Assess each identified risk regularly to decide whether or not it is becoming less or more probable
  - Also assess whether the effects of the risk have changed
  - Each key risk should be discussed at management progress meetings
  - A continuous, iterative process
  - Done right, the risk impact will be minimized
    - Someone IS responsible
    - Watch for risk triggers
    - Communicate
    - Take corrective action - execute
    - Re-evaluate and look for new risks constantly
  - Tools:
    - Risk reviews
    - Risk audits

Risk Analysis (i)

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staff are ill at critical times in the project.	Moderate	Serious
Software components that should be reused contain defects which limit their functionality.	Moderate	Serious
Changes to requirements that require major design rework are proposed.	Moderate	Serious
The organization is restructured so that different management are responsible for the project.	High	Serious

Risk Analysis (ii)

Risk	Probability	Effects
The database used in the system cannot process as many transactions per second as expected.	Moderate	Serious
The time required to develop the software is underestimated.	High	Serious
CASE tools cannot be integrated.	High	Tolerable
Customers fail to understand the impact of requirements changes.	Moderate	Tolerable
Required training for staff is not available.	Moderate	Tolerable
The rate of defect repair is underestimated.	Moderate	Tolerable
The size of the software is underestimated.	High	Tolerable
The code generated by CASE tools is inefficient.	Moderate	Insignificant

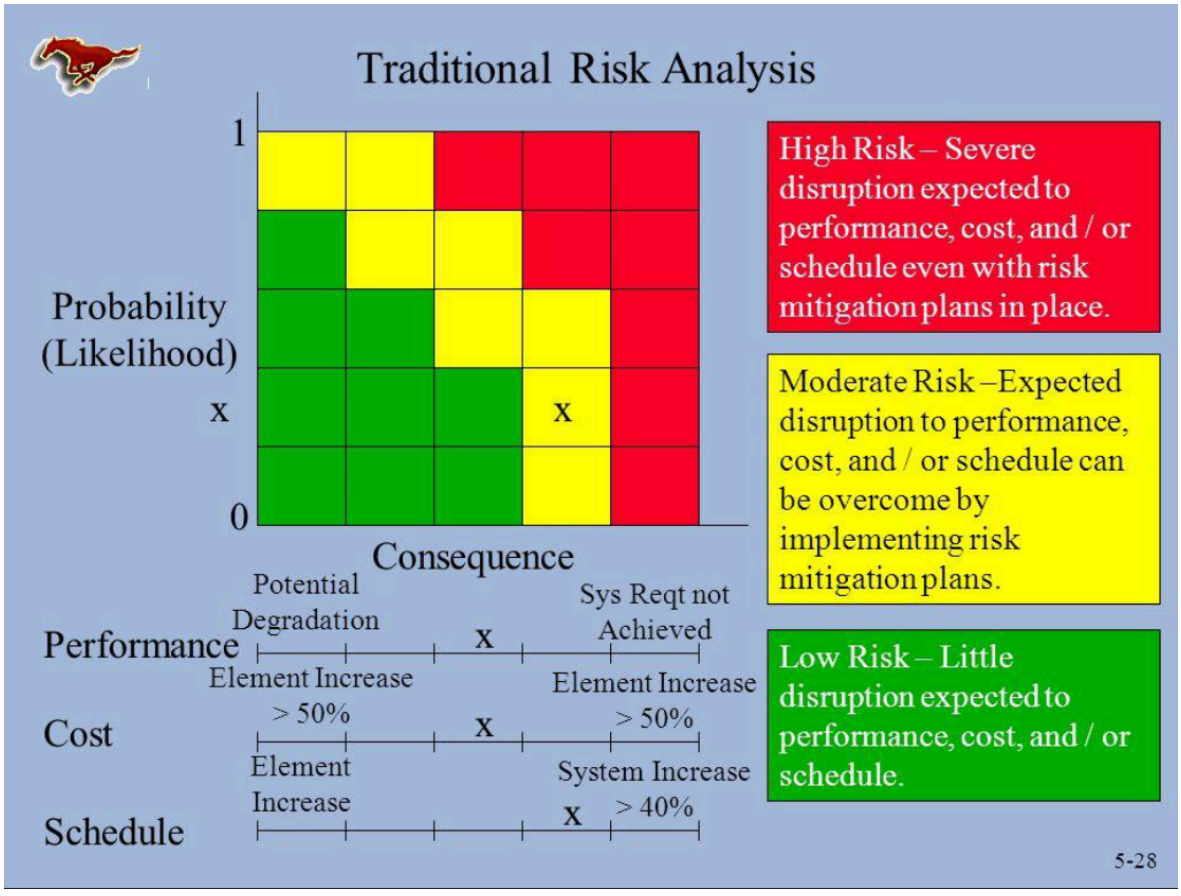
Risk Management Strategies (i)

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Recruitment problems	Alert customer of potential difficulties and the possibility of delays, investigate outsourcing work.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.

Risk Management Strategies (ii)

Risk	Strategy
Requirements changes	Derive traceability information to assess requirements change impact, and maximise information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate outsourcing components, investigate use of a program generator

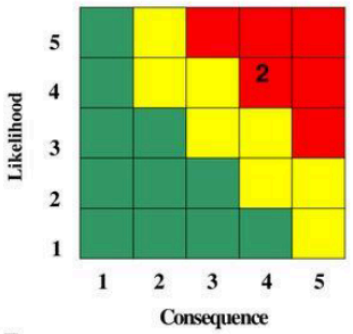
LOCKHEED MARTIN



RISK MITIGATION PLAN EXAMPLE

Identify, evaluate, and select detailed steps that will drive risk to an acceptable level given program constraints and objectives

1. Get new/detailed Program A and SW schedules
2. Identify insertion points for SW updates
3. Work with SW contractor to improve schedule
4. Incentivize SW lab construction for schedule
5. Identify root cause of SW technical issues
6. Correct SW technical issues
7. Improve SW schedule by 2 months
8. Improve SW lab construction by 1 month
9. New SW dates coordinated with Program A leadership



EXAMPLE

Risk #2: If the timeline established for Program A's production is not met because of a delay in receiving software updates, then there will be a program slip of at least 4 months.

Engineering Software Products - Cloud-Based Software:

- The cloud
  - The cloud is made up of a very large number of remote servers that are offered for rent by companies that own these servers
    - Cloud-based servers are 'virtual servers' which means that they are implemented in software rather than hardware
  - You can rent as many servers as you need, run your software on these servers and make them available to your customers
    - Your customers can access these servers from their computers or other networked devices such as a tablet or TV
    - Cloud servers can be started up and shut down as demand changes
  - You may rent a server and install your software, or you may pay for access to software products that are available on the cloud
- Scalability, elasticity, and resilience
  - Scalability reflects the ability of your software to cope with increasing numbers of users
    - As the load on your software increases, your software automatically adapts so that the system performance and response time are maintained
  - Elasticity is related to scalability but also allows for scaling down as well as scaling up
    - That is, you can monitor the demand on your application and add or remove servers dynamically as the number of users changes
  - Resilience means that you can design your software architecture to tolerate server failures

        - You can make several copies of your software concurrently available. If one of these fails, the others continue to provide a service
- Virtual cloud servers
    - A virtual server runs on an underlying physical computer and is made up of an operating system plus a set of software packages that provide the server functionality required
    - A virtual server is a stand-alone system that can run on any hardware in the cloud
        - This ‘run anywhere’ characteristic is possible because the virtual server has no external dependencies
    - Virtual machines running on physical server hardware can be used to implement virtual servers
        - A hypervisor provides hardware emulation that simulates the operation of the underlying hardware
    - If you use a virtual machine to implement virtual servers, you have the same hardware platform as a physical server
- Container-based virtualization
    - If you are running a cloud-based system with many instances of applications or services that these all use the same operating system, you can use a simpler virtualization technology called ‘containers’
    - Using containers accelerates the process of deploying virtual servers on the cloud
        - Containers are usually megabytes in size whereas VMs are gigabytes
        - Containers can be started and shut down in a few seconds rather than a few minutes as required for the VM
    - Containers are an operating system virtualization technology that allows independent servers to share a single operating system
        - They are particularly useful for providing isolated application services where each user sees their version of an application
- Docker
    - Containers were developed by Google around 2007 but containers became a mainstream technology around 2015
    - An open-source project called docker provided a standard means of container management that is fast and easy to use
    - Docker is a container management system that allows users to define the software to be included in a container as a docker image
    - It also includes a run-time system that can create and manage containers using these docker images
- Docker images
    - Docker images are directories that can be archived, shared, and run on different docker hosts. Everything that’s needed to run a software system - binaries, libraries, system tools, etc.. is included in the directory
    - A docker image is a base layer, usually taken from the docker registry, with your software and data added as a layer on top of this

        - The layered model means that updating docker applications is fast and efficient. Each update to the file system is a layer on top of the existing system
        - To change an application, all you have to do is to ship the changes that you have made to its image, often just a small number of files
- Benefits of containers
    - They solve the problem of software dependencies. You don't have to worry about the libraries and other software on the application server being different from those on your development server
        - Instead of shipping your product as stand-alone software, you can ship a container that includes all of the support software that your product needs
    - They provide a mechanism for software portability across different clouds. Docker containers can run on any system or cloud provider where the docker daemon is available
    - They provide an efficient mechanism for implementing software services and so support the development of service-oriented architectures
    - They simplify the adoption of DevOps. This is an approach to software support where the same team is responsible for both developing and supporting operational software
- Everything as a service
    - The idea of a service that is rented rather than owned is fundamental to cloud computing
    - Infrastructure as a service
        - Cloud providers offer different kinds of infrastructure services such as compute service, network service, and storage services that you can use to implement virtual servers
    - Platform as a service
        - This is an intermediate level where you use libraries and frameworks provided by the cloud provider to implement your software. These provide access to a range of functions, including SQL and NoSQL databases
    - Software as a service (SaaS)
        - Your software product runs on the cloud and is accessed by users through a web browser or mobile app
- Software as a service
    - Increasingly, software products are being delivered as a service, rather than installed on the buyer's computers
    - If you deliver your software product as a service, you run the software on your servers, which you may rent from a cloud provider
    - Customers don't have to install software and they access the remote system through a web browser or dedicated mobile app
    - The payment model for software as a service is usually a subscription model
        - Users pay a monthly fee to use the software rather than buy it outright
- SaaS design issues
    - Local/remote processing

        - A software product may be designed so that some features are executed locally in the user's browser or mobile app and some on a remote server
        - Local execution reduces network traffic and so increases user response speed. This is useful when users have a slow network connection
        - Local processing increases the electrical power needed to run the system
    - Authentication
        - If you set up your authentication system, users have to remember another set of authentication credentials
        - Many systems allow authentication using the user's Google, Facebook, or LinkedIn credentials
        - For business products, you may need to set up a federated authentication system, which delegates authentication to the business where the user works
    - Information leakage
        - If you have multiple users from multiple organizations, a security risk is that information leaks from one organization to another
        - There are several different ways that this can happen, so you need to be very careful in designing your security system to avoid this
    - Multi-tenant and multi-instance systems
        - In a multi-tenant system, all customers are served by a single instance of the system and a multitenant database
    - In a multi-instance system, a separate copy of the system and database is made available for each user
- Multi-tenant systems
    - A multi-tenant database is partitioned so that customer companies have their own space and can store and access thrown data
        - There is a single database schema, defined by the SaaS provider, that is shared by all of the system's users
        - Items in the database are tagged with a tenant identifier, representing a company that has stored data in the system. The database access software uses this tenant identifier to provide logical isolation, which means that users seem to be working with their database
- Adding fields to extend the database
    - You add some extra columns to each database table and define a customer profile that maps the column names that the customer wants to these extra columns, however:
        - It is difficult to know how many extra columns you should include. If you have too few, customers will find that there aren't enough for what they need to do
        - If you cater to customers who need a lot of extra columns, however, you will find that most customers don't use them so you will have a lot of wasted space in your database
    - Different customers are likely to need different types of columns

        - For example, some customers may wish to have columns whose items are string types, others may wish to have columns that are integers
        - You can get around this by maintaining everything as strings. However, this means that either you or your customer have to provide conversion software to create items of the correct type
- Extending a database using tables
    - An alternative approach to database extensibility is to allow customers to add any number of additional fields and to define the names, types, and values of these fields
    - The names and types of these values are held in a separate table, accessed using the tenant identifier
    - Unfortunately, using tables in this way adds complexity to the database management software
        - Extra tables must be managed and information from this is integrated into the database
- Database security
    - Information from all customers is stored in the same database in a multi-multi-tenant system so a software bug or an attack could lead to the data of some or all customers being exposed to others
    - Key security issues are multilevel access control and encryption
        - Multilevel access control means that access to data must be controlled at both the organizational level and the individual level
        - You need to have organizational-level access control to ensure that any database operations only act on that organization's data. The individual user accessing the data should also have access permissions
    - Encryption of data in a multitenant database reassures corporate users that their data cannot be viewed by people from other companies if some kind of system failure occurs
- Multi-instance databases
    - Multi-instance systems are SaaS systems where each customer has its own system that is adapted to its needs, including its own database and security controls
    - Multi-instance, cloud-based systems are conceptually simpler than multi-tenant systems and avoid security concerns such as data leakage from one organization to another
    - There are two types of multi-instance systems:
        - VM-based multi-instance systems are multi-instance systems where the software instance and database for each customer run in its virtual machine. All users from the same customer may access the shared system database
        - Container-based multi-instance systems, are multi-instance systems where each user has an isolated version of the software and database running in a set of containers

        - This approach is suited to products in which users mostly work independently, with relatively little data sharing. Therefore, it is best used for software that serves individuals rather than business customers or for business products that are not data-intensive
- Scalability and resilience
    - The scalability of a system reflects its ability to adapt automatically to changes in the load on that system
    - The resilience of a system reflects its ability to continue to deliver critical services in the event of system failure or malicious system use
    - You achieve scalability in a system by making it possible to add new virtual servers or increase the power of a system server in response to increasing load
        - In cloud-based systems, scaling out rather than scaling up is the normal approach used. Your software has to be organized so that individual software components can be replicated and run in parallel
    - To achieve resilience, you need to be able to restart your software quickly after a hardware or software failure
- Resilience
    - Resilience relies on redundancy
        - Replicas of the software and data are maintained in different locations
        - Database updates are mirrored so that the standby database is a working copy of the operational database
        - A system monitor continually checks the system status. It can switch to the standby system automatically if the operating system fails
    - You should use redundant virtual servers that are not hosted on the same physical computer and locate servers in different locations
        - Ideally, these servers should be located in different data centers
        - If a physical server fails or if there is a wider data center failure, then the operation can be switched automatically to the software copies elsewhere
- System structure
    - An object-oriented approach to software engineering has been extensively used for the development of client-server systems built around a shared database
    - The system itself is, logically, a monolithic system with distribution across multiple servers running large software components. The traditional multi-tier client-server architecture is based on this distributed system model
    - The alternative to a monolithic approach to software organization is a service-oriented approach where the system is decomposed into fine-grain, stateless services
        - Because it is stateless, each service is independent and can be replicated, distributed, and migrated from one server to another
        - The service-oriented approach is particularly suitable for cloud-based software, with services deployed in containers
- Cloud platforms
    - Cloud platforms include general-purpose clouds such as Amazon web services or lesser-known platforms oriented around a specific application, such as the

SAP cloud platform. Some smaller national providers provide more limited services but may be more willing to adapt their services to the needs of different customers
    - There is no best platform and you should choose a cloud provider based on your background and experience, the type of product that you are developing, and the expectations of your customers
    - You need to consider both technical issues and business issues when choosing a cloud platform for your product
- Key points
    - The cloud is made up of a large number of virtual servers that you can rent for your use. You and your customers access these servers remotely over the internet and pay for the amount of server time used
    - Virtualization is a technology that allows multiple server instances to be run on the same physical computer. This means that you can create isolated instances of your software for deployment on the cloud
    - Virtual machines are physical server replicas on which you run your own operating system, technology stack, and applications
    - Containers are a lightweight virtualization technology that allows rapid replication and deployment of virtual servers. All containers run the same operating system. Docker is currently the most widely used container technology
    - A fundamental feature of the cloud is that everything can be delivered as a service and accessed over the Internet. A service is rented rather than owned and is shared with other users
    - Infrastructure as a service means computing, storage, and other services are available over the cloud. There is no need to run your physical servers
    - Platform as a service means using services provided by a cloud platform vendor to make it possible to auto-scale your software in response to demand
    - Software as a service means that application software is delivered as a service to users. This has important benefits for users, such as lower capital costs, and software vendors, such as the simpler deployment of new software releases
    - Multi-Tenant systems are SaaS systems where all users share the same database, which may be adapted at runtime to their individual needs. Multi-instance systems are SaaS applications where each user has their own separate database
    - The key architectural issues for cloud-based software are the cloud platform to be used, whether to use a multitenant or multi-instance database, the scalability and resilience requirements, and whether to use objects or services as the basic components in the system

MVC:

- Model View Controller or MVC as it is popularly called is a software design pattern for developing web applications
- MVC is one of three ASP.NET programming models
- Model-view-controller is a software architecture pattern that separates the representation of information from the user’s interaction with it

- History of MVC
    - Presented by Trygve Reenskaug in 1979
    - First used in the Smalltalk-80 framework
        - Used in making Apple interfaces (Lisa and Macintosh)
- MVC uses
    - Smalltalk's MVC implementation inspired many other GUI frameworks such as:
        - The NEXTSTEP and OPENSTEP development environments encourage the use of MVC
        - Cocoa and GNUstep, based on these technologies, also use MVC
        - Microsoft Foundation Classes (MFC), also called document/view architecture
        - Java Swing
        - The Qt Toolkit (since Qt4 Release)
- Parts of MVC
    - A model view controller pattern is made up of the following three parts:
        - Model
        - View
        - Controller
    - The MVC model defines web applications with 3 logic layers
        - The business layer (model logic)
        - The display layer (view logic)
        - The input control (controller logic)
- Model
    - The model is responsible for managing the data of the application
    - It responds to the request from the view and it also responds to instructions from the controller to update itself
    - It is the lowest level of the pattern which is responsible for maintaining data
    - The model represents the application core (for instance a list of database records)
    - It is also called the domain layer
- View
    - The view displays the data (the database records)
    - A view requests information from the model, that it needs to generate an output representation
    - It presents data in a particular format like JSP, ASP, PHP
    - MVC is often seen in web applications, where the view is the HTML page
- Controller
    - The controller is part of the application that handles user interaction
    - Typically controllers read data from a view, control user input, and send input data to the model
    - It handles the input, typically user actions, and may invoke changes on the model and view
- Workflow in MVC

        - Though MVC comes in different flavors, the control flow generally works as follows:
            - The user interacts with the user interface in some way
            - A controller handles the input event from the user interface, often via a registered handler or callback
            - The controller accesses the model, possibly updating it in a way appropriate to the user's action, ex. The controller updates the user's cart
            - A view uses the model to generate an appropriate user interface, ex. View produces a screen listing the shopping cart contents
                - The view gets its data from the model. The model has no direct knowledge of the view
            - The user interface waits for further user interactions, which begins the cycle anew
- Dependence hierarchy
    - There is usually a kind of hierarchy in the MVC pattern
    - The model knows only about itself
    - That is, the source code of the Model has no references to either the View or Controller
    - The View, however, knows about the Model. It will poll the Model about the state, to know what to display
    - That way, the view can display something that is based on what the Model has done
    - But the View knows nothing about the Controller
    - The Controller knows about both the Model and the View
    - Take an example from a game: if you click on the "fire" button on the mouse, the Controller knows what fire function in the Model to call
    - If you press the button for switching between first and third person, the Controller knows what function in the View to call to request the display change
- Why is the dependence hierarchy used?
    - The reason to keep it this way is to minimize dependencies
    - No matter how the View class is modified, the Model will still work
        - Even if the system is moved from a desktop operating system to a smartphone, the Model can be moved with no changes
    - But the View probably needs to be updated, as will the Controller
- Use in web applications
    - Although originally developed for personal computing, Model View Controller has been widely adopted as an architecture for World Wide Web applications in all major programming languages
    - Several commercial and noncommercial application frameworks have been created that enforce the pattern
    - These frameworks vary in their interpretations, mainly in the way that the MVC responsibilities are divided between the client and server
    - Early web MVC frameworks took a thin client approach that placed almost the entire model, view, and controller logic on the server

    - In this approach, the client sends either hyperlink requests or form input to the controller and then receives a complete and updated web page from the view; the model exists entirely on the server
    - As client technologies have matured, frameworks such as JavaScript MVC and Backbone have been created that allow the MVC components to execute partly on the client
- Web forms vs. MVC
    - The MVC programming model is a lighter alternative to traditional ASP.NET (Web Forms)
        - It is a lightweight, highly testable framework, integrated with all existing ASP.NET features, such as Master Pages, Security, and Authentication
- Creating the Web Application
    - If you have a Visual Web Developer installed, start Visual Web Developer and select New Project
    - Steps
        - In the new project dialogue box:
            - Open the Visual C# templates
            - Select the template ASP.NET MVC 3 Web Application
            - Set the project name
            - Set the disk location
            - Click ok
            - When the new project dialogue box opens
                - Select the Internet Application template
                - Select the Razor Engine
                - Select HTML5 Markup
                - Click ok
    - Visual Studio Express will create a project template
    - MVC folders
        - Application information
            - Properties
            - References
        - Application folders
            - App_Data folder
            - Content folder
            - Controllers folder
            - Models folder
            - Scripts folder
            - Views folder
        - Configuration files
            - Global.asax
            - Packages.config
            - Web.config
- Advantages
    - Clear separation between presentation logic and business logic

        - Each object in MVC has distinct responsibilities
        - Parallel development
        - Easy to maintain and future enhancements
        - All objects and classes are independent of each other
    - Disadvantages
        - Increased complexity
        - The inefficiency of data access in view
        - The difficulty of using MVC with a modern user interface too
        - For parallel development, there is a needed multiple programmers
        - Knowledge of multiple technologies is required

DevOps and Code Management:

- Software support
    - Traditionally, separate teams were responsible for software development, software release, and software support
    - The development team passed over a ‘final’ version of the software to a release team. This team then built a release version, tested this, and prepared release documentation before releasing the software to customers
    - A third team was responsible for providing customer support
        - The original development team was sometimes also responsible for implementing software changes
        - Alternatively, the software may have been maintained by a separate ‘maintenance team’
- DevOps
    - There are inevitable delays and overheads in the traditional support model
    - To speed up the release and support processes, an alternative approach called DevOps (Development + Operations) has been developed
    - Three factors led to the development and widespread adoption of DevOps:
        - Agile software engineering reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment
        - Amazon re-engineered its software around services and introduced an approach in which a service was developed and supported by the same team. Amazon’s claim that this led to significant improvements in reliability was widely publicized
        - It became possible to release software as a service, running on a public or private cloud. Software products did not have to be released to users on physical media or downloads
- Code management and DevOps
    - Source code management, combined with automated system building, is essential for professional software engineering
    - In companies that use DevOps, a modern code management system is a fundamental requirement for ‘automating everything
    - Not only does it store the project code that is ultimately deployed, but it also stores all other information that is used in DevOps processes

    - DevOps automation and measurement tools all interact with the code management system
- Code management fundamentals
    - Code management systems provide a set of features that support four general areas:
        - Code transfer - Developers take code into their file store to work on it then return it to the shared code management system
        - Version storage and retrieval - files may be stored in several different versions and specific versions of these files can be retrieved
        - Merging and branching - parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged
        - Version information - information about the different versions maintained in the system may be stored and retrieved
- DevOps automation
    - By using DevOps with automated support, you can dramatically reduce the time and costs for integration, deployment, and delivery
    - Everything that can be should be automated is a fundamental principle of DevOps
    - As well as reducing the costs and time required for integration, deployment, and delivery, process automation also makes these processes more reliable and reproducible
    - Automation information is encoded in scripts and system models that can be checked, reviewed, versioned, and stored in the project repository
- System Integration
    - System integration or system building is the process of gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system
    - Total activities that are part of the system integration process include:
        - Installing database software and setting up the database with the appropriate schema
        - Loading test data into the database
        - Compiling the files that make up the product
        - Linking the compiled code with the libraries and other components used
        - Checking that external services used are operational
        - Deleting old configuration files and moving configuration files to the correct locations
        - Running a set of system tests to check that the integration has been successful
- Continuous integration
    - Continuous integration simply means that an integrated version of the system is created and tested every time a change is pushed to the system’s shared repository

    - On completion of the push operation, the repository sends a message to an integration server to build a new version of the product
    - The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system
    - If you make a small change and some system tests fail, the problem almost certainly lies in the new code that you have pushed to the project repo
    - You can focus on this code to find the bug that's causing the problem
- System building
    - Continuous integration is only effective if the integration process is fast and developers do not have to wait for the results of their tests of the integrated system
    - However, some activities in the build process, such as populating a database or compiling hundreds of system files, are inherently slow
    - It is therefore essential to have an automated build process that minimizes the time spent on these activities
    - Fast system building is achieved using a process of incremental building, where only those parts of the system that have been changed are rebuilt
- Continuous integration
    - An automated build system uses the specification of dependencies to work out what needs to be done. It uses the file modification timestamp to decide if a source code file has been changed
        - The modification date of the compiled code is after the modification date of the source code. The build system infers that no changes have been made to the source code and does nothing
        - The modification date of the compiled code is before the modification date of the compiled code. The build system recompiles the source and replaces the existing file of compiled code with an updated version
        - The modification date of the compiled code is after the modification date of the source code. However, the modification date of Classdef is after the modification date of the source code of Mycode. Therefore, Mycode has to be recompiled to incorporate these changes
- Continuous delivery and deployment
    - Continuous integration means creating an executable version of a software system whenever a change is made to the repository. The CI tool builds the system and runs tests on your development computer or project integration server
    - However, the real environment in which software runs will inevitably be different from your development system
    - When your software runs in its real, operational environment bugs may be revealed that did not show up in the test environment
    - Continuous delivery means that, after making changes to a system, you ensure that the changed system is ready for delivery to customers
    - This means that you have to test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance

- The deployment pipeline
    - After initial integration testing, a staged test environment is created
    - This is a replica of the actual production environment in which the system will run
    - The system acceptance tests, which include functionality, load, and performance tests, are then run to check that the software works as expected. If all of these tests pass, the changed software is installed on the production servers
    - To deploy the system, you then momentarily stop all new requests for service and leave the older version to process the outstanding transactions
    - Once these have been completed, you switch to the new version of the system and restart the processing
- Infrastructure as code
    - In an enterprise environment, there are usually many different physical or virtual servers that do different things. These have different configurations and run different software packages
    - It is therefore difficult to keep track of the software installed on each machine
    - The idea of infrastructure as code was proposed as a way to address this problem. Rather than manually updating the software on a company's servers, the process can be automated using a model of the infrastructure written in a machine-processable language
    - Infrastructure as code is the managing and provisioning of infrastructure through code instead of through manual processes
    - With IaC, configuration files are created that contain your infrastructure specifications, which makes it easier to edit and distribute configurations. It also ensures that you provide the same environment every time. By codifying and documenting your configuration specifications, IaC aids configuration management and helps you to avoid undocumented, ad-hoc configuration changes
    - Version control is an important part of IaC, and your configuration files should be under source control just like any other software source code file. Deploying your infrastructure as code also means that you can divide your infrastructure into modular components that can then be combined in different ways through automation
- Benefits of infrastructure as code
    - Defining your infrastructure as code and using a configuration management system solves two key problems of continuous deployment
        - Your testing environment must be the same as your deployment environment. If you change the deployment environment, you have to mirror those changes in your testing environment
        - When you change service, you have to be able to roll that change out to all of your servers quickly and reliably. If there is a bug in your changed code that affects the system's reliability, you have to be able to seamlessly roll back to the older system

        - The business benefits of defining your infrastructure as code are lower costs of system management and lower risks of unexpected problems arising when infrastructure changes are implemented
- Containers
    - A container provides a stand-alone execution environment running on top of an operating system such as Linux
    - The software installed in a Docker container is specified using a Dockerfile, which is, essentially, a definition of your software infrastructure as code
    - You build an executable container image processing the Dockerfile
    - Using containers makes it very simple to provide identical execution environments
        - For each type of server that you use, you define the environment that you need and build an image for execution. You can run an application container as a test system or as an operational system; there is no distinction between them
        - When you update your software, you rerun the image creation process to create a new image that includes the modified software. You can then start these images alongside the existing system and divert service requests to them
    - Why do you need docker?
        - Compatibility/dependency
        - Long setup time
        - Different dev/test/prod environments
- DevOps measurement
    - After you have adopted DevOps, you should try to continuously improve your DevOps process to achieve faster deployment of better-quality software
    - There are four types of software development measurement
        - Process measurement - you collect and analyze data about your development, testing, and deployment processes
        - Service measurement - you collect and analyze data about the software's performance, reliability, and acceptability to customers
        - Usage measurement - you collect and analyze data about how customers use your product
        - Business success measurement - you collect and analyze data about how your product contributes to the overall success of the business
    - Automating measurement
        - As far as possible, the DevOps principle of automating everything should be applied to software measurement
        - You should instrument your software to collect data about itself and you should use a monitoring system, as explained in Chapter 6, to collect data about your software's performance and availability

SecDevOps:

- Distrust and caution are the parents of security
- What is DevOps?

        - DevOps is a model that blurs the lines between dev and IT operations & consolidates historically segregated management functions to improve efficiency and responsiveness
        - It allows dev teams to automate their build, testing, and deployment processes
        - Often referred to as Continuous Integration (CI) & Continuous Deployment (CD)
- Traditional roles
    - Development
        - Ship code
        - Get around blocks
        - Do minimal operations and security requirements
    - Operations
        - Keep code stable
        - Block changes
        - Get paged because of other people's code
    - Security
        - Lock up the code
        - Block changes
        - Prescribe fixes
        - Work in a locked room
        - Attend Funny-sounding conferences
- Traditional Roles version 2
    - DevOps
        - Optimized for fast iterations
        - Automated deployments
    - Security
        - Optimized for fewer incidents
        - Ads API certifications, architecture review gates
        - Only approved tools
- Security testing
    - A type of software testing that uncovers vulnerabilities, threats, and risks in software applications and prevents malicious attacks from intruders. The purpose of security tests is to identify all possible loopholes and weaknesses of the software system which might result in a loss of information, revenue, and reputation at the hands of the employees or outsiders of the organization. The goal of security testing is to identify the threats in the system and measure its potential vulnerabilities, so the system does not stop functioning or is exploited. It also helps in detecting all possible security risks in the system and helps developers in fixing these problems through coding
- What is SecDevOps?
    - If operations + development automation = DevOps
    - The security + development automation = SecDevOps?
    - SecDevOps is
        - Security automation
        - Security at scale

        - Discussing security and business tradeoffs
    - SecDevOps is not
        - Only there to audit code
        - Ivory tower security
    - The concept of using tools and processing to facilitate collaboration between the engineers who deliver the code/systems and those who must maintain and secure it/them
- Why security automation?
    - Reduce the risk of human error
        - Automation is effective
        - Automation is reliable
        - Automation is scalable
- Total enterprise ratio
    - Developers: 100
    - Operations: 10
    - Security: 1
- Typical enterprise ratio + security champions
    - Developers: 100
    - Operations: 10
    - Security: (+10) 1
        - Security champion: one developer from each team, assuming ten teams, spending a small amount of time to gain proficiency and lead their team in automating and implementing security
- Answer: Security Champions
    - Best practice: teams of 10 or fewer with all the skills needed to push to production
    - This person should be the security champion within the team. They should represent the voice of security while still performing some duties as an application developer
    - Security champion developers still ship code
    - Security champion developers automate security
    - Security champion developers watch for the common security gotchas
    - Security champion benefits
        - Understanding and empathy with the security team aka trust between teams
        - Higher level of security within the application
        - Security in the design phase and throughout the whole lifecycle
            - Top risks (severity and estimated % chance of occurring) are identified early and kept top of mind
        - Higher, more productive discussions with the security team
- Devs developing security automation
    - Infrastructure as code
    - DevOps is generally a trend to automate traditional operation tasks such as deploying code and increasing the availability and uptime of that running code

    - Security as code
    - SecDevOps is enforcing good security patterns and automating traditional security checks
- How to be a security champion?
    - Learn the basics!
        - 8 secure design principles
            - The economy of mechanism - keep the design simple
            - Fail-safe defaults - fail towards denying access
            - Complete mediation - check authorization of every access request
            - Open design - assume the attacker knows the system internals
            - Separation of privilege - require 2 separate keys or other ways to check authorization (2-factor authentication)
            - Least privilege - give only necessary rights
            - Least common mechanisms - ensure failures stay local
            - Psychological acceptability - design security mechanisms that are easy to use
- Summary
    - SecDevOps is not just about tooling and automation - culture is key to success
    - SecDevOps is reshaping security, for the better
    - It allows for security to be seen as an enabler, not a blocker within your business
    - Embracing SecDevOps can positively impact your security culture
    - It isn’t a silver bullet and it doesn’t ‘solve’ security