Chapter 14: Pattern-Based Design

January 27, 2021 9:21 AM

- Design Patterns
  - ★ A <u>design pattern</u> can be thought of as a three-part rule about a <u>context</u>, <u>problem</u>, and <u>solution</u>
  - Context allows the reader to understand the environment of the problem and the appropriate solution
  - ★ We use design patterns so that we don't have to "<u>re-invent the wheel</u>"

Effective design patterns

- Solves a problem =
  - patterns capture solutions, not just abstract principles or strategies
- Proven concept = patterns capture solutions with a proven track record, not theories or speculation
- Solution is not obvious =
  - Best patterns generate a solution to a problem indirectly
- Describe a relationship =
  - Patterns don't just describe modules but describe deeper system structures and mechanisms
- Elegant in it's a approach to unity

Kinds of patterns

- pg. 291
- Rubber necking
- creational patterns =
  - The "creation composition and representation of objects"
- Structural patterns =
  - Focuses on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure
  - Behavioral patterns =
    - Responsibility between objects
- Architectural patterns =
  - Describes broad based design problems that are solved using a structural reproach
- Data patterns =
  - Describes recurring data
- Component patterns (design pattern) =
  - Address problems associated with the development of subsystems
- Interface design patterns =
  - Describe common user interface problems and their solutions within a system
- WebApp patterns =
  - Address a problem set that is encountered when building a WebApp
- Mobile patterns =
  - The context is a mobile platform

Frameworks

- Patterns themselves may not be sufficient to develop a complete design
- In cases it may be necessary to provide a implementation-specific skeletal infrastructure
- ★ You can select a <u>reusable architecture</u> that provides the generic structure and behavior for a family of software

A framework is not an architectural pattern, but rather a skeleton with a collection of plug points(hooks and slots)

Pattern description
- Pg.294
- Pattern name = describes the essence of the pattern in a short but expressive name
- Problem = describes the problem that the pattern address
- Motivation = provides an example of the program
- Context = describes the environment in which the problem resides
- Forces = list the system of forces that affect the manner In which the problem must be solved
- Solution = provides a detailed description of the solution properties
- Intent = describes the pattren and what it does
- Collaborations = describes how other patterns contribute to the solutions
- Consequences = describes the potential trade-offs that must be considered when a pattern is implemented
- Implementation = identifies special issues that should be considered when implementing the pattern
- Known uses = provides examples of actual uses of the design pattern in real applications
- Related patterns =

Pattern based design
- ★ <u>A software designer begins with a requirements model</u> (either explicit or implied) that presents an abstract representation of the system
- The requirements model describes the problem set, establishes context, and identifies the system forces
- If you discover you are faced with a problem, and system of forces that solved before, then use it
- If it is a new problem use methods and modeling to make a new pattern

Thinking in patterns
1. Be sure you understand the big picture (requirements model) / the context
2. Examining the big picture, extract the patterns that are present at the level of abstraction
3. Begin your design with big picture patterns that establish a context or skeleton for further design work
4. Work inward form the context
5. Repeat steps 1 - 4 until the design is completely fleshed out
6. Refine the design by adapting each pattern to the specifics of the software your trying to build

Design task pg.297 - 298
- Exam the requirements model and develop a hierarchy
- Determine if a reliable

Most pattern development is done from the top down rather then bottom up

Pattern organization table
- List patterns and systems

Common mistakes
- Not enough time being spent to understand the underlying problem , its context forces and as a consequence, you select the pattern that looks right but is actually wrong
- Once the wrong pattern is selected don't refuse to see the error and try to force it to work
- In other cases the problem has forces that are not considered by the pattern you have chosen, resulting in a poor erronious fit
- Sometimes a pattern is applied too literally and the required adaption for your problem

spaces are not implemented

Component- level-design

- Component-level design patterns provide a proven solution that addresses one or more sub-problems extracted from the requirements model
- In many cases, design patterns of this type focus on some functional element of a system
- Having enunciated the sub-problem that must be solved, consider context and the system of forces that affect the solution

Anti-patterns

- ★ Anti-patterns describe commonly used solutions to design problems that have a negative affect on the software
- Ant-patterns can provide tools to help developers recognize when these problems exist and may provide detailed plans for reversing the underlying problem causes and implementing

Selected anti-patterns pg.304

- Blob = single class with a large number of attributes, operators or both
- Stovepipe system = a barley maintainable assemblage of ill-related components
- ★ Boat anchor = retaining a part of the system that no longer has any use
- Spaghetti code = program whose structure is barley comprehensible, especially because of misuse code constraints
- Copy and paste programs
- ★ Silver bullet
- Programming by permutation

Chapter 15: Quality Concepts

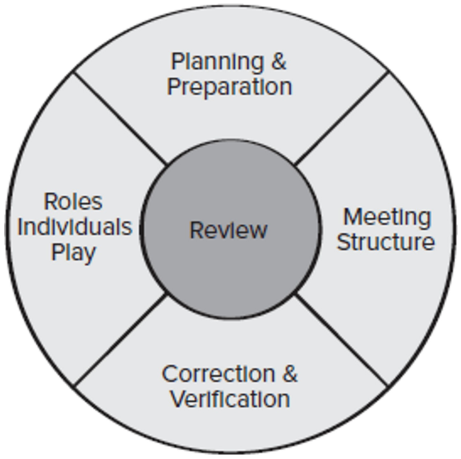
February 8, 2021 8:58 AM

- Views
- Software quality
  - ★ An effective software process defined in a manner that creates a useful product that provides measurable value for those who produce it and those who use it
  - An effective software process establishes infrastructure that supports building a high-quality software product
  - Chaos - determines poor quality
  - Structure, analysis, change management, technical reviews - determines high quality
- Quality in use - ISO25010:2017
- Cost of Quality
  - Prevention costs
    - Quality planning
    - Formal technical reviews
    - Test equipment
    - Training
  - Appraisal costs
    - Conducting technical reviews
    - Data collection and metrics evaluation
    - Testing and debugging
  - Internal failure costs
    - Rework
    - Repair
    - Failure mode analysis
  - External failure costs
    - Complaint resolution
    - Product return and replacement
    - Help line support
    - Warranty work
- Impact of Management Decisions
  - Estimation decisions - irrational delivery dates can effect quality
  - Scheduling decisions - failing to find task dependencies when creating the schedule
  - Risk-oriented decisions - reacting rather than building mechanisms to monitor risks
- Achieving Software Quality
  - Understand the problem to be solved
  - Design that conforms to problem requirements
- Project management - plans for quality and change management
  - Use estimation to verify that delivery dates are achievable
  - Understood schedule and no shortcuts
  - Risk planning to avoid crisis'
- ★ Quality control - series of inspections, reviews, and tests used to ensure conformance to specifications
- ★ Quality assurance - consists of auditing and reporting to provide management with the proper data

Chapter 16: Reviews—A Recommended Approach

February 8, 2021 8:59 AM

- Reviews
  - ★ meetings conducted by technical people for technical people
  - A technical assessment of a work product
  - ★ A software quality assurance mechanism
  - A training ground
- Reviews are not
  - A project summary or progress assessment
  - A meeting intended to solely to impart information
  - A mechanism for political or personal reprisal
- What do we look for?
  - ★ Error - a quality problem found before release
  - ★ Defect - a quality problem found after release
  - These distinctions are not mainstream thinking
- Defect Amplification and Removal
  - Defect amplification - how a defect introduced early in the software engineering work flow and undetected, can and often will be amplified into multiple errors during the design and more errors in the construction
  - Defect propagation - the impact an undiscovered defect has on future development activities or product behavior
  - Technical debt - the costs incurred by failing to find and fix defects early or failing to update documentation following software changes
- Metrics - are measures
  - Effort, E - in-person hours
  - Preparation effort, E_p - the effort required to review a work product prior to the review meeting
  - Assessment effort, E_a - the effort that is expending during the actual review
  - Rework effort, E_r - the effort that is dedicated to the correction of those errors uncovered during the review
  - ★ Work Product Size, WPS - lines of code or number of pages
  - Minor errors found, Err_{minor} - the number of errors found that can be categorized as minor
  - Major errors found, Err_{major} - the number of errors found that can be categorized as major
  - The total review effort
    - $E_{review} = E_p + E_a + E_r$
    - $E_{tot} = Err_{minor} + Err_{major}$
  - Defect density represents the errors found per unit of WPS
    - Defect density = $\frac{Err_{tot}}{WPS}$
  - Example
    - The average defect density for a requirements model is 0.68 errors per page, and a new requirements model is 40 pages long, 0.68 x 40 = approximately 27 errors
    - If you only find 9 errors, you've done an extremely good job in developing the requirements model or your review approach was not thorough enough
  - Example
    - The effort required to correct a minor model error will require 4 person-hours
    - The effort required for major requirements error is 18 person-hours
    - Minor errors occur about 6 times more frequently than major errors
    - The average effort required to correct 1 error will be 6 hours (4 Hours * 6 + 18 Hours)/7 = 6 hours
  - ★ Effort saved per error = $E_{testing} - E_{reviews}$
- Informal Reviews - the benefit is immediate discovery of errors and better work product quality
  - A simple desk check with a colleague
  - A casual meeting (2 or more people)
  - The review-oriented aspects of pair programming



        - ★ pair programming - encourages continuous review as a work product is created
    - Quality can suffer
- Formal Technical Reviews, FTRs, Code Inspections, Code Walkthroughs - 5 key objectives
    - ★ To uncover errors in functions, logic, implementation in any representation of the software
    - ★ To verify that the software meets its requirements
    - ★ To ensure that the software adheres to a standard
    - ★ To achieve software that is developed in a uniform manner
    - ★ To make projects more manageable
- Review Meeting
    - Between three and five people should be involved in the review
    - Advance preparation - should require no more than two hours of work per person
    - The duration should be less than two hours
    - Focus is on a specific work product
- Review Players
    - Producer - the individual who has developed the work product
    - Review leader - evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advanced preparation and facilitates the meeting discussion
    - Reviewers - expected to spend between one and two hours reviewing the product, making notes, and becoming familiar with the work
        - Come prepared to evaluate
    - Recorder - records (in writing) all important issues
- Review Outcome - a decision must be made to:
    - ★ Accept the product without modification
    - ★ Reject the product due to severe errors
    - ★ Accept the product conditionally (correct errors and no additional review)
- Review Reporting and Record Keeping
    - The recorder records all issues raised and summarizes these in an action list for the producer (formal technical review summary report) - this answers three questions:
        - What was reviewed?
        - Who reviewed it?
        - What were the issues and conclusions (Review Outcome)?
    - You should establish a follow-up procedure to ensure that items on the issues list have been properly corrected
- Review Guidelines
    - Review the product, not the producer
    - Stay on-topic
    - Limit the debate
    - Enunciate problems, but don't try to solve every problem
    - Take written notes
    - Limit the number of participants and insist doing advance preparation
    - Develop a checklist for each product that is likely to be reviewed
    - Allocate proper resources and time for FTRs
    - Conduct meaningful training for all reviewers
    - Review your early reviews
- Post-mortem Evaluations, PME
    - A mechanism to determine what went right and what went wrong
    - Attended by members of the software team and stakeholders who focus on excellences and challenges
- Agile reviews
    - User stories are reviewed and ordered by priority
    - Daily scrum meetings to ensure members are working to try to catch and defects
    - Sprint review meetings using guidelines
    - Sprint retrospective meetings to capture the lessons learned from previous meetings

Chapter 17: Software Quality Assurance

February 15, 2021 9:11 AM

- Elements of SQA - Software Quality Assurance
  - Standards
  - Reviews and Audits
  - Testing
  - Error/defect collection and analysis
  - Change management
  - Education
  - Vendor management
  - Security management
  - Safety
  - Risk management
- Tasks of the SQA Group
  - Prepares an SQA plan for a project that identifies
    - Evaluations to be performed
    - Audits and reviews to be performed
    - Standards that are applicable 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
  - Develops 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 plans
  - Verifies compliance with the defined software process
    - Identifies, documents, and tracks deviations from the process and verifies that corrections have been made
    - Periodically reports the results of its work to the project manager
    - Ensures that deviations is documented and handled according to procedure
    - Records and noncompliance and reports to senior management
- SQA Goals
  - ★ Requirements Quality
    - The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow
    - Traceability - the number of requirements not traceable to code
    - Model Clarity
  - ★ Design Quality
    - Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and conforms to requirements
    - Architectural integrity
    - Interface complexity
  - ★ Code Quality
    - Source code and related work products must conform to standards and exhibit maintainability
    - Complexity
  - ★ Quality control effectiveness - QC effectiveness
    - Apply limited resources in the most effective way possible
    - Resource allocation - staff hour percentage per activity

- Formal SQA
  - Assumes that a rigorous syntax and semantics can be defined for every programming language
  - Allows the use of a rigorous approach to the specification of the requirements
  - Applies mathematical proofs to demonstrate that the program conforms to specifications
- Statistical SQA
  - Errors and defects are collected and categorized
  - Try to trace each error and defect's origin
  - ★ Pareto principle - 80% of defects can be traced to 20% of all possible causes
  - Move to correct the vital few 20% of problems
- ★ Six Sigma 6σ - (standard of deviation) for Software Engineering
  - Six standard deviations - 3.4 defects per million occurrences - implying an extremely high quality standard
  - Defines three core steps and two follow-up steps:
    - ★ Define customer requirements and deliverables
    - ★ Measure the existing process and its output to determine quality performance
    - ★ Analyze defect metrics and determine the vital few causes
  - For the same process:
    - ★ Improve the process by eliminating root causes
    - ★ Control the process to ensure future work does not reintroduce the causes
  - For a new process being developed:
    - ★ Design the process to avoid root causes and meet customer requirements
    - ★ Verify that the process will avoid defects and meet customer requirements
- Software Reliability - probability of failure-free operation over a period of time
  - MTBF - Mean-time-between-failure
  - MTTF - Mean-time-to-failure
  - MTTR - Mean-time-to-repair
  - ★ MTBF = MTTF + MTTR
- Software Availability - probability of requirements being met over a period of time
  - ★ Availability = [MTTF/(MTTF + MTTR)] x 100%
- AI and Reliability Models - AI always requires statistics
  - ★ Bayesian inference uses Bayes' theorem to update the probability for a hypothesis as more evidence becomes available
  - Bayesian inference can be used to estimate probabilistic quantities using incomplete historic data
  - ★ Regression model - used to estimate where and what type of defects might occur in future prototypes
  - Genetic algorithms - used to grow reliability models based on historic data
- Software Safety - finds potential hazards that would cause an entire system failure
- ISO 9001:2008 Standard - contains 20 requirements
- SQA Plan Contents
  - Purpose of the plan
  - Description of all products inside the purview of SQA
  - Applicable standards
  - Where SQA tasks are placed throughout the software process
  - Tools for SQA tasks
  - Configuration procedures
  - Safety of SQA records
  - Organize responsibilities

Chapter 19: Software Testing—Component Level

February 22, 2021 9:00 AM

- Strategic Approach to Testing
    - Conduct effective technical reviews before testing begins
    - Testing begins at the component level and works toward integration on the entire system
    - Use different testing techniques for the appropriate software
    - Testing is conducted by the developer of the software and an independent test group
    - Testing and debugging are different, debugging must be used in testing
- Verification and Validation
    - ★ Verification ensures that software correctly implements a function
    - ★ Validation ensures that software is traceable to customer requirements
- Organizing for Testing
    - Software developers are responsible for testing individual program components
    - When software architecture is complete then the independent test group is involved
    - ★ The independent test group (ITG) is there to prevent the builder from testing their own product
    - ITG personnel are paid to find errors
    - Developers and ITG work closely to ensure thorough tests are conducted
- Testing Strategy - figure picture
    - System testing
    - Validation testing
    - Integration testing
- Role of Scaffolding
    - ★ Scaffolding is required to create a testing framework
    - A driver must be developed for each unit test
    - A driver is a "main program" that accepts testcase data
    - Stubs (dummy subprogram) replace modules invoked by the component to be tested
    - A stub uses the module's interface, may do minimal manipulation, prints verification entry, and returns control to the module undergoing testing
- Criteria for Done
    - Testing is never done; the burden is shifted from the engineer to the user (wrong)
    - You're done testing when you are out of time or money (wrong)
    - The statistical quality assurance approach suggests executing tests derived from a statistical sample of all possible program executions by all targeted users
    - ★ By collecting metrics during testing and making use of existing statistical models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?"
- Test Planning
    - Specify quantifiable measures of the requirements 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"
    - ★ Rapid cycle testing tests at the end of every sprint
    - Build a robust software that is designed to test itself
    - Use effective technical reviews as a filter prior to testing
    - Conduct technical reviews to assess the strategy and test cases themselves
    - Develop a continuous improvement approach for the testing process
- Test Recordkeeping
    - Briefly describes the test case

    - Have a pointer to the requirement being tested
    - Have expected output from the test case data on the criteria for success
    - Indicate whether the test was passed or failed
    - Dates the test case was run
    - Should have room for comments about why a test may have failed (aids in debugging)
- Cost Effective Testing
    - Exhaustive testing requires every possible combination and ordering of input values be processed by the test component
    - The return on exhaustive testing is often not worth the effort
    - Testers should work smarter and allocate their testing resources on modules crucial to the success of the project or those that are suspected to be error-prone as the focus of their testing unit
- Test Case Design
    - Design unit test cases before you develop code for a component to ensure that code will pass the tests
    - Test cases are designed to cover the following areas:
        - ★ The module interface is tested to ensure that information properly flows into and out of the program unit.
        - ★ Local data structures are examined to ensure that stored data maintains its integrity during execution
        - ★ Independent paths through control structures are exercised to ensure all statements are executed at least once
        - ★ Boundary conditions are tested to ensure module operates properly at boundaries established to limit or restrict processing
        - ★ All error-handling paths are tested
- 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 be neither too simple nor too complex
- Error Handling
    - A good design anticipates error conditions and establishes error-handling paths which must be tested
    - Among the potential errors that should be tested when error handling is evaluated are:
        - Error description is unintelligible
        - Error noted does not correspond to error encountered
        - Error condition causes system intervention prior to error handling
        - Exception-condition processing is incorrect
        - Error description does not give enough information
- Traceability
    - ★ To ensure that the testing process is auditable, each test case needs to be traceable back to specific functional or non-functional requirements to anti-requirements
    - Often non-functional requirements need to be traceable to specific business or architectural requirements
    - Many test process failures can be traced to missing traceability paths, inconsistent test data, or incomplete test coverage
    - ★ Regression testing requires retesting of selected components that may be affected by changes made to other collaborating software components
- ★ White Box Testing (Glass Box Testing, Clear Box Testing)
    - Using white-box testing methods, you can derive test cases that:
        - Guarantee that all independent paths within a module have been exercised at least once
        - Exercise all logical decisions on their true and false sides

        - Execute all loops at their boundaries and within their operational bounds
        - Exercise internal data structures to ensure validity
- Basic Path Testing
    - Determines the number of independent paths in the program by computing Complexity:
        - ★ The number of regions of the flow graph corresponds to the cyclomatic complexity (book example has 4)
        - ★ Cyclomatic complexity $V(G)$ for a flow graph G is defined as (E = Edge, N = Node, P = Predicate Node)
            - $V(G) = E - N + 2$
            - $V(G) = P + 1$
        - ★ An independent path is any path through the program that introduces at least one new set of processing statements or a new condition (book examples)
            - Path 1: 1-11
            - Path 2: 1-2,3-4,5-10-1-11
            - Path 3: 1-2-3-6-8-9-10-1-11
            - Path 4: 1-2-3-6-7-9-10-1-11
    - Designing test cases
        - Use the code as a foundation
        - Determine the cyclomatic complexity of the flow graph
        - Determine a basis set of linearly independent paths
        - Prepare the test cases that will force execution of each path in the basis set
- Control Structure Testing
    - ★ Condition testing is a test-case design method that exercises logical conditions contained in a program module
    - ★ Data flow testing selects test paths according to the locations of definitions and uses variables in the program
    - ★ Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs
- Loop Testing
    - Test cases for simple loops:
        - Skip the loop entirely
        - One pass through the loop
        - Two passes through the loop
        - m passes through the loop where m < n (normally)
        - n - 1, n, n + 1 passes through the loop (boundaries)
    - Test cases for nested loops:
        - Start at the innermost loop and set all other loops to minimum values
        - Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values
        - Add other tests for excluded values
        - Work outward, conducting tests for the next loop, but keeping all the other outer loops at minimum values and other nested loops to "typical" values
        - Continue until all loops have been tested
- ★ Black Box Testing
    - Black-box (functional) testing attempts to find errors in the following categories:
        - Incorrect or missing functions
        - Interface errors
        - Errors in data structures or external database access
        - Behavior or performance issues
        - Initialization and termination errors
    - Questions
        - How is functional validity tested?
        - How are system behavior and performance tested?

        - What classes of input will make a good test case?
        - Is the system particularly sensitive to certain input values?
    - Interface Testing
        - ★ Interface testing is used to check that a program component accepts information passed to it in the proper order and data types and returns information in proper order and data format
        - Components are not stand-alone programs testing interfaces requires the use of stubs and drivers
        - Stubs and drivers sometimes incorporate test cases to be passed to the component or accessed by the component
- Object-Oriented Testing (OOT)
    - To adequately test OO systems, three things must be done:
        - The definition of testing must be broadened to include error discovery techniques applied to object-oriented analysis and design models
        - The strategy for unit and integration testing must change significantly
        - The design of test cases must account for the unique characteristics of OO software
    - Class Testing (Unit testing)
        - Class testing for OO software is the equivalent of unit testing for conventional software
        - Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flows across the module interface
        - Class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class
    - Behavior Testing
        - ★ A state diagram can be used to help derive a sequence of tests that will exercise dynamic behavior of the class
        - Tests to be designed should achieve full coverage by using operation sequences cause transitions through all allowable states
        - When class behavior results in a collaboration with several classes, multiple state diagrams can be used to track system behavioral flow
    - Boundary Value Analysis (BVA)
        - ★ Boundary value analysis leads to a selection of test cases that exercise bounding values
        - Guidelines for BVA:
            - If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b just above and just below a and b
            - If an input condition specifies a number of values, test cases should be developed that exercise the min and max numbers as well as values just above and below min and max
            - Apply guidelines 1 and 2 to output conditions
            - If internal program data structures have prescribed boundaries be certain to design a test case to exercise the data structure at its boundary

Chapter 20: Software Testing—Integration Level

March 8, 2021 9:12 AM

Midterm---

- Integration Testing
  - Integration testing is a systematic technique for constructing the software architecture while conducting tests to uncover errors associated with interfacing
  - The objective is to take unit-tested components and build a program structure that matches the design
  - In the big bang approach, all components are combined at once and the entire program is tested as a whole. Chaos usually results!
  - In incremental integration a program is constructed and tested in small increments, making errors easier to isolate and correct. Far more cost-effective!
- Top-Down Integration
  - Top-down integration testing is an incremental approach to construction of the software architecture
  - Modules are integrated by moving downward through the control hierarchy, beginning with the control module (main program)
  - Modules subordinate to main control module are incorporated into the structure followed by their subordinates
  - Depth-first integration integrates all components on a major control path before starting another major control path
  - Breadth-first integration incorporates all components directly subordinate at each level, moving across to integrate
- Top-Down Integration Testing
  - The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module
  - Depending on the integration approach selected, subordinate stubs are replaced one at a time with actual components
  - Tests are conducted as each component is integrated
  - On completion of each set of tests, another stub is replaced with the real component
  - Regression testing may be conducted to ensure that new errors have not been introduced
- Bottom-Up Integration Testing
  - Bottom-up integration testing, begins construction and testing with atomic modules components at the lowest levels in the program structure
    - Low-level components are combined into clusters that perform a specific software subfunction
    - A driver is written to coordinate test-case input and output
    - The cluster is tested
    - Drivers are removed and clusters are combined, moving upward in the program structure
- Continuous Integration
  - Continuous integration is the practice of merging components into the evolving software increment at least once a day
  - This is a common practice for teams following agile such as XP or DevOps. Integration testing must be quickly implemented as the program is being built
  - Smoke testing is an integration testing approach that can be used when software is developed by an agile team using short increment build times
- Smoke Testing Integration

    - Software components that have been translated into code are integrated into a build. That includes all data files, libraries, reusable modules, and components required to implement one or more product functions
    - A series of tests is designed to expose "show-stopper" errors that will keep the build from properly performing its function cause the project to fall behind
    - The build is integrated with other builds, and the entire product is smoke tested daily
- Smoke Testing Advantages
    - ★ Integration risk is minimized, since smoke tests are run daily
    - ★ Quality of the end product is improved, functional and architectural problems are uncovered early
    - ★ Error diagnosis and correction are simplified, errors are most likely in the new build
    - ★ Progress is easier to assess, each day more of the final product is complete
    - Smoke testing resembles regression testing by ensuring newly added components do not interfere with the behaviors of existing components
- Integration Testing Work Products
    - ★ An overall plan for integration of the software and a description of specific tests is documented in a test specification
    - Test specification incorporates a test plan and a test procedure and becomes part of the software configuration
    - Testing is divided into phases and incremental builds that address specific functional and behavioral characteristics of the software
    - Time and resources must be allocated to each incremental build along with the test cases needed
    - A history of actual test results, problems, or peculiarities is recorded in a test report and may be appended to the test specification
    - It is often best to implement the test report as a shared Web document to allow all stakeholders access to the latest test results and the current state of the software increment
- 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 the software is corrected, some aspects of the software configuration are changed
    - Regression testing helps to ensure that changes do not introduce unintended behavior
    - Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools
    - AI tools may be able to help select the best subset of test cases to use in regression automatically based on previous experiences of the developers with the evolving software product
- OO Integration Testing
    - ★ Thread-based testing, integrates the set of classes required to respond to one input or event for the system
    - Each thread is integrated and tested individually
    - Regression testing is applied to ensure so side effects occur
    - Use-based testing, begins the construction of the system by testing those classes that use very few server classes
    - Independent classes are tested next
    - The sequence of testing layers of dependent classes continues until the entire system is constructed
- OO Testing - Fault-based Test Case Design
    - ★ The object of fault-based testing is to design tests that have a high likelihood of uncovering plausible faults
    - Because the product or system must conform to customer requirements, fault-based testing begins with the analysis model

    - The strategy for fault-based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis
    - To determine whether these faults exist, test cases are designed to exercise the design or code
- Fault-based OO Integration Testing
    - ★ Fault-based integration testing looks for plausible faults in operation calls or message connection:
        - Unexpected results
        - Wrong operation/message used
        - Incorrect invocation
    - Integration testing applies to attributes and operations - class behaviors are defined by the attributes
    - Focus of integration testing is to determine whether errors exist in the client code, not the server code
    - Scenario-based testing uncovers errors that occur when any actor interacts with the software
    - Concentrates on what the user does, not what the product does
    - This means capturing the tasks that the user has to perform and then applying them and their variants
    - ★ Scenario-based testing uncovers interaction errors
    - Scenario-based testing tends to exercise multiple subsystems in a single test
- OO Testing - Random Test Case Design
    - ★ For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes
    - For each message that is generated, determine the collaborator class and the corresponding operation in the server object
    - For each operation in the server object, determine the messages that it transmits
- Validation Testing
    - ★ Validation testing tries to uncover errors, but the focus is at the requirements level - on user visible actions and user-recognizable output from the system
    - Validation testing begins at the culmination of integration testing, the software is completely assembled as a package and errors have been corrected
    - Each user story has user-visible attributes, and the customer's acceptance criteria which forms the basis for the test cases used in validation-testing
    - A deficiency list is created when a deviation from a specification is uncovered and their resolution is negotiated with all stakeholders
    - An important element of the validation process is a configuration review that ensures the complete system was built properly

Black Box Testing

March 19, 2021 8:58 AM

- ★ Comparison Testing
  - Used only in situations in which the reliability of software is absolutely critical
    - Separate software engineering teams develop independent versions of an application using the same specification
    - Each version can be tested with the same test data to ensure that all provide identical output
    - Then all versions are executed in parallel with real-time comparison results to ensure consistency
- ★ Orthogonal Array Testing
  - Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded

Chapter 21: Software Testing—Specialized Testing for Mobility

March 19, 2021 9:27 AM

- Create a Mobile Test Plan
  - Do you have to build a fully functional prototype before you test with users?
  - Should you test with the user's device or provide a device for testing?
  - What devices and user groups should you include in testing?
  - When is lab testing versus remote testing appropriate?
- Mobile Testing Guidelines
  - Understand the network landscape and device landscape
  - Conduct testing in uncontrolled real-world test conditions
  - Select the right automation test tool
  - Identify the most critical hardware/platform combinations to test
  - Check end-to-end functional flow in all possible platforms at least once
  - Conduct performance, GUI, and compatibility testing using actual devices
  - Measure Mobile performance under realistic load conditions
- Mobile Testing Strategies
  - User-experience testing. Users are involved early in the development process to ensure that the MobileApp lives up to the usability and accessibility expectations of the stakeholders on all supported devices
  - Device compatibility testing. Testers verify that the MobileApp works correctly on all required hardware and software combinations
  - Performance testing. Testing to check non-functional requirements unique to mobile devices
  - Connectivity testing. Testers ensure that the MobileApp can access any needed networks or Web services and can tolerate weak or interrupted network access
  - Security testing. Testers ensure that the MobileApp does not compromise the privacy or security requirements of its users
  - Testing in the wild. The app is tested under realistic conditions on actual user devices in a variety of networking environments around the globe
  - Certification Testing. Testers ensure that the MobileApp meets the standards established by the app stores that will distribute it
- UX (User Experience) - Gesture Testing Issues
  - Touch screens are ubiquitous on mobile devices and developers have added multitouch gestures as a means of augmenting the user interaction possibilities without losing screen real estate
  - Paper prototypes cannot be used to adequately review the adequacy or efficacy of gestures
  - It's difficult to use automated tools to test gesture interface actions
  - Location of screen objects is affected by screen size and resolution making accurate gesture testing difficult
  - Accessibility testing for visually impaired users is challenging because gesture interfaces typically do not provide either tactile or auditory feedback
- UX - Virtual Keyboards
  - Virtual keyboard may obscure part of the display screen when activated, it is important to ensure that important screen information is not hidden from the user while typing
  - Virtual keyboards are smaller than personal computer keyboards, it is hard to type with 10 fingers and they provide no tactile feedback
  - The MobileApp must be tested to ensure that it allows easy error correction and can manage mistyped words without crashing
  - Predictive technologies are often used to help expedite user input

    - It is important to test the correctness of the word completions for the natural language chosen
- UX - Voice Input
    - Voice input has become an increasingly common method for providing input and commands in hands-busy, eyes-busy situations
    - Using voice commands to control a device impresses a greater cognitive load on the user, than pointing to a screen object or pressing a key
    - Testing the quality and reliability of voice input and recognition needs to take environmental conditions and individual voice variation into account
    - The MobileApp should be tested to ensure that bad input does not crash the MobileApp or the device
    - It is important to log errors to help developers improve the ability of the MobileApp to process speech input
- UX - Alerts and Extraordinary Conditions
    - Part of MobileApp testing should focus on the usability issues relating to alerts and pop-up messages
    - Testing should examine the clarity and context of alerts, the appropriateness of their location on the device display screen
    - When foreign languages are involved, verification that the translation from one language to another is correct
    - You should not rely solely on testing in a development environment and you must test MobileApp in the wild on actual devices
    - Apps must recover from faults and resume processing with little or no downtime and in some cases the system must be fault tolerant
    - <u>Recovery testing</u> is a system test that forces the software to fail in a variety of ways verifies that recovery is properly performed
- WebApp Testing Steps
    - The content model for the WebApp is reviewed to uncover errors
    - The Interface model is reviewed to ensure that all use cases can be accommodated
    - The design model for the WebApp is reviewed to uncover navigation errors
    - The user interface is tested to uncover errors in presentation and/or navigation mechanics
    - Each functional component is unit tested
    - Navigation throughout the architecture is tested
    - The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration
    - Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment
    - Performance tests are conducted
    - The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated errors
- ★ WebApp - Content Testing
    - Content testing has three important objectives:
        - To uncover syntactic errors in text-based documents, graphical representations, and other media
        - To uncover semantic errors in any content object presented as navigation errors
        - To find errors in the organization or structure of the content that is presented to the end-user
- WebApp - Interface Testing
    - Interface features tested to ensure design rules, aesthetics, and content is available to user without error
    - Individual interface mechanisms are tested in a manner that is analogous to unit testing
    - Each interface mechanism is tested within the context of a use-case or NSU (Network Semantic Units) for a specific user category

    - Complete interface is tested against selected use-cases and NSUs to uncover errors in interface semantics
    - The interface is tested within a variety of environments to ensure that it will be compatible
- ★ WebApp - Navigation Testing
- Internationalization and Localization
    - Internationalization is the process of creating a software product so that it can be used in several countries with various languages without requiring any engineering changes
    - Localization is the process of adapting a software application for used in targeted global regions by adding locale-specific requirements and translating text elements to appropriate languages
    - Crowdsourcing is a distributed problem-solving model where community members work on solutions to
- Security Testing
    - Designed to probe for vulnerabilities of the client-side environment, the network communications that occur as data are passed from client to server and back again, and server-side environments
    - On the client-side, vulnerabilities can often be traced to pre-existing bugs in browsers, e-mail programs, or communication software
    - On the server-side, vulnerabilities include denial-of-service attacks and malicious scripts that can be passed along to the client-side or used to disable server operations
- Performance Testing
    - Tests response times
    - Tests to see if those times are acceptable
    - Tests to see what is causing performance degradation
    - Tests the average response time under load
    - Tests to see if degradation impacts the security of the system
    - Tests the accuracy of the system under growing loads
    - Tests what happens when servers reach above maximum capacity
- Load Testing
    - The intent is to determine how the WebApp and its server-side environment will respond to various load conditions
        - N, number of concurrent users
        - T, number of on-line transactions per unit of time
        - D, data load processed by server per transaction
    - Overall throughput, P, Is computed in the following manner:
        - P = N x T x D
- Stress Testing
    - Stress testing for mobile apps attempts to find errors that occur under extreme operating conditions such as:
        - Running several mobile apps on the same device
        - Infecting system software with viruses or malware
        - Attempting to take over a device and use it to spread spam
        - Forcing the mobile app to process inordinately large numbers of transactions
        - Storing inordinately large quantities of data on the device
- Creating Weighted Device Platform Matrix
    - To mirror real-world conditions, the demographic characteristics of testers should match those of targeted users, as well as those of their device
    - A weighted device platform matrix (WDPM) helps to ensure that test coverage includes combination of mobile device and context variables
        - List the important operating system variants as the matrix column labels
        - List the targeted devices as the matrix row labels
        - Assign ranking to indicate the relative importance of each operating system and each device

        - Compute the product of each pair of rankings and enter each
- Testing AI Systems
    - Static testing is a software verification technique that focuses on review rather than executable testing. It is important to ensure that human experts agree with the ways in which the developers have represented the information and its use in the AI system.
    - Dynamic testing for AI systems is a validation technique that exercises the source code with test cases. The intent is to show that the AI system conforms to the behaviors specified by the human experts
    - Model-based testing (MBT) a black-box testing technique that uses the requirements model as the basis for the generation of test cases from the behavioral model
- Test Virtual Environments
    - Acceptance tests are a series of specific tests conducted by the customer to uncover product errors before accepting the software from the developer
    - An alpha test is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems
    - A beta test is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. The customer records all problems that are encountered and reports these at regular intervals
- Usability Test Categories
    - Interactivity - are interaction mechanisms easy to understand and use?
    - Layout - are navigation mechanisms, content, and functions placed in a manner that allows the user to find them quickly?
    - Readability - is the text well written and understandable? Are the graphic representations easy to understand?
    - Aesthetics - do layout, color, typeface, and related characteristics lead to ease of use? Do users "feel comfortable" with the look and feel of the app?
    - Display characteristics - does the app make optimal use of screen size and resolution?
    - Time sensitivity - can important features, functions, and content be used acquired in a timely manner?
    - Feedback - do users receive meaningful feedback to their actions? Is the user's work interruptible and recoverable when a system message is displayed?
    - Personalization - does the app tailor itself to the specific needs of different user categories or individual users?
    - Help - is it easy for users to access help and other support options?
    - Accessibility - is the app accessible to people who have disabilities?
    - Trustworthiness - are users able to control how personal information is shared?
- Accessibility Testing
    - Ensure that non-text screen objects are also represented by a text-based description
    - Verify that color is not used exclusively to convey information to the user
    - Demonstrate that high contrast and magnification options are available for visually challenged users
    - Ensure that speech input alternatives have been implemented to accommodate users that might not be able to manipulate a keyboard, keypad, or mouse
    - Demonstrate that blinking, scrolling, auto content updating is avoided to accommodate users with reading difficulties
- Playability Testing
    - Playability is the degree to which a game or simulation is fun to play and usable by the user/player
- Documentation Testing
    - Errors in documentation help facilitate or online program documentation can be devastating to the acceptance of the program
    - Documentation testing should be an important part of every software test plan

- The first phase, technical review examines the document for editorial clarity
- The second phase, a live test, uses the documentation in junction with...

Chapter 22: Software Configuration Management

March 29, 2021 9:00 AM

- Configuration Management System Elements
  - Component elements - a set of tools coupled within a file management system that enables management of each configuration item
  - Process elements - a collection of procedures and tasks that define an effective approach to change management
  - Construction elements - tools that automate the construction of software to ensure proper components are used
  - Human elements - a set of tools to implement SCM (Software Configuration Management)
- Baselines
  - The IEEE defines a baseline as:
    - A specification or product that has been formally reviewed and agreed upon, that thereafter serves as basis for further development of the product, and that can be changed only through formal change control programs
  - A baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCI
- Management of Dependencies and Changes
  - It is important for developers to maintain software work products to ensure dependencies among the SCI are documented
  - Developers must establish discipline when checking items in and out of the SCM repository
  - Impact management involves two complementary aspects:
    - Ensuring that developers employ strategies to minimize the impact of their colleagues' actions on their own work
    - Encouraging software developers to use practices that minimize the impact of their own work on that of their colleagues
- SCM Repository
  - The SCM repository is the set of mechanisms and data structures that provides the following functions that allow a software team to manage change:
    - Data integrity
    - Information sharing
    - Tool integration
    - Data integration
    - Methodology enforcement
    - Document standardization
- SCM Repository Features
  - ★ Versioning - saves versions to manage product releases and allow developers to go back to previous versions
  - ★ Dependency tracking and change management - manages a wide variety of relationships among the data elements stored in it
  - ★ Requirements tracing - provides the ability to track all design and construction components and deliverables resulting from a specific requirement specification
  - ★ Configuration management - tracks series of configurations representing specific project milestones or production releases and provides version management
  - ★ Audit trails - establishes additional information about when, why, and by whom changes are made
- SCM Best Practices
  - Keeping the number of code variants small
  - Test early and often
  - Integrate early and often

    - Use tools to automate testing, building, and code integration
- Continuous Integration Advantages
    - Accelerated feedback - notifying developers immediately when integration fails, allows fixes when the number of changes is small
    - Increased quality - building and integrating software whenever necessary provides confidence into the quality of the product
    - Reduced risk - integrating components early avoids a long integration phase, design failures are discovered and fixed early
    - Improved reporting - providing additional information allows for accurate configuration status accounting
- Change Management Objectives
    - ★ Software change management process defines a series of tasks that have four primary objectives:
        - To identify all items that collectively define the software configurations
        - To manage changes to one or more of these items
        - To facilitate the construction of different versions of an application
        - To ensure that software quality is maintained as the configuration evolves over time
- Impact Management
    - A web of software work product interdependencies must be considered every time a change is made
        - ★ Impact management is accomplished with three actions:
            - An impact network identifies the stakeholders who might effect or be affected by changes that are made to the software based on its architectural documents
            - Forward impact management assesses the impact network and then informs members of the impact of those changes
            - Backward impact management examines changes that are made by other team members and their impact on your work and incorporates mechanisms to mitigate the impact
- Software Configuration Audit
    - ★ Software configuration audit complements a technical review by asking and answering the following questions:
        - Has the change specified in the ECO (Engineering Change Order) been made? Have any additional modifications been incorporated?
        - Has a technical review been conducted to assess technical correctness?
        - Has the software process been followed, and have software engineering standards been properly applied?
        - Has the change been "highlighted" in the SCI (Software Configuration Item)? Do the attributes of the configuration object reflect change?
        - Have SCM procedures for noting the change, recording it, and reporting it been followed?
        - Have all related SCI been properly updated?
- Configuration Status Reporting
    - ★ Configuration status reporting is an SCM task that answers the following questions any time a change or audit occurs:
        - What happened?
        - Who did it?
        - When did it happen?
        - What else will be affected?
- Content Management
    - ★ Collection subsystem encompasses all actions required to create or acquire content, and the technical functions that are necessary to:
        - Convert content into a form that can be represented by a mark-up language (for example, HTML, XML)

        - Organize content into packages that can be displayed effectively on the client-side
    - ★ Management subsystem implements a repository that encompasses:
        - Content database - information structure to store all content objects
        - Database capabilities - functions to search for content objects, store and retrieve objects, and manage the content file structure
        - Configuration management functions - supports content object identification, version control, change management, change auditing
    - ★ Publishing subsystem - extracts content from the repository, converts it to a publishable form, and formats it so that it can be transmitted to client-side browsers
    - The publishing subsystem uses a series of templates for each type:
        - Static elements - text, graphics, media, and scripts that require no further processing are transmitted directly to the client-side
        - Publication services - function calls to specific retrieval and formatting services that personalize content (using predefined rules), perform data conversion, and build appropriate navigation links
        - External services - provide access to external corporate information infrastructure such as enterprise data or "back-room" applications
- Version Control
    - As apps and games evolve through a series of increments different versions may exist at the same time and version control process is required to avoid overwriting changes:
        - A central repository for the app or game project should be established
        - Each developer creates his own working folder
        - The clocks on all developer workstations should be synchronized
        - As new configuration objects are developed or existing objects are changed, they are imported into the central repository
        - As objects are imported or exported from the repository, an automatic, time-stamped log message is made
- Auditing and Reporting
    - Interest of agility, auditing and reporting functions are deemphasized during development of games or apps
    - As objects are checked into or out of the repository the action is recorded in a log that can be reviewed
    - Complete log report can be generated at the time needed so that all members of the team have a chronology of changes
    - An e-mail notification can be generated automatically any time an object is checked in or out of the repository

Chapter 23: Software Metrics and Analytics

April 9, 2021 9:09 AM

- Measures, Metrics, and Indicators
  - An indicator is a metric that provides insight into the product itself
  - A measure provides a quantitative indication of a product or process
  - A metric is a quantitative measure of a system's given attribute
- Attributes of Effective Metrics
  - Simple and computable
  - Empirically and intuitively persuasive
  - Consistent and objective
  - Consistent in its use of units and dimensions
  - Programming language independent
  - Effective mechanism for feedback
- Software Analytics
  - ⭐ Software analytics is a computational analysis that provides meaningful insights so that we can make better decisions
  - Key performance indicators (KPIs) are metrics that track performance and automatically keep a threshold
  - How do you know that metrics are meaningful?
  - We can make better decisions by:
    - Targeted testing
    - Targeted refactoring
    - Release planning
    - Understanding customers
    - Judging stability
    - Targeting inspection
- Requirements Model Metrics
  - Assume there are n_r requirements in a specification: $n_r = n_f + n_{nf}$
  - n_f is the number of functional requirements
  - n_{nf} is the number of nonfunctional requirements
  - Requirement specificity (lack of ambiguity): $Q_1 = n_{ui}/n_r$
  - Where n_{ui} is the number of requirements for which all reviewers had identical interpretations
  - Q_1 close to 1 is good
- Mobile Software Requirements Model Metrics
  - Number of static screen displays (N_{sp})
  - Number of dynamic screen displays (N_{dp})
  - Number of persistent data objects
  - Number of external systems interfaced
  - Number of static content objects
  - Number of dynamic content objects
  - Number of executable functions
- ⭐ Architectural Design Metrics
  - Structural complexity = $(fanout)^2$
  - Data complexity = $\frac{(io\ variables)}{(fanout+1)}$
  - System complexity = structural complexity + data complexity
  - Henry and Kafura (HK) metric: architectural complexity as a function of fan-in and fan-out

        - Complexity = length of procedure $\times (fanin \times fanout)^2$
    - Morphology metrics are a function of the number of modules and the number of interfaces between modules
        - Size = nodes + arcs
        - Arch-to-Node Ratio = arcs / nodes
        - Depth = longest path root to leaf node
        - Width = maximum number of nodes at each level
- Object-Oriented Design Metrics
    - Weighted methods per class (WMC) is the number of methods and their complexity are reasonable indicators of the amount of effort required to implement and test a class
    - Depth of the inheritance tree (DIT) is a deep class hierarchy (DIT is large) leads to greater design complexity
    - Number of children (NOC) as NOC increases, the amount of testing will also increase
    - Coupling between object classes (CBO) where High values of CBO indicate poor reusability and make testing of modifications more complicated
    - Response for a class (RFC) is the number of methods that can potentially be executed in response to a message received by an object of the class
    - Lack of cohesion in methods (LCOM) the number of methods that access one ore more of the same attribute
- User Interface Design Metrics
    - Interface metrics are ergonomic measures (memory load, typing effort, etc.)
    - Aesthetic (graphic design) metrics relies on qualitative judgment but some measures are possible
    - Content metrics focus on content complexity and on clusters of content objects that are organized into pages
    - Navigation metrics address the complexity of the navigational flow they are applicable only for static Web applications
- Source Code Metrics
    - Halstead's Software Science: a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program
        - n_1 Number of distinct operators in a program
        - n_2 Number of distinct operands that appear in a program
        - N_1 Total number of operator occurrences
        - N_2 Total number of operand occurrences
    - Program number $N = n_1 log_2 n_1 + n_2 log_2 n_2$
    - Program volume $V = N log_2(n_1 + n_2)$
    - Volume Ratio $L = \left(\frac{2}{n_1}\right) \times \left(\frac{n_2}{N_2}\right)$
- Testing Metrics
    - Program level $PL = \frac{1}{L}$
    - Effort $e = \frac{V}{PL}$
    - Lack of cohesion in methods (LCOM)
    - Percent public and protected (PAP)
    - Public access to data members (PAD)
    - Number of root classes (NOR)
    - Fan-in (FIN)
    - Number of children (NOC) and depth of inheritance tree (DIT)
- Maintenance Metrics
    - IEEE Std. 982.1-2005 software maturity index (SMI) that provides an indication of the software product stability
        - M_t = number of modules in current release
        - F_c = number of modules in current release that have been changed

        - F_a = number of modules in current release that have been added
        - F_d = number of preceding release modules deleted
        - $SMI = [M_t - (F_a + F_c + F_d)]/M_t$
    - As the SMI approaches 1.0, the product begins to stabilize
- Process and Project Metrics
    - <u>Process metrics</u> are collected across all projects over long periods of time. Their intent is to provide a set of indicators that lead to long-term software process improvement
    - Project metrics enable a software manager to:
        - Assess the status of an ongoing project
        - Track potential risks
        - Uncover problem areas before they go "critical"
        - Adjust workflow or tasks
        - Evaluate project team's ability to control quality of software work products
- Process Measurement
    - We measure efficacy of a software process indirectly - by deriving metrics based on the outcomes that can be derived from the process:
        - Measures of errors uncovered before release of the software
        - Defects delivered to and reported by end-users
        - Work products delivered (productivity)
        - Human effort expended
        - Calendar time expended
        - Schedule conformance
        - Other measures
- Process Metrics Guidelines
    - Use common sense and organizational sensitivity when interpreting metrics data
    - Provide regular feedback to the individuals and teams who collect measures and metrics
    - Don't use metrics to appraise individuals
    - Work with practitioners and teams to set clear goals and metrics that will be used to achieve them
    - Never use metrics to threaten individuals or teams
    - Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement
    - Don't obsess on a single metric to the exclusion of other important metrics
- Software Measurement
    - <u>Direct measures</u> of the software process include cost and effort applied
    - Direct measures of the product include <u>lines of code</u> (LOC) produced, execution speed, memory size, and defects reported over some set period of time
    - <u>Indirect measures</u> of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many others
    - <u>Direct measures</u> are relatively easy to collect, the quality ad functionality of software are more difficult to assess and can be measured only indirectly
- Normalized Size-Oriented Metrics
    - Errors per KLOC (<u>thousand lines of code</u>)
    - Defects per KLOC
    - $ per LOC
    - Pages of documentation per KLOC
    - Errors per person-month
    - Errors per review hour
    - $ per page of documentation
- Normalized Function-Oriented Metrics (pg.514)
    - Errors per FP (thousand lines of code)
    - Defects per PF
    - $ per FP

    - Pages of documentation per FP
    - FP per person-month
- Why Opt For Function-Oriented Metrics
    - Programming language independent
    - Used readily countable characteristics that are determined early in the software process
    - Does not "penalize" inventive (short) implementations that use fewer LOC than other more clumsy versions
    - Makes it easier to measure the impact of reusable components
- Software Quality Metrics
    - Correctness - the degree to which the software performs its required function
    - Maintainability - the degree to which a program is amenable to change (MTTC - mean time to change)
    - Integrity - the degree to which the program is impervious to outside attack
        - Integrity = $\Sigma[1 - (\text{threat} \times (1 - \text{security}))]$
        - Threat = probability specific attack occurs
        - Security = probability specific attack is repelled
    - Usability - quantifies the ease of use
- Defect Removal Efficiency (DRE)
    - DRE is a measure of filtering ability of quality assurance and control actions as they are applied throughout all process framework activities
        - DRE = $\frac{E}{E+D}$
        - E = number of errors found before delivery
        - D = number of errors found after delivery
    - The ideal value for DRE is 1 and no defects (D = 0) are found be the consumers of a work product after delivery
- Goal Driven Metrics Program
    - Identify your business goals
    - Identify what you want to know or learn
    - Identify your subgoals
    - Identify the entities and attributes related to your subgoals
    - Formalize your measurement goals
    - Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals
    - Identify the data elements that you will collect to construct the indicators
    - Identify the measures to be used, and make these definitions operational
    - identify the actions that you will take to implement the measures
    - prepare a plan for implementing the measures
- Metrics for Small Organizations
    - Time (hours or days) elapsed from the time a request is made until evaluation is complete, t_{queue}
    - Effort (person-hours) to perform the evaluation, W_{eval}
    - Time elapsed from completion of evaluation to assignment of change order to personnel, t_{eval}
    - Effort required to make the change, W_{change}
    - Time required to make the change, t_{change}
    - Errors uncovered during work to make change, E_{change}
    - Defects uncovered after change is released to the customer base, D_{change}

Chapter 24: Project Management Concepts

April 16, 2021 9:41 AM

- Management Spectrum - Four P's
  - ⭐ People - the most important element of a successful project
  - ⭐ Product - the software to be built
  - ⭐ Process - the set of framework activities and software engineering tasks to get the job done
  - ⭐ Project - all work required to make the product a reality
- Stakeholders
  - Senior managers (product owners) who define the business issues that often have a significant influence on the project
  - Project (technical) managers (Scrum masters or team leads) who must plan, motivate, organize, and coordinate the practitioners who do software work
  - Practitioners who deliver the technical skills that are necessary to engineer a product or application
  - Customers who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome
  - End-users who interact with the software once it is released for production use
- Team Leaders
  - Model the way - leaders must practice what they preach. They demonstrate commitment to team and project by shared sacrifice
  - Inspire and shared vision - motivate team members to tie their personal aspirations to team goals. Involve stakeholders early
  - Challenge the process - encourage team members to experiment and take risks by helping them generate frequent small successes while learning from their failures
  - Enable others to act - increase the team's sense of competence through sharing decision making and goal setting
  - Encourage the heart - build community spirit by celebrating shared goals and victories
- Factors Affecting Software Team Structure
  - Difficulty of the problem
  - Size of the resultant program(s) in lines of code or function points
  - Team lifetimes - time that the team will stay together
  - Degree to which the problem can be modularized
  - Required quality and reliability of the system to be built
  - Rigidity of the delivery date
  - Communication (degree of sociability) required
- Team Toxicity Factors
  - Frenzied work atmosphere team members waste energy and lose focus on work objectives
  - High frustration caused by personal, business, or technological factors causing team member friction
  - Fragmented or poorly coordinated procedures poorly defined or improperly chosen process model
  - Unclear definition of roles resulting in lack of accountability and resultant finger-pointing
  - Continuous and repeated exposure to failure leads to a loss of confidence and a lowering of morale
- Agile Teams
  - Team members must trust in one another
  - The distribution of skills must be appropriate to the problem

    - Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained
    - Team is "self-organizing"
        - An adaptive team structure
        - Planning is kept to a minimum
        - Team select its own approach constrained by business requirements and organizational standards
- Team Coordination and Communication Issues
    - Scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members
    - Uncertainty is common, resulting in continuing stream of changes that ratchets the project team
    - Interoperability - new software must communicate with existing software and conform to constraints imposed by existing systems or products
    - To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established
    - Formal communication is accomplished through writing, structured meetings, and other relatively non-interactive and impersonal communication channels
    - Informal communication is more personal and allow team members to interact with one another on a daily basis - share ideas on an ad hoc basis and ask for help as problems arise
- Software Scope
    - ★ Software project scope must be unambiguous and understandable at management and technical levels
    - ★ Context - how does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context
    - ★ Information objectives - what customer-visible data objects are produced as output from the software? What data objects are required for input?
    - ★ Function and performance - what function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?
- Problem Decomposition
    - Sometimes called partitioning or problem elaboration
    - Once scope is defined...
        - It is decomposed into constituent
        - It can decomposed into user-visible data objects
          Or
        - It can be decomposed into a set of problem classes
- Process
    - Your team must decide which process model is most appropriate for
        - The customers who have requested the product
        - The people who will do the work
        - The characteristics of the product itself
        - The project environment in which the software team works
    - Team selects the process model and defines a preliminary project plan (based set of process framework activities)
    - Once the preliminary plan is established, process decomposition begins
- Successful Project Characteristics
    - Clear and well-understood requirements accepted by all stakeholders
    - Active and continuous participation of users throughout the development process
    - A project manager with required leadership skills who is able to share project vision with the team
    - A project plan and schedule developed with stakeholder participation to achieve user goals
    - Skilled and engaged team members
    - Development team members with compatible personalities who enjoy working in a collaborative environment

- Realistic schedule and budget estimates which are monitored and maintained
- Customer needs that are understood and satisfied
- Team members who experience a high degree of job satisfaction
- A working product that reflects desired scope and quality

- W^5HH Principle
    - ★ Why is the system being developed?
    - ★ What will be done?
    - ★ When will it be done?
    - ★ Who is responsible for a function?
    - ★ Where are they located organizationally?
    - ★ How will the job be done technically and managerially?
    - ★ How much of each resource is needed?
- Critical Practices
    - Formal risk management
    - Empirical cost and schedule estimation
    - Metrics-based project management
    - Earned value tracking
    - Defect tracking against quality targets
    - People aware project management

Chapter 25: Creating a Viable Software Plan

April 23, 2021 9:10 AM

- Estimation Issues
  - Estimation of resource, cost, and schedule for a software engineering effort requires:
    - Experience
    - Access to good historical information (metrics)
    - The courage to commit to a quantitative predictions when qualitative information is all that exists
  - Estimation carries inherent risk and this risk leads to uncertainty:
    - Project complexity
    - Project size
    - Degree of structural uncertainty
- Project Planning Task Set
  - Establish project scope
  - Determine feasibility
  - Analyze risks (Chapter 26)
  - Define required resources
    - Determine required human resources
    - Define reusable software resources
    - Identify environmental resources
  - Estimate cost and effort
    - Decompose the problem
    - Develop two or more estimates using size, function points, process tasks, or use cases
    - Reconcile the estimates
  - Develop an initial project schedule (Section 25.11)
    - Establish a meaningful task set
    - Define a task network
    - Use scheduling tools to develop a time-line chart
    - Define schedule tracking mechanisms
  - Repeat steps 1 to 6 to create a detailed schedule for each prototype as the scope of each prototype is defined
- What is Scope?
  - Software scope describes
    - ★ Functions and features to be delivered to end-users fitting in any existing systems
    - ★ Data input and output
    - ★ Performance, constraints, interfaces, and reliability that bound the system
  - Scope is defined using one of two techniques:
    - ★ A narrative description of software scope is developed after communication with all stakeholders
    - ★ A set of use-cases is developed by end-users
- Project Feasibility
  - Once scope has been identified, it is reasonable to ask:
    - Can we build software to meet this scope?
    - Is the project feasible?
    - You must try to determine if the system can be created using available technology, dollars, time, and other resources
    - Consideration of business need is important too - it does no good to build high-tech system or product that no one wants

- Data Analytics and Estimation Accuracy
    - To achieve reliable cost and effort estimates several options arise:
        - Delay estimation until late in the project
        - Base estimates on similar projects that have already been completed
        - Use relatively simple decomposition techniques to generate project cost nd effort estimates (similar to divide and conquer)
        - Use one or more empirical models for software cost and effort estimation (often derived using statistical regression models)
- Problem-Based Estimation
    - LOC and FP data are used in two ways during software project estimation:

- LOC-Based Estimation
    - Average productivity for these systems is 620 LOC/pm
    - Burdened labor rate is $8000 per month
    - Cost per line of code is approximately $13
    - Based on LOC estimates and historical data:
        - Estimated project cost is $431,000
        - Estimated effort is 54 person-months
- Problem-Based Estimation
    - Begin with a bounded statement of software scope
    - Decompose the statement of scope into problem functions that can each be estimated individually
    - LOC of FP is then estimated for each function
    - Baseline productivity metrics are then applied to the appropriate estimation variable
- ★ Process-Based Estimation
    - ★ Process-based estimation begins with a delineation of software functions obtained from the project scope
    - ★ A series of framework activities are performed for each function

- Use Case Point Estimation
  - Computation of use case point takes the following into account:
    - The number of complexity of the use cases in the system
    - The number and complexity of the actors on the system
    - Various non-functional requirements not written as use cases
    - The environment in which the project will be developed
      - UCP = (UUCW + UAW) × TCF × ECF
      - UUCW - unadjusted sum of use cases
      - UAW - unadjusted sum of actor weight
      - TCF - technical complexity 13 factors
      - ECF - environment complexity 8 actors
- Agile Project Estimation
  - Each user story is considered separately for estimation purposes
  - Each user story is decomposed into the set of software engineering tasks that will be required to develop it
  - Each task is estimated separately (historic data, empirical model, experience, or planning poker)
    - Alternatively the "volume" of the user story can be estimated in LOC, FP, or use case point
  - Estimates for each task are summed to estimate the user story
  - Alternatively, the volume translated into effort using historical data
  - Effort estimates for all user stories are summed to create effort estimates for the increment
- Reconciling Estimates
  - Any estimation technique must be checked by computing at least one other estimate using a different approach
  - If you have created multiple estimates they need to be compared and reconciled
  - If both estimates show agreement, there is a good reason to believe that the estimates are reliable
  - Widely divergent estimates can often be traced to one of two causes:
    - The scope of the project is not adequately understood or has been misinterpreted by the planner
    - Productivity data used for problem-based estimation techniques in inappropriate for the application or has been misapplied
- Why Are Projects Late?

  - Unrealistic deadline established by someone outside the software team and focused on managers and practitioners on the group
  - Changing requirements not reflected in schedule changes
  - An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job
  - Predictable and/or unpredictable risks that were not considered when the project commenced
  - Technical difficulties that could not have been foreseen in advance
  - Human difficulties that could not have been foreseen in advance
  - Miscommunication among project staff that results in delays
  - Failure by management to recognize that the project is falling behind schedule and lack of action to correct the problem
- Scheduling Principles
  - Compartmentalization - the project must be compartmentalized by decomposing the product and the process
  - Interdependency - the interdependency of each compartmentalized activity or task must be determined
  - Time allocation - each task must be allocated some number of work activity or task must be determined
  - Effort validation - ensure that no more than the allocated number of people has been scheduled at any given time
  - Define responsibilities - every task that is scheduled should be assigned to a specific team member
  - Define outcomes - every task should have a defined outcome
  - Define milestones - every task or group of tasks should be associated with a project milestone
- Schedule Tracking
  - Conducting periodic project status meetings in which each team member reports progress and problems
  - Evaluating the results of all reviews conducted throughout the software engineering process
  - Determining whether formal project milestones have been accomplished by the schedule date
  - Comparing the actual start date to the planned start date for each...

Chapter 26: Risk Management

April 28, 2021 9:03 AM

- Reactive Risk Management
  - Project team reacts to risks when they occur
  - Mitigation - plan for additional resources in anticipation of fire fighting
  - Fix on failure - resources are found and applied when the risk strikes
  - Crisis management - failure does not respond to applied resources and project is in jeopardy
- Proactive Risk Management
  - Potential risks are identified, their probability and impact are assessed, and they are ranked by importance
  - Software team establishes a plan for managing risk
  - Primary objective is to avoid risk, but because not all risks can be avoided
  - Team works to develop a contingency plan that will enable it to respond in a controlled and effective manner
- Software Risk
  - Project risks threaten the project plan
  - Technical risks threaten the quality and timelines of the software to be produced
  - Business risks threaten the viability of the software to be built and often jeopardize the project or the product
  - Known risks are those that can be uncovered after careful evaluation project plan
  - Predictable risks are extrapolated from past experience
  - Unpredictable risks can and do occur, but they are extremely difficult to identify in advance
- Risk Management Principles
  - Maintain a global perspective - view software risks within the context of the system and the business problem
  - Take a forward-looking view - think about the risks that may arise in the future; establish contingency plans
  - Encourage open communication - if someone states a potential risk, don't discount it
  - Integrate - a consideration of risk must be integrated into the software process
  - Emphasize a continuous process - team must be vigilant throughout the software process, modifying identified risks as information is known and adding new ones as better insight is achieved
  - Develop a shared product vision - if all stakeholders share the same vision of the software, it's likely that better risk identification and assessment will occur
  - Encourage teamwork - the talents, skills, and knowledge of all stakeholders should be pooled
- Risk Identification
  - Product size - risks associated with overall size if the software to be built or modified
  - Business impact - risks associated with constraints imposed by management or the marketplace
  - Customer characteristics - risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
  - Process definition - risks associated with the degree to which the software process has been defined and is followed by the development organization

    - Development environment - risks associated with the availability and quality of the tools to be used to build the product
    - Technology to be built - risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system
    - Staff size and experience - risks associated with the overall technical and project experience of the software engineers who will do the work
- Risk Components
    - Performance risk - the degree of uncertainty that the product will meet its requirements and be fit for intended use
    - Cost risk - the degree of uncertainty that the product budget will be maintained
    - Support risk - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
    - Schedule risk - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time
- Risk Projection (risk estimation)
    - Risk projection attempts to rate each risk in two ways:
        - Likelihood or probability that the risk is real
        - Consequences of the problems associated with the risk,
    - There are four risk projection steps:
        - Establish a scale that reflects the perceived likelihood of a risk
        - Delineate the consequences of the risk
        - Estimate the impact of the risk on the project and the product
        - Note the overall accuracy of the risk projection so that there will be no misunderstandings
- Building Risk Table
    - Estimate probability of occurrence
    - Estimate the impact on the project on a scale of 1 to 5, where,
        - 1 = low impact on project success
        - 5 = catastrophic impact on project success
    - Sort the table by probability and impact
- Risk Impact (Exposure)
    - The overall risk exposure, RE, is determined using the following relationship:
        - RE = P x C
    - P is the probability of occurrence for a risk
    - C is the cost to the project should the risk occur
- Risk Mitigation, Monitoring, and Management (RMMM)
    - Mitigation - how can we avoid the risk?
    - Monitoring - what factors can we track that will enable us to determine if the risk is becoming more or less likely?
    - Management - what contingency plans do we have if the risk becomes a reality?

Chapter 27: A Strategy for Software Support

May 3, 2021 9:32 AM

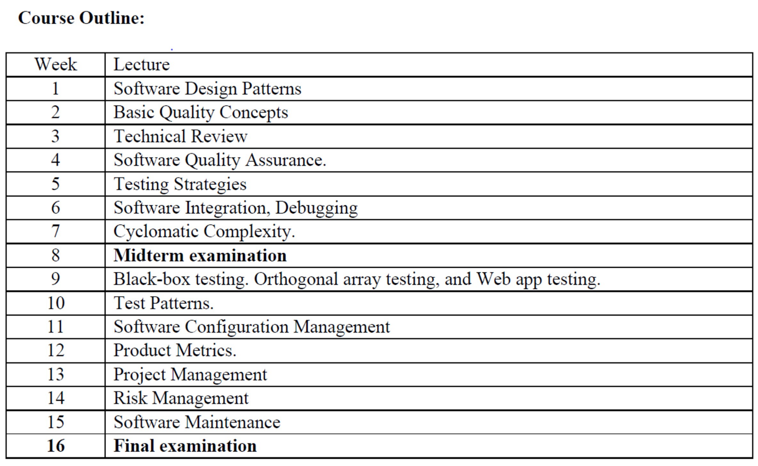
- Lehman's Laws of Software Evolution
    - Law of continuing change (1974). Software that has been implemented in a real-world computing context and will therefore evolve over time (called E-type systems) must be continually adapted else they become progressively less satisfactory.
    - Law of increasing complexity (1974). As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
    - Law of conservation of familiarity (1980). As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain knowledge of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that knowledge. Hence the average incremental growth remains invariant as the system evolves.
    - Law of continuing growth (1980). The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.
    - Law of declining quality (1996). The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
- Software Support
    - Software support can be considered an umbrella activity that includes:
        - Change management
        - Proactive risk management
        - Process management
        - Configuration management
        - Quality assurance
        - Release management
- ★ Release Management
    - ★ Release management - process that brings high-quality code from developer's workspace to the end user includes:
        - Code change integration
        - Continuous integration - the more integration, the less issues
        - Build system specifications
        - Infrastructure-as-code
        - Deployment and release
        - Retirement
- Software Supportability
    - Capability of supporting software over its whole lifetime
    - Implies satisfying all necessary requirements and also the provision of resources, support infrastructure, additional software facilities, and manpower needed to ensure software is capable of performing its functions
- Maintainable Software
    - Maintainable software exhibits effective modularity
    - It makes use of design patterns that allow ease of understanding
    - It has been constructed using well-defined coding standards, leading to understandable source code
    - It has undergone quality assurance techniques that uncovered maintenance problems before release
    - It was created by software engineers who recognize that they might not be around when changes made
    - The design and implementation of the software must "assist" the person who is making the change
- Maintenance and Support
    - Reverse engineering - process of analyzing a software system to create representations of the system at a higher level of abstraction. Often used to rediscover and redocument system design elements prior to modifying the system source code
        - Reverse Engineering to Understand Data - first reengineering task often begins by constructing UML class diagram
        - Reverse Engineering of Internal Data Structures - focuses on the definition of object classes
        - Reverse Engineering of Database Structure - reengineering one database schema into another requires an understanding of existing objects and their relationships
        - Reverse Engineering to understand user interfaces - may need to be done as part of the maintenance task (for example, adding a GUI)
    - Refactoring - process of changing a software system to improve its internal structure without altering its external behavior. Often used to improve the quality of a software product and make it easier to understand and maintain
    - Reengineering (evolution) - process of taking an existing software system and generating a new system that has the same quality as software created using modern software engineering practices
    - Inventory Analysis
        - Every software organization should have an inventory of all applications
        - The inventory can be a spreadsheet containing information that provides a detailed description (Fr example size, age, business criticality) of every active application
        - Sorting this information according to business criticality, longevity, current maintainability, and other criteria, helps to identify candidates for reengineering
    - Document Restructuring
        - Weak documentation is the trademark of many legacy systems
        - In some cases, creating documentation when none exists is simply too costly
        - In other cases, some documentation must be created, but only when changes are made
    - Code Restructuring
        - Source code is analyzed using refactoring tools
        - Poorly design code segments are redesigned
        - Violations of structured programming are noted and code is refactored (this can be done automatically)

    - Code Restructuring
        - Source code is analyzed using refactoring tools
        - Poorly design code segments are redesigned
        - Violations of structured programming are noted and code is refactored (this can be done automatically)
        - The resultant factored code is reviewed and tested to ensure that no anomalies have been introduced
        - Internal code documentation is updated
    - Data Restructuring
        - Data refactoring is a full-scale reengineering activity
        - The current data architecture is analyzed and necessary data models are defined
        - Data objects and attributes are identified, and existing data structures are reviewed for quality
        - When data structure is weak (for example, flat files are currently implemented, when a relational approach would greatly simplify processing), the data is reengineered
    - Forward Engineering
        - In an ideal world, applications would be rebuilt using an automated reengineering engine
        - Forward engineering recovers design information from existing software and uses this information to alter or reconstitute the existing system to improve its overall quality

- Data Refactoring
    - Data refactoring should be preceded by source code analysis
    - Data analysis requires the evaluation of programming language statements containing data definitions, file descriptions, I/O, and interface descriptions are evaluated (high cost on maintenance)
- Code Refactoring
    - Code refactoring is performed to yield a design that produces the same function but with higher quality that the original program
    - The objective is to take "spaghetti-bowl" code and derive a design that conforms to the quality factors defined for the product
- Architectural Refactoring
    - Architectural refactoring as one of the design trade-off options for dealing with a messy program:
        - You can struggle through modification after modification, fighting the ad hoc design and tangled source code to implement the necessary changes
        - You can attempt to understand broader inner workings of the program to make modifications more effectively
        - You can revise (redesign, recode, and test) those portions of the software that require modification, applying a meaningful software engineering approach to all revised segments - somewhat costly, high benefit
        - You can completely redo (redesign, recode, and test) the program using reengineering tools to understand the current design - most costly, most benefit
- Agile Maintenance
    - Use sprints to organize the maintenance work. You should balance the goal of keeping customers happy with the technical needs of the developers.
    - Allow for urgent customer requests to interrupt scheduled maintenance sprints, by including time for them during maintenance sprint planning.
    - Facilitate team learning by ensuring that more experienced developers are able to mentor less experienced team members even when working on their own discrete tasks.
    - Allow multiple team members to accept customer requests as they arise and coordinate their processing with the other maintenance team members.
    - Balance the use of written documentation with face-to-face communication to ensure planning meeting time is used wisely.
    - Write informal use cases to supplement the other documentation being used for communications with stakeholders.
    - Have developers test each other's work (both defect repairs and new feature implementations). This allows for shared learning and improves the feelings of product ownership by the team members.
    - Make sure developers are empowered to share knowledge with one another. This can motivate people to improve the skills and knowledge (allows developers to learn new things, improves their professional skills, and distributes tasks more evenly).
    - Keep planning meetings short, frequent, and focused.

Finals Notes (Syllabus Outline)

April 30, 2021 3:36 PM

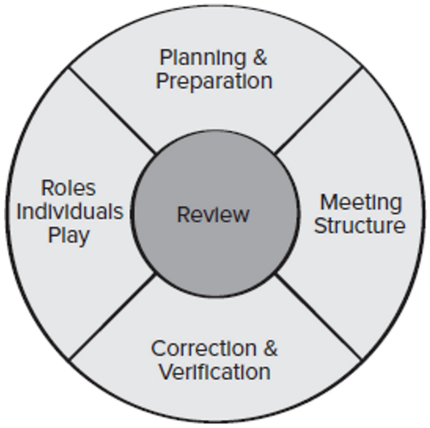


Course Outline:

Week	Lecture
1	Software Design Patterns
2	Basic Quality Concepts
3	Technical Review
4	Software Quality Assurance.
5	Testing Strategies
6	Software Integration, Debugging
7	Cyclomatic Complexity.
8	**Midterm examination**
9	Black-box testing. Orthogonal array testing, and Web app testing.
10	Test Patterns.
11	Software Configuration Management
12	Product Metrics.
13	Project Management
14	Risk Management
15	Software Maintenance
16	**Final examination**

- Design Patterns
  - ★ A design pattern can be thought of as a three-part rule about a context, problem, and solution
  - Context allows the reader to understand the environment of the problem and the appropriate solution
  - ★ We use design patterns so that we don't have to "re-invent the wheel"
- Frameworks
  - Patterns themselves may not be sufficient to develop a complete design
  - In cases it may be necessary to provide a implementation-specific skeletal infrastructure
  - ★ You can select a reusable architecture that provides the generic structure and behavior for a family of software
- Pattern based design
  - ★ A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system
- Anti-patterns
  - ★ Anti-patterns describe commonly used solutions to design problems that have a negative affect on the software
- Selected anti-patterns pg.304
  - ★ Blob = single class with a large number of attributes, operators or both
  - Stovepipe system = a barley maintainable assemblage of ill-related components
  - ★ Boat anchor = retaining a part of the system that no longer has any use
  - Spaghetti code = program whose structure is barley comprehensible, especially because of misuse code constraints
  - Copy and paste programs
  - ★ Silver bullet
  - Programming by permutation
- Views
  - ★ Transcendental view - quality is something that you recognize, but cannot explicitly define
  - ★ User's view - quality as an end user's specific goals
  - ★ Manufacturer's view - defines quality in terms of the original specification of the product
  - ★ Product view - quality can be tied to inherent characteristics
  - ★ Value-based view - quality based on how much a customer is willing to pay for the product
- Software quality
  - ★ An effective software process defined in a manner that creates a useful product that provides measurable value for those who produce it and those who use it
  - An effective software process establishes infrastructure that supports building a high-quality software product
  - Chaos - determines poor quality
  - Structure, analysis, change management, technical reviews - determines high quality
- ★ Quality control - series of inspections, reviews, and tests used to ensure conformance to specifications
- ★ Quality assurance - consists of auditing and reporting to provide management with the proper data
- Reviews
  - ★ meetings conducted by technical people for technical people
  - A technical assessment of a work product
  - ★ A software quality assurance mechanism
  - A training ground
- Metrics - are measures
  - Effort, E - in-person hours

-
    - Effort, E - in-person hours
    - Preparation effort, E_p - the effort required to review a work product prior to the review meeting
    - Assessment effort, E_a - the effort that is expending during the actual review
    - Rework effort, E_r - the effort that is dedicated to the correction of those errors uncovered during the review
    - ★ Work Product Size, WPS - lines of code or number of pages
    - Minor errors found, Err_{minor} - the number of errors found that can be categorized as minor
    - Major errors found, Err_{major} - the number of errors found that can be categorized as major
    - The total review effort
        - $E_{review} = E_p + E_a + E_r$
        - $E_{tot} = Err_{minor} + Err_{major}$
    - Defect density represents the errors found per unit of WPS
        - Defect density = $\frac{Err_{tot}}{WPS}$
    - ★ Effort saved per error = $E_{testing} - E_{reviews}$



- Informal Reviews - the benefit is immediate discovery of errors and better work product quality
    - A simple desk check with a colleague
    - A casual meeting (2 or more people)
    - The review-oriented aspects of pair programming
        - ★ pair programming - encourages continuous review as a work product is created
- Formal Technical Reviews, FTRs, Code Inspections, Code Walkthroughs - 5 key objectives
    - ★ To uncover errors in functions, logic, implementation in any representation of the software
    - ★ To verify that the software meets its requirements
    - ★ To ensure that the software adheres to a standard
    - ★ To achieve software that is developed in a uniform manner
    - ★ To make projects more manageable
- Review Players
    - ★ Producer - the individual who has developed the work product
    - ★ Review leader - evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advanced preparation and facilitates the meeting discussion
    - ★ Reviewers - expected to spend between one and two hours reviewing the product, making notes, and becoming familiar with the work
        - ★ Come prepared to evaluate
    - ★ Recorder - records (in writing) all important issues
- Review Outcome - a decision must be made to:
    - ★ Accept the product without modification
    - ★ Reject the product due to severe errors
    - ★ Accept the product conditionally (correct errors and no additional review)
- SQA Goals
    - ★ Requirements Quality
        - The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow
        - Traceability - the number of requirements not traceable to code
        - Model Clarity
    - ★ Design Quality
        - Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and conforms to requirements
        - Architectural integrity
        - Interface complexity
    - ★ Code Quality
        - Source code and related work products must conform to standards and exhibit maintainability
        - Complexity
    - ★ Quality control effectiveness - QC effectiveness
        - Apply limited resources in the most effective way possible
        - Resource allocation - staff hour percentage per activity
    - ★ Pareto principle - 80% of defects can be traced to 20% of all possible causes
    - Move to correct the vital few 20% of problems
- Six Sigma 6σ - (standard of deviation) for Software Engineering
    - ★ Six standard deviations - 3.4 defects per million occurrences - implying an extremely high quality standard
    - Defines three core steps and two follow-up steps:
        - ★ Define customer requirements and deliverables
        - ★ Measure the quality performance
        - ★ Analyze and determine the vital few causes
    - For the same process:
        - ★ Improve the process by eliminating root causes
        - ★ Control the process to ensure future work does not reintroduce the causes
    - For a new process being developed:

        - ★ Design the process to avoid root causes and meet customer requirements
        - ★ Verify that the process will avoid defects and meet customer requirements
- Software Reliability - probability of failure-free operation over a period of time
    - MTTF = Mean-time-to-failure
    - MTTR = Mean-time-to-repair
    - MTBF = Mean-time-between-failure
    - ★ MTBF = MTTF + MTTR
- Software Availability - probability of requirements being met over a period of time
    - ★ Availability = [MTTF/(MTTF + MTTR)] x 100%
- AI and Reliability Models - AI always requires statistics
    - ★ Bayesian inference uses Bayes' theorem to update the probability for a hypothesis as more evidence becomes available
    - ★ Regression model - used to estimate where and what type of defects might occur in future prototypes
    - Genetic algorithms - used to grow reliability models based on historic data
- Verification and Validation
    - ★ Verification ensures that software correctly implements a function
    - ★ Validation ensures that software is traceable to customer requirements
- Organizing for Testing
    - When software architecture is complete then the independent test group is involved
    - ★ The independent test group (ITG) is there to prevent the builder from testing their own product
- Role of Scaffolding
    - ★ Scaffolding is requirefd to create a testing framework
    - ★ A driver must be developed for each unit test
    - ★ A driver is a "main program" that accepts testcase data
    - ★ Stubs (dummy subprogram) replace modules invoked by the component to be tested
- ★ By collecting metrics during testing and making use of existing statistical models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?"
- Test Planning
    - Specify quantifiable measures of the requirements before testing commences
    - State testing objectives explicitly
    - Develop a testing plan that emphasizes "rapid cycle testing"
    - ★ Rapid cycle testing tests at the end of every sprint
- Test Case Design
    - Design unit test cases before you develop code for a component to ensure that code will pass the tests
    - Test cases are designed to cover the following areas:
        - ★ The module interface is tested to ensure that information properly flows into and out of the program unit.
        - ★ Local data structures are examined to ensure that stored data maintains its integrity during execution
        - ★ Independent paths through control structures are exercised to ensure all statements are executed at least once
        - ★ Boundary conditions are tested to ensure module operates properly at boundaries established to limit or restrict processing
        - ★ All error-handling paths are tested
- Traceability
    - ★ To ensure that the testing process is auditable, each test case needs to be traceable back to specific functional or non-functional requirements to anti-requirements
    - ★ Regression testing requires retesting of selected components that may be affected by changes
- Basic Path Testing
    - Determines the number of independent paths in the program by computing Cyclomatic Complexity:
        - ★ The number of regions of the flow graph corresponds to the cyclomatic complexity (book example has 4)
        - ★ Cyclomatic complexity $V(G)$ for a flow graph G is defined as (E = Edge, N = Node, P = Predicate Node)
            - $V(G) = E - N + 2$
            - $V(G) = P + 1$
        - ★ An independent path is any path through the program that introduces at least one new set of processing statements or a new condition (book examples)
            - Path 1: 1-11
            - Path 2: 1-2,3-4,5-10-1-11
            - Path 3: 1-2-3-6-8-9-10-1-11
            - Path 4: 1-2-3-6-7-9-10-1-11
- Control Structure Testing
    - ★ Condition testing is a test-case design method that exercises logical conditions contained in a program module
    - ★ Data flow testing selects test paths according to the locations of definitions and uses variables in the program

- ★ Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs

- Interface Testing
    - ★ Interface testing is used to check that a program component accepts information passed to it in the proper order and data types and returns information in proper order and data format
    - Components are not stand-alone programs testing interfaces requires the use of stubs and drivers
    - Stubs and drivers sometimes incorporate test cases to be passed to the component or accessed by the component
- Behavior Testing
    - ★ A state diagram can be used to help derive a sequence of tests that will exercise dynamic behavior of the class
- Boundary Value Analysis (BVA)
    - ★ Boundary value analysis leads to a selection of test cases that exercise bounding values
    - Guidelines for BVA:
        - If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b just above and just below a and b
        - If an input condition specifies a number of values, test cases should be developed that exercise the min and max numbers as well as values just above and below min and max
        - Apply guidelines 1 and 2 to output conditions
        - If internal program data structures have prescribed boundaries be certain to design a test case to exercise the data structure at its boundary
- Integration Testing
    - ★ Integration testing is a systematic technique for constructing the software architecture while conducting tests to uncover errors associated with interfacing
    - ★ The objective is to take unit-tested components and build a program structure that matches the design
    - ★ In the big bang approach, all components are combined at once and the entire program is tested as a whole. Chaos usually results!
    - ★ In incremental integration a program is constructed and tested in small increments, making errors easier to isolate and correct. Far more cost-effective!
- Top-Down Integration
    - ★ Top-down integration testing is an incremental approach to construction of the software architecture
    - Modules are integrated by moving downward through the control hierarchy, beginning with the control module (main program)
- Top-Down Integration Testing
    - The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module
- Bottom-Up Integration Testing
    - ★ Bottom-up integration testing, begins construction and testing with atomic modules components at the lowest levels in the program structure
        - Low-level components are combined into clusters that perform a specific software subfunction
        - A driver is written to coordinate test-case input and output
        - The cluster is tested
        - Drivers are removed and clusters are combined, moving upward in the program structure
- Continuous Integration
    - ★ Continuous integration is the practice of merging components into the evolving software increment at least once a day
    - This is a common practice for teams following agile such as XP or DevOps. Integration testing must be quickly implemented as the program is being built
    - ★ Smoke testing is an integration testing approach that can be used when software is developed by an agile team using short increment build times
- Smoke Testing Integration
    - Software components that have been translated into code are integrated into a build. That includes all data files, libraries, reusable modules, and components required to implement one or more product functions
    - A series of tests is designed to expose "show-stopper" errors that will keep the build from properly performing its function cause the project to fall behind
    - The build is integrated with other builds, and the entire product is smoke tested daily
- Smoke Testing Advantages
    - ★ Integration risk is minimized, since smoke tests are run daily
    - ★ Quality of the end product is improved, functional and architectural problems are uncovered early
    - ★ Error diagnosis and correction are simplified, errors are most likely in the new build
    - ★ Progress is easier to assess, each day more of the final product is complete
    - Smoke testing resembles regression testing by ensuring newly added components do not interfere with the behaviors of existing components
- Integration Testing Work Products
    - ★ An overall plan for integration of the software and a description of specific tests is documented in a test specification
- 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

- OO Integration Testing
    - ★ Thread-based testing, integrates the set of classes required to respond to one input or event for the system
- OO Testing - Fault-based Test Case Design
    - ★ The object of fault-based testing is to design tests that have a high likelihood of uncovering plausible faults
- Fault-based OO Integration Testing
    - ★ Fault-based integration testing looks for plausible faults in operation calls or message connection:
        - Unexpected results
        - Wrong operation/message used
        - Incorrect invocation
    - ★ Scenario-based testing uncovers interaction errors
    - Scenario-based testing tends to exercise multiple subsystems in a single test
- OO Testing - Random Test Case Design
    - ★ For each client class, use the list of class operations to generate a series of random test sequences
- Validation Testing
    - ★ Validation testing tries to uncover errors, but the focus is at the requirements level - on user visible actions and user-recognizable output from the system
    - Validation testing begins at the culmination of integration testing, the software is completely assembled as a package and errors have been corrected

Midterms---

- Black Box Testing
    - ★ Comparison Testing
        - Used only in situations in which the reliability of software is absolutely critical
            - Separate software engineering teams develop independent versions of an application using the same specification
            - Each version can be tested with the same test data to ensure that all provide identical output
            - Then all versions are executed in parallel with real-time comparison results to ensure consistency
    - ★ Orthogonal Array Testing
        - Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded
- ★ WebApp - Content Testing
    - Content testing has three important objectives:
        - To uncover syntactic errors in text-based documents, graphical representations, and other media
        - To uncover semantic errors in any content object presented as navigation errors
        - To find errors in the organization or structure of the content that is presented to the end-user
- ★ WebApp - Interface Testing
    - Interface features tested to ensure design rules, aesthetics, and content is available to user without error
    - Individual interface mechanisms are tested in a manner that is analogous to unit testing
    - Each interface mechanism is tested within the context of a use-case or NSU (Network Semantic Units) for a specific user category
    - Complete interface is tested against selected use-cases and NSUs to uncover errors in interface semantics
    - The interface is tested within a variety of environments to ensure that it will be compatible
- ★ WebApp - Navigation Testing
    - The job of navigation testing is:
        - To ensure that the mechanisms that allow the WebApp user to travel through the WebApp are all functional
        - To validate that each NSU (navigation semantic unit) can be achieved by the appropriate user category
- ★ Security Testing
    - Designed to probe for vulnerabilities of the client-side environment, the network communications that occur as data are passed from client to server and back again, and server-side environments
    - On the client-side, vulnerabilities can often be traced to pre-existing bugs in browsers, e-mail programs, or communication software
    - On the server-side, vulnerabilities include denial-of-service attacks and malicious scripts that can be passed along to the client-side or used to disable server operations
- ★ Load Testing
    - The intent is to determine how the WebApp and its server-side environment will respond to various load conditions
        - N, number of concurrent users
        - T, number of on-line transactions per unit of time
        - D, data load processed by server per transaction
    - Overall throughput, P, Is computed in the following manner:

        - P = N x T x D
- Creating Weighted Device Platform Matrix
    - To mirror real-world conditions, the demographic characteristics of testers should match those of targeted users, as well as those of their device
    - ★ A weighted device platform matrix (WDPM) helps to ensure that test coverage includes combination of mobile device and context variables
        - List the important operating system variants as the matrix column labels
        - List the targeted devices as the matrix row labels
        - Assign ranking to indicate the relative importance of each operating system and each device
        - Compute the product of each pair of rankings and enter each
- Configuration Management System Elements
    - ★ Component elements - a set of tools coupled within a file management system that enables management of each configuration item
    - ★ Process elements - a collection of procedures and tasks that define an effective approach to change management
    - ★ Construction elements - tools that automate the construction of software to ensure proper components are used
    - ★ Human elements - used to implement SCM (Software Configuration Management)
- Baselines
    - ★ The IEEE defines a baseline as:
        - A specification or product that has been formally reviewed and agreed upon, that thereafter serves as basis for further development of the product, and that can be changed only through formal change control programs
- SCM Repository Features
    - ★ Versioning - saves versions to manage product releases and allow developers to go back to previous versions
    - ★ Dependency tracking and change management - manages a wide variety of relationships among the data elements stored in it
    - ★ Requirements tracing - provides the ability to track all design and construction components and deliverables resulting from a specific requirement specification
    - ★ Configuration management - tracks series of configurations representing specific project milestones or production releases and provides version management
    - ★ Audit trails - establishes additional information about when, why, and by whom changes are made
- SCM Best Practices
    - Keeping the number of code variants small
    - Test early and often
    - Integrate early and often
    - Use tools to automate testing, building, and code integration
- Continuous Integration Advantages
    - Accelerated feedback - notifying developers immediately when integration fails, allows fixes when the number of changes is small
    - Increased quality - building and integrating software whenever necessary provides confidence into the quality of the product
    - Reduced risk - integrating components early avoids a long integration phase, design failures are discovered and fixed early
    - Improved reporting - providing additional information allows for accurate configuration status accounting
- Change Management Objectives
    - ★ Software change management process defines a series of tasks that have four primary objectives:
        - To identify all items that collectively define the software configurations
        - To manage changes to one or more of these items
        - To facilitate the construction of different versions of an application
        - To ensure that software quality is maintained as the configuration evolves over time
- Impact Management
    - ★ A web of software work product interdependencies must be considered every time a change is made
        - ★ Impact management is accomplished with three actions:
            - An impact network identifies the stakeholders who might effect or be affected by changes that are made to the software based on its architectural documents
            - Forward impact management assesses the impact network and then informs members of the impact of those changes
            - Backward impact management examines changes that are made by other team members and

their impact on your work and incorporates mechanisms to mitigate the impact

- Software Configuration Audit
    - ★ Software configuration audit complements a technical review by asking and answering the following questions:
        - Has the change specified in the ECO (Engineering Change Order) been made? Have any additional modifications been incorporated?
        - Has a technical review been conducted to assess technical correctness?
        - Has the software process been followed, and have software engineering standards been properly applied?
        - Has the change been "highlighted" in the SCI (Software Configuration Item)? Do the attributes of the configuration object reflect change?
        - Have SCM procedures for noting the change, recording it, and reporting it been followed?
        - Have all related SCI been properly updated?
- Content Management
    - ★ Collection subsystem encompasses all actions required to create or acquire content, and the technical functions that are necessary to:
        - Convert content into a form that can be represented by a mark-up language (for example, HTML, XML)
        - Organize content into packages that can be displayed effectively on the client-side
    - ★ Management subsystem implements a repository that encompasses:
        - Content database - information structure to store all content objects
        - Database capabilities - functions to search for content objects, store and retrieve objects, and manage the content file structure
        - Configuration management functions - supports content object identification, version control, change management, change auditing
    - ★ Publishing subsystem - extracts content from the repository, converts it to a publishable form, and formats it so that it can be transmitted to client-side browsers
        - The publishing subsystem uses a series of templates for each type:
            - Static elements - text, graphics, media, and scripts that require no further processing are transmitted directly to the client-side
            - Publication services - function calls to specific retrieval and formatting services that personalize content (using predefined rules), perform data conversion, and build appropriate navigation links
            - External services - provide access to external corporate information infrastructure such as enterprise data or "back-room" applications
- Software Analytics
    - ★ Software analytics is a computational analysis that provides meaningful insights so that we can make better decisions
    - Key performance indicators (KPIs) are metrics that track performance and automatically keep a threshold
- ★ Architectural Design Metrics
    - Structural complexity = $fanout^2$
    - Data complexity = $\frac{(io\ variables)}{(fanout+1)}$
    - System complexity = structural complexity + data complexity
    - Henry and Kafura (HK) metric: architectural complexity as a function of fan-in and fan-out
        - Complexity = length of procedure $\times (fanin\ \times fanout)^2$
    - Morphology metrics are a function of the number of modules and the number of interfaces between modules
        - Size = nodes + arcs
        - Arch-to-Node Ratio = arcs / nodes
        - Depth = longest path root to leaf node
        - Width = maximum number of nodes at each level
- ★ Source Code Metrics
    - Halstead's Software Science: a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program
        - n_1 Number of distinct operators in a program
        - n_2 Number of distinct operands that appear in a program
        - N_1 Total number of operator occurrences
        - N_2 Total number of operand occurrences
    - Program number $N = n_1 log_2 n_1 + n_2 log_2 n_2$
    - Program volume $V = N log_2(n_1 + n_2)$
    - Volume Ratio $L = \left(\frac{2}{n_1}\right) \times \left(\frac{n_2}{N_2}\right)$
- Testing Metrics
    - Program level $PL = \frac{1}{L}$
    - Effort $e = \frac{V}{PL}$
    - Lack of cohesion in methods (LCOM)
    - Percent public and protected (PAP)

    - Public access to data members (PAD)
    - Number of root classes (NOR)
    - Fan-in (FIN)
    - Number of children (NOC) and depth of inheritance tree (DIT)
- ★ Defect Removal Efficiency (DRE)
    - DRE is a measure of filtering ability of quality assurance and control actions as they are applied throughout all process framework activities
        - DRE = $\frac{E}{E+D}$
        - E = number of errors found before delivery
        - D = number of errors found after delivery
    - The ideal value for DRE is 1 and no defects (D = 0) are found be the consumers of a work product after delivery
- Management Spectrum - Four P's
    - ★ People - the most important element of a successful project
    - ★ Product - the software to be built
    - ★ Process - the set of framework activities and software engineering tasks to get the job done
    - ★ Project - all work required to make the product a reality
- Stakeholders
    - Senior managers (product owners) who define the business issues that often have a significant influence on the project
    - Project (technical) managers (Scrum masters or team leads) who must plan, motivate, organize, and coordinate the practitioners who do software work
    - Practitioners who deliver the technical skills that are necessary to engineer a product or application
    - Customers who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome
    - End-users who interact with the software once it is released for production use
- Software Scope
    - ★ Software project scope must be unambiguous and understandable at management and technical levels
    - ★ Context - how does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context
    - ★ Information objectives - what customer-visible data objects are produced as output from the software? What data objects are required for input?
    - ★ Function and performance - what function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?
- W^5HH Principle
    - ★ Why is the system being developed?
    - ★ What will be done?
    - ★ When will it be done?
    - ★ Who is responsible for a function?
    - ★ Where are they located organizationally?
    - ★ How will the job be done technically and managerially?
    - ★ How much of each resource is needed?
- What is Scope?
    - Software scope describes
        - ★ Functions and features to be delivered to end-users fitting in any existing systems
        - ★ Data input and output
        - ★ Performance, constraints, interfaces, and reliability that bound the system
    - Scope is defined using one of two techniques:
        - ★ A narrative description of software scope is developed after communication with all stakeholders
        - ★ A set of use-cases is developed by end-users
- ★ Process-Based Estimation
    - ★ Process-based estimation begins with a delineation of software functions obtained from the project scope
    - ★ A series of framework activities are performed for each function
- Use Case Point Estimation
    - Computation of use case point takes the following into account:
        - The number of complexity of the use cases in the system
        - The number and complexity of the actors on the system
        - Various non-functional requirements not written as use cases
        - The environment in which the project will be developed
            - ★ UCP = (UUCW + UAW) × TCF × ECF
            - UUCW - unadjusted sum of use cases
            - UAW - unadjusted sum of actor weight
            - TCF - technical complexity 13 factors
            - ECF - environment complexity 8 actors
- Risk Impact (Exposure)
    - The overall risk exposure, RE, is determined using the following relationship:

        - RE = P x C
    - P is the probability of occurrence for a risk
    - C is the cost to the project should the risk occur
- Risk Mitigation, Monitoring, and Management (RMMM)
    - ★ Mitigation - how can we avoid the risk?
    - ★ Monitoring - what factors can we track that will enable us to determine if the risk is becoming more or less likely?
    - ★ Management - what contingency plans do we have if the risk becomes a reality?
- Release Management
    - ★ Release management - process that brings high-quality code from developer's workspace to the end user includes:
        - Code change integration
        - Continuous integration
        - Build system specifications
        - Infrastructure-as-code
        - Deployment and release
        - Retirement
- Maintenance and Support
    - Reverse engineering - process of analyzing a software system to create representations of the system at a higher level of abstraction. Often used to rediscover and redocument system design elements prior to modifying the system source code
    - Refactoring - process of changing a software system to improve its internal structure without altering its external behavior. Often used to improve the quality of a software product and make it easier to understand and maintain
    - Reengineering (evolution) - process of taking an existing software system and generating a new system that has the same quality as software created using modern software engineering practices
    - Inventory Analysis
        - Every software organization should have an inventory of all applications
        - The inventory can be a spreadsheet containing information that provides a detailed description (Fr example size, age, business criticality) of every active application
        - Sorting this information according to business criticality, longevity, current maintainability, and other criteria, helps to identify candidates for reengineering
    - Document Restructuring
        - Weak documentation is the trademark of many legacy systems
        - In some cases, creating documentation when none exists is simply too costly
        - In other cases, some documentation must be created, but only when changes are made
    - Code Restructuring
        - Source code is analyzed using refactoring tools
        - Poorly design code segments are redesigned
        - Violations of structured programming are noted and code is refactored (this can be done automatically)
        - The resultant factored code is reviewed and tested to ensure that no anomalies have been introduced
        - Internal code documentation is updated
    - Data Restructuring
        - Data refactoring is a full-scale reengineering activity
        - The current data architecture is analyzed and necessary data models are defined
        - Data objects and attributes are identified, and existing data structures are reviewed for quality
        - When data structure is weak (for example, flat files are currently implemented, when a relational approach would greatly simplify processing), the data is reengineered
    - Forward Engineering
        - In an ideal world, applications would be rebuilt using an automated reengineering engine
        - Forward engineering recovers design information from existing software and uses this information to alter or reconstitute the existing system to improve its overall quality
- Data Refactoring
    - Data refactoring should be preceded by source code analysis
    - Data analysis requires the evaluation of programming language statements containing data definitions, file descriptions, I/O, and interface descriptions are evaluated (high cost on maintenance)
- Code Refactoring
    - Code refactoring is performed to yield a design that produces the same function but with higher quality that the original program
    - The objective is to take "spaghetti-bowl" code and derive a design that conforms to the quality factors defined for the product
- Architectural Refactoring
    - Architectural refactoring as one of the design trade-off options for dealing with a messy program:
        - You can struggle through modification after modification, fighting the ad hoc design and tangled source code to implement the necessary changes
        - You can attempt to understand broader inner workings of the program to make modifications more

effectively
        - You can revise (redesign, recode, and test) those portions of the software that require modification, applying a meaningful software engineering approach to all revised segments - somewhat costly, high benefit
        - You can completely redo (redesign, recode, and test) the program using reengineering tools to understand the current design - most costly, most benefit

- Cost of Support
    - Nine parameters are defined:

P1	=	current annual maintenance cost for an application
P2	=	current annual operations cost for an application
P3	=	current annual business value of an application
P4	=	predicted annual maintenance cost after reengineering
P5	=	predicted annual operations cost after reengineering
P6	=	predicted annual business value after reengineering
P7	=	estimated reengineering costs
P8	=	estimated reengineering calendar time
P9	=	reengineering risk factor (P9 = 1.0 is nominal)
L	=	expected life of the system

    - The cost associated with continuing maintenance of a candidate application can be defined as:
        - $C_{maint} = [P3 - (P1 + P2)] \times L$
    - The costs associated with reengineering are defined using the following relationship:
        - $C_{reeng} = P6 - (P4 + P5) \times (L - P8) - (P7 \times P9)$
    - Using the costs, the overall benefit of reengineering can be computed as:
        - Cost benefit = $C_{reeng} - C_{maint}$