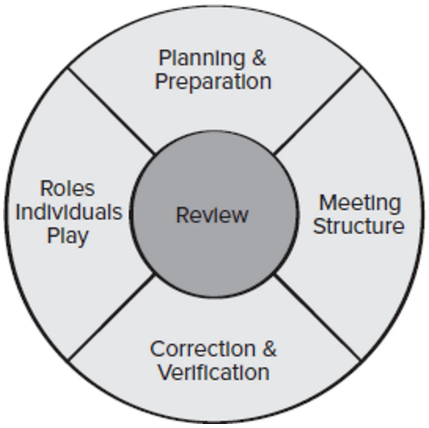
Finals Notes (Syllabus Outline)

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

TABLE 21.1 Weighted device platform matrix

		OS1	OS2	OS3
	Ranking	3	4	7
Device1	7	N/A	28	49
Device2	3	9	N/A	N/A
Device3	4	12	N/A	N/A
Device4	9	N/A	36	63

- 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 × $(fanin \times fanout)^2$
  - 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 $L = \left(\frac{2}{n_1}\right)\left(\frac{n_2}{N_2}\right)$
- 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)] × L
    - The costs associated with reengineering are defined using the following relationship:
        - C_{reeng} = P6 − (P4 + P5) × (L − P8) − (P7 × P9)
    - Using the costs, the overall benefit of reengineering can be computed as:
        - Cost benefit = $C_{reeng} - C_{maint}$