**CIS-375 – Software Engineering I - SUMMER 22 NOTES – DEMETRIUS JOHNSON**

# CIS 375 – SUMER FULL– with Dr. Abou Nassir-Mouhammad

# Lecture 1 notes – 5-10-22 – What is software and software engineering?

* What is software?
  + Remember from CIS-285 software engineering tools class (which is exactly like this course and uses same course book); software engineering is one of the last engineering disciplines to come about; so software engineering is derived from the engineering principles of all the other disciplines. And remember, software engineering is largely an entrepreneurial undertaking.
  + Software came bout to abstract hardware and control it, and make it more readily understandable by more people (without it taking so much time to learn and write on the lower, more complex levels).
* Remember software does not “wear out” as sense of time in the Law of Thermodynamics and entropy sense, but rather it, but in a sense of time of the evolution of technology so as to make older software become obsolete.
  + So, we want to make software to be flexible, expandible, and adaptable over time – this is what constitutes integrity and durability of a piece of software engineering.
* Need to be aware of side effects in code (when one part of code negatively affects another part due to some failure or bug).
* Remember, releasing software is like releasing a new model of a car; over the years a given model each year gets better and better as they work out the bug – but the advantage of software is that each version can be rapidly changed (but a set of cars already manufactured is a lot harder to change, so you usually can only make the next batch/version better).
* Embedded software (software that controls the hardware/microprocessors/chips inside of smart devices, cars, etc.).
* Software products v software projects
  + Products: facebook, Microsoft office, video games, etc.
  + Projects: custom software made internally for a given company/government/organization.
  + Remember: software engineers who make software as a product are visionaries – entrepreneurs.
* Software engineers (just like entrepreneurs), need to first understand what the customer (could be internally in the company, or a customer product, etc.) wants/needs before they continue to develop the project/product – so they need to gather that in order to then move to find the requirements and determine the best path/if it is viable or even profitable.
* Many great products started as projects (facebook, Microsoft DOS, etc.).
  + Key to a good product: solves a problem.
* Need the business acumen and software engineering discipline (for example, you may come up with a great idea, but never knew the idea already has been manifested by some other person/company that you could either work off of and save time, or realize that it may not be a good venture).
  + You don’t want to repeat the same mistakes/solve problems in the same way or in a way that is not as good as what is already out there (that you may not be aware of).
* Software execution models: stand-alone (download and install and run solely from local machine), hybrid (video game/cloud), as a service (cloud).
* Product vision: ask what, who, why?
  + Moore’s vision template: FOR, WHO, PRODUCT NAME, THAT, UNLIKE, OUR PRODUCT.
  + Need small, catchy names, and use words such as “innovative”.
* Software Product Management
  + Manage the software production: communication, and knowing how to mitigate issues, someone who is calm and collected under pressure.
  + Need to have some idea of how current technology works so you can put the right team together and to help manage everything.
* The Grand Challenge
  + Trend: software continues to become more widespread and intertwined and collaborated and more complex; how do we manage this growing complexity and domain?
    - We especially need better software building models, and better software tools.
* Model-Driven Development
  + For example: domain-specific development; if you need to build an engine, then perhaps there is some language that is specific to engines to make building/modeling engines easier.
* Test-Driven Development
  + Test as part of the development.

# Lecture 2 notes – 5-12-22 – The Software Process

* Software engineering
  + Remember, just like entrepreneurs, software engineers (and engineers in general) must understand the problem before developing a software solution.
  + Design = make decisions.
    - We try to make the best performance decisions.
  + Make maintainable, high quality software – if it is done well, then customers will keep “feeding” it, meaning they want you to keep developing it to do more and keep up with current trends.
  + IEEE Definition: application of engineering to software.
* Framework Activities for the software engineering process
  + **The Waterfall Model** (very rigid model; cannot be easily changed and adapted, especially later in the planning stage and especially in the coding stage): Communicate, plan, model (pseudo code), construct, deploy.
    - If you understand how to deploy, then likely you have a deep, broad and also detailed understanding of how technology works and fit together.
  + We have to think a lot before construction (coding). Understand problem 🡪Gather requirements 🡪 think a lot (plan extensively) 🡪 code. The actual coding step should be as smooth as possible by excellent understanding of problem and planning.
  + Money does not come easy; organization have to plan it – how will they finance a project?
  + You must test to make sure software is bug-free (including in production environment); that is the only way to make sure your program really works (but likely, there will always be some bug depending on how complex the software is).
    - Roll back is inevitable during testing process in production environment.
  + For other engineering processes, it is very costly to make a design mistake due to not thinking through the process enough. Software is more adaptable and able to change, but still it can be costly if a very complex project has some major flaw that was not well thought out.
  + Customers will get nervous if a lot of time and money is being spent and no product is getting produced or making good progress.
    - Sometimes, you may over document versus actually getting something done. So, based on the complexity of the project, make sure not to over document. Need to have some rigidity in your process, because if it is too flexible an change too often, it may not get done in a timely enough manner.
* The V-Model
  + Similar to waterfall model, but a little more detailed with more steps.
* The Incremental Model
  + An iterative process – implement small increments, test/share, then repeat that step and grow the software project. Don’t wait for all planning to get done (say a year later) then start all the coding at once. Do it incrementally alongside planning.
* Evolutionary Models: Prototyping
  + Greatly reduces the emphasis on documentation in order to save time and be more flexible (especially in the short term/ here and now).
  + Prototype is not a finish product but gives customers and developers a quick idea of how a software could look/work.
* Agile Software Engineering
  + More of an evolutionary/incremental model approach to push out software quickly.
  + Good for getting a jump on the market!
  + Greatly reduce overhead of documentation to increase development speed.
  + Customers don’t care about paying for you effort as much as the actual product you deliver to them.
  + Overhead = any effort spent that is not the actual construction of the product – essentially, any thinking/planning/documentation processes.
  + The sooner the customer sees the product making progress or gaining functionality/working, sometimes this is a better approach for the customer.
    - Financial overseers of companies are usually very cautious; they don’t like to spend money and not see any returns. It makes them uncomfortable. Seeing some development of the product is reassuring.
  + Agile = quick, and dynamic.
  + Follow incremental development.
    - Think about the product as a bunch of features, and involve customers as you get each feature working and iteratively tying them together as you move along with each increment.
    - Most important features are implemented first; decided by the developer/company and the client.
      * This is much more rapid and often cost effective, so that the most important features can be up and running and possibly used (and thus tested) while the other features are being developed.
  + Test-based development.
    - Automated testing saves time: batch tests into a file, or some other program so that you can run multiple tests at a time automatically (i.e.: xv6 RunAllTests program that ran all tests to see if xv6/CPU scheduler function was working properly).
    - Run all automated tests every time; this helps because each test can be ran whenever changes or more tests are added – adds a layer of security to your testing to make sure program works.
  + Make the customer apart of the team so that you can get good feedback on changes that need to be made to the software.
  + Usually, the picture becomes clearer as you go through iteration and learn more, which is why developers and customers often make/request changes.
    - This necessitates that you cannot make your product very rigid; your code needs to be flexible and modular.
  + Maintain simplicity!
    - Eliminate complexity as much as possible from the software and development process.
* Extreme Programming (XP)
  + Pushing iterative development to ‘extreme’ level.
  + Some of these XP techniques are not widely used.
    - Incremental planning/user stories
    - Small releases – start with most important functionality
    - Test-driven development
    - Continuous integration
    - Refactoring (improving the structure/readability of released features).
  + So, you see, it is the same as agile development but more extreme version of it.

# Lecture 3 notes – 5-17-22 – Software Engineering Models

* Effective Software Team Attributes
  + Need: purpose, involvement, trust, improvement, diverse skillsets.
  + Also need procedures to give the team organization and accountability.
* Factors affecting Team Structure
  + Factors: Difficulty of problem, size of programs, lifetime of a team, modularization of a problem (so that work can more easily be distributed), quality and reliability of program, rigidity of due date, degree of communication required.
  + Software teams need to have a spokesperson – an entrepreneur: someone who can find and acquire business opportunity so that there is work to be done.
* Organizational Paradigms – how to organize the team?
  + Closed (hierarchy) paradigm, random paradigm, open (hybrid of closed and random) paradigm, synchronous paradigm (specialization and compartmentalization of problem naturally into teams).
* Generic Agile Teams
  + Focus on individual and team collaboration abilities.
  + Adaptive
  + Self-organizing, autonomy
  + Planning kept minimal
* XP (extreme programming) Team Values
  + Communication, simplicity, feedback, courage (to not be pressured into looking to far ahead and completing specifications that should be left to future iteration), respect.
* Impact of Social Media
  + Blogs, microblogs, targeted online forums, social networking sites, social book marking.
* Software Engineering using the Cloud
  + Pros: available on all platforms, easy to share,
  + Cons: easier for trade secrets and confidential information to be leaked, as home network team has less control if the cloud is via a service from another company.
* Collaboration Tools
  + Namespace (domain) of access, calendar, templates, metrics, communication and analysis, artifact clustering (prototypes and product specifications).
* Team Decisions Making Complications
  + Decisions can affect/have: Problem complexity, uncertain risk, unintended consequences, different perspectives which create confusion, collaboration and coordination.
  + Think about the consequences of moving things around (even people resources).
* More on Agile Teams…”Scrum” team/agile method
  + An agile method that gives an agile project organization.
  + Based on short-term plans, which are more reliable that long term plans.
* Scrum Terminology
  + Product, Product owner, product backlog (to do list), development team, sprint (product increment created every 2-4 weeks), scrum (a daily team meeting where progress is reviewed and work to be done that days is discussed), scrum master (team coach/similar to project manager), potentially shippable product increment, velocity (estimate how much work a team can do in 1 sprint).
* Key Roles in Scrum
  + The product owner: make sure team is focused on product and product goals; equivalent to project manager, or project manager should take on this role.
  + Scrum and Sprints: software is developed in sprints (2-4 week time frame, developing a set of features), scrums occur each day during a sprint. This should result in a shippable product increment (complete, and ready to deploy software).
  + Product backlog = entire list of features to implement (to-do list); sprint backlog = a set of features to be currently developed, which is a subset of the product backlog.
* Key Scrum Practices
  + Product backlog (to do list)
  + Timeboxed sprints (fixed period, usually 2-4 weeks, to complete a set of features).
    - Work stops at the end of the time allotted, whether work is complete or not.
  + Self-organizing teams – makes decisions by consensus.
* Benefits of timeboxing method for sprints
  + Demonstratable progress
  + Problem discovery (also limits spending too much time, and more quickly change direction).
  + Work planning (start to learn as a team/based on project requirements, good time estimates of how long it takes to develop various features).
* Sprint Activities
  + Some panning: Items/features selected from product backlog.
  + Execution: work on the selected features in the allotted time.
  + Reviewing: analyze how the performance and progress of the sprint when time ends; team does reflection.
* Scrum meeting
  + Should be short and focused.
  + Everyone should give updates on their progress, so that everyone can plan their day accordingly and be updated.
* Agile Activities (should be done during a sprint)
  + Test automation
  + Continuous integration
* Code completeness checklist
  + Reviewed (by others)
  + Unit tested
  + Integrated
  + Integration tested
  + Accepted (as having passed all tests and review)
* Sprint Reviews
  + Did sprint meat goal?
  + New problems
  + Team reflection
* Team Size and composition
  + Ideal size is 5-8 people (not too large, but big enough for enough diversity)
  + Diverse skillsets
  + Diverse experience levels (less experienced people may have newer, updated information)
  + Self-organizing team that can adapt to change (including losing or gaining members), and all members will learn about the other people’s stronger areas.
* External Interactions
  + Interactions with people outside of the team
  + Usually, only the ScrumMaster and Product Owner should be involved in external interactions
  + Idea is that most team members work only with each other (internal only)
* Project Management
  + Self-organizing team appoints manager, often it is the scrum master.

# Lecture 4 notes – 5-19-22 – Software Modeling

* Principles that guide the process
  + Be agile
  + Focus on quality at every step
  + Be ready to adapt
  + Build an effective team
  + Establish mechanisms for communication and coordination
  + Manage change, which is inevitable
  + Assess risk (i.e., of making some change)
  + Create work products that provide **value** for others (think about ENT-400 entrepreneurship class – develop a product that actually captures value for some customer segment).
* Requirements Engineering
  + Inception: ask questions to understand some problem.
  + Elicitation: gather requirements.
  + Elaboration: create analysis model of the software product.
  + Negotiation: agree on a system to be built.
  + Specification: set of models and/or documents, mathematical formula, etc.
  + Validation: testing/review mechanism to find errors or missing/misunderstood specifications/requirements.
    - Check this during each iteration
  + Requirements management (version management, releases, etc.)
* Inception
  + ID stakeholders
  + See many POVs
  + Who will use the solution? (who is the customer)
  + What is the value captured by the proposed solution?
* Eliciting Requirements
  + Conduct meetings with client
  + Meetings have an organized agenda
  + Use charts, bulletin board, power point, etc.
  + Goal is to ID the problem more precisely
  + End of meeting should result in statements of need, feasibility, and scope, and a list of requirements organized by features.
    - This can include several prototypes/proposed solution models.
* Use-Cases – helps understand problem/requirements
  + Collection of user scenarios – shows that you understand the problem and requirements of your client.
  + Each scenario deals with a set of questions that can help showcase the problem and requirements.
  + Can make a use-case diagram.
* Negotiating Requirements
  + Identify key stakeholders, determine each of their respective “win conditions”, and negotiate toward a set of requirements that lead to a win-win.
  + Note: Requirements is simply getting information from your clients; specifications is putting those requirements into technical terms (such as UML diagrams).
* Requirements Monitoring
  + Use each increment to fulfill requirements and monitor each one to make sure requirements are correctly being fulfilled.
  + Run-time verification and validation – match specs and meets user goals.
  + Business activity monitoring – make sure business goals met.
  + Evolution
* Validating requirements
  + Has the overall objective been met and all requirements met?
  + Is there enough or too much abstraction in the system?
  + Are all requirements achievable/still relevant?
  + Make sure each requirement can be tested.
* Object-Oriented Analysis, Design and Modeling (modeling of a software to be implemented)
  + We represent OOA (object-oriented analysis) using UML (unified modeling language).
* Three steps of OOA
  + 1) Use-case modeling
  + 2) Class (object) modeling
  + 3) Dynamic modeling (action-oriented)
  + It is an iterative process
* OOA Elevator problem
  + Make use-case diagram
  + Focus on normal scenarios first
  + Then address exception scenarios
  + Now make a class (object) model (class diagram)
    - Extract classes and their attributes and relationships to each other
    - Use Noun or CRC (class responsibility collaboration) extraction methods.
    - get rid of many-to-many relationships, if possible, by expanding the diagram.
    - CRC cards must not be vague – clearly state which class is responsible to perform some action/responsibility.
    - When new responsibilities/requirements are identified, do another iteration of the use-case and class diagrams.
* Challenges of OOA Phase
  + Don’t make the analysis the actual design – don’t build/design the product yet. Mainly gather requirements.
* Design Phase (OOD – Object Oriented Design)
  + 4 steps:
    - Step 1: construct interaction diagrams for each scenario (use sequence or collaboration diagram); actions on diagram are essentially methods/function calls.
    - Step 2: Construct Detailed Class Diagram (based on the one made from OOA phase, but with more details)
    - Step 3: Design Product in terms of clients and objects (make a client-object relations diagram)
    - Step 4: perform detailed design (detailed pseudo code).
      * Important for determining which language is best to use based on your design.
  + Challenges of design phase
    - Design team should not do too much – do not code it as you are design it.
    - But also don’t do too little – the people who implement it need enough details.
* RSA – Rational Software Architect tool (used to store each iteration, so that you can roll back and track the progress…etc.).
  + EA – Enterprise Architecture is an alternative CASE tool to RSA.

# Lecture 5 notes – 5-24-22 – Software Design

* Software Design
  + Set of development principles
* Software Engineering Design
  + Data/class design
  + Architectural design
  + Interface design
  + Component-level design
* Analysis Model 🡪 Design Model
  + Take an analysis model and map it to a design model.
* Design Quality
  + Must implement all explicit requirements and accommodate implicit requirements
  + Must be readable and understandable
  + Should give complete picture of software
* Design Principles
  + Process should not suffer from tunnel vision
    - Cannot focus on just one thing when designing; must cover all sub-processes
  + Should be traceable
  + Don’t reinvent wheel (so software that it can fit into the rest of the world), and to save time (i.e. using libraries – reusability is part of good software)
  + Minimize intellectual distance between design and real world
  + Exhibit uniformity
  + Design able to accommodate change
  + Can be integrated into other systems
  + Design should allow software to deteriorate slowly (so that it can be better maintained and updated as errors occur and technology changes)
  + Keep design and coding as separate steps
  + Assess and review design as it is being created for quality, semantics, and conceptual errors
  + Cannot always use agile processing for situations where stakes are high, and when experience on developing software for a certain task has never been done by the group that will implement.
    - Depending on product, client, and background of the people who will implement the software will determine if you can use the agile method.
* Fundamental Concepts
  + Abstraction
  + Architecture
  + Patterns
  + Separation ofc concerns
  + Modularity
  + Information Hiding
  + Stepwise Refinement
  + Functional Independence
  + Refactoring
* Abstraction
  + Two kinds: object abstraction (some physical thing abstracted), and procedural abstraction (some event/process abstracted)
* Architecture
  + Structural properties
  + Extra-functional properties
  + Families of relates systems
  + The “structure of software and how it provides system integrity”
* Patterns
  + Use patterns to find and/or implement and customize some library or template
    - i.e. the vector template in c++
    - reusability of software is good and increases reliability
    - creativity does not mean you have to invent things from scratch
* Separation of Concerns
  + Break complex problem into smaller pieces and solve them independently (if possible). The sum of those will be the full solution to the complex problem.
* Modularity
  + Allows a complex program to be manageable
  + Solves each of the separation of concerns
  + Tradeoff: more modules = more integration cost but less development cost; less modules = less integration cost, but more development cost; you need to find the happy medium. **Also, too many classes in your design can reverse the direction of abstraction and start to make things more complex again.**
* Information Hiding
  + The point of public and private variables is to make sure only people or other functions/programs that should have access to the variable are only allowed access, and also to abstract things that others should not worry about.
  + Reduce chance of side effects
  + Limit global impact of design (if all variables global, then one issue or change can cause many global side effects and changes)
    - You have to inform everyone who uses a global variable (or anyone in some variable scope) of any changes to global variables
  + Emphasize controlled communication
  + Discourage use of global data
  + Allows for encapsulation
  + Higher quality software
* Stepwise Refinement
  + Don’t put all details in right away (part of modularization)
    - Start with simple version, then make subsequent versions and gradually add detail
  + Try to have overview of design/problem
  + Need-to-know basis at each design blueprint
* Functional Independence
  + Develop modules with a single mind
    - Don’t make a module do more than what that module is assigned to do.
  + Maximize cohesion of modules
  + Minimize Coupling - does not need other functions in the system – measure relative independence
* Refactoring (NOT redesign – only rearrangement)
  + Output of software is the same – but internally we “rearrange the furniture” in terms of making maintenance easier, more efficient resource use, better syntax/readability.
  + Exams design for redundancy, unused elements/objects, inefficient/unused algorithms, etc.
* OO Design Concepts
  + Design classes
  + Inheritance
  + Messages (between or to an object)
  + Polymorphism (multiple versions of the same function; determined during run time = dynamic bonding instead of classic static binding)
* What is a Relationship
  + Objects connected to each other in some way
* Why Architecture?
  + Enables engineer to have some representation to:
    - Analyze effectiveness of design
    - Consider architectural alternatives
    - Reduce risks of construction
  + Representations of software as a means of communication
  + Highlight early design decisions
  + Conceptual point of view
    - Intellectually graspable model
* Architectural Styles
  + A set of components and connections
* Types of styles
  + Data-centered Architecture
  + Data Flow Architecture
  + Call and Return Architecture (modular)
  + Layered Architecture
    - Design of operating systems and networking (i.e. Network layer, hardware layer, application layer, etc).
* Architectural Patterns
  + Concurrency (OS scheduler)
  + Persistence (of data)
  + Distribution (components communicate in distributed/shared environment)
* Architecture Reviews
  + Assess ability of software to meet architectural design
* Agility and Architecture
  + Avoid rework by getting feedback from users
  + Hybrid models
  + Deliver work after sprints
  + Review code emerging from each sprint and learn/change architecture based on review

# Lecture 6 notes – 5-24-22 – Software Quality

* Effective software process
  + Establish structure needed for the effort of a quality product
  + Management keeps checks and balances
  + SE process practice allows for analysis and design
  + Find possible problems before product is developed is key for quality
  + Adds value to those who will use the software
* Useful Product
  + Contains content and features required by end user
  + Reliable
  + Explicit requirements fulfilled
  + Satisfy implicit requirements (ease of use, security)
  + “surprise and delight”
* Adding Value
  + Add value for producer and user of the software
  + Less maintenance, fewer bugs
  + Bottom line = profit overall
    - Product revenue, profitability, availability of information
* Quality Dimensions
  + Performance quality
  + Feature quality
  + Reliability
  + Conformance (to local and external standards for integration)
  + Durability
  + Serviceability
  + Perception
  + Aesthetics
* Measuring Quality
  + Need a concrete way to measure software quality
  + Project team needs to develop some targeted method to determine quality
  + Need to quantify the measurement of the software quality
* The Software Quality Dilemma
  + How do we determine quality metrics?
  + How do we not be too obsessed with quality metrics so as to have “paralysis by analysis”?
  + But also, how do we make the product with enough quality that we lose in the market because no one buys it?
  + So we have a time dilemma of quality vs time/speed of delivery, where each one could have financial ramifications.
  + ***Need to find the middle ground of quality v speed.***
    - Startups fail quickly because customers have a more critical eye and judge more harshly and with less patience and leniency for companies that don’t already have an established name; hence finding middle ground can be harder since mistakes may have larger ramifications.
* “Good Enough” Software (***not an option for software that can cause damage or cost lives***)
  + Delivers high quality for main features and afford bugs in non-essential features that can be remedied later or “lived with”
  + There are some arguments against “good enough”:
    - May work for some applications; but small companies risk more and never get the chance to deliver better software versions.
    - Ramifications of bugs in a given project may not allow for “good enough” (i.e. for a car engine or nuclear power plant).
    - Embedded software can cause trickle down effects to other software, so quality is a must and you need more than “good enough”.
* Cost of Quality
  + Prevention costs (planning, test equipment)
  + Internal failure cost (rework/repair)
  + External failure cost (product return/replacement, warranty work)
  + Need to find balance of spending money now to save on costs in the future, but not making quality so high that the cost of installing those quality methods is too costly.
  + There is a cost to quality (nothing is free!)
* Cost
  + Cost of allowing bug past the coding phase is when costs rise dramatically (exponentially)!
* Quality and Risk
  + Software is used for everything nowadays! It had better be right!
  + You can go to prison for bad software depending on the situation!
* Negligence and Liability
  + Can be fined
  + Sent to court
  + Sent to prison
  + Requirements and quality must be delivered
* Low Quality Software
  + Increases risk for both developers and end users
  + Easier to hack
  + Secure systems depend solely on quality – it is the essential key for the design of such systems.
  + Bugs
* Impact of Management Decisions on quality
  + Estimation decisions (rushing team – cause shortcuts/cut corners)
  + Scheduling decision (not paying attention to dependencies (timeline))
  + Risk-oriented decisions (bandage/patch method = reduce quality)
  + **It is essential as a project manger to do proper time estimation.**
  + Need to have contingency – anticipate failure and have a plan B, C…etc.
    - Some balanced level of “what-if/risk analysis”
* Achieving Software Quality
  + Good project management and engineering practices
  + Understand problem
  + Project management (anticipate changes)
  + Quality control (inspect, review, test to ensure specs met)
  + Quality assurance (audit to get data for decisions)

# Lecture 7 notes – 5-31-22 – Reviewing Software (Gathering and using Review Data)

* Defect amplification
  + Not reviewing software can cost 300% more and have 300% more defects (errors after release).
  + When reviews are done before release, then there can be 3 times the amount of errors found before releasing the software
* Metrics
  + E\_review = E\_p + E\_a + E\_r
  + Err\_tot = Err\_minor + Err\_major
  + Defect density = errors found per unit of work product reviewed = Err\_tot/WPS (i.e. 0.6 errors per page of code)
  + Where E = effort, p = preparation, a = assessment effort (in person-hours), r = reword effort, WPS = work product size, minor = minor errors, major = major errors.
  + Each field is representing time unit in hours.
  + Effort saved per process = E\_testing – E\_reviews
  + It is better to find requirements-related errors during reviews than when during testing; finding these type of errors during late/major testing will cost significantly more person hours per error.
* Overall
  + You need balance; too little software reviewing will cost a lot of hours (and thus money), but too much reviewing can cost time and money because it can stunt/slow down progress too much and start to have a negative effect.
  + But overall expenditure is better to have reviews than no reviews before releasing software.
* Reference model
  + Reviews steps: plan and prep, meeting structure, correction + verify, roles of individuals.
* Informal Reviews
  + Pair programming (two people program side by side 🡪 higher chance to catch errors).
  + Casual meeting involving 2 or more people to review a work product.
* Formal Technical Review (FTR)
  + Objective: uncover design and implementation errors and make sure code meets requirements.
  + Includes walkthroughs (walk through logic and flow of code) and inspections (check for a certain set of areas).
* The Review Meeting
  + 3-5 people usually (too many people can cause waste of time).
  + Advance preparation expected (1-2hrs per person).
  + Meeting < 2 hrs.
  + Focus on a work product.
* Meeting Players
  + Standards bearer (SQA = Software Quality Assurance)
  + Producer (of work product)
  + Review leader (moderator)
  + Reviewer
  + Recorder
* Conducting the Review
  + Review the product, not the producer (per say).
  + Set an agenda and maintain it, limit debate and rebuttal.
  + Note all problems.
  + Allocate resources and schedule time for FTRs.
  + Review previous reviews.
  + Goal is primarily to find errors (not solutions).
* Sample-Driven Reviews (SDR)
  + Find only major work products to do full FTRs for; we may not be able to afford the time or money to do FTR for every work product, and also we may overall be able to save time and money by not over-reviewing on the overall project.
  + Inspect a fraction a\_i of each work product i. record number of faults f\_i found in a\_i.
  + Gross estimate = f\_i \* 1/a\_i = faults within a work product i.
  + Choose FTR for work product i that has the highest number of faults.
  + So: sample, sort, and prioritize based on SDR to select FDR work products.
* Metrics Derived from Reviews
  + There are many metrics, including the ones we mentioned above; these can help to make analysis and decisions during a software project.

# Lecture 8 notes – 6-2-22 – Software Testing

* What testing shows
  + Errors
  + Requirements conformance
  + Performance
  + Indication of quality
* Strategic Approach
  + Use technical reviews to eliminate errors.
  + Test each component independently first, then integration.
* V & V
  + Verification (set of tasks to ensure functionality)
  + Validation (set of tasks to ensure customer requirements)
* Who Tests the software?
  + Developers and independent testers
* Testing Strategy
  + Inward to outward 🡪 unit/component-level to integration testing to validation and system testing.
  + Integration testing: all internal units work together; system testing = integration in terms of does your software work with other software systems (i.e. network, database, etc.).
  + “testing in the small to testing in the large”
* Strategic Issues
  + Specify/quantify requirements (so that we have a way to measure and test).
  + State test objectives and develop a plan to implement testing.
  + Build robust software designed to test itself – embedded testing.
  + Use rapid cycle testing (scripting/automation essentially).
  + Use continuous improvement to better the testing process.
* Unit Testing
  + Test: interface, local data structures, boundary conditions (i.e. array boundary not crossed – what happens if attempted to cross in a loop?, etc.), independent paths (if statements, or case statements, etc. some path structure – test all paths/conditions), error handling paths.
* Unit Test Environment
  + Sometimes you cannot fully test a module if other related modules are not yet developed; so just develop a “driver” which is a shell/pseudo module of the module to be developed, that way you can still run all test and make calls to the undeveloped modules.
  + Later after the modules are developed, then the test strategy is already in place and you can simply re-run them and call the actual completed modules that a given unit module needs in its testing.
* Integration Testing Strategies
  + Options:
    - “big bang” approach (finish all or most modules, then put them all together and test for errors/issues).
      * Can be hard sometimes to identify sources of problems when you test so many modules together.
    - Incremental construction strategy (develop a few modules and continually test as modules come online/get completed).
* Top Down Integration (incremental construction strategy)
  + Test top module with stubs/drivers (shells).
  + Work down as each stub becomes available with its real completed module and re-run tests.
* Bottom up strategy
  + Drivers replaced from bottom up as the come online and continuously and incrementally test as new modules com online, continuing to build to the final top module.
* Sandwich testing
  + Start from top and bottom, work toward middle drivers and replace with real completed modules…incrementally run tests as each module keeps coming online..etc…just like top down and bottom up.
* Regression Testing
  + Any time you change something (anything is touched in any module’s code), you MUST run a series of tests to ensure nothing is broken.
  + Reduces chances of unintended behavior or errors.
  + Automation/script testing especially important here in order to quickly run series of tests.
* General Testing Criteria
  + Should test for:
    - Interface integrity
    - Functional validity
    - Information content (local and global data structures) (important for security and for preventing corruption).
    - Performance (performance bounds tested against some criteria).
* High Order Testing
  + Validation testing (requirement satisfied).
  + System testing (system integration).
  + Recovery testing (force software to fail to test recovery ability).
  + Security testing
  + Stress testing (worse-case resource scenario – will software still run?)
  + Performance testing (real world testing – integrated in some system context)

# Lecture 9 notes – 6-7-22 – Software Configuration Management (SCM)

* Baselines
  + We use baselines in order to track and manage versions of a project/code in order to have a checkpoint that is reliable and agreed upon and by which work on the project can continue to branch/fork off of and be developed until it is finalized into another baseline.
  + We need some way to automate this process; so, we use various types of software (like GitHub) to manage a repository that centralizes all of the documentation and code and clones of baselines/versions of code/software in a project.
  + It is essentially the idea of Forking (make exact copy of a program/project, and then make changes/additions to it).
* SCM Repository (SCM Repo)
  + A set of mechanism and data structures that allows management of changes in an effective manner.
  + Performs and allows for: data integrity, info sharing, tool integration, data integration, methodology enforcement, documentation standardization.
* Repository Content
  + Verification and Validation documents/testing
  + Business content
  + Project management content
  + Documents
  + Construction content (code)
  + Model content (UML, Pseudo code)
* Repository Features
  + Versioning
  + Dependency tracking and change management
  + Requirements tracing
  + Configuration management (repository, for example, to manage milestones/baselines)
  + Audit trails
  + These features help to branch/fork and continue to develop the other modules or enhancements and corrections to the software, and allow for tracking of defects and making sure requirements are being met.
  + Think about routing and switching when doing change overs with new configurations; there always needs to be a repo with the rollback images/versions in case there are defects; we need to have that baseline or several baselines for backup to work with certain systems or for finding issues with the software releases.
* SCM Elements
  + Component elements (database/repo)
  + Process elements (procedures)
  + Construction elements (versioning of code)
  + Human elements (training of people to use the tools (software) the team uses to implement the above components of SCM)
* The SCM Process
  + Addresses:
    - How to control changes of software
    - Responsible parties of managing changes
    - How to identify the elements of a software configuration
    - How to ensure changes made properly
    - Mechanism to inform other necessary parties of a change made, since there could be defects or side effects of a change that others may need to know or be able to identify.
* Version Control
  + Combines procedures and tools to manage versions
  + Versional control system contains/integrates: project database (repository), version management, make facility (a place with all necessary files of a version/baseline, such as a MakeFile and a folder with all files, in order to build a software version; can also link and build modules from different version into an executable), and issues (bug) tracking of each version.
  + Must be a way to save storage; so some SCM tool does not store the entire copy of every change made, but only the differences between each version so as to save storage space – it is a form of smart storage. (GitHub and sites/software like it can allow for this version control automatically).
* Change Control
  + One of the most important aspects of SCM and baselines
  + Cannot casually make changes; each version, even from the smallest change, must be tracked, and managed and technically reviewed.
    - This is especially critical for software that control dangerous real-life machines/aspects. So technical review and critical.
  + Need to track who made a change, who tested, who approved and reviewed a change, what change was made, etc.
* Change control process
  + Need for change recognized 🡪request from user 🡪developer evaluates 🡪 change report is generated 🡪 authority decided 🡪 (change may be denied) 🡪 queued for action 🡪assign people to SCIs (software config items) 🡪 check-out SCI from repository (everyone from the team will then know some smaller team has checked out some component for changes)🡪make change 🡪 review/audit change 🡪establish a “baseline” for testing 🡪perform SQA tests 🡪check changed SCI back into repo 🡪promote SCI for inclusion in next release (seek for adding it to the next build) 🡪rebuild appropriate version 🡪 review/audit change again 🡪 include all changes in release.
* Auditing
  + Must audit SCIs, change requests, SQA Plan
  + Must make sure procedures are followed and review if procedures are good or need to be improved.
  + Monitors and collects data for the quality of all plans of the project.

# Lecture 12 notes – 6-30-22 – Software Scheduling

* Schedule Tracking
  + Need periodic meeting to ensure things are on schedule.
  + Determine if we hit milestones.
  + Status check among team members, managers, and even the customers to measure progress as compared to the schedule.
  + Need to make team members comfortable to bring up problems so that they do not hide it.
  + Use metrics by placing value on certain tasks in order to measure progress quantitively.
* Earned Value Analysis (EVA)
  + Measure of progress
  + Quantitative analysis
  + More accurate and objective form of measurement of progress/performance
  + BAC = SUM(BCWS\_k) for all tasks k, where BAC = budget at completion, BCWS = Budgeted cost of work scheduled.

# Lecture 15 notes – 7-12-22 – Software Release and Support

* DevOps = development and operations/maintenance/support of a software merged together.
  + Came about because of Agile Process – but this agility does not necessarily carry over to the support group of a software.
  + Gave the “software as a service model”
  + Think about Minecraft. Many of the developers are also the same people who run the support team for releases; and as long as people subscribe/buy server space, we are paying the salary and giving profit for the software as a service.
  + Main principles: everyone responsible for everything, automate everything that can be, measure first (test) and change later (then release).
  + Benefits: faster deployment, reduced risk, faster repair, more productivity from teams.
* Code management
  + Need a way to manage the large code files – including making changes, testing, etc.
  + Example of a code management tool: Git Hub (manages a code repository).
  + Helps so that everyone knows what changes were made and when for a given release/branch of a software.
  + Repository can also track bugs for various versions (from users, or from some other quality assurance metrics defined by the team/group).
    - Think about the prompts you can get when using a program “would you like to automatically send bugs/errors to …?”
  + Many code management have efficient memory storage systems: for example, only store the differences between versions so as to not have to have multiple duplicate copies of large projects.
  + Git is a distributed file version control system.
    - Think about shared variables etc.
    - Think about google docs, or google drive, and other distributed systems.
  + Code management Repository applications such Git maintains a master branch, and allows others to branch off and remerge, or leave off a branch as experimentation or as another specialized version of a software.
* Need DevOps with automation to greatly reduce error. EVERYTHING THAT CAN BE SHOULD BE AUTOMATED (as a fundamental principle).
  + Also increases reliability and reproducibility of a software.
* System integration: for example 🡪 MakeFiles
  + Gather all elements necessary and placing in appropriate directories so that a software will compile and run.
  + Need continuous integration (CI): every time a change is made, we need to reintegrate the new version to make sure it works.
    - Need to automate this process too.
  + However, we need a system that can do fast system builds; to increase build speed during integration testing, we need an automated build process that only rebuilds parts of the system that have been changed.
    - They use dependency trees (thus the automated system can use some search and sort algorithm), in order to determine which modules need to be rebuilt based on what code/module was changed.
* Continuous delivery and deployment
  + Depends on continuous integration, but involves another test environment that is the actual environment the software will run in (or an environment that can simulate the actual environment).
* DevOps is a software project on its own; it is the manual work done in order to automate the work (testing in particular) of another software project.
* Infrastructure as code: use Configuration Management tool (CM Tool) in order to automate how and which version of software to install on a given system. This allows for many different types of systems to be updated with the correct code and in the correct way for a given system in an automated process.
  + Also, this helps automate which testing environment a given software version must be tested in (or for example it can test in all necessary environments, and/or with the correct version of code for the target environment/machine).
  + Helps greatly increase rollout speed.
  + Also allows automation and faster rolling back to older versions of code when necessary.
* Containers
  + Virtual environment within an environment.
  + Allows for testing a virtual environment needed on some local machine that is a different environment. Think about Hypervisors.
  + Dockerfiles: build executable container image by processing dockerfile, which is a definition of your software infrastructure as code.
* DevOps Measurements
  + Metrics (such as business, user, bugs, etc.) used to measure software versions and which can be used to help inform decisions made in later versions or in other aspects of maintenance of the software.