**CIS-285 SUMMER 22 NOTES – DEMETRIUS JOHNSON**

# CIS 285 – SUMMER I – with Dr. Abou Nassir-Mouhammad

# Lecture 1 notes – 5-9-22 – What is software and software engineering?

* Goal for software development: follow the model of current engineering that came before software engineering; how large projects are managed and worked on by many different groups and teams in order to get a working product that can make profit and is useful.
* Many people are involved in a software engineering project just like any other engineering project; software engineering is one of the oldest engineering disciplines to come to humanity.
  + So, we borrow from real engineering; in order to really understand software engineering, learn how engineering projects are done in general since this is how software engineering was derived.
  + Software will “deteriorate” just like any other engineering project; but not because it is being weathered or broken down in that sense of time, but the sense of time where things will get newer and better and technology keeps changing, and thus a software project will deteriorate in the sense that it becomes obsolete over time, even to the point where it is completely unusable in today’s world – provided it is not maintained. So, often software engineers keep in mind maintenance and expansion of the future when they design software so that it does not deteriorate (or as fast) and so that it can evolve with the times.
  + Try to design software to be flexible; you don’t want your software to be so fragile that you cannot change anything without breaking the entire program.
* Often, we want to make software that works across platforms.
* Software project v software product
  + Projects are often done in house for a company, i.e. company software.
  + Products are supposed to be marketable; it is not as custom made as a software project for an immediate need for a company or some person. Products are more generic so that it can be useful for a range of customers.
* You might think of a software engineer as more of a technical entrepreneur – someone who is a businessman who understands how to sell technical skills of software engineering.
  + Facebook started as a project at a university. Then, they saw the opportunity and made it into a product – they were visionaries, and they worked to try and make their software profitable! So, you see, they were opportunists: entrepreneurs.
  + Software engineers must of a product vision, just like any other entrepreneur.
    - Ask/give: FOR, WHO, PRODUCT NAME, THAT (key benefit), UNLIKE (competitors), OUR PRODUCT (statement of differentiation).
    - Use words such as “innovative” in product name or phrase.
* Deadlines and resource management are key to successfully marketed software products.
* Three types of software execution models: stand alone, hybrid, and software as a service.
* Product Management
  + Make sure the product is on a good schedule and mitigating any issues by anticipation and finding solution.
  + Have to know how to deal with people (customers, employees, etc.), and calm and cool mannered.

# Lecture 2 notes – 5-16-22 – Object-Oriented Analysis

* Review from last lecture:
  + Complexity = the interdependency between (many) components.
    - We *might* even say software engineering is exactly the same as all other engineering disciplines, but even more complex.
  + Specifications/requirements/think 🡪 design/architecture 🡪 prototype/implementation 🡪 testing; just like in entrepreneurship, you start off with more divergent thinking, but then narrow it down (convergent thinking).
  + Like any other engineer, you need tools for the job as a software engineer to help get things done faster, more efficient, more accurate, and more precise. Tools make the engineer a better engineer because you can do your job better.
* For this lecture, let’s take a scenario where we are doing object-oriented analysis to use object-oriented methodology for a software project.
* Objected Oriented Paradigm
  + Objects are independent units with data (attributes/variables) and actions (methods/functions).
  + A program may be simply interactions of several objects.
  + OOA = Object Oriented Analysis
  + UML = Unified Modeling Language – shows interaction of classes.
* 3 steps of OOA
  + 1) use case (testing), 2) class modeling (data model), 3) dynamic modeling (method/functions modeling).
  + These 3 steps are an iterative process 🡪 there are different iterations in each of these steps as the object-oriented analysis builds/elaborates/narrows down.
* Elevator Problem: OOA
  + (an idea I have): an AI/machine learning elevator software; the elevator learns the uses of the elevators and incorporates that data in to probabilistic models into its software to determine when/where the elevator should wait for the next user. Incorporate AI with fairness (kind of like I/O with disc processes). Would be especially nice for large buildings, or even small ones too, and building such as hotels, where certain floors may be holding more customers who are using the elevator more often during some period of time. We could have testing by tracking how many floors the elevator has to pass before reaching the user, in order to see if our AI/machine learning elevator software works/is effective. Test program could have a set of user requests from various floors, and based on probabilistic spread, the next group of tests ran that is (perhaps exactly the same requests) should be able to complete all requests without passing as many floors (meaning all requests handled faster). Use CIS-450 disk I/O request models to help with the design of this AI elevator software.
    - Also, what if you could tell “who” is on the elevator, simply by measuring the current force applied to the elevator. With enough significant figures, you could tell exactly who gets on and off. In doing this, if a user gets off the elevator before their desired destination, perhaps then their request is automatically cancelled to save time for other users.
* Class Modeling
  + Extract and represent classes only by their data variables.
  + Be careful not to have too many classes, which could raise complexity more than necessary.
* Two approaches to class modeling (both are iterative, and follow the 3 general steps of OOA)
  + 1) noun extraction
    - 1 – define product in sentence, use the nouns used in the sentence to get a set of possible classes that need to be defined and used.
    - 2 – informal strategy (divergent)
    - 3 – formalized the strategy (convergent, narrow down which classes (nouns) will be used and which nouns will be combined into other classes)
  + 2)CRC classes (class, responsibility, and collaboration)
    - Use CRC cards
      * Need name (class), functionality/methods (responsibility), invocation of other classes (collaboration), now automated CASE tools.
      * Best to use this method when someone knows about the topic of which the software is being developed for.
    - Encapsulate data (hide info) in the CRC card so abstract and use more sentences about what a class is responsible for.
* Challenges of the OOA Phase
  + Don’t be too eager/force an object-oriented design for a program/software project.
* Design Phase (OOA design phase)
  + Design product based on objects extracted during OOA.
  + Classes by themselves are useless (usually). They need to interact with other classes (often/usually).
  + Sequence diagrams (drawn using CASE Tools) show scenario in a special way as you move down the diagram time is moving forward.
  + 1) construct interaction diagrams for each scenario
  + 2)construct detailed class diagram
  + 3)design product in terms of clients of object (determine which class is a client to which class, kind of like a client-server model relationship, where also a server could become the client of the same class it was acting as a server to).
  + 4)perform detailed design (detailed design method; pseudo code).
* Challenges of the Design Phase
  + Don’t turn project into coding yet (don’t do too much), but also don’t do too little – do pseudo code. So, you need to find that balance.
* One CASE tool is RSA (Rational Software Architect)
  + **The software supports model-to-code (forward transformation) and code-to-model transformations (reverse transformations)!**
  + These kinds of tools in the future could allow practically anybody to “code” by using a modeling language.

# Lecture 3 notes – 5-23-22 (chapter 31 – project management concepts)

* The Four P’s of project management
  + People
  + Product
  + Process
  + Project
* Stakeholders
  + Anyone who has a part in the project: senior managers (not necessarily technically sound, but more business oriented), project (technical skills and some business background) managers, practitioners (technicians/workers), customers, end-users.
* Software Teams
  + Project manager must think about the team: how to lead, organize, collaborate, motivate, create good ideas?
* Team Leader
  + The MOI model: motivate, organization, Ideas (or innovation).
  + Ultimate goal is to deliver a product that meets the requirements.
* Software Teams
  + Must consider:
    - Difficulty of problem
    - Size of resultant program (in lines of code/function points/complexity)
    - Time team will stay together (team lifetime)
    - Degree of modularization of a problem
    - Required quality and reliability
    - Rigidity of delivery date
    - Degree of sociability (communication) required
* Organizational Paradigms
  + Closed paradigm (traditional hierarchy)
  + Random paradigm (individual initiative)
  + Open paradigm (mixture of random + closed)
  + Synchronous paradigm (depends on modularization of problem)
* Avoid Team “Toxicity”
  + Frenzy work atmosphere
  + High frustration due to technical constraints
  + Fragmented/poorly coordinated or poorly defined procedures.
  + Unclear definition of roles
  + Continuous repeated failure
    - When you are successful as a project manager with a team, that builds confidence – the team will have more confidence in their team leader or manager. They will tolerate more problems in future projects until there is enough failure.
* Agile Teams (increment/iterative/test-driven)
  + Team members must trust each other
  + Distribute skills appropriate to problem
  + Mavericks can disrupt team chemistry
  + Team is self-organizing and adaptive
  + Minimal documentation
  + Primary idea: team members must deliver/complete their tasks so that overall the team delivers a product.
* Team Coordination and Communication
  + Formal, impersonal approach (technical memos, SE documents and work products)
  + Formal, interpersonal procedures (review meetings, code inspections)
  + Informal interpersonal procedures (group meetings for info dissemination)
  + Electronic communication (emails, etc.)
  + Interpersonal networking (informal discussion with team + outsiders who may have insight)
* The Product Scope
  + Scope
    - Context
    - Information objectives
    - Function and performance (input 🡪 output)
  + Must be unambiguous and clear at management and technical levels
* Problem Decomposition
  + Partitioning an elaborate problem to establish a simplified framework
* The Process
  + Consider project characteristics
  + Degree of rigor required
  + Define a task set for each activity
    - Task set =
      * SE tasks
      * Products
      * Quality assurance
      * Milestones (to help measure overall project completion and motivate team)
* Melding the Problem and the Process
  + You need to meld the aspects of the problem with the process to complete/accomplish each aspect.
* The Project can get into trouble when…
  + Software team does not understand customer need
  + Poor definition of product
  + Poor changes
  + Technology changes
  + Business need change/bad define
  + Unrealistic deadline
  + Resistant users
  + Loss of finances
  + Lack of skills
  + Best practices not followed
  + Not learning from mistakes (lack of continuous improvement)
* Common-sense Approach to Projects
  + Start on right foot – work hard and understand objectives
  + Maintain momentum
  + Track progress (quality assurance)
  + Make smart decisions (i.e. with time management; keep it simple as possible)
  + Conduct a postmortem analysis (learn from each project/process; continuous improvement)
* To Get to the Essence of a Project
  + Why is system being developed
  + What will be done
  + When accomplished
  + Who is response
  + Where organizationally located
  + How job will be done
  + How much of each resource
* Critical Practices
  + Formal risk management; manager who has contingency plans in place
  + Empirical cost and schedule estimation
  + Metrics-based project management
  + Earned value tracking
  + Defect tracking
  + People-aware project management

# Lecture 4 notes – 6-06-22 (chapter 29 – software configuration management)

* Why does software change?
  + Technology, data, and requirements are always changing and evolving.
* The Software Configuration
  + 3 pieces: programs, documents, data.
  + You need something to help you manage the changes; that is why we need software configuration 🡪 like Microsoft Project.
  + SCI = Software Configuration Item
* Baselines
  + A milestone in the development of a software; a foundation that is **formally agreed upon** of a version of the software that is then built off of – helps everyone developing modules to start on the same level.
  + Can build baselines from other baselines (updated/modified version of the first or other baselines).
  + Same idea as having ability of rolling back to older or previous versions of software so that it is safe to test and/or make changes 🡪 these rollback images/version of code are stored in repositories.
    - Exactly the same idea of how I backed up images of my virtual box machine for xv6 development.
* PDL = program description language (pseudo code)
* SCM Repository (software configuration management repository)
  + GitHub, etc.
  + Allows for baselines of a software, so that any changes to the baseline can be managed or even changed in an effective manner.
  + Allows for checking data integrity, information sharing, tool and data integration, methodology enforcement, document standardization.
  + Can allow for branching off of baseline in several directions, and then even deciding when those branches (different versions with some respective module developed) should merge again (and perhaps dubbed into a new baseline).
* Repository Content
  + Baseline code, model content, project management content, other documentation, etc.
* Repository Features
  + Versioning
  + Dependency tracking and change management
  + Requirements tracing
  + Configuration management (configs representing baselines/milestones of versions + documentation associated with that milestone)
  + Audit trails (who, when, what, and why for some change)
  + Can have space optimization when saving versions by only storing the changes (but may be more robust and safer to save the entire version as a full copy for sake of redundancy and robustness).
* SCM Elements
  + Component elements (set of tools for file management system)
  + Process elements (set of procedures)
  + Construction elements (set of tools for construction of software)
  + Human elements (team that uses the above components)
* The SCM Process
  + Addresses: how are discrete elements of software defined, how are versions managed, how to control changes, who has responsibility of approving change, audit to ensure changes done correctly, what mechanism is there to inform others of a change.
* Version Control (System)
  + Combines procedure and tools to manage version of a software.
  + Should be: a project database (repository), version management (smart storage; only save changed configuration objects; and control naming of versions), make facility (a place where a software engineer will have all files associated with a version in order to build it -> like a makeFile), and issues tracking (to track issues associated with changes).
* Change Control
  + Others must know about changes made so we can manage issues/defects and side effects that may arise, and for any other modules/people working on those modules may be affected by a change.
  + There needs to a be a formal process.
  + Change process must be thoroughly tested and audited and reviewed.
* Auditing
  + These items must be audited in order to maintain a robust project: SCIs, Change Requests, SQA Plan.

# Lecture 5 notes – 6-13-22 – Integrated Development Environment (IDE)

* History
  + Before IDEs, programs were written down in text editors by programmers
  + You had to close the editor and then compile it and find errors.
  + Nowadays, such as in Visual Studio, you can write, get syntax errors from compiler, and compile, link and run the program all from one environment (IDE).
* Turbo Pascal (debated that Visual Basic was first as it came to market first)
  + First software that had both editor and compiler in one – which is why today we just say “the compiler finds the errors”.
* Some IDE focus on One Language
  + i.e.: CodeLite and C-Free for C and C++
  + in Visual Studio, you can compile and run many, many languages, as it has many languages integrated.
* Features of an IDE
  + Main feature: offers central interface with all tools needed by the developer including:
    - Text editor (LSEs: Language Sensitive Editors, as most text editors detect errors).
    - Ability to compile (right from IDE; do not need to go to command line and call the compiler separately..etc..)
    - Debug your code on the go (step through, and break points, look at variable values, stop conditions, etc.)
    - Build Automation Tools (automate building the executable code; link with other objects (.o) and create executable); i.e., do not need a make facility (i.e. you do not need to create and self-build a makeFile); the IDE will automatically do linking and creating the final, runnable .exe file.
* Some IDE’s support Multiple Programming Languages
  + i.e.: Visual Studio
  + also Eclipse Software (open source)
  + etc.
* The Visual Studio Environment (in terms of C#, which is common to c++, etc.)
  + Includes designer Window, solution explorer window, and properties window.
  + Multiple projects can belong to a unique solution (but not vice versa). This allows for modularity: i.e. one project could be device drivers, another project gui interface; but then the solution will have all of the projects available to be brought together into one solution/program.