Midterm Review

JD Kilgallin

CPSC:480

10/03/22

Pressman Ch 1-11, 24-25, Appendix 1

"I really need 5 hours of Facebook to balance out my 5 minutes of studying."
-Thabang Gideon Magaola

Midterm

- Anything covered in class or assignments is examinable with the exception of items struck in following "learning objectives" slides.
- The selected lecture slides following the learning objectives only serve to highlight essential topics, but do not cover everything.
- Reviewing the textbook chapters is recommended.
- Will reuse quiz and textbook problems.
- Questions will be a mix of short-answer and diagramming.
- Exam will be long, but curved.
- **1 page of notes (2-sided) allowed.** You will turn this in with the test.

Topics

- Software engineering fundamentals
- History of software
- Software engineering terminology
- Software products and projects
- Software development lifecycle
- Software engineering methodologies
- Software development organizations
- Software engineering roles
- Tools for software engineering
- Version control/Git/GitHub
- Collaborative development
- Requirements engineering
- Software project planning
- Software product design
- Software systems modeling
- Formal methods

Learning Objectives lectures 2-3

- Invention of software and software engineering
- ~~History of software 1950s-today.~~
- Classification of software systems
- Role of a Software Engineer in an organization
- Core concepts of the software engineering profession
- Software development process models
- Phases of development and their elements
- Common and historical software engineering methodologies
- Evolution of software systems
- Core DevOps concepts

Learning Objectives lectures 4-5

- Participants in software development and their roles
- Organization of software development teams
- Career tracks in software engineering
- Relationships to other parts of an organization
- ~~Applying to software engineering roles~~
- Types of tools involved in software engineering
- Features of project management tools
- Features of development tools
- Features of testing tools
- Features of tools for builds, releases, and version control
- ~~Using Markdown~~

Learning Objectives lectures 6-7

- ~~Navigating GitHub~~
- ~~Creating a GitHub repository~~
- Working with Git repositories on the command line
- Making, sharing, and merging changes in Git
- Branches and forks of Git repositories
- Process of establishing requirements for a software project
- Roles involved in establishing requirements
- Functional vs non-functional product requirements
- Software specification
- Modeling requirements

Learning Objectives lectures 8, 10, 12

- Project planning concepts
- Estimating effort required
- Measuring progress according to project plan
- Design concepts
- Software architecture
- Design patterns
- Effective design principles
- UML diagrams (rigorous UML standard **not** required)
- Formal methods concepts
- Methods of formal modeling
- ~~Temporal Logic of Actions system~~

Basic Terms

- Software: The practice of storing computer instructions as code in the same form as program input data.
- Software engineering: The practical application of scientific knowledge to the creative design and building of computer programs.
- Software project: A planned iteration of software development.
- Software product: The artifact of a commercial software project.

Invention of software

- Charles Babbage and Ada Lovelace credited for first program design
- Gottfried Leibniz, David Hilbert, Alonzo Church, Kurt Gödel, Claude Shannon (and others) developed mathematical/statistical foundations
- Church's student Alan Turing developed a formal theory of computing
- Electrical computing hardware developed for Allied artillery and corporate business processes in first half of 20th century.
- Hilbert's student John von Neumann applied Turing's theory to build a general-purpose digital computer following World War II.
- The von Neumann model, including program-as-data, persists to modern computing systems, and forms the basis for *software*.

Why is software *engineered?*

- The past 70 years of software development has led to identification of effective principles, techniques, and processes for software systems.
- Well-engineered software is easier to enhance and adapt, and therefore *cheaper* to develop and maintain.
- Poorly engineered software will ultimately fail to meet user expectations in the presence of competing/disrupting products.
- Software is ubiquitous and present in important settings including medical, automotive and financial applications. Faulty software causes real physical and economic harm.

What types of software are there?

- Form factor – desktop, mobile, web, embedded, distributed, cloud
- Purpose – application, utility, system, drivers, etc.
- Context – standalone executable, script, plugin, service, library
- Consumer – individual, enterprise, advertisers, developers
- Framework – Java, .NET, Ruby on Rails, etc.
- Architecture – Monolith, microservices, layers, Model-View-Controller
- Source language is *not* a distinguishing feature of software. Most software contains components written in many different languages, and it is not generally possible to tell what language was used.

Software application domains

- System – Written to support other software (OS, compiler, editor)
- Application – Standalone software that meets a specific need
- Scientific/engineering – Data processing software for computation
- Embedded – Control software in an appliance (e.g. car, microwave)
- Product-line – Software systems built from reusable components
- Mobile/web – Software designed for portability and remote access
- Artificial Intelligence – Uses large datasets to solve problems that are intractable by other means, e.g. language processing.
- *Gaming – Uses computer graphics for interactive entertainment

Software development activities

- Communication (Requirements) – Deciding what the software will do
- Planning – Deciding what needs to be done to achieve requirements, including allocation of time for product design/modeling
- Modelling (Product Design) – Deciding how the software will be built
- Construction
  - Implementation – Creating the software according to design
  - Validation (Testing, aka Quality Assurance (QA)) – Confirming the software meets requirements
- Release (Deployment) – Delivering software package to the consumer

Process models

- Waterfall
- Parallel waterfall
- Prototyping
- Iterative
- Agile
- DevOps
  - Infrastructure as Code
  - Continuous Integration/Continuous Deployment

Agile Alliance Principles (paraphrased)

- Highest priority is customer satisfaction.
- Welcome changing requirements for competitive advantage.
- Deliver working software as frequently as possible (down to 2 weeks).
- Business and development staff must work together.
- Get work done by giving motivated people a conducive environment.
- Prefer face-to-face communication.
- Working software is the primary measure of progress.
- Promote an indefinitely-sustainable pace of development.
- Continuously pay attention to technical excellence and good design.
- Simplicity – maximizing the amount of work not done – is essential.
- The best designs emerge from self-organizing teams.
- Routinely reflect on how to become more effective and adjust accordingly.

Agile processes

- Scrum meetings – Short "stand-up" meetings at the beginning of each day to describe previous day's work, plans for that day, and obstacles.
- Product backlog – A list of work tasks to be prioritized and assigned.
- User stories – Capturing requirements in the form "As a ___, I want to ___"
- Product owner – Responsible for backlog item definition and priority.
- Sprints – A short, 2-4-week block in which selected work items and bug fixes are completed by developers, resulting in a software increment.
- Sprint review meeting – Demonstrate features developed that sprint.
- Sprint planning – Work to be done next sprint is estimated and assigned. No new work *should* be added to the sprint once it has begun.
- Pair programming (XP) – Two developers, one computer for each item.
- Task (Kanban) board – Table showing current status on each work item.

Software maintenance

- Occurs through the entire life of the product. More expensive than the original development due to the long duration.
- *Perfective maintenance* occurs to proactively improve product performance, existing functionality, architecture, and documentation. This is often planned before the version to be maintained is released.
- *Corrective maintenance* occurs in reaction to a bug or defect whose existence or impact is only identified after a release is deployed.
- *Adaptive maintenance* occurs in reaction to changes in the environment in which the software needs to run (e.g. new version of underlying database server has new security requirements).
- *Preventive maintenance* occurs to proactively fix potential issues before they cause impact to customers (e.g. developers, testers, or users find an exploitable, but not yet exploited, weakness in application security).

Lehman's 8 Laws of Software Evolution

- "Continuing Change"
- "Increasing Complexity"
- "Self-Regulation"
- "Conservation of Organizational Stability"
- "Conservation of Familiarity"
- "Continuing Growth"
- "Declining Quality"
- "Feedback System"

Job titles in software engineering teams

- Software engineer/developer (build software as primary focus)
  - Game/web/mobile developer (design and build software in specific target domain; particular flavors of software engineer)
  - IoT/Embedded software engineer (software for connected devices)
- Product/project manager, UX engineer (design software elements)
- Quality assurance engineer (test software, build test infrastructure)
- Technical writer (Develop software documentation and guides)
- Integration engineer (writing code to interface technologies)
- Software/systems/solutions architect (typically a higher-level role responsible for design and maintenance of program structure)

Software engineering team structure

- A functional *software development team* typically consists of a team lead, 2-10 SWE, 1-4 QA engineers, 1-3 product managers, and possibly a technical writer, integration engineer, and/or UX engineer.
- A *software development organization* is responsible for a suite of products and typically contains 1-50 development teams.
- Team leads report to a director or VP of engineering, or directly to CTO in smaller organizations. Conversely, larger organizations may have intermediate engineering managers for additional hierarchy.
- Higher-level and ancillary roles, such as software architect, product manager/owner, integration engineer, technical writer may report directly to organization leader and fall outside individual teams.

Project management tools

- Track the work currently in progress, completed, and still to be started.
- Hierarchical structure of projects, features, items, and tasks.
- Title, description, priorities, history, product/area, target product versions/dates, time estimate, supporting files, comments.
- Current status, personnel assignment, and *definition of done.*
- Bug reports, root cause analysis, severity and impact notes, criteria for fix.
- Links to parent/child items, progress-blocked-by, duplicates, related items.
- May incorporate source code, code reviews, test cases, and relationship between code checkins/authors, tests/signoff, bugs, original work items.
- Common tools include Trello, Azure DevOps, JIRA, Bugzilla.

Testing tools

- *Unit tests* (should) cover individual code paths of a single method.
- *Integration tests* cover interactions between methods/components.
- *End-to-end tests* cover all steps of realistic user activities.
- *Mock* components are used to simulate functionality of components not under test (e.g. returning data from a database or web service).
- Developers usually write unit tests with code, QA Engineers do the rest.
- Test software may be part of IDE, version control, standalone (e.g. Selenium), developed in-house for specific needs, or a combination.
- Selected tests may run automatically each checkin, full set daily w/report.
- Some tools test GUI components by emulating mouse motion/clicks.
- Some tools allow *static analysis* (search for common bugs or vulnerabilities like SQL injection). May use external assessment firm specializing in this.

Git – repositories

- SCM system built to record changes to files. Allows users to merge simultaneous changes, revert changes, recover old versions, see who changed something and when, compare files at different times, etc.
- A *repository* is a directory/folder representing a project and tracks changes to files and subdirectories. Always has a ".git" directory.
- A repository can exist solely on one machine, or it can be shared, in which case multiple copies can be *cloned* from the *origin* copy.
- While git can be used without a central repository, typically each developer working on a single project will clone a copy from a central source hosted and managed by the company or product owner.
- Git is very complex and we will cover only a subset of features, commands, and options needed to support core use cases.

Git – commits

- A *working tree* is a local copy of a git repository where changes are made. Changed files are *staged* for a tracked update, and staged files are *committed*, creating a new local snapshot of the repository.
- New snapshots are *pushed* to origin, and others' changes are *pulled*. Any remote changes must be pulled/merged before yours can be pushed.
- Git stores each commit with a snapshot of changed files, metadata like author and message/title, and pointer to parent commit(s).
- A commit is identified by a hash of the snapshot. You can always checkout a commit to restore the local copy to a previous state.
- A *tag* can be created that serves as a **static** pointer to a specific commit. The main use is to tag a release with the version number.

Git – additional topics

- Branches
- Forks
- Merges
- Pull requests
- Commands

What is requirements engineering?

- "Requirements engineering (RE) is the process of discovering the purpose for which a software system is intended, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation."
- "A disciplined mechanism to understand the customer’s desires, negotiate for a reasonable solution, describe the intended behavior of the proposed system & constraints associated with it for transfer to the working system."
- Requirements engineering is a function of project management and product ownership that occurs throughout the lifetime of a product and is integral to success of the people, process, project, & product.

Why are requirements *engineered?*

- Software requirements are complex and difficult to capture.
- Stakeholders have differing levels of sophistication and understanding of what is feasible for software systems to do, and an individual's vision may be difficult to translate into actionable development tasks.
- Users in a specific domain are rarely familiar with software development processes, and developers are frequently unfamiliar with the application domain, leading to communication barriers.
- Without a working product to reference, there are an endless number of options for reaching a solution to a problem stated in basic terms.
- Poor understanding and assumptions about needed software behavior leads to products unsuitable for their intended purpose, resulting in lower rates of adoption and commercial success.

What types of requirements are there?

- *Functional requirements* define what the software must do. "The software must allow users to search for a student by id number".
- *Non-functional requirements* define *how* the software must do what it does. "The software must support more than 1000 users at a time."
- A requirement may blur the lines, but this frequently means it can be decomposed into two or more requirements "The software must display a list of students in three seconds or less"
    1. The software must display a list of students
    2. The software must perform an operation in 3 seconds or less.
- Requirements may be explicit or implicit (e.g. "Must not crash during routine use", or "Must allow keyboard input for text").

How are requirements engineered?

- Inception – Someone identifies a need for software to address a specific use case.
- Elicitation – Product team works to define basic requirements.
- Elaboration – Requirement details filled in to understand exact needs.
- Negotiation – Budgets, timelines, and resource availability considered
- Specification – A full(ish) statement of the work to be performed.
- Validation – Specification is confirmed to meet customer needs.

Modeling requirements

- *Class models* describe objects to be modeled within the software, the operations that can be done with them, and their relationship. Object-oriented programming is to some extent based on class diagrams from requirements modeling.
- *Scenario models* describe a use case and the sequence of actions a user might take to accomplish the task at hand. There are many ways to represent this such as a *sequence* (or *swimlane*) diagram.
- *Behavioral models* describe what input and output the software may consume or produce at a given point, and how the software transitions to other states over time. Usually uses state diagrams.

Planning

- Aims to optimize cost and schedule of a project to meet requirements and make best use of resources. Primary concern is estimating effort.
- Happens to some degree in conjunction with requirements engineering and product design.
  - Requirements must be known in order to build a plan, but a plan is needed to assess cost/timeline and whether the project is feasible at all.
  - Some design is needed to build a plan, but design work should be planned.
- Iterative process with rough task outline and estimates for the whole project, gradually refined and broken down to sprint plans.
- Some aspects unique to software vs other engineering disciplines (lack of reference, difficulty capturing requirements, scope/complexity).

Time vs effort

- Attempts to accelerate product release schedule requires exponential increase in effort and the number of personnel needed to complete that effort, so available resources must be considered carefully.
    - Additional communication/management overhead grows quadratically.
    - Additional complexity integrating changes; more incompatibilities introduced.
    - More code rewrites needed to maintain project integrity ("too many cooks")
    - Harder to have all developers productive all the time (training, blocked work)
    - Requirement changes and dependency delays affect more people.
- Effort is frequently measured in person-months, but more person-months are required for faster projects.
- *Putnam-Norden-Rayleigh (PNR) curve* suggests effort is proportional to $(\text{Lines of Code})^3/(\text{Time})^4$. Delivering 10% faster requires 50% more staff; 25% faster => ~4x the staff. Impossible to go much beyond this.

Planning Poker

- Method for producing consensus estimates of effort needed for tasks.
- Team meets at the beginning of a product iteration or development sprint with a list of pre-defined tasks to be estimated.
- Each developer has a set of cards with numbers on one side (the "face").
    - The set of numbers varies by team, as do the units (hours, days, generic "points").
    - Keyfactor uses points in multiples of Fibonacci numbers: ½, 1, 2, 3, 5, 8, 13, 20, 30.
- For each task to be estimated, the team:
    - Reads the task to ensure everyone understands the task to be done.
    - Has each member select a card from their "deck" for their individual estimate.
    - Places cards face down on a table, then flips them when everyone has played.
    - Discusses any major differences to identify causes of disagreement (may redo round)
    - Takes the mean or median as the consensus estimate for that task.
    - Tries to break down any tasks that don't fit in a reasonable amount of effort.

Why is software *designed?*

- A software design can be assessed for quality and suitability before investment in construction of the software as designed.
- Allows final product to be assessed for completeness and quality.
- A software design defines an end goal that will determine when software construction is complete, and allow measurement and forecast of progress toward completion.
- A software design can provide a common understanding among non-technical stakeholders and engineers who will implement the design.
- Allows for changes in requirements to be translated into changes in software implementation.

How is software designed?

- Incrementally decreasing levels of abstraction.
- Requirements translate into a list of use cases, and objects to be modeled become evident. Requirements models translate to design.
- Five major types of design:
  - Architecture design describes core features and basic relationships needed to model use cases.
  - Component-level design gives more granular specifications of sub-systems and precise relationships within and between them, listing individual object types, their attributes and possible interactions.
  - Interface design describes interactions with users and other systems.
  - Deployment design describes where and how software runs.
  - Data design describes information representation.

Separation of concerns

- A *concern* is a feature or behavior of the software.
- Decomposition of a program into independent components allows a complex system to be broken down into manageable pieces. "The [complexity of the] whole is greater than the sum of its parts" implies "each part is less [complex] than its share of the whole".
- Modularity – Separate, named components that are integrated to comprise the whole program.
- Functional independence – Modules created in a way that each serves a single concern and communication between modules is simplified.
  - Cohesion – Degree to which all parts of a module serve one concern. Maximize this.
  - Coupling – Degree to which a module depends on others for its concern. Minimize.
- Encapsulation – The ability for modules to work together without needing any information from other modules outside of their defined interfaces.

Effective design principles

- Primitiveness – A method should do one task, and there should only be one method to do a task. Code should be reused rather than duplicated.
- Traceability – A method or other code snippet should be able to be traced back to the design of that component, and back to the requirement that led to that design.
- Symmetry – Design of components should be similar to each other within a single product.
- Dependency Inversion – Components should depend on abstractions, not implementations.
- Interface Segregation – Many narrowly-tailored interfaces between systems or modules are better than one all-purpose interface.
- Common Closure – Classes that change together, belong together.

UML

- Unified Modeling Language, or UML, was developed to support a prescriptive software development process ("Rational Unified Process"), now used in other contexts regardless of process model.
- A standard language for specifying, documenting, communicating and visualizing 13 types of software models.
- Usually more formal than required; informal diagrams can still easily capture design and provide flexibility with less work.
- May be automatically generated by an IDE or other visualization tool, or may be used as a basis for automating construction of a program skeleton, database, or documentation. Interoperability is the main benefit of following a rigorous standard.

Formal methods

- *Formal Methods* is a class of techniques for assessing correctness of a system revolving around the modelling of that system in a mathematical language.
- A software system is modeled in a formal logic with a finite state machine (or similar), and the specification is a set of logical formulas describing requirements of the system.
- *Formal verification* or *model-checking:* Given state machine M along with specification h, algorithmically determine whether or not the behavior of M meets the terms of h.
- Determining whether the formal specification captures the intended requirements is called "pleasantness" and is outside the scope of formal methods.

Benefits of formal verification

- Formal specifications can be model-checked programmatically to verify that they meet certain properties. Enables automation.
- Increases confidence in software reliability.
- Rigor of formal spec necessitates precise design, which helps catch design flaws and ambiguities.
- Gives a machine-readable reference for the system's behavior.
- Test cases and some documentation can be automatically extracted from a model - and can be updated automatically on program change.
- Some form of formal verification required for some certifications.

Drawbacks of formal verification

- Cumbersome due to advanced math, and harder to write than English + code, and is inaccessible to much of the team.
- Limited compatibility matrix of programming languages, formal specification languages, and model-checking tools compounds complexity and learning requirements.
- Specifications of systems of any complexity quickly become impossible to model or comprehend.
- Errors (or just excessive abstraction) in the model may lead to false expectations about the product.
- No guarantee of providing actionable proof or finding errors that wouldn't have been found otherwise.