CSE 110 Notes

1 9/27

Why is it so hard to properly get a software project done?

- Scale - larger projects require more time; the longer it takes, the less likely the estimated time will be accurate
  - More likely to be cancelled
- Misunderstood and changing requirements - if software is already in operation, the cost to change it is higher

Class is meant to help deliver larger and better quality software projects
Quality control - early manufacturing revolved around:

- Inspecting the product
- Fixing the product
- Reworking the production line

Led to Process-centric quality control
Still test the product, but also measure the process elements
Use cause-and-effect model to adjust production process
Statistical Process Control (SPC) - use statistics to track production variation
SE is Process-centric

What is a Software Process?
Produce quality software - what the customer wants, on time, under budget, no flaws
Steps include planning, execution, and measurement of product and process, and improvement

Discusses techniques for managing scale and risk/uncertainty
Process is just the beginning; also about quality decision-making
Needs good architecture, design, teamwork, and quality assurance

How to built what is needed, vs what is thought to be needed? Through frequent iteration and feedback from users
Robust code through good design and architecture

Project is self-decided
As long as choices make sense, you can get an A
Each student is graded on contributions to the team
Wisdom is better than quantity

Goals of the course:
Work effectively in a team using Agile development process
Design and document software systems according to stakeholder needs
Implement and debug complex software systems
Think about tradeoffs and risks

Courses need to teach technologies and principles, but principles need to be taught in context
Lecture focuses on principles, lab focuses on technologies

Team project: you choose the requirements
TA to manage project
Graded on ongoing quality and progress
Submit peer feedback every week, and TAs will give scores in Independence, Teamwork, and Technical contributions
Focus groups to provide insight regarding what users want
Vision document to say what you plan to create
A **mockup** will show the application in detail
In **five sprints**, build the app

2 9/30

Requirements Engineering - trying to figure out what is needed and record that down to communicate to the team
Goal is to elicit requirements from stakeholders through interviews
Write user stories according to the INVEST criteria
Stakeholder: ”person with an interest or concern in something, particularly a business”
Methods:
Survey – might not know what to count
Interviews – better
Focus groups – more participants, less time, but quiet people may not be heard
How to write questions:
Can cause biases if the wording is not neutral
Needs to be simple, open-ended, speak the user’s language, and ask for demonstrations or recall of concrete events
Recording data: write notes, or record audio and transcribe
Rapport: Be non-judgmental
Conducting the interview:
Start with easy questions
Listen and provide opportunities for the interviewee to continue
Ask for clarification
Express requirements as User Stories - ”As a [role], I want [something] so that [need]”
User story criteria: INVEST
Independent, Negotiable, Valuable, Estimable, Small, Testable
Independent: want to implement requirements in any order - helps with collaboration
Negotiable: can be changed during development; user story doesn’t specify everything
Valuable: provides value to the user
Estimable: keeps the size small, need to complete user story in 1-2 weeks or less
Small: Fit on a 3x5 card, at most two person-weeks of work – too big means unable to estimate, can’t finish in time for delivery
Testable: can be tested to see if it is done or not

3 10/2

What makes software great?
User stories aren’t enough – lots of ways to write the code
How to figure out which implementation is desided?
Goal is to learn *abstraction* - key to writing good software
Learn key principles for making software maintainable
Quality Attributes:
Express ”non-functional” requirements

Not what the system should do, but how it should do it
Examples: modifiability, maintainability, performance, robustness
Good design promotes some quality attributes, sometimes at the expense of other
High level design is called "architecture"
SOLID Principles for Design:
Single Responsibility Principle (SRP): a class should be responsible for one thing

- Thing, capability, computation
- Object only does its own calculations
- Don't cram related functionality into one class
- Helps know where to find code and prevents propogation of mistakes

Open/Closed Principle: a class should be open for extension, but closed for modification

- Can extend class without modifying it

Liskov Substitution Principle: properties of a class should hold of subclasses

- Subclasses should be able to be used in place of the superclass

Interface Segregation Principle: clients shouldn't have to implement interfaces/depend on methods they don't use Dependency Inversion Principle: high-level modules should not depend on low-level modules; both should depend on abstractions

- Abstractions should not depend on details; details should depend on abstractions
- Goal is to avoid tight coupling

DRY: Don't Repeat Yourself
Each thing or computational idea should be expressed just once in the code
Violations are the result of copy-pasting code or incomplete classes (violate SRP), but also over-specialization of classes

4 10/4

React
Learn the HTML Document Object Model (DOM)
What are web apps? Website that changes depending on what the user does - dynamic web site
Content is interactive
Webpages are trees
Old way was to remake the tree and reload the page
Modern way is to manipulate the DOM
React: don't write HTML directly, write code that emits HTML
React is a framework
A software framework is an abstraction in which software, probiding generic functionality, can be selectively changed with additional user specified code
Instead of writing code in order, write the functions, and framework calls functions in the right order
Frameworks are "opinionated" - designer of a framework has a way they want you to write code
Benefits of a framework:
Provides useful defaults, standard behaviour Downsides:
Have to understand how the framework works
Knowledge will be incomplete

5 10/7

Design Patterns
Same problems in multiple contexts
”each pattern represents our current best guess as to what arrangement of the physical environment will work to solve the problem presented. The empirical questions center on the problem—does it occur and is it felt in the way we describe it? —and the solution—does the arrangement we propose solve the problem?”
Object-oriented design patterns: so common
Each pattern solves certain problems, but not all problems can be solved with named patterns
Factory pattern:
Object needs to be hooked up, or which object to create depends on something
Repeating this pattern violates DRY, so put the logic in a ”factory”
Singleton Pattern:
Sometimes there should only be one of something - only one ”factory”
Other examples: logger, cache, thread pool
Use sparingly; a lot like global variables
Private constructors are a good example
Observer Pattern:
Multiple different ways to update one model - different controllers
Model stores state
Views need to know state, but the model shouldn’t have to communicate to the different views through controllers
Controllers have to know about the model, but the model shouldn’t have to know about the controllers
Model has a notification center, and the controllers **listen** to the notification center
Controllers are the Observers

6 10/9

Why study HCI? Software Engineering is about meeting user Needs
Building the wrong software is costly
Apply principles of HCI, iterate, hire a designer
Contextual design: People are experts at their own work, but can’t articulate their work process
Observe people in their work environment
Build personas to bring users alive
Personas build empathy, help you think like the user, but are not necessarily real people
Design a prototype with Figma
Principles:
Ways of assessing designs: usability studies, ”discount methods” (users are expensive)
Nielsen’s 10 Usability Heuristics:
From 1994, but still relevant
No need for user tests, instead, need an expert evaluator
Goal is to identify usability problems

- Visibility of system status: keep users informed about what is going on; progress indicators
- Match between system and the real world: speak the user’s language, follow real-world conventions
- User control and freedom: undo, redo, escape, exit
- Consistency and standards: follow platform conventions - people use other products more than yours
- Error prevention: eliminate error-prone conditions, check for errors, confirm before destructive actions
- Recognition rather than recall: minimize memory load, make objects, actions, and options visible

- Flexibility and efficiency of use: accelerators, shortcuts (hidden from novice users), allow users to tailor frequent actions: novices need discoverability; experts need efficiency
- Aesthetic and minimalist design: don't include information that is irrelevant or rarely needed
- Help users recognize, diagnose, and recover from errors: error messages in plain language, precisely indicate the problem, constructively suggest a solution
- Help and documentation: even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation: enable search in documentation, present documentation in context
  - Proactive help: provide help before the user gets stuck (tutorials): Push - not relevant to user goals, pull - relevant to user goals
  - Reactive help: provide help when the user gets stuck (error messages, FAQ)

7 10/14

Process:
Software Process: a set of activities and associated results that produce a software product
How to develop software?
Discuss the software that needs to be written
write some code
test the code
debug the code to find causes of defects
fix the bugs Productive development: (coding, testing, making progress towards goals)
Not spending effort into addressing inefficiences and following process leads to more work later on
Hypothesis: Process increases efficiency and flexibility
Ideal curve is upfront investment in process for later returns
Estimating time is hard
How to measure progress?
Developer judgement, lines of code, functionality, quality
Milestones and Deliverables make progress observable
Milestone: Clear end point of a (sub)task(s)
For a project manager; reports, prototypes, completed subprojects
Deliverables: Result for customer
Similar to milestones, but for customers
Waterfall Model:
Requirements, design, implement, verification, maintenance
Need to center around rapid feedback instead
Lean production: adapt to variable demand
Build only what is needed, when it is needed
Use "pull" system to avoid overproduction
Stop to fix problems to prevent defects from propogating
Workers are multi-skilled and can understand the whole process; take ownership
Just-in-time, DevOps, Shift-Left

Agile Overview:
Keep a prioritized list of user stories in a backlog
Product owner sets priorities in backlog
Divide work into sprints (2ish weeks)
Conceptually, at the end of each sprint, you could ship
The **scrum master** keeps progress on track; removes barriers to success

Sprint structure:

Start with planning meeting
Estimate user stories, commit to them individually
Daily standup meetings: what did I do yesterday, what will I do today, am I stuck
Then sprint review and sprint retrospective

Sprint Review:
For each user story, demo
If acceptance criteria are met, story is done, else it goes back to the backlog

Sprint Retrospective:
Discuss how the sprint went, refine iterations, processes, tools; identify and solve problems; decide on changes to improve effectiveness

8 10/16

Testing:
Defining correct behaviour
Purposes of testing:
Find bugs
Prevent bugs from sneaking in when enhancing
Give high confidence in correctness
Explore class/method design
Specify expected behaviour
Three big ideas:
Coverage: seek to execute all possibilities
Test equivalence classes: tests should all cover different things
Bottom-up testing: test the smallest things first, when testing things you shouldn't have to test the components
Black box vs White box testing:
Black box: test the software without knowing the internal structure
Not biased by implementation details, can't leverage opportunities
White box: test the software with knowledge of the internal structure
Can exploit bugs, will miss "impossible" bugs
Agile Testing (hierarchical, diverse):
3 kinds of tests, bottom up:
Unit tests: test individual classes and methods (task level)
Integration tests: test how classes work together, BDD personality tests (story level)
End-to-end scenario tests: (milestone level)
Mocking as a way to test
Testing can only show the presence of bugs, not the absence
Fuzz testing: security and robustness
Bugs don't always result in crashes
Automatic oracles: sanitizers
Pros: cheap to generate inputs, easy to debug when issues are found
Cons: hard to know when to stop, hard to know if you've found all the bugs, random inputs don't make sense

9 10/18

Software Architecture:
High-level design of a software system
Structure of the system, includes software elements, externally visible properties, relationships between them

Two kinds of requirements:
Functional requirements: what the system should do
Quality attributes: the degree to which the software works as needed
Modifiability, performance, scalability, availabilty, usability, maintainability, etc.
Levels of abstraction:
Requirements: what needs to be done
Architecture: how it will be done, mid-level "what"
OO-Design (low level): mid-level how it will be done, low-level "what"
Views:
Different ways of looking at the same system
Why document architecture?
Blueprint for the system, helps with maintenance
Documents architect's knowledge/decisions
Supports traceability
Each view alights with a purpose
Only represents info relevant to the purpose
Different views for different reasoning models
Module views (static):
Shows structure defined by the code
Modules (systems, subsystems) and their relations (dependencies)
Often shows decompositions (what contains what) and uses (what calls what)
Composition: has to include the other
Aggregation: can exist without the other
UML for the diagram
Component views (dynamic):
Shows entities at runtime
Components (processes, runnable entities) and their connectors (data flow, messages)
Do not exist until program runs, can't be shown in static view
Physical view (deployment):
Hardware structures and their components
What parts of the system run on which physical machines
How do those machines connect

10 10/21

Reminder that software architecture is about promoting quality attributes, sometimes at the expense of other quality attributes A *style* is a class of architecture with a typical structure

- Pipes and filters: "Data flow" family of styles
  - Ex. Compilers - language source code to compiler to intermediate code optimizer to machine code
  - UNIX Pipes: filters are processes (different ports), pipes are buffered streams
  - Difference with procedures - control is asynchronous and data-driven rather than synchronous and blocking; semantics are functional rather than hierarchical, data is streamed rather than parameter/return value variations include buffering, EOF behavior as opposed to binding time, exception handling, polymorphism
  - Analysis: promotes modifiability (can insert/remove filters, redirect pipes), reuse, and performance (parallel computing)
  - Inhibits usability (harder to build interactive apps this way), performance (translating data to send through pipes), cost (writing filters complex due to common pipe data format), correctness (synchronizing across pipes)
- Layered Styles: layers of useful systems, basic utility, and core level

  - Ex. Internet protocol suite
  - Static entities (classes, modules)
  - Constraint: only invoke code at lower levels; variation is only next level down
  - Benefits: changes only affect layers above, not whole system, and can reuse (swap implementation of layer)
  - Considerations: hard to choose right layers, what layer does which code go in

- Tiers: organize clients and servers into tiers - provide services above, rely on services below
  - Seen in runtime view, as opposed to layers seen in module/static view
  - Promotes: security (user can't access data directly), performance (separate tiers on separate hardware), availabilty (replicate tiers)
  - Rules: Each componenet in exactly one tier, can use services in lower tier/next tier down, depending on variation, can or can't use components on same tier
  - Advantages: tiers reflect clean abstractions, promotes reuse
  - Disadvantages: unclear which tier components belont to, what if they fit multiple, performance implications motivate inappropriate connections around layers (tunneling)
  - Tunneling: connection between layers not next to each other - violates layering, but is convenient and can improve performance

- Client-Server Architecture: clients know who the server is, server knows little about clients with previously agreed upon protocol
  - Promotes: scalability (easy to add more clients, servers), modifiability (can swap out clients and servers separately)
  - Inhibits: reliability (server/network may be down), performance (network bandwidth, latency), security (open ports), simplicity (more failure modes to test)
  - Publish-Subscribe style (Implicit Invocation): publishers don't know who is subscribed
    - Benefits: decouples publishers from subscribers, promotes reuse because adding a component just requires registering it for events
    - Problems: Order of event delivery not guaranteed; causes bugs when order is depended on; choose between synchronous or asynchronous event processing

11 10/23

Focus on Modifiability: before deployment
Responsibilities: action, knowledge to be maintained, or decision to be carried out by software system or element of the system
Responsibilities are assigned to modules; what is the cost of modifiying a responsibility?
Coupling: cost of modifiying module depends on how tightly-coupled it is to other modules
Reducing coupling may reduce modification costs - minimize relationships among elements not in same module, maximize relationships among elements in same module
Cohesion: put related responsibilties in the same module
Maximize cohesion and minimize coupling
Tactics:
Reduce cost of modifiying a singe responsibility by splitting it
Split responsibility so the new modules can be modified independently; enables deferred binding (replacing a module at runtime)
Increase cohesion by maintaining semantic coherence and abstracting common services
More responsibilities from one module to another, put related responsibilities together

Reduce coupling by using encapsulation, using a wrapper, raising the abstraction level, using an intermediary, or restricting communication paths
Encapsulation: hide information with an interface
Wrapper transforms invocations, difference between wrapper and encapsulation a little fuzzy
Raise abstraction level by adding parameters to interface; makes module more abstract, enables flexibility
Intermediary: restricts communication paths, breaks direct dependencies by adding one in between

12 10/28

Code Review:
Why Code Review?
Testing has limitations: costly to get 100% coverage; use 80/20 rule
Not all properties can be checked at runtime: good design, simple implementation, understandable code, follows coding conventions, UI looks as intended, follows UI guidelines, are tests adequate?
Code review: systematic reading/examination of the code
Focused on what can't be tested
Usually done asynchronously, at least one is non-author, work through complex problems like design
Pair programming is instantaneous code review
Bus factor: how many people need to be hit by a bus for the project to fail
Previously: formal code review (inspection)
Sit in meeting, read all code - effective at finding bugs, too slow for practical use
Modern code review: change-based code review - every change gets reviewed by someone
Purposes of code review: find defects, code improvement, alternative solution, knowledge transfer, team awareness, improving dev process, share code ownership, avoid build breaks track rationale, team assessment
Code review at Google: each directory is owned by some people; owners must review and approve changes
"Readability": some developers are "readability" certified - each change must be reviewed by someone certified in that language
Create a change, preview results of static analyzers, reviewers write comments, unresolved comments must be addressed, feedback is addressed, and reviewers mark as "LGTM" (looks good to me)
Few commits, few number of lines modified per change
Split changes into multiple smaller commits
Tone: code is team's code, not the author's code; use "we" language, not :you" language
Avoid blame - use "what if"s Subject of the review is the code itself, not the design
Questions are not very helpful in code review
Most useful comments:
Identification of functional issues (rare, bery useful)
Validation issues, corner cases
For new developers: API suggestions, design ideas, coding conventions
Somewhat useful: nit-picking (identifier naming, comments); refactoring ideas
Not useful: questions, future tasks
Systematic review:
Checklists: what to look for
Specific techniques for specific issues
Use tools in GitHub or IDE

13 10/30

Integrating security into the development workflow, and technical risks and mitigations
Old methods: write code, give to security people, then let operations people host it
Usually too late to fix security issues; security has architectural implications

DevSecOps: integrate security into the development workflow
Kinds of security challenges:
Undefined behavior - don't use unsafe languages
Incorrect security-related code - review, test, control changes
Higher-level design mistakes - architectural review, penetration testing
Users (social engineering) - HCI techniques, training, compromise procedures
Train, define security requirements, define metrics and compliance reporting, use software composition analysis and governance, perform threat modeling, use tools and automation, keep credentials safe, use continuous learning and monitoring
Security requirements:
Legal and industry requirements
Internal standards and coding practices(strcpy)
Review of previous incidents and known threats
Traditional requirements analysis with security focus
Metrics and Compliance reporting:
How will you know whether you've succeeded? Does one breach mean you've failed? Better to focus on progress than success/failure
Threat modeling:
Goal: enumerate all possible threats; use STRIDE model to identify threats
Spoofing identity: attacker pretends to be someone they are not
Tampering with data: changing data in the system
Repudiation: lying about something done
Information disclosure: unauthorized access to data
Denial of service: making the system unusable
Elevation of privilege: gaining access to something you shouldn't have
Software composition analysis and governance:
Vulnerabilities in third-party libraries
Use Tools and Automation:
Integrate tools into CI/CD pipeline
Must not require security expertise
Must avoid a high false-positive rate
Keep credentials safe:
Scan for keys in source code
Put keys in a .env file
Put .env in .gitignore
Use continuous learning and monitoring:
Continuous integration/delivery
Run analyses automatically
Mean time to identify, mean time to contain
Top 10 Threats (OWASP)
Broken access control, cryptographic failures, injection, insecure design, security misconfiguration, vulnerable and outdated components, identification and authentication failures, software and data integrity failures, security logging and monitoring failures, server-side request forgery
Mitigating key threats:
Untrusted Data:
Injection attacks: validate input, avoid eval(), sanitize inputs when constructing SQL queries
Cross site scripting attack (XSS): untrusted data enters app, data included in content sent to user
Bad authentication: use TLS to check server certificate, check user credentials
Authentication vs Authorization:
Authentication: are you who you say you are?
Authorization: given who you are, are you allowed to do what you're trying to do? Needs access control
letsencrypt.com: free certificates
Password cracking:
Brute force to try all strings: use large space of passwords, avoid commonly used passwords

Rainbow table attack (precompute hashes of common passwords): use large space of passwords, avoid commonly used passwords
Salts: users with same password have different salts, so same password doesn't have same hash; random salts added to user passwords
Passwords do not go in the repo or database! Passwords in config files, hashed passwords in database
Principle of Least Privilege:
Only authorize access that is needed
Defense in Depth:
Don't have just one security check; encrypt traffic and restrict access by VPN

14 11/4

Debugging large software systems:
Avoidance path: some bugs can't be fixed by delivery date
Sometimes bugs are encoded in the specifications
Either change the code or change the specifications
Judge costs, assess risks, and measure severity
Fixing bugs risks introducing regressions
Bug reports shoud show how to reproduce bug, what the observed behavior is
Two phases: fault localization (where bug occurs) and fault repair (what to do about it)
Be deliberate: one bug at a time, write test cases, commit after each bug
Fault localization is most of the work
Come up with hypotheses, test them according to likelihood and ease of elimination
Goal is to fix the bug while understanding no more than necessary; assuming that reading all code is impossible
Test case minimization:
Remove all elements of the test case that are unnecessary to reproduce the bug
Narrowing down the responsible code:
Replace modules with mock modules that do the right thing
Try to show the bug is in the framework you're using, or you learn what the bug actually is
Divide and conquer:
Bug is either because component doesn't do what it's supposed to, or it's not what some other component needed
Be explicit about assumptions (preconditions), and expectations (postconditions)
Regressions: use git bisect to fix
Bad state: after doing X, some state is wrong.
Sprinkle assertions throughout code for X; find exactly where the state is wrong
No clue where to start: search code for relevant words, ask expert, git blame
Unusual situations:
Heisenbugs: bugs that disappear when you try to debug them
Usual suspects: race conditions (timing matters), compiler optimizations, hardware failures
Careful recording: can't keep track of everything in your head:
Record each hypothesis, test inputs and results and conclusion
Change one thing at a time

15 11/6

Risk is essential to software engineering: every bug costs more to fix later, but fixing bugs may introduce new bugs
Technical risks: choosing to rely on a framework that's "almost done" but runs late; rely on a platform/service/framework that does not meet needs /adds complexity without adding value; underestimate the

complexity of your own components
Financial risks: running out of money (startup), getting sued, need to give raises for retention but now can't afford to hire more people
Requirements risks: releasing software that does not meet user needs (even if it's high quality), or if it frustrates users, or if it is released too late
People-related risks: by choice (better job offer), by circumstance (disaster), being fired, being stolen by Management to work on a higher priority project
Management-related risks: management changes priorities, de-prioritizes features you invested in, prioritizes features you didn't invest in, management turnover (spends time managing up (managing the manager))
Osborne effect: announcing a product too early, causing sales of current product to drop
Mythical Man Month: book by Fred Brooks
Brooks' Law: Adding people to a late project makes it later
New people consume resources getting up to speed, introduce new bugs, reintroduce old bugs, increases communication overhead
Group intercommunication formula: n(n-1)/2
Need to keep teams small
Second System Effect: the first time you design something, you know you don't know what you're doing
The second time, you think you know, and you fix all the things that were wrong the first time
The second system is the riskiest
Incremental slippage: continually delaying by a day causes a year delay
Awareness-Understanding matrix:
Aware + Understand: Known knowns: things you know you know
Aware + Don't Understand: Known unknowns: things you know you don't know
Unaware + Understand: Unknown knowns: things you aren't aware of but you know implicitly
Unaware + Don't Understand: Unknown unknowns: things you don't know you don't know
Inherent vs Accidental complexity: complexity is a source of risk
Inherent complexity: complexity that is inherent to the problem - tax software has to be as complex as tax law
Accidental complexity: complexity that is not inherent to the problem - making the problem harder than it should be
Conway's Law: Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations
If you have 3 teams working on a project, there will be 3 separate modules
Organizational structure poses architectural risks
Surfacing risks: ask team members what might go wrong (diverse team more likely to identify risks), make mitigation plans

16 11/8