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