CSE-110 Final Review

This version is created by FTP. This extends the previous collaborative review doc. It synthesises every possible detail covered in Gary's lecture slides, past quizzes (including those from quizlet), website's artifact sections, agile videos, labs and design pattern textbook. **To get perfect scores on Gary's CSE110's bi-weekly quizzes,** refer the the last page.

Official Review Guide

- Old quizzes
- Design patterns and design principles (definitions and basics of functionality)
- End of chapter bullet point pages
- Testing
- Cohesion and coupling
- Agile and waterfall
- Financial lecture focusing on high level ideas

Resume

- Purpose: biased representation of yourself to get an interview
- Audience: to the employer
- Content: items where you can impress
- Teasing: keep engaged

- Layout:
    - Top ¼
        - <u>Introduction:</u> name, address, email, phone, professional link
        - <u>Objective:</u> outcome you desire when giving resume (customized), show desire to contribute.
        - <u>Education availability:</u> graduation data determines year, assessing experience level and availability.
        - <u>Skills:</u> table of contents
    - Mid ⅔ - 1/4:
        - <u>Experiences</u>: focused on transferable skills, match objective, know your point.
    - Final:
        - <u>Polish</u>: accomplishments, hornors.

Interview

- Mutual: you are interviewing the company the same time they interviewing you.

- Goals:
    - Engage interviewer
    - Display thought process
    - Show interest and attitude
    - Demonstrate what a work day be like with you in it
    - Get a follow-up interview
    - Get a position

- Interviewer
    - Determine if they want to spend more time with you than their family
    - Determine if they can take a chance on you to complete the tasks

- Rejection
    - All they can say is “no”
    - Have a come back response
    - Don’t be salty

- Acceptance
    - Stop interviewing
    - Other jobs might look better.
    - Don’t renege and wait for a year.
    - If you want something better, then go.

<u>Engineering Projects</u>

- **M**inimal **V**iable **P**roject: prioritize essential features

- **The most important quality of a project: USEFUL**

- First and second questions
    - What is the problem
    - Who is the customer

- Three (or four) variables:
    - Time
    - Resources
    - Scope (functionality)
    - *Quality*

- One constant
    - Change

- Three unwanted
    - Hero
    - Miracle
    - Surprise

- Age of interruption:
    - **O**nly **H**andle **I**t **O**nce

- Two methodologies
    - Waterfall: vintage, more formal
    - Agile: newer, less document, more customer interaction, based on a rugby tactic

<u>Team Building</u>

- Professional interaction
    - Rational, brief, beneficial.
    - Consider before start: **what is the goal**.

- Make mistakes
    - Own up right away

Project Manager	Responsible for the entire project
Senior System Analyst	Coordinator of use case, user stories and requirements
Database Specialist	Maintains database and related issues
Software Architect	Coordinator of design and selector of technologies.
Software Development Lead	Coordinates software development
Business Analyst	Researches the content of application and how business rules are integrated into the project
UI Specialist	Focus on the view, feel and user experience
Algorithm Specialist	Designer of algorithm and module interfaces

Quality Assurance	Coordinates testing phases and ensures procedure followed.

- **Leadership**:
    - Everyone is a leader and everyone is a follower
    - Each week's artifact is led by role
    - Qualities:
        - GOOD: Honest, Responsive, Responsible, Patient, Parental
            - Earns and maintain the respect of team members
        - BAD: Didn't follow through.
    - Criticism:
        - Welcome it
        - Be **tactful and respectful**
        - Don't be **defensive or offensive**
        - Look up the source
        - Don't embarrass your teammates in front of others
        - **Correcting yet motivating**
    - Team member:
        - Accept the teammates you have:
            - don't expect them to be who they are not.
            - don't taunt them.
        - Have collaboration area**:** github, dropbox, google drive.
    - Team meeting:
        - Productive time use: respect everyone has other things
        - Agenda:
            - information to discuss
            - decisions to be made
            - follow up issues
        - Record**:**
            - who are present
            - what's announced
            - decisions made
        - Allow team members to speak:
            - What you've been doing
            - Acknowledge accomplishments of members

- Project Manager: responsible for the project
    - **Delegate to be most productive**
    - Aware of all aspects of the project
    - Allocate resources for team to accomplish tasks
    - Has a final say
    - Take ownership

- Supports the team members
- Earns the respect of the team
- The go-to boy or girl
- Deputy as backup

- Team members
    - **Responsibility**
        - **Look at job title: take initiative, volunteer**
    - Accept tasks from managers / requests from teammates
        - Tasking done with tact
        - Manipulation vs persuasion
        - Goal of interaction: **beneficial**
    - Team discussion
        - **Recognize the time for input**
        - **Recognize the time when decision is made**
        - **Sideline unresolved issues**
    - Team members
        - Support members and PM
        - Receptive of requests made by others
        - Offer help when you can.

- Trust and betrayal
    - Trust is earned
    - Betrayal is trust violated
    - Teammates are your **colleagues**, not friends.

- Dispute resolution
    - Respect
    - Logic and reason should prevail over loud voices and profanity

- Benefits of blame: **none**

- **Pigs and Chicken**
    - Pig:
        - totally committed to the project and accountable for its outcome
        - sacrifice for the team by providing bacon
    - Chicken:
        - consult on the project and are informed of its progress
        - Provides non-sacrificial eggs

<u>Software Development Methodologies</u>

- Software development cycle
    - Requirement
    - Design
    - Code
    - Unit testing
    - Integration testing
    - Acceptance / formal testing
    - Maintenance

- Risk Matrix
    - Items of uncertainty listed to avoid surprise

- **Agile**
    - Iterative and incremental development
    - Requirements and solutions evolve via collaboration between **cross-functional, self-organizing** teams
    - **Flexible** responds to changes
    - **Adaptive planning** rather than predictive planning
    - Software delivery is a measure of progress
    - “Spike testing”:Doing a burst of activity, seeing how much you get done, using that to estimate how much time it'll take to get everything else done

    - <u>Benefits</u>:
        - Adaptive methods focusing on adapting quickly to changing realities
        - Best for developmental and non-sequential projects

    - <u>Drawback</u>
        - Difficulty in describing what will happen in the future
        - Lack of scalability and design flaws
        - Insufficient training cited as most significant cause for failed agile projects
        - Team members get boxed into different roles preventing cross training
        - Teams not focused to meet project commitment
        - Lacking test automation
        - Allows technical debt to build up if only focusing on increased functionality
        - Insufficient for large projects

- **Scrum** (An Agile framework)
    - Team roles
        - Product owner: the voice of customer
        - Scrum master:
            - **Facilitator-in-chief**
            - Schedule needed resources (human and logistical) for sprint planning, stand-up, review and retrospective

        - Resolve impediments and distractions for the team
        - NOT PM
    - Scrum team:
        - Developers, QA, designers, etc.
- **Sprint**: Basic unit of development of a fixed period of time.
    - Typically two weeks
    - <u>Planning meeting:</u> (start)
        - Tasks identified
        - Estimated commitment of goal
    - <u>Daily scrum meeting:</u> (stand up)
        - What did you do
        - What do you plan to do
        - What impediments you have
        - No longer than 15 minutes
    - <u>Sprint review/retrospective meeting:</u> (end)
        - Progress reviewed
        - Lessons identified
    - <u>Result:</u>
        - Working product: integrated, tested, documented in Sprint.
- **Sprint backlog**: items (user stories) needed to be done, prioritized by risk, business value, dependencies.
    - Prioritized feature list, containing short descriptions of all functionality desired in the product.
    - Product owner's assessment of business value
    - Developer's assessment of effort
    - Scrum starts with backlog
- **Velocity**: number of units of work / interval
- **Burndown chart**: chart of work to do versus time, updated daily. (how much work is remaining to be done)
- **Burnup chart**: work completed and total amount of work versus time, updated daily. (how much work has been completed)
    - Project scope
    - Planned progress
    - Actual progress
    - X-axis as iterations, y-axis as story points.
- **Success slider:**
    - To define the success of the project: scope, time, quality, cost
    - Adjust these four as appropriate, to get an idea of what your project will need.
    - the sponsor and product owner need to be involved
    - Slider between fixed and flexible.
- The manner of restarting after minor infraction
- Key principles:

        - Customer can change their minds about what they want during development process. (**requirement churn)**
        - Unpredicted challenges are hard to address in planned manner
        - Accept problems can't be fully defined
        - focus on team's ability to deliver quickly.

- **Waterfall**
    - Project is divided into **sequential phases**, with some overlap and splashback acceptable between phases
    - Emphasis on **planning, time schedule, target dates, budgets and implementation** of an entire system at one time.
    - **Tight control** is maintained over the life of project via
        - extensive **documentation**, **formal reviews**
        - **approval/sign off** by the user and IT management occurring at most phases before beginning the next phase.
    - Eight phases:
        - Conception
        - Initiation
        - Analysis
        - Design
        - Construction
        - Testing
        - Production/Implementation
        - Maintenance
    - <u>Benefits</u>
        - Time spent early in software production cycle can reduce costs at larger stages
        - Well suited for projects where
            - requirements and scope are fixed
            - product is firm and stable
            - technology is clearly understood.
    - <u>Drawback</u>
        - Hard to change direction if planning goes wrong
        - Clients may **change their requirements**
            - Clients may not know exact requirements until they see working software
        - Changing requirements leads to **increased costs**
            - redesign, development, retesting, long development to market timeline
        - Designers may not be aware of **future difficulties**
        - Project stakeholders may not be fully aware of the capabilities of the technology being implemented
        - Impossible to perfect one phase before moving on to the next phase

- **Lengthy delivery cycle**.

<u>Layered Architecture</u>

- **M**odel **V**iew **C**ontroller
    - Software architectural pattern for implementing user interfaces.
    - Model: directly manages data/logic/rules of application
        - Should be **lightweight**
    - View: the presentation of data to user
        - Should be **stupid**
    - Controller: the interaction between above two

- Presentation Layer: layer of code **processing input** from screens.
    - Form classes:
        - performs form validation **without** database access
        - Send flow of control to action classes to process data or error handling
        - Ex: AddUser in UserForm class (validate, call action)
    - Action classes:
        - perform user requested action with valid data
        - Ex: AddUser in UserAction class (collect, process, call dispatch)

- Business Logic: **high level functionality** invoked from presentation layer
    - Dispatch classes:
        - performs validation **with** database access
        - Send flow of control to manager class to process data or error handling
        - Ex: AddUser in UserDispatch (validate authority, business logic, call manager)
    - Manager classes:
        - Manages data access objects
        - Ex: AddUser in UserManager class. (No validation, call DA methods)

- Data Access: **let ow level database interface** methods invoked from Manager layer.
    - Data Access Object (DAO classes)
        - AddUser in UserDAO class (add to database, no validation)

- Database: database connectivity code.

- <u>Benefits:</u>
    - Simplifies design considerable
    - Enables different roles to effectively work at different layers of abstraction
    - Supports portability of software artifacts

Artifacts

- **U**nified **M**odeling **L**anguage: standard way to visualize the design of a system.

- THEY: gender-neutral singular pronoun for a known person, as a non-binary identifier

- **Use Cases** (waterfall)
    - Detailed description and the steps involved with a user's interaction with application on how it provides one specific functionality **without specifying technology, implementation or specific user entry**.
    - A use case needed for each **complete idea implemented** by the application. A user would buy or use the application due to the functionality described by use case.
        - Display home screen is not a functionality
        - Login into system is a functionality (provides benefit)
        - Store to database is behind the scenes
    - Start with: the user shall, the system shall.
    - Components: priority, status, description, user goal, desired outcome, actor, dependent use cases, requirements, pre-condition, post-condition, trigger, **workflow**, alternative workflow.
    - Content
        - Start with hyperlinked table of contents and a legend of terms
        - Followed by overview essay
    - Need **requirement**

- **System Requirement**
    - The actions that a software system is required to accomplish. The functionality that a system is required to provide
    - Desired Qualities:
        - **Atomic**, **Testable**
        - Unitary: address only one thing
        - Complete: fully stated in one place
        - Consistent: don't contradict any other document or requirement
        - Traceable, current, unambiguous, specify importance, verifiable.
    - Bad quality:
        - Vague terms: not testable
        - **Extensible, polymorphic**
        - If, when, why.
    - Start with:
        - Functional: the system **shall**
        - Non-functional: the system **shall** be
    - Should be:
        - Cross referenced to original use case

    - Numbered as subsystem based to be able to easily extend.
  - Functional requirement:
    - Describe **a function user want**.
    - Especially for system, can't require from user
  - Data model / Business rule requirement:
    - Defines the **fields of an object**
  - Needed by use case. Often derived from steps of the Use Cases.

- **User Stories** (agile)
  - An informal, natural language description of one or more features of a software system.
  - Capture the description of a software feature from an end-user's perspective. Short description of something that your user will do when they come to your website.
    - Display home screen is not a functionality
    - Login into system is a functionality (provides benefit)
  - Described
    - Type of user
    - What they want and why
  - Format
    - As a <role>, I can <capability>, so that <receive benefit>.
  - Benefit
    - Facilitate sense-making without undue problem structuring.
  - Limitation
    - <u>Scale up</u>: difficult to scale to large project
    - <u>Vage, informal, incomplete</u>: do not stage all the details necessary to implement a feature.
    - <u>Lack of non-functional requirement</u>: rarely including performance or non-functional requirement details.
  - **Acceptance test** is needed to complete a user story
  - Should have
    - Glossary of terms
    - Numbered user stories
  - **M**ust **S**hould **C**ould **W**on't have
    - Prioritize user stories
  - No requirements needed

- **Design Use Case**
  - Use case from the perspective of what the system needs to do to achieve the functionality described in corresponding use case.
  - Main difference are the **workflow**:
    - Include implementation details
    - List what needs to be done for each step in use case

- Examples:
  - **C**reate **R**ead **U**pdate **D**elete
    - Four basic functions of consistent storage
  - Validation
  - Creation
  - Screen pages to display

- Use Case Diagram

- Database schema
  - Blueprint of the database
  - Structure described in a formal language
  - Supported by the database management system.

- System Test Plan
  - Test cases match use case
  - Acceptance tests match user stories
  - Test Case workflow
    - Written from user's perspective
    - Indicate **exactly** what to enter
    - Need to demonstrate functionality
  - Acceptance Test workflow
    - General instructions in steps
    - General description to determine passing
    - Can be explicit

Design Principles

- Software project can either be object-oriented or aspected-oriented, or both at the same time.

- **A**spect-**O**riented **D**evelopment
  - Code that spans/touches all aspects of the project (supporting functionality).
  - **Separation of concerns**
    - Separating a computer program into distinct sections
    - Sections can be reused, developed, updated independently
    - **A program that embodies a SoC is Modular**
  - Ex: calls to the audit system
    - Made from most manager class methods
    - Maintain table of state transitions for messages and users.
  - Ex: error processing

        - Made from most methods
        - Exceptions thrown when encountered
        - Exceptions caught at central location

- **T**est **D**riven **D**evelopment
    - Key ideas:
        - **Failing test is written to verify new functionality**
        - Code is **only written to pass tests**
        - Code is refactored into code base with acceptable standards
    - Steps: fail, pass, refactoring.
    - <u>Benefits</u>:
        - More tests are written
        - Programmers tend to be more productive
        - Less need for debuggers
        - Resultant codes are ized, flexible and extensible.
        - Automated test cover every code path.
        - Drive design focusing on customer interaction with system
        - Limits defeats shortening implementation time
    - <u>Drawbacks:</u>
        - Difficult enough to use when functional tests **determine success/failure of the system**
            - **User interfaces**
            - **Database backends**
            - **Network configurations**
        - **Not all managers are believers**
        - Refactoring or design changes may result in many changes in test
        - **Developer and tester the same person leads to blind spot**
        - **Passing tests creates false sense of security**

    - True/false question :
        - When the developer and test author is the same person, TDD leads to the same blind spots in test coverage. - T
        - Using TDD, writing code to implement new functionality is done only after writing a failing test verifying the new functionality is not working. - T

<u>Code Review / Software Inspection</u>

- **Code review**
    - **Systematic examination of computer source code**
    - Find and fix defects in code that are overlooked in initial development phase

- Software inspection
    - Peer review by trained individuals looking for defects using a defined process

- Inspection Roles:
    - Moderator: leader and coordinator of code review
        - NOT PM
        - Rotates among team members
    - Reader: reads the document aloud to group
        - Not the code author
    - Author: the person who created the code to review
    - Recorder: documents the defects found
    - Inspector: person examining the code

Pair Programming

- Roles:
    - Driver
    - Navigator
- Benefit:
    - Catch defeat quickly
    - Produce shorter program
    - More alternatives are considered
    - Arrive at more maintainable designs
    - Difficult problems are easier when worked together
- Drawback**:**
    - Time consuming
    - Net productivity drop in simple task
- **Promiscuous**:
    - Pair programming with the entire team, not a single partner
    - Result in team knowledge speed
    - Don't lead to higher error rate.

Industry Insights (Employment, Compensation, Tax)

- Working hours (official hours)
    - Typically 8am - 5pm with one hour lunch break
    - Administrative employees typically must follow.
        - Receptionist, secretaries, payroll, etc.
        - Services must be staffed during set times
    - Executives, managers, engineers typically don't follow:

        - Flexibility to come and go as long as jobs get done, hours are worked, present for meetings, co-workers know your schedule.
    - **40 hours** is week is full time (5 days * 8 hours or 9 hours * 9 days w/o Friday)

- Overtime
    - Exempt employees working over 40 hours a week or 8 hours a day
    - **Casual** overtime: unpaid time given to company
        - Sometimes expected, sometimes unusual
    - **Extended** work week: paid time for hours worked (rare)
        - Must be approved by management
        - Impact salary budgets
        - Begins after initial amount of casual overtime
        - Paid straight hourly rate
    - Sometimes forbidden
        - Violates the employment contract
        - Cause for termination

- **Compensation**
    - **Salary**: money paid to you for your service
        - Determined at time of hire
        - Adjusted at performance review
        - Paid a short delay after work performed
            - Weekly, alternating weeks, bi-monthly, monthly.
        - Exempt (No) or Non-exempt (Yes):
            - Eligible for overtime pay.
    - **Vacations / Holidays**: paid time off for recognized holidays
        - Each company has a list
        - Floating holiday: day off your choice
        - Comprehensive leave: doctor's appointment, sick time, combined.
        - Lump award at beginning of year or accrued over time
        - Duration extends with years of service
        - **The best time to take vacation: never**
    - Jury duty: fixed or unlimited number of days
    - Bereavement: on approval, fixed number of days
    - Tuition Reimbursement: on approval. Get A or B for job related classes.
    - **Bonus**: one time payment to recognize achievement, employee sign on or referral.
        - Cash, stock, stock options or combinations
        - **Golden Handcuffs**
            - Stock and stock options are often vesting
            - Money is yours if you stay long enough
    - Life insurance, legal services, discounts at amusement parks, etc.
    - **Employment Stock Ownership Plan (ESOP)**

        - Buying stock directly from the company through payroll deduction
        - Limit to percentage of salary
        - Defined enrollment period
        - Discounted price or with company match
        - Set price: private stock or public stock
    - Health/dental/vision insurance
        - Company purchases and/or contributes with employee contributions
        - Selections made during enrollment period
    - Optional FLEX spending accounts
        - **Pre-tax dollars** in an account used for eligible expenses
        - Contributions made in paycheck
        - Reimbursed in cash for expenses or` charge direct to account
    - Severance: payment when company terminates employment without cause
        - May be 0, 1 week salary, or 3 months salary
        - Depend on size of comp
        - any
    - Relocation: benefit for packing and moving to new city
    - Training: in-house or off-site training
        - **Brown bag** lunch: company provides lunch training on employee time

- **Stock Option**
    - Option to purchase a set amount of stock for a duration in the future at set price
        - Price established at time of stock option award
    - Exercise: purchase stock using your stock option
    - **Underwater**: price at purchase is less than stock option price
    - Tax basis: reset to value at date of purchase
        - Capital gain taxable in year of stock purchase
        - Sometimes taxed as ordinary income

- Employment policy
    - Employment “at will”:
        - employer is free to discharge individuals for any cause
        - the employee is equally free to quit, strike or cease work.
    - Probation employment:
        - Specified employment period with a termination
        - Must be stated in contract
    - Contract employees
        - Fixed contract duration
        - Often paid more due to no benefits (health care, retirement, etc)

- Performance review
    - Company vehicle for feedback, growth, salary adjustments, promotion
    - Typically done annually at your anniversary date of hire

    - Out of cycle still possible
  - **"Focal Point"** review: entire company does review the same time

- Performance review process
  - Employee phase:
    - Employee writes assesses goals made during the prior cycle
      - Initial goals done at hire
    - Employee writes goals for following year
    - Ethics statements are reviewed
    - Time Charging procedures are reviewed
  - Manager phase
    - Manager writes assessment of employee's goals
    - Manager approved goals for the following year
    - Manager writes performance evaluation listing strengths and areas for growth
  - Compensation phase
    - Salary adjustment announced with effective date in future
      - **Merit** – increase due to job performance or increased responsibilities
      - **Promotion** – increase due to increased responsibilities
      - **Equity** – increase to raise to market values: Amazon & Google: 10%

- **RETIREMENT OPTIONS**
  - Many plans allow loans against balance up to **50%**
    - Paid back with interest via payroll deductions
    - Early penalty applies if not paid back
  - **E**mployee **S**tock **R**etirement **P**lan and/or profile sharing
    - Company contributions to retirement fund
    - Earnings tax-deferred
    - % salary given to all employees eligible
      - May need to be employee for fixed hours, employed on certain time period, sale restrictions, etc.
      - http://ieng6.ucsd.edu/~cs110x/static/lectures_a00/CSE_110_Week_Five_Lecture_Two.pdf
  - **401k**
    - Traditional: employee **pre-tax** dollars diverted into investment account. Earnings are **tax-deferred.**
    - Roth: **post-tax** dollars diverted into investment account. Earnings are **tax-free.**
    - Withdrawn at age of **59.5.**
    - Employers match funds diverted?

  - Highly compensated employees **maximum based on average of deferrals** by non-highly compensated employees.
  - Maximum dollar limit per year, adjusted every year
  - Contributions through payroll deduction
    - Max contribution: 75%
    - Employee selects percentage
    - Enrollment / disenrollment period quarterly
    - **Front loading**: diverting higher % limit than annual allowance to reach limit sonner.
      - Reduce paycheck
      - Allows earlier investment choices than if equal allotments
      - Assurance of reaching limit in spite of employment conditions
- **403b**
  - Employee **pre-tax** dollars diverted into investment account for **non-profit** organizations.
- **457b**
  - Employee **pre-tax** dollars diverted into investment account for **governmental** employees
  - Limits same as 401k /403b in addition to their limits

- **Roth IRA** (Individual Retirement Account)
  - **$5,500** contribution to retirement account in **post-tax** dollars
  - Age 50+ add additional ($1000)
  - Grows **tax free**
  - Contribution limit dwindles as income rises
- **Traditional IRA**
  - **$5,500** contribution to retirement account in **pre-tax** dollars
  - Age 50+ add additional ($1000)
  - Grows **tax deferred**
  - Contribution limit dwindles as income rises if combined with 401k

- Required minimum distribution
  - Must start taking distribution on April 1 of year turning at 70.5
  - Amount is a percentage of the combined pre-tax retirement assets as of 12/31 of the prior year, percentage based on age.
  - 50% penalty for amount not distributed that should have been distributed
  - No distribution requirement on **Roth** assets.

- **A**djusted **G**ross **I**ncome
  - **S**alary + **I**nterest + **D**ividends +/- **C**apital Gains - **P**ersonal Exemption - **A**djustment
  - Personal exemption $4,050 (2016)

  - If your adjustments exceed your personal exemption: itemize them.
  - FICA: social security tax and medicare tax rate: 7.65% of AGI.
  - State income tax.

- **Tax forms**
  - W2: employer listing
  - W4: employee to determine withholding rates
  - 1099 forms
    - Report income other than wages
  - 1040 starting form for Federal Income Tax
    - Schedule A: deduction
    - Schedule B: Interests and dividends
    - Schedule C: self employment expenses
    - Schedule D: capital gains and losses
    - Schedule K1: partnership
  - California 540: starting form for California income tax

<u>Testing</u>

White	Grey	Black	All
unit test, developer integration test, system Integration test	System integration test, formal testing, verification and validation testing	Verification and validation testing, alpha (or beta) testing, acceptance	Regression testing

- **Unit testing**
  - Testing **individual units** of codes
  - Verifies individual components work individually
  - Performed **locally** by software developer
  - Usually performed along with coding, sometimes as an independent phase of development
  - JUnits support unit testing
  - Sometimes informal

- **Developer Integration Testing**
  - Testing **all code**
  - Verifies that all code works together
  - Performed **locally** by software developer
  - Usually performed as a separate phase of development
  - Sometimes informal, sometimes formal

- **System Integration Testing**
    - Testing **all code on development platform**
    - Verifies system works when installed **from scratch**
    - Non-code verification
        - Security, load testing, performance
        - Destructive testing: verification within allowable failure limits
    - Performed on **test platform** by software developers and **others**.
    - Usually performed as a separate phase of development
    - Sometimes informal, sometimes formal
    - Code usually requires **authorization** to make changes

- **Formal Testing**
    - Independent testing on **test platform**
    - Verifies system works when tested by **non-developers**
    - System **installed fresh**
    - Performed on **test platform** by **test engineers**
    - Performed as a separate phase
    - Code requires **authorization** to change
    - Formal **test plan** followed, **report** generated.

- **Verification and Validation Testing**
    - Independent testing on **test platform**
    - Performed by **outside agency or customer**
    - Performed as a separate phase
    - Code changes require **customer authorization**
    - Formal **test plan** followed, **report** generated

- **Alpha (or Beta) Testing**
    - Limited release to few **customer** on **production site**
    - Coordinated with customer
    - Performed by **end users**
    - Performed as a separate project hase
    - Code changes require **new build, formal testing approval, customer approval**

- **Acceptance**
    - Successful passing of alpha or beta testing

- **Regression Testing**
    - Verifies that **existing functionality** still works
    - Crosses all phases of testing

Coupling / Cohesion

- **Cohesion**
    - Measure of how strongly **related** is functionality in a source code module
    - **High Cohesion** (loose coupling) is **prefered**
        - Robustness, reliability, reusability, understandability
    - Low cohesion is undesirable
        - Functionality of class methods have **little in common**
        - Methods carry out varied activities on **unrelated data**
        - Difficult to maintain, test, reuse and understand

Coincidental	Only relationship is that they have been grouped together	A utility class
Logical	Logically categorized to do the same thing, even if differ by name	Grouping all input of mouse and keyboard routines
Temporal	Grouping by when processed in program execution	A function called after catching an exception that closes files, creates an error log and notifies the user
Procedural	Grouped because they follow a sequence of execution	A function that checks file permissions and then opens the file
Communicational	Grouped because they operate on the same data	Code which operates on the same record of information
Sequential	The output of one part is the input of another	A function which reads file and processes the data
Functional	Grouped due to contribution to a single well-defined task	Tokenizing a string of XML

- **Coupling**
    - Degree to which each program module **relies** on one of other modules
    - **Low coupling is preferred**
        - Sign of good design
        - Supports high readability and maintainability
        - Promotes the opposite of tight coupling disadvantages
    - Tight coupling:
        - A change has **ripple** effect of changes in other modules

- Assembly of modules requires more effort due to increased inter-module dependencies
- Reuse is difficult due to dependent modules that must be included

Content	One module modifies or relies on inner workings of another module	Accessing local data of another module
Common	Two modules share the same global data	Changing the shared resources means changing all modules using it
External	Two modules share an externally imposed data format, communication protocol or device interface	Modules that all take JSON files as input
Control	One module controlling the flow of another	Pass a what-to-do flag
Stamp / Data structured	Modules share a data structure and use only a part of it	Passing a whole object to a method needing one field
Data	Modules share data by passing parameters	Pass an integer to a function to compute square root
Message	State decentralization with communication done via parameters or message passing	Event driven code
No	Modules that do not communicate with one another	

Decisions and Ethics

- Etiquette
    - Expectation for social behavior according to contemporary conventional norms within a society, social class or group.
    - Hygiene Manners
    - Courtesy Manners:
        - put interests of others before oneself

        - Display self-control and good intent for the purpose of being trusted
    - Written and unwritten rules of conduct that make social interaction smooth.

- Guideline
    - Determine a course of action
    - Streamline particular processes according to a set routine or sound practice
    - Not binding and not enforced

- Policy
    - **Deliberate system of principles** to guide decisions and achieve rational outcome.
    - Statement of intent, implemented as procedure or protocol.
    - Generally adopted by governance body within an organization

- Law
    - System of rules
    - **Enforced through social or governmental institutions**
    - Regulate behavior

- Decision Making
    - Process of identifying and choosing alternatives based on the **values, preferences and beliefs** of the decision maker.

- Programming Ethics
    - Ethical guidelines for developers.
    - Shape and differentiate good practices and attitudes from the wrong ones
    - Basis for ethical decision-making skills in the conduct of professional work

<u>Guest Lecture</u>

- A successful software engineer
    - Requirement, planning, integration, testing and design
    - SWE > Programmer

- Software engineering is all about **managing complexity**
- Software engineering is a set of **accepted** practices implemented to produce software
- Programming is about **getting code to work**
- Evaluate a project with two main things:
    - Layer
    - MVC
- Measure of software maturity
    - Capability

- Maturity
- Model
- Integration

- **Avoid clever code**
- **P**seudocode **P**rogramming **P**rocess

- Design principles
    - Abstraction: hiding the implementation details from the user, provide only functionalities.
    - Encapsulation: wrapping the data and code acting on the data together as a single unit. (also known as data hiding).
    - Strong **cohesion**
    - Loose **coupling**
    - Modular design: (Separation of concern)
    - Object-oriented
    - Inheritance

- Most successful app from our guest speaker was based **jigsaw puzzle**
- Advertise your app in another app

- Over-engineering
    - There is a **cost** to SW engineering processes
    - Make sure the cost is not higher than it yields

- Personal relations
    - **Attitude is important**
        - Best SW + Bad Attitude = Dead Career
        - Constantly negative
        - Overly sarcastic / snarky
    - Nobody will want to work with you (including those hiring and promoting)
    - Never respond to email while angry
        - Wait and not send because people remembers
    - Best colleagues
        - Humble
        - Make time for others
        - Don’t make you feel dumb
        - Say “hi”

- Ethics
    - Good ethics is necessary for career success, while bad ones are as toxic as bad attitude
    - Be proactive (milestones), be curious (learn new things)

- Enterprise systems
    - Not all software is consumer-base
        - Databases, order management, etc.
    - Other devs may be your customer

- Career
    - Enjoy what you do
    - Don't be afraid to change
    - Don't have to be a SWE

- Come out of shell
    - More personal interaction
    - Meetings, code reviews, work with manager, designers and QA
    - Participate, present, collaborate
    - Practice to not be stress

- Time management
    - Non-standard hours
        - Release & client deployments (weekend rollouts, on-call, shifts)
        - Production outages (emergency work)
    - Set aside work time
    - Plan your day first (look at calendar)
    - Handling interruptions
        - Keep context switching low
        - Respond to email at set time
        - Acceptable response time is **24** hours.
        - Great software developer deliver: what is needed, on time, on budget

Design Patterns Summary

- Overall:
    - Good OO design are **reusable, extensible, maintainable**

- Patterns
    - Proven **object-oriented experiences**
    - Address the **issue of changes**
    - Not invented but discovered
    - **Take what varies in a system and encapsulate it**
    - Provide a share language that max value your communication with others developers
    - **Recur in many application**

  - **Most patterns allow some part of systems to vary independently of all other parts**
  -

- Observer
  - Defines **one-to-many** relationships between objects
  - Observable update osververs using a common interface
  - **Do not depend on specific order of notification for your observers**
  - Loosely coupled in that observable knows nothing about observers.
  - Observable is a **class**

- Factory
  - Encapsulate object creation
  - Intent: defer instantiation to its subclasses. Abstract factory create families of related object without depending on their concrete classes.
  - Decouple your clients from concrete classes
  - Coding to **abstractions, not concrete classes**
  - Factory method relies on inheritance: **delegate object creation to subclasses** that implement this interface
  - Abstract factory relies on object composition: **object creation is implemented in methods exposed** in interface

- Singleton
  - At most **one** instance of your class and provides **global access** point
  - A **private** constructor, **static** methods, and **static** variable
  - Caution when using multiple class loaders.

- Command
  - Encapsulates a request to an object letting you parameterize clients with different requests
  - Command object is at the center of decoupling and encapsulates a receiver with an (or set of) actions.
  - An invoke makes a request of command object by calling its execute(), which invokes those actions on the receiver
  - Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called

- Adapter
  - **Adapter converts the interface** of a class into another interface client expect
  - Use an **existing class** and its **interface is not the one you need**
  - Two Forms:
    - Object adapter
    - Class adapter: require multiple inheritance

- Facade
    - **Simplify and unify** large interface or complex set of interfaces
    - Decoupled client from a complex subsystem
    - Required to **compose** facade with its subsystem and use **delegation**
    - Can have more than one facade

- Decorator
    - Add new behaviours and responsibilities
    - **Must be of the same super-type of the object it decorates**

- **An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviours and responsibilities, and a facade wraps a set of objects to simplify.**

- Template
    - Defines the steps of an algorithm, deferring to subclasses for the implementation
    - The template methods' **abstract class** may define **concrete methods, abstract methods and hooks**.
        - Abstract methods implemented by subclasses
        - Hooks are methods do nothing, but may be overridden
    - Keep subclasses from changing algorithm, declare methods as **final**
    - Code **reuse**
    - **Hollywood principle**: don't call us, we will call you. Put decision-making in high level modules that decide how and when to call low-level modules.

- **The strategy encapsulates algorithm by inheritance and template encapsulates algorithms by composition. Factory method is a specialization of template method.**

- Iterator
    - Allow **access** to an aggregate element **without exposing its internal structure**
    - Provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism.

- Composite
    - A structure to hold both individual objects and composites, which allows **clients to treat them uniformly.**
    - Component is any object in a composite structure.
    - Tradeoff between transparency and safety with your needs.
    - Compose objects into **tree structures**

- State

    - Allows an object to have many different behaviours that are based on its internal state. Represent state as a full-blown class.
    - The **context gets its behavior by delegating to current state object** it is composed with. Changes to be made are **localized**.
    - Allows context to change its behaviour as the state of context change
    - Transition controlled by state/context classes.
    - Greater number of classes
    - May be shared among context instances

- Strategy
    - Define **a family of algorithms** and encapsulate each one and make them interchangeable. **By inheritance**.
    - Configures context classes with a behavior or algorithm

- Proxy
    - Provides a **representative** for another object to **control the client's access** to the object.
        - **Remote**: manages interaction between a client and a remote object.
        - **Virtual**: controls access to an object that is expensive to instantiate
        - **Projection**: controls access to the methods of an object based on the caller.
    - Increase the number of classes and objects in your designs

- MVC
    - A **compound** pattern consisting of **observer, strategy, composite and adapter** patterns.
    - Combines two or more patterns into a solution that solves a recurring or general problem

        - Model **keeps observers updated** yet stay decoupled from them.
        - Controller is the **strategy** for view, and view can use different implementation of controller to get different behavior.
        - View uses **composite** pattern to implement user interface (nested component like panels, buttons, etc)
        - New model use **adapter** to an existing view and controller

<u>Design Choice</u>

- Favor **composition** over **inheritance**
- Classes should be **closed** for modification, **open** for extension
- Program to **interface,** not **implementation**
- **Encapsulates** aspects of your application that **varies** and **separate** them from **stays the same.**

- Depend on **abstraction**, do not depend on **concrete classes**.

Ilities

Reusability	The ability to use some or all aspects of **existing** software **in other projects** with little to no modifications.
Security	The ability to withstand and resist **hostile** acts and influences
Maintainability	A measure of **how easily** a bug fixed and function modification can be accomplished
Portability	Software usable across **different conditions and environment**
Performance	Performs its task within acceptable time frame and does **not require too much memory**
Scalability	Adapts well to **increasing** users or data
Compatibility	Operate with **other products** that are designed for interoperability with another product
Reliability	Perform required function under stated conditions for specified period of time
Usability	User interface must be **usable** for target audience
Robustness	Operate under stress or tolerate **unpredictable** or **invalid input**
Extensibility	New capabilities can be added to software without major changes to the underlying architecture.
Fault tolerance	Resistant to and able to recover from component failure.

Other things mentioned / tested

- Simple requests are important requests: **48 hours**.
- Delegate as much as you can, so you can do as much as you can
- Blue skying: could have, must have.
- Course logistics:
    - Team name / Acronym
    - Customer and team meeting time
    - Status report
    - Peer review
    - Lab hours
    - Grade boost
- Pig in the team
- HTML, CSS, Virtual machine.
- CRUD: create, read, update, delete
- http://ieng6.ucsd.edu/~cs110x/static/lectures_a00/CSE_110_Week_Three_Lecture_Two.pdf
    - Social contracts:
        - A promise of behavior by setting up a baseline
        - Can change over time
    - Stand-ups
        - No longer than 15 minutes
        - A meeting where every member report what they did, what they plan to do, and what impediment.
    - Planning poker
- Definition of Done. https://www.youtube.com/watch?v=9FnZsaRujpw
- Good enough design is better than perfect design
- Code should have **only one reason** to change
- Optimal variable name length **10 to 16 characters**.
- A **null** object **is useful** when you want to eliminate the responsibility from the client for handling null pointers.
- Controller may get too large.

- Talk only to your friends (Principle of Least Knowledge)
- Don't call us, we'll call you (Hollywood principle)
- A class should only have one reason to change (Single Responsibility Principle)
- Pseudocode programming:
    - Plain english(no syntax)
    - Precise
    - Turn into comments
- Weekend rollouts shifts---release and client deployment
- Emergency work--production outrage

Labs

- Lab 1: HTML 5, CSS, Bootstrap

- Lab 2: Android studio to create basic android calculator

- Lab 3: React JS
    - An open source Javascript library
    - Easy learning curve,
    - Easier syntax **JSX** (html with javascript),
    - Efficient with its own **virtual DOM**
    - Dynamic data
    - Component:
        - Send information to each other
        - Props, states used for dynamic UI

- Lab 4: Google Firebase, database

- Lab 5: IntelliJ, Travis CI
    - Continuous integration**: automation to repeatedly verify developer changes, integrating those changes into the code baseline, building and deploying the entire software package, and performing unit and functional testing.**

- Lab 6: Node.js
    - Javascript runtime system that allows to build server-side applications.
    - Asynchronous and event driven
    - Fast
    - Single threaded but highly scalable
    - Reading streams in chunks

- Lab 7: Docker and AWS
    - Docker:
        - platform that allows o**ne to package an application with all of its dependencies and surrounding environment in an isolated container.**
        - **Flexible, lightweight, interchangeable, portable, scalable**
    - AWS
        - EC2, AWS.

- Lab 9: Refactoring

- Code refactoring is the **process of restructuring existing computer code – changing the fact**oring – **without changing its external behavior. Refactoring improves portability and reusability of the software**

Design Pattern Complete Points

Overall

- Knowing the OO basics does not make you a OO designer
- Good OO designs are **reusable, extensible, maintainable**
- Patterns show you how to build systems with good OO design qualities
- Patterns are proven object-oriented experience
- Patterns do not give you code, they give you general solutions to design problems. You apply them to your specific application
- Patterns are not invented, they are discovered
- Most patterns and principles address **issues of change** in software
- Most patterns allow some part of system to vary independently of all other parts
- We often try to take what varies in a system and encapsulate it
- Patterns provide a shared language that can max the value of your communication with other developers

Observer pattern

- The observer pattern defines a one-to-many relationship between objects
- Subjects or as we also know them, observables, update observers using a common interface
- Observers are **loosely** coupled in that the observable knows nothing about them, other than that they implement the observer interface
- You can push or pull data from observable when using the pattern
- **Do not depend on a specific order of notification** for your observers
- Java has several implementations of the observer pattern, including the general purpose java.util.Observable
- Watch out for issues with the java.util.Observable implementation
- Do not be afraid to create your own Observable implementation if needed
- Swing makes heavy use of the observer pattern, as do many GUI frameworks
- You will also find the pattern in many other places, including JavaBeans and RMI

Factory pattern

- All factories encapsulate object creation
- Simple factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes
- Factory method relies on inheritance: object creation is delegated to subclasses which implement the factory interface

- Abstract factory relies on object composition:
    - Object creation is implemented in methods exposed in the factory interface
- All factory patterns promote **loose coupling by reducing the dependency of your application on concrete classes**
- The intent of factory method is to allow a class to defer instantiation to its subclasses
- The intent of abstract factory is to create families of related objects without having to depend on their concrete classes
- The dependency inversion principle guides us to avoid dependencies on concrete types and to strive for abstractions
- Factories are a powerful technique for coding to **abstractions**, **not concrete classes**

Singleton pattern

- At most **one instance of** a class in your application
- provides a **global access** point to that instance
- A **private** constructor, a **public static** get instance method combined with a **static** unique instance
- Examine your performance and resource constraints and carefully choose an appropriate singleton implementation for multithreaded applications
- Beware of the double-checked locking implementation; it is not thread-safe in versions before java 2, version 5
- Be careful if you are using multiple class loaders; this could defeat the singleton implementation and result in multiple instances
- If you are using a JVM earlier than 1.2, you will need to create a registry of singleton to defeat the garbage collector

Command pattern

- decouples an object, making a request from the one that knows how to perform it.
- A command object is at the center of this decoupling and encapsulates a receiver with an action (or set of actions).
- An invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver.
- Invokers can be parameterized with Commands, even dynamically at runtime.

- Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.

- Macro Commands are a simple extension of Commands to be invoked. Likewise, Macro Commands can easily support undo().
- In practice, it is not uncommon for “smart” Command objects to implement the request themselves rather than delegating to a receivers.
- Commands may also be used to implement logging and transactional systems.

Adapter & Facade pattern

- When you need to use an **existing** class and its interface is not the one you need, use an adapter.
- An adapter changes an interface into one a client expects.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- There are two forms of the Adapter Pattern: object and class adapters. Class adapters require multiple **inheritance.**

- When you need to **simplify** and **unify** a large interface or complex set of interfaces, use a facade.
- A facade decouples a client from a complex subsystem.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- You can implement more than one facade for a subsystem.

- An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade “wraps” a set of objects to simplify.

Template method pattern

- Defines the steps of an algorithm, deferring to subclasses for the implementation of those steps
- Gives us an important technique for code reuse
- The template method’s abstract class may define **concrete methods, abstract methods and hooks**
- Abstract methods are implemented by subclasses
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclasses

- To prevent subclasses from changing the algorithm in the template method, declare template method as **final**

- The **hollywood principle** guides is to put decision-making in high-level modules that can decide how and when to call low level modules
- ~~You will see lots of uses of the template method pattern in real world code, but do not expect it all to be designed by the book~~
- The strategy and template method pattern both encapsulate algorithms, one by inheritance and one by composition
- The factory method is a specialization of template method

Iterator pattern

- Allows access to an aggregate's elements **without exposing its internal structure**
- Takes the job of iterating over an aggregate and encapsulates it in another object
- When using an iterator we relieve the aggregate of the responsibility of supporting operations for traversing its data
- An iterator provides a common interface for traversing the items of an aggregate, allowing you to use polymorphism when writing code that makes use of the items of the aggregate
- We should strive to assign only one responsibility to each class

Composite pattern

- Provides a structure to hold both individual objects and composites
- Allows clients to treat composites and individual objects uniformly
- A component is any object in a composite structure. Components may be other composites or leaf nodes
- There are many design trade-off in implementing composite. You need to balance transparency and safety with your needs

State & Strategy pattern

- Allows an object to have many different behaviors that are based on its internal state.
- Unlike a procedural state machine, the State Pattern represents state as a full-blown class.

- The context gets its behavior by delegating to the current state object it is composed with.
- By encapsulating each state into a class, we localize any changes that will need to be made.
- The State and Strategy Patterns have the same class diagram, but they differ in intent.
- Strategy Pattern typically configures Context classes with a behavior or algorithm.
- State Pattern allows a Context to change its behavior as the state of the Context changes.
- State transitions can be controlled by the State classes or by the Context classes.
- Using the State Pattern will typically result in a greater number of classes in your design.
- State classes may be shared among Context instances.

Proxy pattern

- Provides a representative for another object in order to control the client's access to it. There are a number of ways it can manage that access.
- A **Remote** Proxy manages interaction between a client and a remote object.
- A **Virtual** Proxy controls access to an object that is expensive to instantiate.
- A **Protection** Proxy controls access to the methods of an object based on the caller.
- Many other variants of the Proxy Pattern exist including caching proxies, firewall proxies, copy-on-write proxies, and so on.
- Proxy is structurally similar to Decorator, but the two differ in their purpose.
- The Decorator Pattern adds behavior to an object, while a Proxy **controls access**.
- Java's built-in support for Proxy can build a dynamic proxy class on demand and dispatch all calls on it to a handler of your choosing.
- Like any wrapper, proxies will increase the number of classes and objects in your designs.

MVC

- The model view controller pattern is a **compound** pattern consisting of the **observer, strategy and composite** patterns

- The model makes use of the observer pattern so that it can keep observers updated yet stay decoupled from them
- The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior

- The view uses the composite pattern to implement the user interface, which usually consists of nested components like panels, frames and buttons
- These patterns work together to decouple the three players in the MVC model, which keeps designs clear and flexible
- The adapter pattern can be used to adapt a new model to an existing view and controller
- Model 2 is an adaption of MVC for web applications
- In model 2, the controller is implemented as a servlet and JSP & HTML implement the view

Quizlet List to Study

To get perfect scores on Gary's CSE110's bi-weekly quizzes, study all the quizlets below.

Quiz 1

- Software project, team, leadership, interview, criticism
- Status report, peer review, grade calculation, html, css

Quiz 2

- Design use case, waterfall/agile, code review, requirement
- Waterfall/Agile details, layered architecture, risk matrix
- User story, team members, team meetings, betrayal/trust
- Team related, pig/chicken, interaction, user story

Quiz 3

- MVC, layered architecture, design use case, AOP, requirement, TDD, pair programming.
- Layered architecture, requirement, AOP, TDD
- MVC, requirement, design use case, layered architecture, AOP, TDD

Quiz 4

- Design patterns, retirement, employment
- Design patterns as we tested in our quiz 3.
- Design pattern details, retirement, stock option, testing.

Quiz 5

- Testing, coupling, cohesion, social norms, frequent small release, continuous feedback
- Design patterns, testing, coupling, cohesion, employment

Final
- Everything 277 terms
- Everything reduced 126 terms
- Collaborative review doc based 334 terms