CS 2400 January 23rd Wed.

Wednesday, January 23, 2019 11:51 AM

Digital representation of information. Numerical: Binary codes. Binary Arithmetic

- Positional normal count systems (or so called radix systems): every number n can be represented in a positional count system
    - $N = a_n p^n + a_{n-1} p^{n-1} + \ldots + a_1 p^1 + a_0 p^0$
    - $1010.01_{(2)} =$

CS 3250 Mon. January 28th

Monday, January 28, 2019 3:52 PM

Version control and revision control

- **Sometimes called (VCS)**

WHAT TO DO

- Every time you have something running, check it in (commit)
- Every time you make a significant change, check it in
- Every half hour or so, check it in
- Now you'll know what you broke

HISTORY – LOCAL STORE

- Source code control system
- SCCS
- Early: 1972
- Store first version and then all deltas

CLIENT/SERVER

- Multiple users
    - Either file locking or merging of diffs
- Concurrent version system
    - Originally built on top of RCS
- Subversion
    - Inspired by CVS

COMMERCIAL

- Clear case
- Perforce
- SourceSafe

CS 3250 Wed. January 30th

Wednesday, January 30, 2019 4:02 PM

Coverage command

Coverage run –omit=*test* -m unittest

Travis

- Jenkins
  - CI tool that can be used locally

Tags

- "git tag"
  - Often used for release versions
  - Shows all of the files of a particular version number

Branching, Initially

- Do not work on master

Git branch testing

- Creates another point of reference to a particular branch

Switching

- HEAD --> refers to the master branch
- Git checkout testing
  - HEAD no longer refers to master branch, but now to the testing branch

Git branch <branch name> --create new branch
Vi <file name>-- creates file and opens for editing
Git checkout <branch name> -- checkout named branch
Git merge <branch> --merges changes with current and specified branches
Git revert – undoes a commit

CS 3250 Mon, Feb. 4th

Monday, February 4, 2019 3:55 PM

Story points and velocity

- As an 'x' I want to do 'y' because of 'z'
- Story points
  - Length of time estimated for a story point
- Velocity
  - Number of story points per sprint

GITFLOW

- Semantic Versioning
  - "In the world of software management exists a dreaded place called 'dependency hell.' The bigger your system grows the more packages you integrate into your software, the more likely you are to find yourself, one day, in this pit of despair."
  - Essentially, sometimes you install packages that need other packages in order to function, the more of these you need, the further you fall into hell
- Version numbers
  - Give a version number MAJOR.MINOR.PATCH, increment the:
    - MAJOR version when you make incompatible API changes
    - MINOR version when you add functionality in a backwards-compatible manner, and
    - PATCH version when you make backwards-compatible bug fixes.
  - Additional labels for pre-release and build metadata are available are extensions to the MAJOR.MINOR.PATCH format
- Rules
  - Software using semantic versioning must declare a public API
  - A normal version number MUST take the form X.Y.Z where x, y, and z are non-negative integers, and MUST NOT contain leading zeroes
  - Once a versioned package has been released, the contents of that version MUST NOT be modified
  - Major version zero (0.y.z) is for initial dev
  - Version 1.0.0 defines public API

Merging branches (move to the branch you wish to merge to, and name the branch you were working on)

- Git status
- Git commit –a
- Git checkout master
- Git merge dev
- Git push

- Git branch justin <create branch>
- Git checkout justin <move to branch>
- |
- |
- Git checkout dev <move back to dev>
- Git merge justin <merge justin with dev>
- Git push <push changes>
- Git branch –d justin <delete a branch>

CS3250 Mon, Feb. 11th

Monday, February 11, 2019 3:52 PM

SCRUM

- OVERVIEW
  - Scrum is a framework for managing software dev
  - Designed for teams of three to nine developers who break their work into two-week cycles, called "sprints", check progress daily in 15-minute stand-up meetings, and deliver workable software at the end of every sprint
- AGILE
  - A key principle of scrum is the recognition that customers will change their minds about what they want or need (often called requirements volatility) and that there will be unpredictable challenges – for which a predictive or planned approach is not suited
  - As such, scrum adopts an evidence-based empirical approach – accepting that the problem cannot be fully understood or defined up front, and instead focusing on how to maximize the team's ability to deliver quickly, to respond to emerging requirements, and to adapt to evolving technologies and changes in market conditions
- PRODUCT OWNER
  - Represents the product's stakeholders and the voice of the customer; and is accountable for ensuring that the team delivers value to the business
  - They define the product in customer-centric terms (typically user stories), adds them to the product backlog, and prioritizes them based on importance and dependencies.
  - Scrum teams should have one product owner. This role should not be combined with that of the scrum master.
  - The following are some of the communication tasks of the product owner to the stakeholders
    - Demonstrates the solution to key stakeholders who were not present at sprint review
    - Defines and announces releases
    - Communicates team status
    - Organizes milestones reviews
    - Educates stakeholders in the development process
    - Negotiates priorities, scope, funding, and schedule
    - Ensures that the product backlog is visible, transparent, and clear
- SCRUM MASTER
  - Scrum is facilitated by the scrum master, who is accountable for removing impediments to the ability of the team to deliver product goals and deliverables
  - Not a team lead or project manager but acts as a buffer between the team and any distracting influences
  - Ensures that the scrum framework is followed
  - Helps product owner maintain product backlog in a way that ensures the needed work is well understood so the team can continually make forward progress
  - Helping the team to determine the definition of done for the product, the input from key stakeholders
  - Coaching the team, within scrum principles, in order to deliver high quality features for its product
  - Facilitating team events to ensure regular progress
  - Helping team to remove impediments to its progress, whether internal or external to the team
  - Educating key stakeholders in the product on scrum principles

CS3250 wed, Feb 13th

Wednesday, February 13, 2019 5:35 PM

SCRUM CONT.

- Sprint
  - Time-boxed effort; that is, it's restricted to a specific duration
  - Duration is fixed in advanced for each sprint and is normally between one week and one month, with two weeks being the most common
  - Each sprint starts with a sprint planning event that aims to define sprint backlog, identify the work for the sprint, and make an estimated forecast for the sprint goal.
  - Each sprint ends with a sprint review and sprint retrospective, that reviews progress to show to stakeholders and identify lessons and improvements for the next sprints.

- Increment
  - Scrum emphasizes a working product at the end of the sprint that is really done.
  - In the case of software, this likely includes that the software has been fully integrated, tested and documented, and is potentially shippable
- Standup/daily scrum
  - All members of the dev team come prepared
  - Daily scrum:
    - Starts precisely on time even if some dev team members are missing
    - Should happen at the same time and place every day
    - Is limited (timed-boxed) to fifteen minutes
  - Anyone is welcome, though only dev team members should contribute
  - During the scrum, each team member typically answers three question:
    - What did I complete yesterday that contributed to the team meeting our sprint goal
    - What will I complete today to contribute to the sprint goal
    - Do I see any impediments that are keeping me from completing our sprint goal
- Sprint review/demo day
  - Demo completed sprint goals
- Retrospective
  - Reflects on the past sprint
  - Identifies and agrees on continuous process improvement actions
  - Two main questions are asked during this
    - What went well during this sprint?
    - What could be improved on in the next sprint?
-

CS3250 Mon, Feb 18th

Monday, February 18, 2019 3:55 PM

- PRODUCT BACKLOG
  - Comprises an ordered list of requirements that a scrum team maintains for a product
  - Consists of features, bug fixes, non-functional requirements, etc.
  - Items added to a backlog are commonly written in story format
  - Non-functional requirements are:
    - Programming language used
    - Environment developed in
    - Test coverage
  - Functional requirements:
    - Requirements the customer can see
  - Visible to everyone but may only be changed with consent of the product owner, who is ultimately responsible for the ordering of product backlog items for the development team to choose
  - Often, but not always, stated in story points using the rounded Fibonacci scale
- SPRINT BACKLOG
  - Is the list of work the dev team must address during the next sprint
  - The list is derived by the scrum team progressively selecting product backlog items in priority order from the top of the product backlog until they feel they have enough work to fill the sprint
  - Dev team should keep in mind past performance assessing its capacity for completing backlog items
  - Property of the dev team
  - Once a sprint backlog is committed, no additional work can be added to the sprint backlog except by the team
- TYPICAL ADDITIONS
  - Burn-down chart is a public displayed chart showing remaining work in the sprint backlog
  - Definition of done: the exit criteria to determine whether a product backlog item is complete
- VELOCITY
  - Total effort a team is capable of in a sprint
  - Number is derived by evaluating the work (typically in user story points) completed in the last sprint
  - Collection of historical velocity data is a guideline for assisting the team in understanding how much work they can likely achieve in a future sprint
- SPIKES
  - A time-boxed period used to research a concept or create a simple prototype
- THREE PILLARS
  - Transparency
  - Inspection
  - Adaptation
- FIVE VALUES
  - Commitment: team members individually commit to achieving their team goals, each and every sprint
  - Courage: know they have the courage to work through conflict and challenges together so that they can do the right thing

- Focus: team members focus exclusively on their team goals and sprint backlog; there should be no work done other than their backlog
- Openness: team members and their stakeholders agree to be transparent about their work and any challenges they face
- Respect: team members respect each other to be technically capable and to work with good intent

- The important stuff
  - Acts confidently and is secure
  - Respects rights and property of others
  - Works cooperatively with others
  - Courteous; respects authority, polite
  - Practices self-control
  - Accepts responsibility for behavior
  - Resolves conflict appropriately
  - Exhibits positive attitude toward learning

Cs3250 wed, Feb 27th

Wednesday, February 27, 2019 4:49 PM

- **Testing**
  - **Many different types**
    - Acceptance
    - Unit: method
    - Integration: class
    - System
- **Unit test**
  - Arguably the most important
  - Many modern SE methods use "test first"
    - Write a test
    - Make sure it fails
    - Write just enough code to make it work
  - If you don't know how to test it, how can you write it?
  - Correct frame of mind
  - Test-driven development

- Implementation
  - SE tends to focus on requirements, design, and processes
  - A bad implementation will ruin everything else
- **Guideline: style**
  - Follow language styles
    - E.g.: the elements of java style by Scott Ambler and Trevor Misfeldt
- **Guideline: naming**
  - Use good, descriptive names
  - Probably not Hungarian notation
- **Guidelines**
  - **DRY**
    - **Don't repeat yourself**
  - **YAGNI**
    - **You ain't gonna need it** (don't write a single line of code that you don't need right now)
  - **SOLID**

DESIGN PATTERNS
- **Wikipedia**
  - A general usual solution to a commonly-occurring problem within a given context in software design
  - It is not a finished design that can be transformed directly into source or machine code
  - It is a description or a template that can be applied to many different situations
  - Formalized best practices a programmer can use to solve common problems when designing an application or system
  - OO (object oriented) design patterns
- Why talk about design patterns?
  - Creates a shared vocabulary
    - Developers can interact in richer terms

    - Keeps thinking/designing at the abstract (pattern) level
        - Creates better (more flexible, reusable, etc.) designs
- **Origin**
    - Patterns originated as an Architectual concept by Christopher Alexander (1977/79)
    - In 1987, Kent Beck and Ward Cummingham began experimenting with the idea of applying patterns to programming – specifically pattern languages – and presented their results at the OOPSLA conference that year

- Groups and concepts
    - Design patterns
        - Creational
        - Structural
        - Behavioral
    - Described using the concepts of
        - Delegation,
        - Aggregation
        - Consultation
- Big four concepts of object oriented programming (AEI and off by 1 error)
    - abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism

- First design principle
    - Also creates more easily reusable objects
    - Object delegate behavior to other objects
- Second design principle
    - Program to an interface, not an implementation
        - Not necessarily a java interface
        - Program to a supertype
            - Private map<String, Boolean> urls – new ConcurrentHashMap<String, Boolean> ():
        - Can then better use polymorphism
        - Can more easily change implementation

CS3250 Mon, March 4th

Monday, March 4, 2019 4:11 PM

Design principles cont.

- **Third design principal**
  - Favor composition over inheritance
  - Favor has-a(composition) relationships over is-a(inheritance) relationships
  - Inheritance limits reusability
- **Strategy pattern**
  - Defines a family of algorithms, encapsulates each one, and makes them interchangeable
  - Strategy lets the algorithm vary independently from clients that use it

Chapter 2: the observer pattern

- **The observer pattern**
  - Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and update automati cally
- More on observer
  - The observer pattern is a software design pattern in which an object called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state change
  - Also a key part in the familiar model-view-controller (MVC) architectural pattern
- **Observer UM**L
  - Subject (has a list of observers)
    - Observers attach to subject
    - Attach()
    - Dethatch()
    - Notify{} ------------> for each o in observers: o.update()
  - Observer
    - Update()
- Languages
  - Java has observer and observable, etc.
  - JavaScript has addEventListener etc.
- Design principle
  - Strive for loosely-coupled designs between objects that interact
  - Objects have very little information about each other
  - No shared state (two objects don't have the same instance variables)
- **Decoupling**
  - The observer pattern helps decouple objects
  - The subject knows only that the observer implements the observer interface
  - New observers can be added at any time
  - New types of observers can be added
  - Can reuse subjects and observers independently
  - Changes have no effect on one another
- Warnings
  - Don’t depend on order of evaluation of notifications
  - Java observable is a class
    - Not an interface
    - Must inherit
    - SetChanged() is protected
-

CS3250 Wed, March 6th

Wednesday, March 6, 2019 5:21 PM

Chapter 3: the decorator pattern

- The open close principle
    - Design should be open for extension, but closed for modification
- Decorators
    - Has the same super type of the objects that they decorate
    - Can have one or more decorators
    - Can pass wrapped object anywhere original could be passed
    - Adds behavior before and/or after delegating object
    - Objects can be decorated at runtime
- Decorator pattern
    - Attaches additional responsibilities to an object dynamically
- Inheritance
    - Using inheritance to get type matching but not behavior
        - Java requires this, other languages don't

Chapter 4: the factory pattern

- Factories
    - Handles the details of object creation
        - Encapsulates the creation in a subclass
        - Decouples interface from creation
    - Can return a variety of types
    - Client doesn't care which type
    - Can add additional types
    - If static, can't subtype to extend
- Wikipedia (source of the following…)
    - …the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created
- UML
    - Creator creates product
        - Factory method()
        - Operation()
    - Product (interface) pointed to by product(s)
        - Appropriate product is created based on the request

CS3250 Mon, March 11th

Monday, March 11, 2019 4:34 PM

Factory continued…

- Abstraction
  - Factories don't have abstraction
    - Can have default and that can call down if necessary
- Dependency inversion principle
  - Depend upon abstractions
  - Do not depend upon concrete classes
  - We want to program towards the abstractions
- Wikipedia
  - High-level modules should not depend on low-level modules
    - Both should depend on abstractions
  - Abstractions should not depend on details
    - Details should depend upon abstractions
  - By dictating that both high-level and low-level objects must depend on the same abstraction, this design principle inverts the way some people may think about OO programming
- Therefore
  - No variable should hold a reference to a concrete class
    - A general rule, there are exceptions
  - No class should derive from a concrete class
    - Again, general rule
  - No method should override an implemented method of any of its classes

- Dependency injection
  - A technique whereby one object (or static method) supplies dependencies of another object
  - A dependency is an object that can be used (a service)
  - An injection is the passing of a dependency to a dependent object (a client) that would use it
  - The service is made part of the client's state
  - Passing the service to the client rather than allowing a client to build or find the service, is the fundamental requirement of the pattern
  - This fundamental requirement means that using values (services) produced within the class from a new or static methods is prohibited
  - The client should accept values passed in from outside
  - This allows the client to make acquiring dependencies someone else's problem
  - The intent is to decouple objects to the extent that no client code has to be changed imply because an object it depends on needs to be changed to a different one
- Inversion of control
  - A design principle in which custom-written portions of a computer program receive the flow of control from a generic framework
  - A software architecture with this design inverts control as compared to traditional procedural programming
    - In traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks
    - With inversion of control, it is the framework that calls into the custom, or task-specific, code.
- Abstract Factory Pattern
  - Provides an interface for creating families of related or depended objects without specifying

their concrete classes

- Abstract Vs. Non
    - Factory
        - Creation through inheritance
        - Creates objects of a single type
    - Abstract factory
        - Creation through composition
        - Instantiated via new and passed
        - Creates families of related objects via factories

Midterm stuff

Monday, March 11, 2019 5:08 PM

- Git
  - It is crucial to have version control in modern SW development
    - We are not capable of handling the complexity of programs
    - We are not good at communication
  - Complexity
    - Using git to help deal with it
    - Also using design patterns
- Software development
  - We haven't gotten a lot better at it
  - Traditional waterfall model
    - Requirements
      - Architecture
        - Design
          - Implementation
            - Release
    - Not good for the chaotic nature of sw dev
- AGILE
  - Happened because waterfall didn't work
  - SCRUM
    - A form of AGILE development
    - See scrum guide
- Design patterns book
  - High-level picture
  - Interfaces
  - Inheritance
  - AEIP (big OO four)
  - Design principle
    - What changes and what stays the same
    - Encapsulate what stays together
  - Has a… vs is a…
    - When to apply
  - Chapter one
  - Chapter two
    - Loosely coupled designs
    - Don’t want one class seeing the internals of another class
    - As long as behavior doesn’t change, you don't care
  - Observer pattern
    - We have a subject to which we can subscribe any number of observers
      - Each observer implements an interface
      - When a subject changes, it updates and notifies each observer
      - Multiple listeners to the same event
      - Event driven
  - Chapter 3
    - Decorator pattern
    - Open close principle
      - Open for extension
      - Closed for modification

            - We can wrap an object inside another object of the same type
                - Continue to have the same type, but added some functionality
    - Factory pattern
        - Factory not abstract factory
- Type of question
    - Is encapsulation required for abstraction?
- At least one question on the 5 dysfunctions
    - 5 dysfunctions themselves
        - Not story
        - concepts

CS3250 Wed, April, 3rd

Wednesday, March 20, 2019 4:55 PM

DESIGN PATTERNS 08

Templates

Wikipedia

- Behavioral design pattern that defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses
- It lets one redefine certain steps of an algorithm without changing the algorithm's overall structure

Book

- Defines the skeleton of an algorithm in a method.

Hot beverage

- For both cofee and tea
    - Boil water(same, in base class)
    - Use hot water to extract (different, abstract in base class)
    - Pour into cup (same)
    - Add condiments (different)
- PrepareRecipie is template method

prepareRecipie

- Is template method
- All steps present
- Some are handled by base class
- Some by subclass

Hooks

- Can define concrete methods that do nothing unless subclass overrides them
- Use abstract when subclass must implement, hooks when optional

Hollywood principle

- Don’t call us, we'll call you
- Low-level hooks into system, high-level calls at the appropriate time
- Java Arrays.sort calls to compareTo()

Summary

- To prevent subclasses from changing the algorithm, make the template method final
- Both strategy and template patterns encapsulate algorithms
    - Strategy via composition
    - Template via inheritance
- Factory is a very specialized template
    - Returns result from sublcass

Chapter 9

Iterators and composites

Iterator pattern

- Provides a way to access the elements of an aggregate object sequentially without exposing the underlying representation
- This places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.
- (task of traversal is placed on the iterator and not the class)
- (aggregator does not participate in the iteration)

JAVA

- Enumeration is the older that has been replaced by iterator

    - Iterator allows removal

Design principle

- A class should have only one reason to change
    - Single-responsibility principle
- High cohesion
    - All methods related to purpose
- All public methods should be dealing with the instance variables
- (Malcolm gladwell: Outliers)

Composite pattern

- Allows you to compose objects into tree structures to represent part/whole hierarchies
- Composite allows clients to treat individual objects and compositions of objects uniformly
- We can apply the same operations over both composites and individual objects
- Can ignore differences between the two
- Think recursion

Part/whole

- Animals, mammals, cats,…
    - Respiration, locomotion, etc.
- Objects in a scene
    - Texture, placement, etc.

Getting a job

Monday, April 8, 2019 4:32 PM

<u>Getting a Job</u>

- The great American know who
    - Networking helps
    - Maintaining your GPA is the #1 thing you can do
    - Impressing people in this class is the other #1 thing you can do
- Linked In
    - Frickin' join it
        - It's free
        - It's a good way to network
        - They send you jobs that match your skills
        - Pulse is usually pretty good
        - Recruiters rely on it heavily
- Meetup
    - Frickin' join it
    - Find a meetup that matches your interest in a job
    - Pizza and recruiters are free at most meetings
    - Would it kill you to interact with people you don't know?
        - And talk about things you and they are interested in?
- Twitter
    - Maybe join it
    - Don't just re-tweet or like
    - Follow those whom you're interested in and see what they tweet
- ACM
    - $20
    - Periodical tech news
    - Good resource
- Denver devs
- Stop
    - Gaming
        - Gaming friends aren't going to hire you
        - They don't teach you skills
        - Get off the couch
    - FB'ing
    - Watching your favorite sports
    - Wasting your time
        - Time is all we have
    - Everyone is an expert at something
        - Choose what it is
- Start
    - Learning about your field
        - ACM, IEEE
    - Being relatable
        - Be able to interact with the non-geek
    - Being nice
    - Branching out
        - New languages etc.
- How to get any job that you want

    - It turns out that people applying for a job care more about their own credentials than the people who are hiring them
    - Companies can usually get over a hundred applications for a single position
        - Filters
        - Have to get through HR
    - Do the job before you get the job
        - Do research on the company and the position
        - What tech do they use?
        - Whom have they bought and been bought by?
        - Their strengths and weakness and those of their competitors
    - It's OK if you're a few years below the minimum experience level, but not TOO far below
    - It's OK if your education level is a little below the required amount, but again, not TOO much below
    - Make sure you can actually DO the job
    - "but my field is different!"
        - Not it's not
        - It's just human behavior.
- Market yourself
    - Print up some cards
        - Include
        - A non-school email address
        - Your twitter, Linked-In, github URLs
    - Github
        - Your school work
        - Gists
    - Blog
    - Create a presentation or two
    - Write a paper or two

INTERVIWING

- Many kinds
    - Typically start with a phone screen
    - Possibly followed by an online quiz
    - May be half- or full-day
    - May be interviewed by group
        - Or individuals
        - Or mix
    - Typically have to eat with them
        - Lean some table manners
- Overall
    - Arrive at least 15 minutes early, but enter 8-10 minutes early
    - Bring a notepad and make notes if necessary
    - Bring extra resumes
    - Follow up!
        - The same day
        - Say thanks, you know how much work it is, if there is anything else you can provide to help
    - 1 page starting out
- Personal
    - Dress well
    - Eye contact
    - Body language
        - Don't cross your arms and legs
        - Lean in

        - Avoid your nervous habits
    - Don't wear Axe
        - Many people sensitive/allergic to artificial scents
- What you want
    - You're interviewing them as much as they you
        - You don't want to work for just any place
        - Unless you can grit your teeth for a year and build your resume
        - Maybe check out http://www.glassdoor.com
    - Working at some places can be worse than not working in the field for a time
        - Trailing edge technology that traps you
        - A place you can't excel in
        - Churn and burn
- What to ask
    - Why do you work here?
    - What are the working conditions?
    - What SE lifecycle is used?
    - What is the tool chain?
    - HR
        - How are the benefits?
            - Health, vacation, retirement?
- Types of questions
    - Computer science
    - What you've done
    - Language and frameworks
    - Algorithms and data structures
    - O of algorithms and choosing
    - Imperative vs functional
    - http://www.nerdparadise.com/tech/interview
- What you've done
    - Make sure you did what is on your resume and are ready to talk about it
    - They are looking for specifics
    - Give a concrete example
    - If you don't have an example say so
- Language
    - When would you use an anonymous inner class in java?
    - What's a lambda?
        - Know your versions
    - What's a closure?
- OO
    - What are the major features of OO?
        - Abstraction, encapsulation, inheritance, polymorphism
    - What is an interface?
    - What is a virtual method?
    - What is multiple inheritance?
        - Java doesn't have multiple inheritance
    - Deep vs shallow copies
    - Dependency inversion/inversion of control
        - Frameworks
    - Design patterns
- Do me a SOLID
    - S - single-responsibility principle
        - Classes do one thing

    - O - open-closed principle
    - L- liskov substitution principle
    - I - interface segregation
    - D- dependency inversion
- OS
    - Concurrency
        - Threads and processes
- Networks
    - Ipv4 and ipv6
    - TCP vs UDP
    - Client/server vs P2P
        - P2P - single program is both a client an a server
            - Torrents
- Software engineering
    - Git
    - Agile manifesto
    - TDD
    - DRY, YAGNI
    - CI/CD
- Web
    - MVC
    - ORM
        - Object relational mapper
        - Object document mapper
    - Ajax

Getting a job cont.

Monday, April 15, 2019 4:48 PM

- Questions
  - Why are manhole covers round?
    - Cover can't fall through the hole
  - What is 2^24?
    - 2^10^2 (2^4)
  - How many usable addresses in a class c network?
    - 2^21
  - What are your strengths and weaknesses
    - No kidding
  - Why do you want to work here?
    - We all like to eat and not sleep under bridges
    - Give another reason
- Tough questions
  - They typically will ask a few very difficult questions
    - To show off
    - To see how you reason
  - How many trees are in CONUS?
  - How far away is the moon?
    - 238,900 mi
- Verbal hints
  - Don't say "I feel like"
    - No one cares how you feel
    - You hopefully mean "I think" or "I mean"
- Try to avoid "Like"
  - Unless you're making a simile
  - Also "um", "you know", and "got"
- Don't use "literally"
  - Unless you almost always talk figuratively
- Irregardless is not a word
  - Regardless is
  - Irrespective is
- Don't say methodology
  - Study of a method
  - You don't study methods
  - You apply them
- The word data is a plural
  - 'The data are…'
  - Singular is datum
- Don't make jokes
  - In general, you're bad at it
  - Making a joke is difficult
  - If you absolutely want to
    - Make it abstract
    - Or about yourself
  - The wrong joke can get you fired
- Be terse and don't BS

Guerrilla interviewing

- Looking for smart people who get things done
- Introduction
- Question about recent project candidate worked on
- Design patterns
- What is your approach when you have a technical problem
- What is your approach to dealing with conflict/relationships?
- Easy programming question
- Pointer/recursion question
  - You can do arithmetic on pointers
  - You can't on references
- What are the four major aspects of OO
- Do you have any questions?
  - You do.
  - It shows interest in the workplace
  - What's you're software dev lifecycle?
  - What are the chances for advancement?
  - Do you do lateral moves?
  - What's your turnover rate?
  - How long will I typically be in a position before being eligible for elevation or change?
  - Why do you work here?
  - What would a successful candidate look like in this position?
  - What are you hoping to get out of this job?
    - Goes both ways

COVER LETTER
- If going through a recruiter
  - You don't need a cover letter
  - But the split is about 50/50
- Need to be targeted
  - A generic cover letter is probably worse than not having one

Some typical questions
- Why did you chose this field?
- How did your college experience prepare you for a career in this field?
- Describe the work environment that makes you thrive

Resources
- Cracking the code interview (book)
  - Interview process
  - Behind the scenes
  - Special situations
  - Before the interview
  - Behavioral preparation
  - Big O
  - Technical questions (+ 5 algorithm approaches)
  - The offer and beyond
  - 189 programming questions and answers
- Where do you see yourself in 5-10 years?
  - I really don't know, I haven't planned that far ahead because I don't know what opportunities I'll have, where technology is going, where the company is going 5 or even 2 years from now.
- How comfortable are you with change and how do you deal with it?
  - Come up with a couple examples that demonstrate you're comfortable with change

Functional programming
- Scala
- Elixir

CS3250 Wed. April, 17th

Wednesday, April 17, 2019 3:59 PM

THE STATE PATTERN

- STATE
  - The combination of the value of all the variables in an object
  - We use state machines all the time
    - NFAs, DFAs
  - Automata
    - Combinational logic
      - Ands ors nots
    - FSMs
      - Finite state machines
    - Pushdown
      - FSMs with a stack
    - Turing machines
      - A machine that can compute anything that can be computed
  - Vending machines, elevator, locks, traffic lights, etc.
    - Current state and future state etc.
  - FSMs limited to the amount of memory (states) it has
  - Turnstile
    - Initial condition <LOCKED>
      - Push, nothing happens (loop back to initial)
      - Drop a coin in (go to unlocked)
    - Future condition <UNLOCKED>
      - Add a coin (loop back to unlocked)
      - Push (go to locked)
- State pattern
  - Allows an object to alter its behavior when its internal state changes
  - The object will appear to change its class
  - Very similar to strategy pattern
    - Strategy an alternative to sub-classing as it uses composition
    - State is an alternative to having lots of conditionals
- UML
  - State interface
    - Operation defined on the interface
    - Typically called "next state"

EFFECTIVE JAVA

- CHAPTER 2
  - Consider static factory methods instead of constructors
    - One advantage is they have names
      - Constructors do not and one has to differentiate between parameters
      - This can be confusing and lead to errors
    - A class can have only one constructor with a given name
      - Don't change order of constructor parameters to differentiate
    - Static factory methods don't have to create a new object
      - Constructors always do
      - Maybe there's an object already created that works
      - Helps with immutable classes and preconstructed instances
    - Singletons, flyweights, non-instantiable

            - Can return a subtype
                - Java.util.collections contains all static methods that work on many types
                - Polymorphic
                - addAll, binarySearch, disjoint, frequency, min, max, sort, shuffle, reverse, …
                - Type returned can be non-public
                    - You can return private types from a factory
                - Can vary implementation
            - Returned class need not exist at the time the class is written
                - Allows run-time specification
                - JDBC (java database connector) an example of a service provider framework
                    - Consults the configuration file and returns the right class at runtime
            - Service provider framework
                - Service interface
                - Provider registration
                    - My SQL can say, "here's how you implement me"
                - Service access
            - Disadvantages
                - Classes that have neither a public nor a protected constructor cannot be sub-classed
                - Not called out in Javadoc
            - Popular java static factory name
                - valueOf, getInstance, newInstance, get*Type*, …
        - Consider a builder when faced with many constructor parameters
            - If a class has many fields that need initializing, constructors have long lists of parameters
                - Constructors often chained/telescoped
            - Create empty instance and have many set()s
                - Problem: instance in inconsistent state
            - Builder pattern
            - Build() is a parameter-less static method
            - Required parameters passed in to constructor
                - Optionals set()
                - Other languages have optional parameters instead
        - Enforce the singleton property with a private constructor or an Enum type
            - Up to 1.5, two ways to ensure only a single instance of an object
            - Statics are executed before anything else in a class
                - Statics are essentially pre-existing
                - This is why you need "public static void main" in java
            - 1.5: Enumeration
                - 1.5 introduced a set of symbols and we don't know what those symbols are
                - You cannot extend an Enumeration
                    - Can't be sub-classed
        - Enforce non-insatiability with private constructor
            - Just have a private no-args constructor, the default isn't created
            - Class cannot be sub-classed
                - Sub-classes would need to call constructor
            - Might want to have constructor throw an assertion error
                - Just to be safe
        - Avoid creating unnecessary objects
            - Use literals and valueOf()
        - So
            - Prefer primitives to boxed primitives
            - Be careful of unintended auto-boxing

- So
    - Try very hard not to manage memory
    - Nulling object references should be very unusual
- Avoid finalizers
    - Don't use finalizers
    - Unpredictable, often dangerous, generally unnecessary
    - Unlick C++ destructors
        - These are called immediately
        - Java uses try/finally for these types of uses
    - One never knows when a finalizer is called
        - Part of garbage collection
        - Might not be called at all
    - Don't e.g. close files as there is a limited number of open files
    - Finalizers are slow
    - Finalizers are not chained
    - If really need functionality, provide explicit termination method

CS3250 Mon. April 22nd

Monday, April 22, 2019 4:15 PM

Chapter 3

- Obey the general contract when overriding equals
  - Sometimes you don't need to
    - When all objects are unique, such as threads
      - Lumbok (in java)
    - When you don't need it, such as with random number generators
- When to implement
  - When logical equality (.equals) is different from simple object identity (==)
  - This is the typical case as classes have state, kept by variables with values
  - Test for equivalence, not the same object
  - When we need the class to be map keys or set elements
- Must implement an equivalence relation
  - Must be reflexive: x.equals(x) must return true
  - Must be symmetric: x.equals(y) must be true if and only if y.equals(x)
  - Must be transitive: if x.equals(y) is true and y.equals(z) is true, x.equals(z) must be true
  - Must be consistent: multiple calls to x.equals(y) must always return the same value
  - For any non-null reference, x.equals(null) must return false
- So?
  - There is no way to extend an instantiable class and add a value while preserving the equals contract
  - You can safely add values to a subclass of an abstract class
- Consistency
  - Do not write an equals method that depends on unreliable resources
  - Java's URL equals relies on IP address comparison
    - What happens when not on network?
    - What happens when network addresses change?
- Recipe
  - Check for object == this
  - Use instanceof to check for correct type
  - Cast argument to correct type
  - Test == for all significant fields
    - Except for Float.compare, Double.compare, and Arrays.equal
  - Also override hashCode
  - Use @Override
- Always override hashCode when you override equals
  - When invoked on the same object, and the object hasn't changed to affect equals, always return the same integer
    - Does not have to be the same integer from runtime to runtime
  - If two objects are equals, both hashCodes must be the same
  - If they are equals, it is not required to product distinct hashCodes
    - If not, hash table performance can be affected
    - "return 42" is legal, but horrible
- Creating a hashCode
  - Set result = 17
  - For all the fields
    - If boolean, c = f?1:0

        - If byte, char, short, or int, c = (int)f
        - If long, c = (int)(f^(f>>>32))
        - Float, c = float.floattointbits(f)
        - Double, c = (int)(Double.doubletolongbits(f) ^(double.doubletolongbits(f) >>> 32))
        - If reference, call hashCode
    - Update result = 31*result +c
    - Exclude any redundant fields
        - Which you really don't' want anyway
    - Ignore fields ignored by…
- Always override toString
    - Make class much more pleasant to use
    - When practical, toString should return all interesting information in an object
    - One has to choose the format returned
        - Good idea to create constructor or static factory that takes string representation and creates object
    - Provide access to values in toString via getters
- Override clone judiciously
    - Creates and returns a copy of an object
        - X.clone() != x
        - X.clone().getClass() == x.getclass()
        - X.clone().equals(X)
        - Constructors are not called
    - If you override the clone method in a non-final class, return an object obtained by invoking super.clone
    - If class implements Cloneable, must have properly functioning public clone method
    - Beaty's general rule: Don't clone.
- Clone elementes too
    - The clone method is effectively another constructor
        - You must ensure that it does not harm original object and properly establishes invariants
    - The clone architecture is incompatible with normal use of final fields referring to mutable objects
- More clones
    - No need to provide a clone for immutable objects
    - Instead, can provide copy constructor or factory
        - Public Foo(Foo foo)
        - Public static Foo newInstance(Foo foo)
    - Interfaces should not extend Cloneable
    - Classes designed for inheritance should not implement it
- Consider implementing comparable
    - Similar to equals
        - But provides ordering information
        - Is generic
        - Useful in e.g. Arrays.sort()
    - Returns comparison between two objects
        - -1 if first is less than second
        - 0 if equal to
        - 1 if greater than
    - compareTo
        - X.compareTo(y) == -y.compareTo(x)
        - X.compareto(y) >0 and y.compareto(z) > 0 then x.compareto(z) > 0
        - X.compareto(y) == 0 --> x.compareto(z) == y.compareto(z) for all z
        - X.compareto(y) == 0 --> x.equals(y)

Effective Java 04

Wednesday, April 24, 2019 4:57 PM

CHAPTER 4

- Minimize accessibility of classes and members
  - The single most important factor that distinguishes a well-designed component from a poorly designed one is the degree to which the component hides its internal data and other implementation details from other Modules
  - Encapsulation
  - Decouples modules allowing them to be developed, tested, optimized, used, understood, and modified in isolation
  - Make each class or member as inaccessible as possible
  - If used nowhere else, nest a class within the class that uses it
  - Don't' make any variable/field/attribute public
    - At worst, make it package private
  - Try to avoid protected too
    - Must always support
    - Exposes implementation detail to subclasses
    - Should be rare
  - If a method overrides a superclass method, it must have the same access level
    - To not violate the Liskov inversion principle
  - Implementing an interface requires all methods to be public
    - Implicit in implementing an interface
  - Instance field should never be public
    - Never have a setter, because a setter makes an instance field public
    - Limits typing
    - Limits invariants
    - Are not thread-safe
  - Arrays are always mutable
    - Never have a public static final array field
    - Or an accessor that returns such a beast
    - Be careful of IDEs that create accessors automatically
- In public classes use accessor methods, not public fields
  - Book still insists on using lame examples of sets instead of simply making fields public
    - With the ostensible argument that we can change the internal representation
      - But we never do
      - And if we do, we break the preexisting API contract
    - Less harmful if immutable
- Minimize mutability
  - All information provided at construction
  - Any changes result in new objects
    - Which is in general true
  - Don't provide methods that modify an object's state
    - Mutators
  - Ensure class cannot be extended
    - Subclasses can't change intent
  - Make all fields final
  - Make all fields private
  - Ensure the client cannot obtain references to mutable data
    - Don't' use client-provided reference
    - Don't return direct object reference
    - Never initialize such a field to a client-provided object reference or return the field

        - from an accessor
        - Make defensive copies
    - Immutable objects are simple
        - Always the same behavior
        - Never global data
    - Immutable objects are thread-safe
        - Implicitly parallelizable
        - No synchronization needed
    - Only possible downside is the need for an object for each value
        - But: objects are, in general, cheap
        - Are you sure it's inefficient?
    - "classes should be immutable unless there is a very good reason to make them mutable"
        - At the very least, from an external point of view
    - If cannot be immutable. Limit mutability as much as possible
        - Make every field final unless there is a compelling reason not to
- Favor composition over inheritance
    - GO4
    - Inheritance violates encapsulation
        - Subclass depends on superclass's implementation
- Compose instead
    - An instrumented hashset that **has a** hash set instead of **is** a hashset
        - And extends a forwardingset class
- Inheritance
    - Is-A relationship
    - Is every instance of a subclass really an instance of the superclass?
        - If not, have a private instance of the referred-to class
- Design and document for inheritance
    - The only way to test a class designed for inheritance is to write subclasses
    - Constructors must not call over-ridable methods
        - Directly or indirectly
        - A superclass constructor runs before a subclass constructor, so any subclass methods that are overridden will be called before constructor called
- Prefer interfaces to abstract classes
    - Classes force inheritance
        - Java is single inheritance
    - Existing classes can be easily changed to implement interface
    - Interface are ideal for defining mixins
        - Loosely, a mixin is an additional type for a class
        - Useful for polymorphism
        - Know what methods are available to client, which in general define a type
    - Can create skeletal implementation for each interface
        - Generally call abstract*interface* (skeleton*interface* might be better)
        - abstractCollection, map, list, set
    - Abstract classes do permit multiple implementations
        - Easier to evolve
        - If you want to add a method, can add and implement
        - Everything else still works
    - Once an interface is released, much more difficult to change
        - Requires all dependent classes to implement new method
- Use interfaces only to define types
    - 'nuff said
- Prefer class hierarchies to tagged classes
    - Verbose, error-prone, and inefficient

- Imitation of a real class hierarchy

Chapter 4 continued…

Monday, April 29, 2019 4:57 PM

…

- Use function objects to represent strategies
  - Java didn't have method references or lambda
  - Could create objects
- Favor static member classes over non-static member classes
  - A nested class is defined within another class
  - Only serves the enclosing class
  - Four kinds
    - Static
    - Non-static
    - Anonymous
    - Local
  - Last three are called inner classes
  - For example, in a linked-list, nodes do not need to refer to head tail, etc., from list class
  - No need for node to contain all the data in list so it can be static member class
  - One way to think of this is that the static member class could be a separate class, but the code reads better with it inside
- Anonymous classes
  - Have no names
  - Not a member of enclosing class
  - Declared and instantiated at the same place
  - Permitted wherever expressions are allowed
  - Cannot have static members
  - Useful for creating functions objects on the fly

Effective Java 05

Wednesday, May 1, 2019 4:06 PM

EFFECTIVE JAVA 05: GENERICS

- Typing
  - Java's type system is very complex
  - It adds various mechanisms to add "generics"
  - Other languages simply have references to objects and duck-typing
- Generic types
  - Generic classes and interfaces are known as generic types
  - Generic types define sets of parameterized types
    - List<String>
    - Raw type is List
- Prefer Lists to Arrays
  - Arrays are covariant
  - If sub is a subtype of super, sub[] is a subtype of super[]
- Generics are invariant
  - List<t1> is never a subtype of List<t2>
  - Arrays vs Generics
- Arrays are reified
  - Their element types are enforced at runtime
- Generics are implemented by type erasure
  - Types enforced at compile time and erase type at runtime
  - Cannot create arrays of generic types, parameterized types, or type parameters
- Bounded wildcards
  - List<String> is not a subtype of List<object>
  - However, every object is a subtype of itself
  - So, can have a parameter that "extends" a generic
  - And have a parameter that is the supertype
  - Bit of a hack, really
- When to use
  - Use bounded wildcards in methods that have producer or consumer parameters
    - Maybe not a great idea anyway
  - PECS: Producer/Extends, Consumer/Super
  - Do not use wildcard return types
    - Client of class shouldn't have to know about wildcards

Effective Java 06

Wednesday, May 1, 2019 4:13 PM

ENUMS AND ANNOTATIONS

- Use enums instead of int constants
- Java
  - Java's enumerations are more powerful than other languages'
  - Almost classes
    - Can't extend, but can implement an interface
  - Export one instance of each enumeration via a public static final field
  - Enums are final
    - Only one instance
  - Provide compile-time type safety
  - Can't pass or assign incorrectly
- Can construct
- Enumerations
  - Don't count on ordinal values
  - Use constructor instead
  - Use EnumSets instead of bit fields
- Annotations
  - Prefer annotations to nameing patterns
    - JUnit a major example
  - Consistently use @Override
    - Makes sure you are actually overriding
    - Especially for equals, toString, hashCode
- Use marker interfaces to define types
  - A marker interface is one with no methods
  - Serializable is an example
    - Indicates object can be written via ObjectOutputStream

Effective Java 07 (08)

Wednesday, May 1, 2019 4:44 PM

Methods

- Check parameters for validity
    - Most parameters have restrictions on their validity
        - Positive, non-null, not zero-length, etc.
    - AKA preconditions
    - Program defensively
    - Catch problems as soon as possible
    - Fail fast
    - If it fails
        - Method may die
            - Or worse, work but in an unexpected way
        - Throw an exception
            - An illegalArgumentException a good choice
    - Assertions
        - Optional in java
            - Must enable with -ea
            - Or in first class (and doesn't enable them there):
                - Static {
                    - classLoader.getSystemClassLoader().setDefaultAssertionStatus(true);
                - }
    - Check parameters for validity
        - Important for maintaining object consistency
            - Values stored for later use must be good
            - Checks in constructors important
        - Check before doing any calculation
            - Unless calculation does the checks for you
    - Make defensive copies
        - Assume the worst of your class's clients
            - They will modify your invariants
    - Arrays
        - Non-zero-length arrays are always mutable
        - Make defensive copies
    - Design method signatures carefully
        - Good names
        - Short parameter lists
            - If all identical, maybe use varargs
    - Short parameter lists
        - Maybe pass a class that encapsulates multiple parameters
        - Maybe use the builder pattern
        - Prefer interfaces for parameters
        - Avoid Booleans, favor enumerations
    - Use overloading judiciously
        - Overloading happens at compile time
        - Here, there was an array of Collection<?>
        - Overloading is static (happens at compile time), overriding is dynamic (happens at runtime)

        - Overloading is: having a method with the same name within the same class
    - Overriding
        - Overriding is: the same method name from a super class
    - Overloaded meaning
    - Don't return nulls
        - Return empty arrays or empty collections
        - Collections.emptyset, emptyList, emptyMap
    - Write good JavaDoc Documentation
        - For all your visible items