Computer Science

- ⇨ Problem Solving using computer effectively
- ⇨ **Program = Data Structure + Algorithm**

Data Science

- ⇨ Computer Science + Statistics + Math
- ⇨ Machine learning vs. data mining
    - Machine Learning
        - Small size data
        - Can afford to use complex algorithms, lots of statistics
    - Data Mining – Hard-core database
        - Involves big data: trade time/space for efficiency, hashing, indexing
        - Knowledge discovery → Try to extract knowledge from raw data
        - Beer and diapers data mining story
- ⇨ Deep learning
    - Huge data + deep algorithms

Big Data Era

- ⇨ O(1) – Fixed number of auxiliary variables
- ⇨ O(N) – No-No, aim for O(1)
- ⇨ $O(N^3)$ – Not Acceptable
    - Even gives optimal result, still need to settle for sub-optimal result for less time
- ⇨ sublinear: $O(N^x)$ where $x < 1$

Algorithm

- ⇨ In-place sorting: QuickSort
- ⇨ Search: O(log N) for already sorted, O(1) for hashing H(val) → index
- ⇨ Downside for Hashing
    - Collisions, new time complexity O(log log N)
    - Space Complexity: To avoid collision, hashing need 2N space requirement
    - Time Complexity: Better if no collision

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

Math/Statics Need

- ⇨ Machines Learning: Linear Algebra, Taylor Series,
    - Gradient, descent based on Taylor series
- ⇨ Cluster / Dimensionality Reduction
- ⇨ Matrix Analysis Ax = lambda x eigen values/vector
- ⇨ Tensor: differential geometry, Gauss vector, matrix, higher dimensional space

- ⇨ MATLAB: Matrix Lab

Skip List

- ⇨ Widely used in big data applications
- ⇨ A data structure that allows fast search within an ordered sequence of elements
  - A linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one
- ⇨ Copycat of the subway system of NYC
  - Transfer to express lines
  - If all local stations: O(N)
  - If there is an express line: $O(N^x)$ → sublinear
- ⇨ N local stations, Using Divide and Conquer
  - Avg # of local stops before adding express stop: **N/2**
    - Time Complexity O(N)
  - Assume have **m** express stations
    - Avg # of express stops: **m/2**
    - Avg # of local stops: **N/m * ½** → local stops btw 2 express stops: N /m
    - Avg # of stops in total:
      - **F(m) = ½ * (m+ N /m)** → Formula of Avg # of stops in total
    - Best/Optimal m in which minimizes F(m):
      - $F'(m) = \frac{1}{2} * (1-N/m^2) = 0 \rightarrow m = sqrt(N)$
    - Time Complexity: ½(sqrt(N) + sqrt(N)) = sqrt(N)
    - O(sqrt(N)) sublinear: N^x where x < 1

Three System Description Techniques

- ⇨ **Informal:** Natural Languages
  - Common Features: Redundancies, Robustness by redundancies
  - **Pros**: Easy to Understand, More Robust
  - **Cons**: Ambiguity on both parties (Fatal)
- ⇨ **Semiformal**: Combination of Math + Picture + Natural Language
  - Example: Use-Case Diagram, E/R diagram
  - **Pros:** Reduce Ambiguity, Easy to Understand
  - **Cons:** Room for Ambiguity is big
- ⇨ **Formal**: Totally based on Math/Logic + Diagram, No need Natural Language
  - Example: Finite State Machine, Petri-net
  - **Pros:** No Ambiguity, get code automatically
  - **Cons:** Hard to Develop & Understand, not good to describe large system

Computable vs. Infeasible

- ⇨ **Three Complexities**
  - o **Time** - relative to the input size
    - ▪ Number of steps a program takes to run. Take highest order of time complexity
  - o **Space -** relative to the input size
    - ▪ How much memory or space is taken up when running a program. Only care about the highest order or degree.
  - o **Kolmogorov -** Chaitin-Complexity (1969)
    - ▪ Concise and Understandability
    - ▪ Measure Number of Lines – More line implies more complexity
- ⇨ **Efficiency**
  - o Measure in combine Time and Space Complexities together
  - o Trade-off between time and space, choice of smart algorithm or smart algorithm

Measure the Software Quality

- ⇨ **Satisfy Specification** (to get paid $$$) (most important quality)
- ⇨ **User-friendliness** (reason why apple is so rich, to customer)
- ⇨ **Understandability** to other coders
  - o more comments, naming convention,
  - o clear logic: subjective
- ⇨ Correctness: no correct system b/c no complete induction
- ⇨ **Efficiency (Complexity)**: Time and Space
- ⇨ **Reliability vs Robustness** (crucial quality measure of software):
  - o Defensive → **Robust**: if silly input, system won't crash (handle reasonable errors);
    - ▪ Assert, Try/Catch, Throw
    - ▪ Without try-catch, can't mix functional code and error handling code together; **understandability** not good
    - ▪ Try is the functional part, catch is error processing
    - ▪ logically robust: random anchor in quicksort
  - o **Reliability**:
    - ▪ program performs as intended (do what supposed to do)

Software Crisis in the 1960s/70s

- ⇨ History
  - o First computer showed up in 1940s

- Computers originally used for military

- ⇨ **Software Crisis** in 1960s/70s
    - Projects were **always late** and took years to complete
    - **Unreliable, Hard to Maintain**: succeeds one day and fails another,
    - **Expensive**: $1-10 million to develop, maintain, and upgrade

- ⇨ **Software development** (before software engineering)
    - Original Way Develop A System: Try-and-Fix
        - Run/Using the system until a bug show up and then fix it
    - **(Develop + Use) + (Find Bugs + Fix) → Code + Fix (Very bad, Origin of Crisis)**

- ⇨ **Solution to software crisis**: Software Engineering
    - Software engineer is a copycat of conventional engineering
    - Example: Build a Bridge
        - Wrong Way: "code + fix" policy:
            - Build bridge quickly, Let people walk/run/drive in it,
            - If some dropped out the bridge, then fix it.
        - Right Way: Engineering Way **(Water-Hall Life Cycle Model)**
            - **Analysis/Research in-field studies**
                - System Analysis → What to do
                - Feasible: know NOT every problem can be computerized)
            - **Planning** (software/hardware/personnel)
            - **Design the Project**
                - High Level (Architecture [num class, connections])
                - Low Level (Type data structure and algorithm))
            - **Construction** (Implementation(coding) + debugging)
            - **Testing**
                - Internal testing: IBM testing group alpha version
                - External testing: beta version, experts, release system for free (google beta)
                - Dynamic: initially beta = alpha, finally == deliver version, beta version may have many bugs not completely finish.
                - Integration for some group is part of testing
            - **Delivery**
            - **Maintain**: Fix bugs/troubles (Maintain System)
            - **Death/Retirement** of System

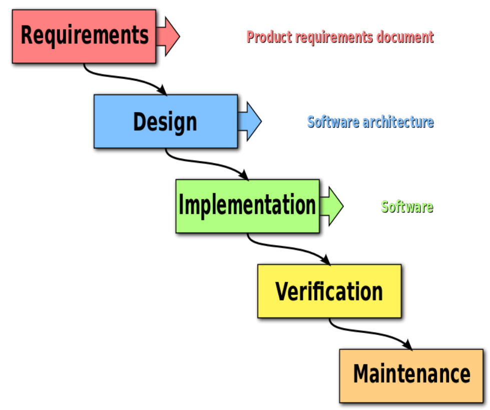
Software Engineer VS Conventional Engineering (Major Difference)

- ⇨ Software Physical Thing: Tangible/Concrete
- ⇨ S.E. → Cause: Lack of visibility of efforts, cannot visualize the software system
- ⇨ S.E. → Result: **Moving Target**, client ask more change specification in software project
    - Not easy to convince work, change small target, require much effort

- Difficult communication between software engineers and clients
- OO Programs is better to deal with Moving Targets

Life Cycle Concept

- **Life Cycle Model – 5 Types**
  - **DEF**: A series of steps taken to develop a software system
  - Build and Fix not a Life Cycle Model
- **Water-Fall Life Cycle Model (W.F.)** – 1st Life Cycle Model
  - **DEF**: Process flow step by step:
    1. Investigation: specification
    2. Planning: people/money/s/h
    3. System design
    4. Implementation
    5. Integration
    6. Testing
    7. Delivery
    8. Maintenance
    9. Retire



  - **Pros**
    - Well Defined Stages, Better Quality Control
    - Documentation = Specification + Plan + Design + Test + Maintenance
  - **Cons**
    - Time Consuming
    - Lack of Communication between Clients/Developer, Developer/Developer
- **Rapid Prototyping (R.P.)**
  - Quickly Build a Model reflect on Major function/feature of system by using Scripting PL without worry about low-level features
  - Reduce Misunderstanding
    - Scripting PL: Python, R, SQL
    - Productional Code: C/C++/Java → Using to build important/actual system
  - **Use W.F. + R.P.** (used 90% of times)
    - **Pros:**
      - **Improve Communication**
      - Reduce the overall risk & Quick respond and feedback
    - **Cons:**
      - Scaling, toy system not fit for large system (lot people)
      - **More moving targets** due to more communication
- **Evolution** Not popular at all < 3%
  - Four Steps (**DEF**) Start from **Simple** → **Complex**
    - **Partition** system into number of builds
    - Use **W.F.** + **R.P.** to construct each build

        - Client decide likeness (get feedback)
            - If client like → keep work on 1st partition
            - If client don't like → continue if client & developer like each other
        - Continue build partition until finished
    - **Pros**
        - Reduce overall risk on both side
            - Developer: Build one part and get feedback immediately
            - Client: Can discontinue work with developer if the developer bad
        - Improve Communication
        - Reduce number of errors
    - **Cons** - Fatal Problem
        - **Hard (almost impossible) to integrate/put together**
        - Need wisely partition, Problems need to solve in later partition
        - More Client idea, may change specification during the development
- **Spiral**
    - **W.F. + R.P. + Risk analysis for each phase** (prob/stats, reliability/robustness)
    - Rigorous Risk Analysis
        - Require Math + Statistics to identify risks, way to reduce risk to target threshold
        - Estimate Rate of Success
            - Time & Money Consuming, usually only used for military systems
        - Ex: A thief steal with 99% success rate, 1% get caught by police. Chance of get caught within 1 year?
            - P (success everyday) = 0.99^{365}
            - P (get caught) = 1 - P(success everyday) = $1- 0.99^{365}$
                - As $x < 1$, limit $n \to$ infinity, $x^n = 0$
                - Eventually he will get caught as days increase
    - **Pros:**
        - Reduce overall risk by statistics (not by communication)
        - Safer, more robust, more secure of the system
    - **Cons:**
        - Utility is limited to **military** or national security or **big bank** only
        - Takes much longer, Annoying
- **Agile**
    - **DEF**: **More Micro-management** alone with development.
        - Values human communication and feedback, adapting to changes
        - daily standup (everyone)
        - 1-2 week sprint (leader only): ticket
        - Dream for project manager
    - **Pros:**
        - More Communication (Standup, example below)
            - among developer
            - leaderboard meeting/sprint between developer/client (more frequent)

- **Cons**
  - Less defined stages/phases
  - Hard for risk Analysis at early stage
  - Hard to assess
  - More moving target

*Choose which life cycle model use for different situations, need justify choice

Stream Data Processing

- ⇨ Computer Programming Paradigm
  - Similar: Dataflow Programming / Event Stream Processing / Reactive Programming
- ⇨ Easily exploit a limited form of parallel processing
- ⇨ **Data Streaming**
  - River of Data (Huge in Size)
  - A **sequence of digitally encoded coherent signals** (packets of data) used to transmit or receive information that is **in the process of being transmitted**
  - At any time, we have knowledge and should be able to respond
  - Our knowledge/belief may change over time
- ⇨ Example 1: N customer, $N \rightarrow \infty$. From 1 up to N, have a bonus $$, each one only takes $1, N-1 customer already claimed the bonus. Find who didn't claim in most efficient way
  - Parallel Processing, $$$ not good solution in this case
  - Solution only work with only one missing – Sum Total ID

    Sum = 0

    Sum += ID // for each came customer

    Total sum = 1 + 2 + … + N-1 = N(N-1)/2

    Total sum – Sum = Missing ID

    O(1): Random Shuffle of N id's by blocking the last one
- ⇨ Example 2: Similar as Example 1, but now with two people missing
  - Base-Einstein Number: N Unknown, N Equation
  - Algorithmic Solution (Don't do the product)

    Sum1 += id
    Sum2 += id**2
    …
    Sumk += id**k

  Find the unknown:

    id1 + id2 +… + idk = Sum1
    …
    id1**k + id2 **k+… + idk**k = Sumk
- ⇨ Example 3: Flowing data, how to computer the average value on the fly?
  - Wrong Way that will crash OS, b/c space O(N) too

```
int A[N]                    N → ∞
for i in range(N):
        sum += A[i]
print(sum)
```

- Right Way, Only need O(1) Space

  Initial avg = A[0]   // variable to save current average

  avg = (A[0] + A[1])/2 = avg + (A[1]-avg)/2 = avg + diff/2

  avg = (A[0] + A[1]+A[3])/3 = (2*avg+A[2])/3 = avg + (A[2] – avg)/3

  …

  avg = avg + (A[k-1] – avg)/k = avg + diff/2 = avg + *eta**diff
- *eta* is learning factor (1/k) in **Machine Learning**
  - when k is very small, update belief a lot, since learning factor is 1/k
  - when k is very large, will end up no learning
- **Convergence**: Learning factor should fade as time goes on, 1/k is perfect learning. The more data the better, however too much data too little to learn
- *** **new_belief = prior_belief + learning_factor * difference(evidence)**
- **Bayesian theory**
  - Update the prior belief (old_avg) based on new evidence (new_data)

Bayes' Rule

- ⇨ **Probability/Statics**
  - Statics is collection and make sense of data, more like induction
  - **Classic Static**: Based on **frequency** (Fisher, 1940) → Strict
    - Fair coin: p(h) = p(t) = 0.5, prove by large amount of trials
  - **Subjective/Bayesian Static**: Based on **Believed** (Bayes > 200 years age died)
    - Modern, Soft
    - Subjective/Believe system
    - P(coin is fair | john) = 1 or P(coin is fair | jane) = 0
- ⇨ **Bayes' Rule** (Used In CS)
  - Prior belief/knowledge can be changed by experiment/evident (new data)
  - Every probability is **conditional probability** → P(k | conditional) : your belief
  - $\boldsymbol{P(A|B) = \frac{P(A)*P(B|A)}{P(B)} = \frac{P(B|A)*P(A|Original\ Knowledge\ Base)}{P(B)}}$
  - $-log(P(A|B)) = -log(P(A) - log(P(B|A)) + constant$
    - Information theory → Learning factor * evidence
- ⇨ **Shannon Theorem** (A theorem of information theory)
  - entropy or information of certain event -log p(e) as the info
  - 1st Axiom – Compression
    - $H(X) = -\sum_{j=1}^{m} p_j log_2(p_j) = Entropy\ of\ X$, X can be one of m values
    - High Entropy → X is from uniform distribution
    - Low Entropy means X is frim varied (peaks and valleys) distribution
  - 2nd Axiom – Channel Theory: Communication
    - Bar code: parity code
  - IT: surprise measure p(e)

- Any news worthy item should have threshold of entropy surprise
- p(e) based on how often/rare certain event is
- Ex: p(e) = 1, log(1) = 0, not surprise at all
    - e = "the sun rises from the east"
    - e = "a dog bite a human being"

- Statistic: p(e) more power to represent real world
    - Use small number to represent a population

Classic statistic: Gaussian distribution/Normal distribution

- **Median** = the middle value in a group of ordered numbers
    - The median is **more robust**(strong) than the average
    - Median is **not affected by outliers**
- **Mean** Average NOT **robust**
    - Sensitive to outliers
    - If you square a value that is very large, the result will be even larger
- Robustness/Unrobustness of Mean/Median
    - Gaussian formation:
        - Many data: modeling, then optimize
    - Given N data items, find one number faithfully represent the data
        - Formulate Math: Fitting Errors
            - N difference close to Zero $x_1 - a,\ x_2 - a, \dots, x_n - a$
        - Mean Square Error (**MSE**) → Make Outlier Value too much
            - $f(a) = (x_1 - a)^2 + (x_2 - a)^2 + \dots + (x_n - a)^2$
            - Optimal a → Minimze $f(a)$
                - $f'(a) = -2 * (x_1 - a) - 2 * (x_2 - a) - \dots - 2 * (x_n - a) = 0$
            - $n * a = \sum x_i \rightarrow a = \frac{\sum x_i}{n} \rightarrow a$ is the mean value
            - **Mean Value minimizes MSE(Gaussian),** but the square makes the outlier too much, SO NOT GOOD
        - Mean Absolute Difference (**MAD**)
            - A measure of statistical dispersion equal to the average absolute difference of two independent values drawn from a probability distribution
            - $f(a) = |x_1 - a| + |x_2 - a| + \cdots + |x_n - a|$
            - **Piecewise function**
                - $f'(a) = \begin{cases} -1, & |x_i - a| < 0 \\ 1, & |x_i - a| > 0 \\ 0, & |x_i - a| = 0 \end{cases}$
            - If a is the Median value of x_i
                - Optimal: $f'(a) = 0$
                    - $-1 * \frac{n}{2}\ for\ half\ difference < 0$
                    - $1 * \frac{n}{2}\ for\ half\ difference > 0$

        - Sum up and cancel out, so $f'(a) = 0$
      - **Median value minimized MAD**, thus more robust than mean
  - Robust statistic: how to make classics stat insensitive to outliers
    - Robust created in 1950-60s since Gauss failed to make robust something
    - Get Median: Sort the array
      - If odd # of data, just use the middle value
      - If even # of data, just take the mean of the middle 2 values

Gold Standard of Science

- Criteria of science: **Repeatable**
- **Randomized** programming and testing, major component of S.E.
  - Fisher (Father of Modern Science) → Major contribution is Randomization
    - Let it be comparing crops: f1, f2, which one more effective
      - Many factors out of control
      - **Distinguish cause & effect vs correlation**
      - Test in same year, partition the field into many small fields and randomly assign crops into those small field, compare performance
  - Stat test: Size/population should be large
    - **Poll**: error of margin 3 %, if >3% useless info
- **Double Blindness**: scientific way to decide goodness
  - Test/Experiment design fairly, well design queries
  - Observation ≠ Science
  - Search: NOT know which search engine is being used
  - Evaluate Performance: NOT know which results are evaluate blind
- Example: 3 ppl in a leaking boat, sharks around, if one out of boat other two will survive
  - Fairness: randomized algorithm
    - All 3 sign a contract before anything
    - Throw a dice or something

Mathematics Modeling

- Essential to Software Engineering
- **Hashing Function**
  - Goodness - Math feature: totally uniformly distributed
    - Avoid Collision， randomness
  - Ex: n balls, n bins, n throws by monkey randomly, chance of a bin to be empty after n throws:
    - Possibility of a bin be empty after 1 throw: $(1 - 1/n)$
    - Possibility of a bin be empty after n throw: $(1 - 1/n)^n = e^{-1} = 38\%$
      - Throw are independent $(1 + 1/n)^n = e$

Reliability Analysis

- **Explore and Exploit** (Think as Machine Learning)
  - Explore – See around world gain experience
  - Exploit – Take advantage of experience to make gains

- Example – Save early birds as Exploring experience
    - Total M candidates → Testing Sample: M-k; Training Sample: k
        - Too Small k– inadequate exploration
        - Too Big k– over learning
        - Right choice of k: 1/e, 38% for exploring
    - Exploring: Ranking/explore: first k candidates, see how good/bad first k candidates are, and never make any offer to them
    - Exploit: After See Enough make decision/exploit our experience

4 Generation of Programming Language (PL)

- ⇨ 1st Generation: **Binary Machine Code** in 1950s
  - Good for machines (CPU), impossible to understand/communicate/share
  - Ex: 00010110
- ⇨ 2nd Generation: **Assembly =>** Hacker Must Know
  - Size smaller than any other PL
  - Better Understandability VS Machine Code
  - 1-1 correspondence between assembly and machine language
  - Ex: add ax, 01: 010101: 01: add, 01: ax, 01: #
- ⇨ 3rd Generation: **Procedural Programming Language**
  - Big Three Types of Statements
    - Conditions
    - Loop/Recursion, should be ~10 lines or < 20 lines
    - Sequence (van Neumann: Binary System, merge Sort, Sequence)
  - **Turing Machine (TM)**
    - Read/Write/Erase header on the tape
    - **Turing Complete**
      - A PL is TC if can used to simulate any Turing Machine
    - Turing-Church thesis:
      - A function on the natural numbers is computable by a human being following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.
  - **Functional Programming (FP) – Programming Paradigm** (3 Types):
    - **Imperative** (All PL)
      - Uses Statements that change a program's state
      - Focus on describing how a program operates
      - Consists commands for the computer to perform
    - **Functional**
      - Treats Computation as the evaluation of mathematical functions
      - Avoid change state
      - Declarative Programming Paradigm
        - Program done with expressions/declaration instead of statements
      - Ex: LISP – influenced Scala, Haskell, …
        - No assignments, No Loops, only Recursion
    - **Logic Programming**
      - Any program written in a logic programming language is essentially a set of sentences in logical form
      - **Prolog**: Programming Logic
      - **Develop new concept based on existed concept**
      - Ex: Predicate:

        Tiger (x): true or false x is tiger

        Tiger ('Tiger Woods'): False

Tiger(x): Animal(x) and Stripes(x) and eatMeat(x) and…

- Ex: Pascal language
  - First programming language in the whole world for many years
  - Useless by industry – Pure 3rd-gen language
  - C: 2.5-gen, assembly + pascal.

⇨ 4^{th} Generation – Several Direction

- **Object Oriented (OO) languages**
  - Java → pure OO
  - C++ → partially OO + Assembly + Procedural PL
  - Important feature of OO
    - **Inheritance** (1^{st})
      - Better Reusable, all of the code is in one place
      - Easier to fix the bug
        - localize features in order to localize errors
      - Better than copy and paste
        - Copy and paste is bad, **no correct software system**
        - Ex: if we have a student, then a grad_std (graduate student) would just be a student + a new feature
    - **Polymorphism** (2^{nd})
- System Query Language (**SQL)**
  - Database reliant, **Not TM** equivalent
  - Directly process DB
  - Type of **declarative PL**
    - Tell what to do NOT how to do
  - DB – Small OS
    - Decide which order to perform filter
    - Meta data – make DB smart
    - Big company: with many hotness/popularity indexes, keep update
      - Machine learning/adaptive
  - Example Code

```
Select name
From std
Where gpa>3.2 && gender = 'M'
```

GPA Filter and gender filter, which filter are more selective,
Reduce data volume

There is no **correct** software system

⇨ All testing: incomplete induction

⇨ In CS, test as many cases as possible

Complete Induction

⇨ Prove by completely enumerate all cases: derivation

⇨ Disprove by example, one counter-example is enough

Any System is **Error-prone**

⇨ Easy fix bugs

⇨ Inheritance: fix once, felt everywhere

Lazy Evaluation: being lazy: yield in python
First Class Function: Pass a function as parameter

Computer Aid Software Engineering → CASE

- **Tools**
  - Aid to improve efficiency
    - not need to work from scratch, used existed tool or self-build library
  - Citation: Give Credit to others if used
  - Smart Editors: Sublime, vim, idle,
  - Collaboration/Organize Project: Git (hub, lab), bitbucket
    - Gitlab/bitbucket: Privacy
    - Version control using git
  - OS: Linux / MacBook (base on Unix): Shell Programming
    - Windows: Batch Programming
    - Automate Action Use Shell/Batch/Python
- **Concept Exploration → Exploratory Data Analysis (EDA)**
  - Python: #1 language in data science
  - Data Wrangling
    - matplotlib (copycat of MATLAB plot), pandas (copycat of R), seaborn

Feasibility Study

- Decide whether or not the system is possible to implement the system, by determine whether or not the algorithm is NP-hard.
- Not all problem is solvable.
- NP-Complete: Hamilton Circuit/TSP: Traveling Salesman Problem,
  - P or not? Lower order exponential
  - Example 1: Digital Camera + Computer Replace 95% Security Guard?
    - 5 Years Ago: NOT Tech Feasibility
    - Now: Yes, with Deep Learning
    - Human Face: Too many variations without deep learning
      - Deep Learning study with many samples of single subject

Mutual Misunderstanding

- Software Developer and Domain Experts from different world(view)
  - Different people can get different meaning from same statement.
- Between developers and clients, developers and developers
  - *** Ensure true meaning of any words/sentences
  - You think you understand, but you really don't
- Example: Replace Handle
  - Understand 1: sub handle with a new one
  - Understand 2: Re-place, put it back
- **MM Causes**

- Natural Language: Ambiguity
  - bank → can be river bank or money bank
  - 汽车 → car in Chinese but train in Japanese

⇨ **Reduce MM**
- More Nature Language, more easy.
- Use Universal Language: Math (Galileo), formula
- **Figures/Diagram**
  - Unified Modeling Language (UML) standard software engineering toolset
  - Finite State Machine (FSM) not UML
    - petri-net (superset of FSM)??? petri-net: dynamic (UML)
  - Regular Language, **Class Diagram**
    - E/R Diagram (Entity-Relation)
      - one type of Class Diagram that represents dataset
  - Design = Algorithm (petri, FSM, …) + data (E/R)
- **Rapid Prototyping**: Build small model for the significant part of the system

Data Structure – Data Base

- Example Student Database: std_id, name, dept, dept_chair, college, college_president

1. **Correctness of Database**
   a. Remove the space redundancy, update redundancy, ex: dept_chair, college_president
   b. Create **Data Dictionary**, Class def / DB refinement
      i. Relational Data Base (**RDB)**
      ii. So student DB not include college_president, but college_president is link to college in the Data Dictionary
   c. **Fewer Attribute the Better, but not too small**
2. **Identify the Key**: Set of Attribute uniquely ID the tuple
   a. Don't Want to introduce new attribute just for convenience
   b. Used existed data for KEY, key can have more than one attributes
   c. Example Roster DataBase

ssn	course_id	grade	|	Semester	Year
999	32200	A	|	Spring	2018
888	32200	F	|	Spring	2018
999	22100	B	|	Spring	2019
888	32200	A	|	Spring	2019

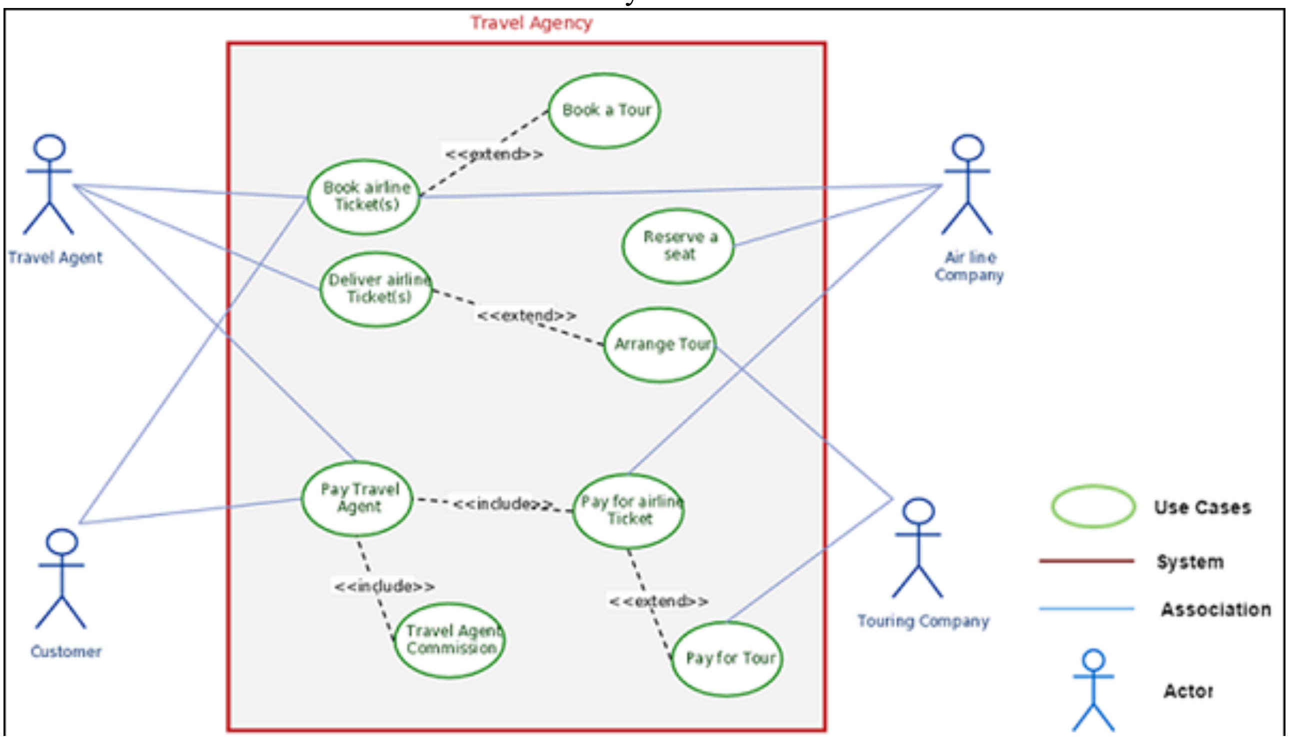
- SSN cannot be the KEY because of duplicate, any single attribute cannot be key
- (ssn, course_id) is the key at left, but not work if a person retakes the class

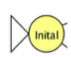
        - Need more attributes, ex: Semester and Year
    - Then the key is 4 attributes (ssn, course_id, semester, year)

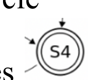
Python: type (inefficiency)
Cython: Add some type declaration == C/C++

Unified Modeling Language (UML) **Methods**

- **Use-Case Diagram** – Need identify all the users
  - Drawing Steps
    - **System Boundary** → Large Rectangle
    - **Ordinary Users** → left side of the large rectangle
    - **Privilege Users**(insider) → Right side of the large rectangle
    - **Features** → **Ovals** Inside Rectangle
      - Run through all possible features
    - **Line Connections** → Identify who will be users of each features

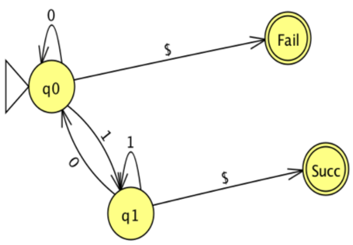


- **Finite State Machine** (FSM) **–** Formal Method, not UML
  - **Three States**
    - **Initial** State: Denote by pointer to a circle 

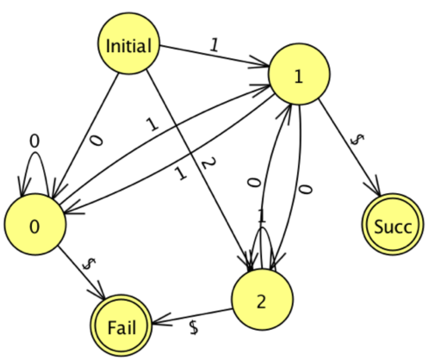
    - **Final** State(s): Denote by double circles 

    - **Intermediate** States: Any states between Initial/Final, denote by single circle
  - All States exclude Final State need to handle **All Tokens**
  - Could have more than one Final State (e.g. Success and Fail)
  - **Binary Stream**
    - **Tokens**: 0, 1, $(terminating symbol)
  - **Ternary Stream**
    - **Tokens**: 0, 1, 2, $(terminating symbol)
  - **Octal Stream**
    - **Tokens**: 0, 1, 2, 3, 4, 5, 6, 7, $(terminating symbol)
  - $ may not need for certain FSM

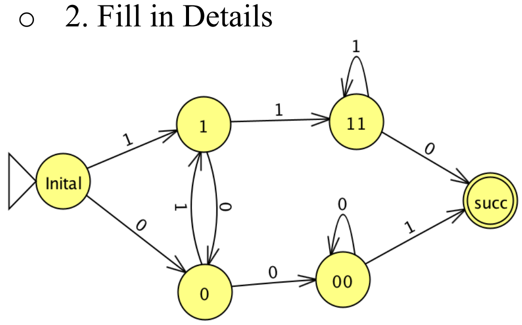
- Stream read from Left to Right
- **Examples – All in Binary Stream**
    - FSM recognize odd Number
        - Two Final States: Succ for odd number, Fail for Even number
        - Let q0 = 0, q1=1
            - q0 is initial state only if the current last digit



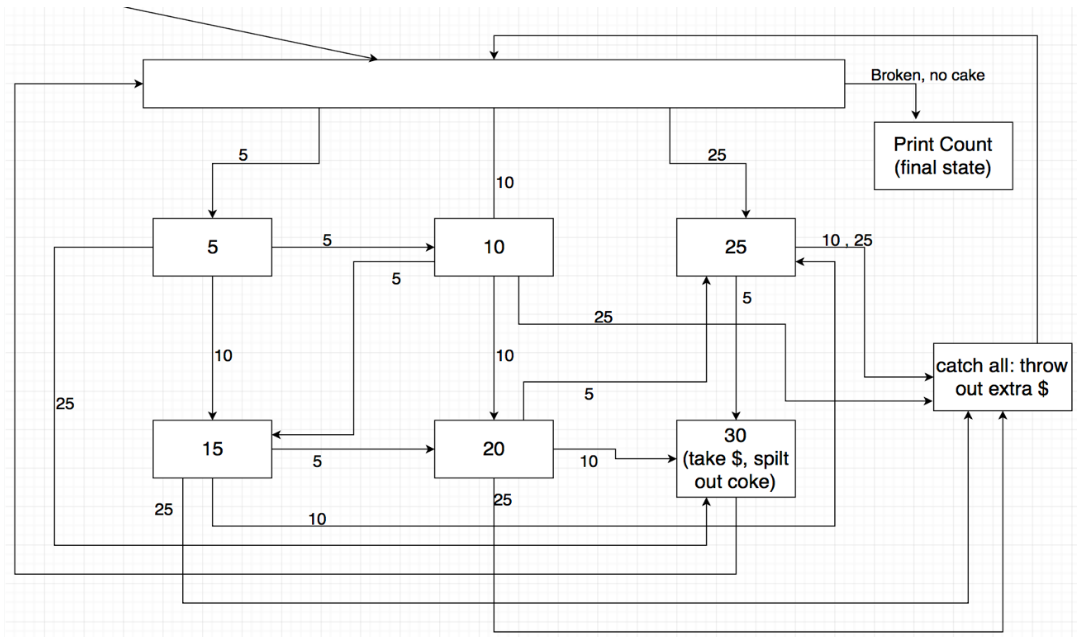
    - FSM: n%3 = 1, n is binary number
        - Three possible remainder of 3: 0, 1, 2
            - So, three intermediate states
        - If current remainder is 0, let current number be 3
            - If incoming symbol = 0
                - New number = (current number*2) + incoming symbol
                - (3*2) + 0 = 6   6 % 3 = 0
                - The new remainder is 0, stay in its state
            - If incoming symbol = 1
                - (3*2) +1 = 7   7 % 3 = 1
                - The new remainder is 1, change state to 1
        - If current remainder is 1, let current number be 4
            - If incoming symbol = 0
                - (4*2) +0 = 8   8 % 3 = 2
                - The new remainder is 2, change state to 2
            - If incoming symbol = 1
                - (4*2) +1 = 9   9 % 3 = 0
                - The new remainder is 0, change state to 0
        - If current remainder is 2, let current number be 5
            - If incoming symbol = 0
                - (5*2) +0 = 10 10 % 3 = 1
                - The new remainder is 1, change state to 1
            - If incoming symbol = 1
                - (5*2) +1 = 11 11 % 3 = 2
                - The new remainder is 2, stay in its state



    - **Pattern Recognition**
        - Binary Stream, recognize any pattern with form of 110, 001
        - Suppose no terminating token $
        - **Steps**
            - 1. Construct Prototype
            - 2. Fill in Details

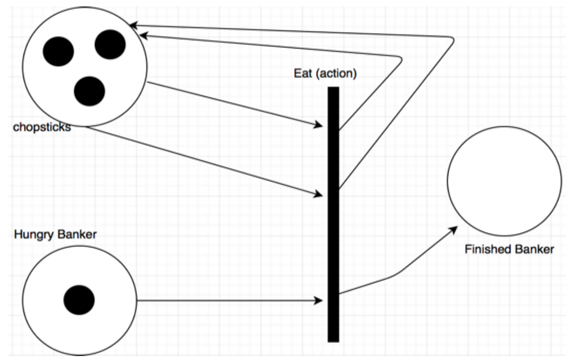


- ⇨ **State Diagram** – Behavior Diagram, UML
    - ALL **Rectangle** VS Circles in FSM
    - Long Bar → Initial State / Based State
    - May not have a Final State
    - State Diagram represent Loop/iteration of a system VS FSM for One-shot
    - Example: Vending Machine only accept 5, 10, and 25 cents
        - **Number of tokens** = possible coins can have
            - In this case is 3 tokens

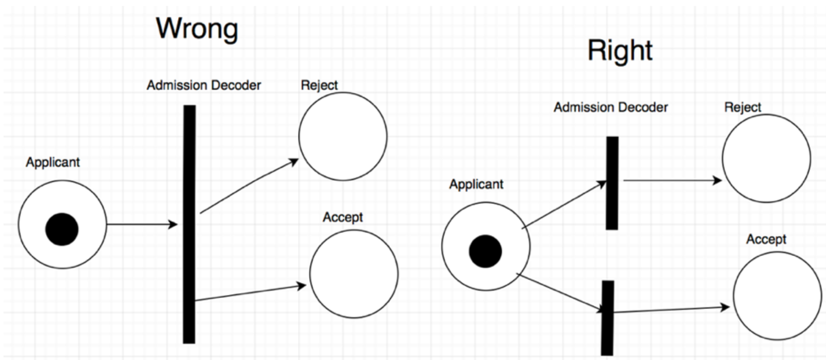


- ⇨ **Petri-Net** – Not UML, too difficult
    - Superset of FSM and State Diagram, TM-equivalent
    - Motivation: System Resource / Dynamics
    - Avoid all too powerful transitions, don't put too much
    - Never connect 2 transactions/places directly
    - Avoid **OR** gate, all transactions are **AND** gates (Input and Output**), separate actions**
        - If #tokens_in_s1 >= 2 AND #tokens_in_s2 >= 3: fired
    - Components
        - States → Represent by circles
            - Tokens → Represent by dots inside the circles
                - Tokens represent system resources
        - Transition → Represent by arrows (→ | →)
            - Either fired (there is a dynamic action) or blocked
            - Fire/dynamic action
                - $\# \ of \ tokens \ in \ each \ input \ place \geq \# \ of \ input \ legs$
                - If fired, removed # of token in the input place per # of input legs, add # of token in the output place per # of output legs

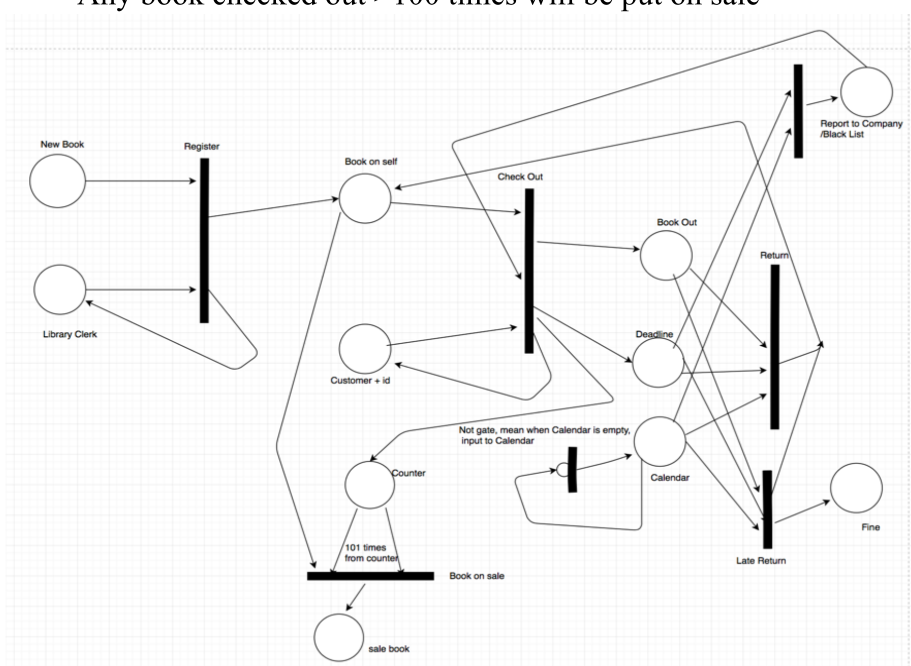
- Example: Bankers need two chopsticks to eat
  - Initial Condition: 1 banker, 3 chopsticks
  - Banker able to eat (fire action)
  - After eating, two chopsticks return to the poll of resource
  - But if there are 3 bankers, deadlock and not action is fired



- Example 2: Admit student – Two output
  - Two Version: Right and Wrong
  - Separate OR gate by using two **AND** gate



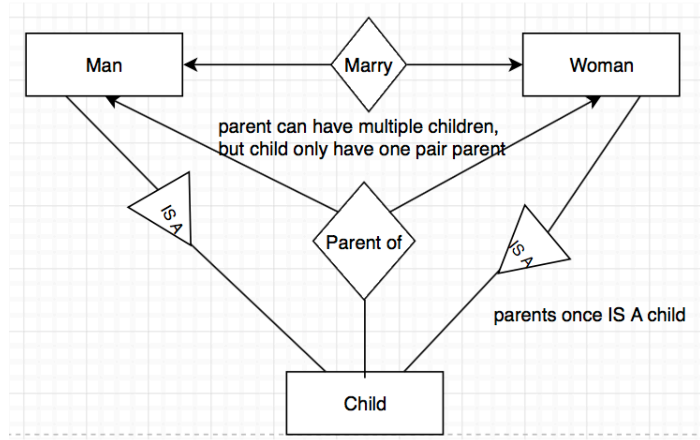
- Example 3: Below is list of requirements, Draw one large petri-net, not individuals
  - New book should be registered before shelfing to check out a book
  - Customer should have legal id
  - Book returned after deadline will be fined book not
  - Book return after 1 year will be reported to collection company.
  - Any customer reported will be barred from borrowing,
  - Any book checked out >100 times will be put on sale

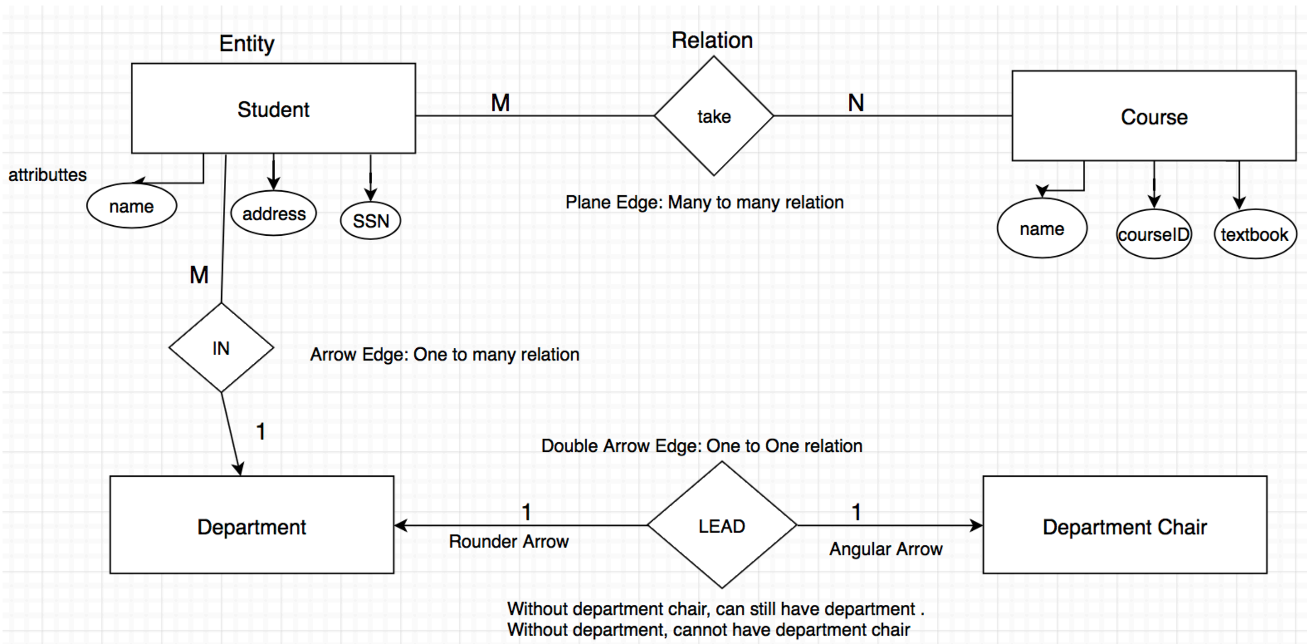


- Look at the not-gate of calendar, shows never run out of resource

➪ **Entity Relation Diagram**

- **Rectangle**: Entity
- **Oval**: Attributes
- **Diamond**: Relation
- **Edges/Lines**: 3 type relation
  - Many-to-Many: Plane edge
  - One-to-One: Two Arrow
  - One-to-Many: One Arrow
- **Two Arrow Types**
  - Rounded arrow: mandatory need
  - Angular arrow: optional need
- **ISA** Relationship (**SubClass**), show by triangle





➪ Regular Language (subset of TM)

- Example: FSM and State Diagram
- **Cons**:
  - Limit to only small part
  - Not good at handling dynamic of the system
    - Ex: banker's paradox: deadlock, require know availability of chopsticks

➪ Data Flow Diagram VS Sequence Class Diagram

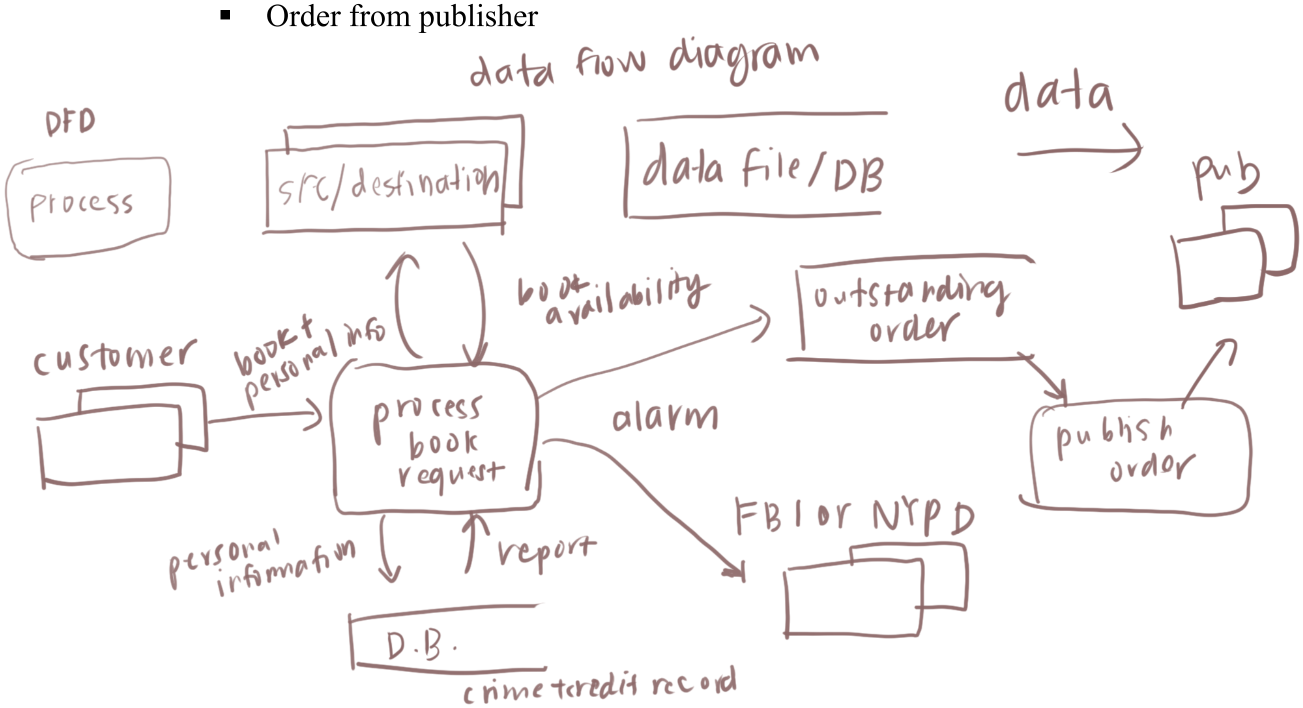
- Similarities
  - Flow of Diagram, Semiformal methods, show data from resource
- Differences
  - Data Flow Diagram is more functional programming
  - Sequence Class Diagram is more object oriented

Software Cost Estimation - Methods of Measurement

- ⇨ **Measure Number of Lines**
  - **Pros:** Easy/No arguments/Disputes
  - **Cons:** Enough Cheating, reward bad codes/punish good codes (DRY)
- ⇨ **Based on Features**
  - **Pros:** Cheating is meaningless, rewarded good coders
  - **Cons:** How fair the $$ to assign (who decide), subjective
- ⇨ **Number of Programming Hours (Popular method)**
  - **Pros:** Easy/No arguments
  - **Cons:** Difficult to record, hard to hold responsible, Easy to cheat
  - Popularly used method: Mutual trust and Reputation
- ⇨ **Poll Many - Several Experts mitigate fairness**
  - **Pros**: Less biased, more professional
  - **Cons:** subjective, trust fake/dishonest expert, can't invite too many experts
  - Number of Expert: **More** and **diversified** the better – Solution Machine Learning
    - Ensemble and Committee Method
      - By vote of weight majority
    - Proven: # expert → ∞, optimal performance
      - Many weak classifiers (better than random guess) 50% 60%
    - Top Three Method for check learning
      - Deep Learning, Boosting, Random Forest
- ⇨ **Self - By analogy and own experience (transfer learning)**
  - Transfer Learning: transfer knowledge from one domain to another
    - Ex: C++ -> Java
  - **Pros:** Total Trust, transfer learning possible
  - **Cons:** Lack of real/transferable experience; over-confidence
- ⇨ **Bidding**
  - Bidding strategy: reward the purchasing to the 2nd highest bidder
    - Prevent from over-bid and under-bid
    - Bidding mostly likely equal to the real utility
    - gaming theory -> Fairness
  - **Pros:** Fair processes, limit room for cheating
  - **Cons:** Less $$ for developer, more effort
- ⇨ **Weighted Sum: Different Statements have different price**
  - Ex: OS for $20, Loop/recursion for $10, Condition for $5, Assignment $ for 1
  - **Pros:** Fairer than # of Lines
  - **Cons:** Cheating inspire

Data Flow Diagram (DFD)

- ➪ Definition
  - Use to descript Logic, Not UML, used to be the king of diagram
  - Not UML, Belong to Semi-Formal
  - Notations
    - Date Source/Destination → 3D square
    - Process/Procedure → Rounded rectangle
    - Data Stored (DB) → Open rectangle
    - Data Flow (must have data) → Arrow with text describe data
  - Hierarchical graph
    - High level is only good for CEO (high management)
    - Further refine DFD with more information to help developers
    - Class diagram derived from DFD
- ➪ Example: Mini-Amazon
  - Customers will put in book order and personal information
  - Checks the criminal record/credit record of the customer
    - Wrong → No service + Call FBI/NYPD
  - Check DB/DW if book is available
    - Yes → Cash return book
    - No → Feedback to order system
  - If book has many outstanding orders
    - Order from publisher

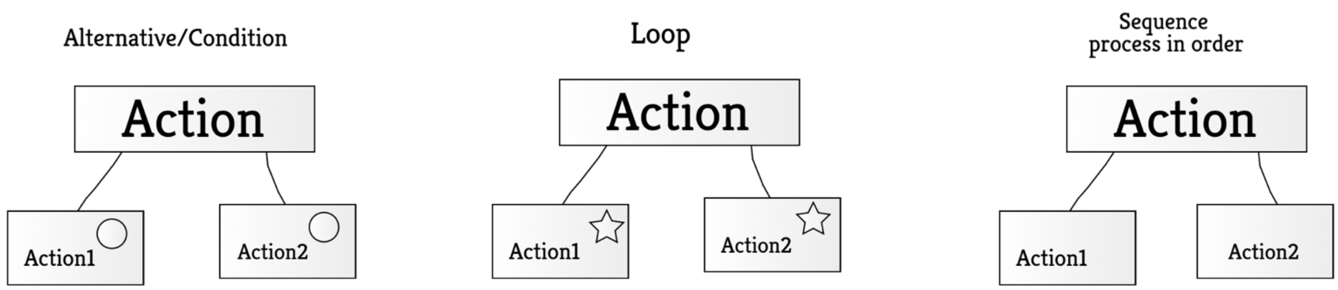


Concepts

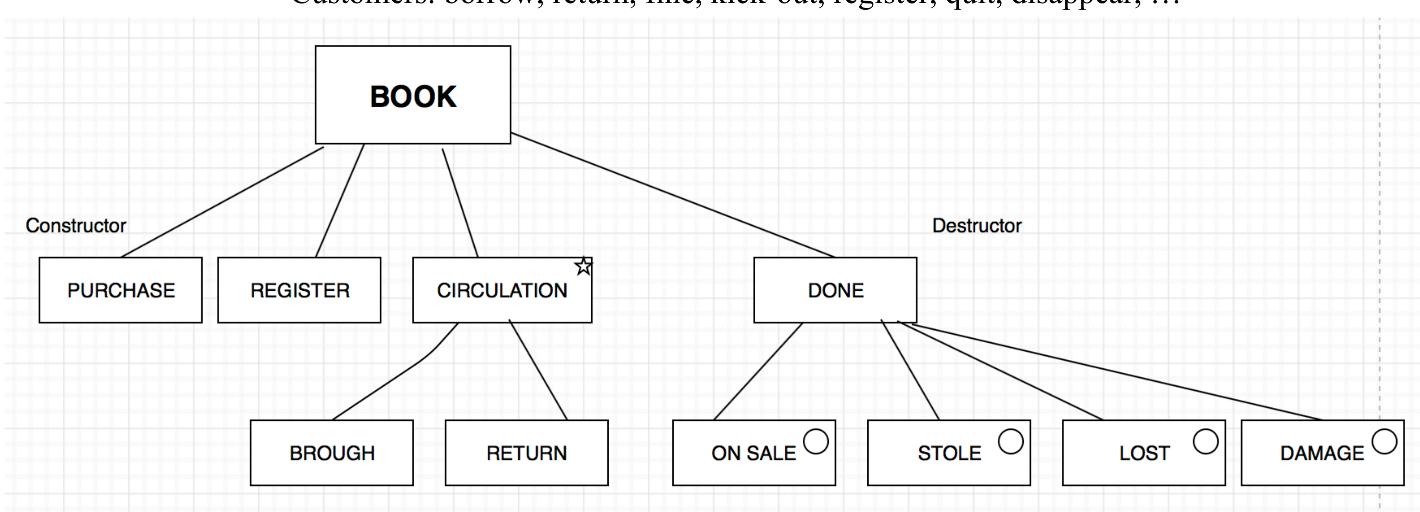
- ➪ Object Oriented Analysis (OOA) analysis/specification system based on objects
- ➪ Object Oriented Design (OOD) design system after analysis, want system based on objects
- ➪ Object Oriented Programming (OOP) coding, example Java
  - Not Determine Each Other, a way to classify the analysis and designing of a system

Jackson System Development (JSD)

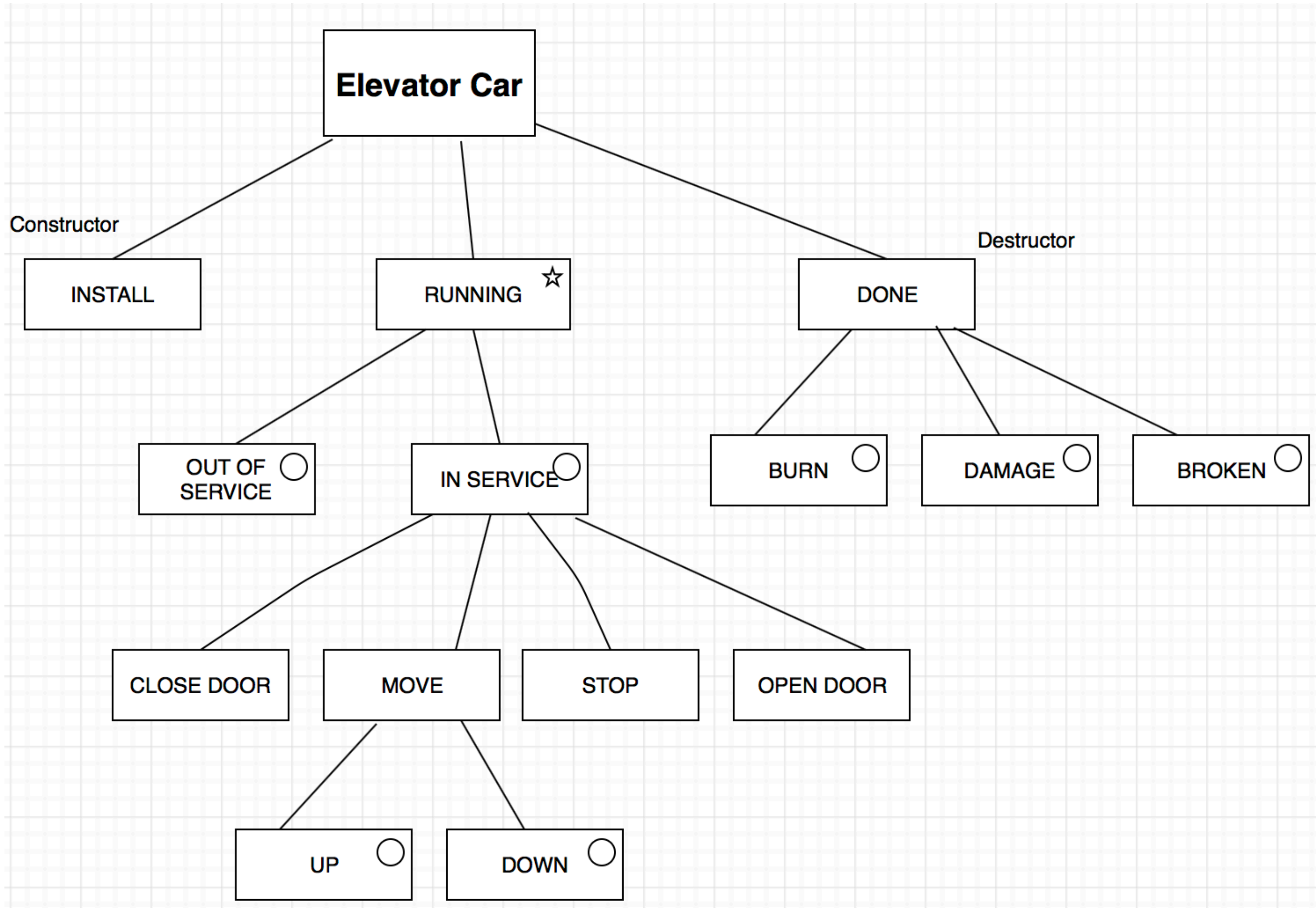
- ⇨ Definition
  - NOT UML, Semi-Formal Diagram, Tree Structure, OO based design (OOA/OOD)
  - Identify entities/objects and functionality of each entity, service should be action
- ⇨ Organize the actions for every entity
  - Always START draw the constructor and destructor
  - At same level, must have same nature(C/L/S)
    - Like tree structure
  - 3 ways to organize actions
    - **Condition**
      - Small **circle** in the upper right corner of rectangle
    - **Loop**
      - Small **star** in the upper right corner of rectangle
    - **Sequence**
      - **Blank** symbol in the rectangle
  - For each rectangle: write one sentence to describe it functionality



- ⇨ Example: CCNY Library
  - Identify most important entities and service
    - Books: borrow, return, purchase, register, on sale, stolen, lost, damage
    - Customers: borrow, return, fine, kick-out, register, quit, disappear, …

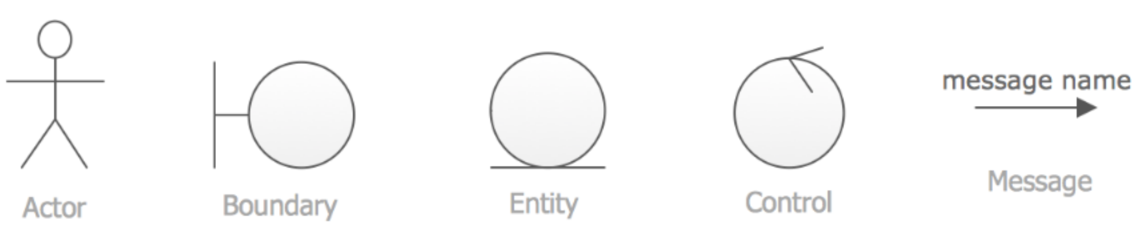


- ⇨ Classic Example in Software Engineer: Elevator Control System
    - o Identify entities:
        - ▪ Elevator car: go up, go down, open door, close door, stop, out of service, install, broken, burned, damage

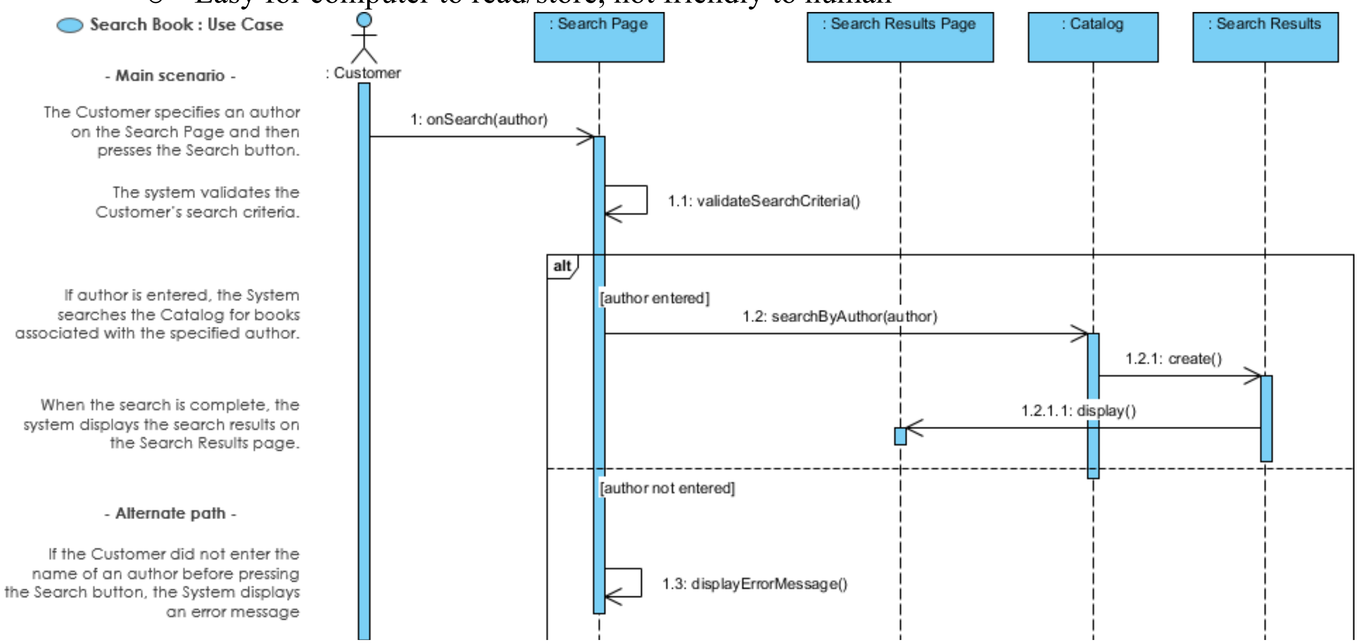


Class Diagram

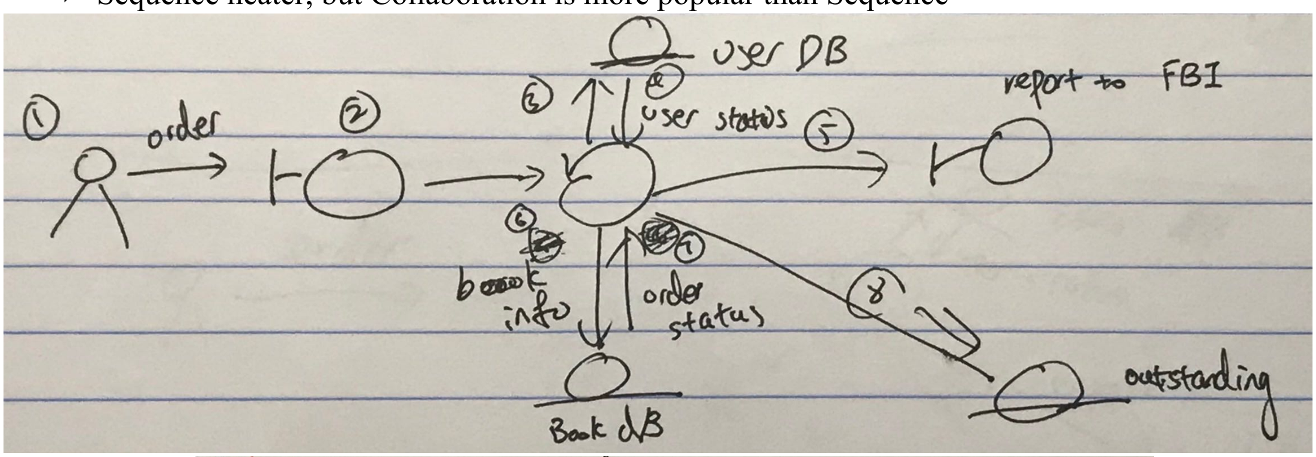
- ⇨ Definition
    - o Based on OO version of DFD
    - o UML
- ⇨ Collaboration Diagram
    - o Need Order (numbered) the arrow (sequence, general path)
    - o Action data in arrow must be numbered, idea of sequence, order or timing

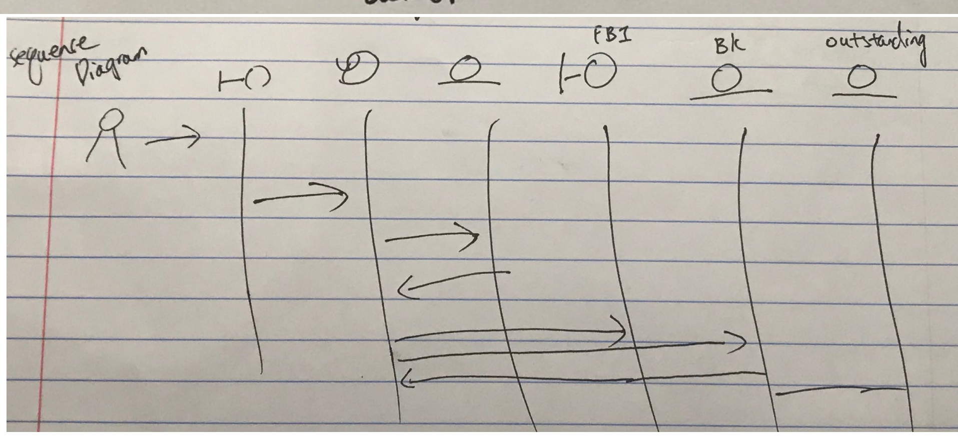


- ⇨ Sequence Diagram
  - Rewrite collaboration diagram in table (sparse matrix)
  - Header (1st row) of table are classes, sequence action/data in the table to tell the connection
  - Easy for computer to read/store, not friendly to human



- ⇨ Sequence neater, but Collaboration is more popular than Sequence





Leadership in software development projects

- ⇨ No Leaders/Democratic → All Members are equal
  - Pros
    - Move diversified opinion, egoless coding
    - No hierarchy; people feel like they're at the same level
    - Willing to help each other
  - Cons
    - Lack of leadership (major/fatal problem)
    - Too many communicate channels
    - Lazy (excuse), minority opinion suppressed
- ⇨ Dictatorship: One team leader dictate everything
  - Pros:
    - Capability VS ethical/fair Leader
    - Easy to set up meetings and get things done, talk/focus
  - Cons:
    - Who should be that leader? No such ideal leader
      - People person vs nerds
    - It's questionable whether the leader is qualified for the job
- ⇨ Two Leaders: one manager + one tech leader
  - Compromise: Golden Rule, Ideal Team Format

Team Cohesiveness

- ⇨ Close teammates
  - Like siblings, best friends
  - Pros
    - Feel Happy, Egoless developing
  - Cons
    - Group thinking → lower quality, no objective, may kill outstanding idea
    - Unwilling to give negative/objective feedback
    - Example in the tech industry (databases): powerful companies (in the past)
      - Ingres
        - Used to be very successful because the team was very close
        - They were too close and ending up leaving Ingres altogether
      - Oracle → The current biggest thing in Databases
- ⇨ Too Loose (Like enemies)
  - Game theoretic style: foes/enemy
  - Example: quicksort: choose 1st element as anchor vs random anchor
    - Don't matter in theory
    - But if enemy read the code and say 1st element is wrong, then better to implement as random anchor
  - Pros
    - Competitive → More careful in your development → Improve quality
  - Cons
    - Stressful when working with enemies who bring you down
    - Sabotaging inside of the team

- ⇨ Optimal format
  - 2 sub-teams
    - Split the close people into separate groups
    - Example: in school, twins are separated into two different classes
  - Professional ← basically what we have today
    - Not too close, not too loose

System Design

- ⇨ **High Level** → Macro (architecture design/ modularization)
  - Hierarchical Modularization
    - Partition System into many modules
    - Number of modules, Communication between the modules
  - Criteria in Functionality → Example CPU System Design
    - Top 3 Modules based on functionality
      - ALU, Registers, Memory (Cache)
    - Critical in moduling, three gates: AND, NOT, OR
  - **Coupling**
    - Relation between **two** modules
    - **Loose the better**
      - Individual development/test/maintain
    - Close the worse
      - Too much communication and limit time to focus
      - May merged into one module, bad design
  - **Cohesion**
    - Relation between code blocks **within a** module
    - **Close cohesion**
      - Smaller module is better (for reuse)
      - Should merge if too close
    - Loose cohesion
      - Need Further partition the module into relatively independent submodules
      - Should be separated into multiple modules
  - **Software Quality Measure**
    - High Level Design System Only
      - **Close Cohesion and Loose Coupling**
- ⇨ **Low Level** → Micro
  - Within each module (Imperative or functional programming)/ class (OO)
  - Define data structure, logic, and algorithm choice for each module
  - Divide and Conquer
    - Greedy Algorithm → NOt globally optimal
  - Dynamic Programming (DP)
    - Save intermediate result for checking, top choice in optimally allocate resource

- **Fibonacci Number**
  - `F(n) = F(n-1) + F(n-2), F(0) = 0; F(1) = 1;`
  - Recursion

```python
### Recursion Way
def fib(n):
    if n==0:
        return 0;
    elif n ==1:
        return 1;
    return fib(n-1) + fib(n-2)
```

    - Problem: fib (-2), need assert(n>=0), too slow
    - Time complexity: T(n) = T(n-2) + T(n-1) = 1.618**n (1.618 golden ratio)
    - Exponential algorithm, infeasible; repeat computing 2 times
    - Computer only can hander polynomial algorithm
  - Loop

```python
def fib(n):
    prev1 = 1
    prev2 = 2
    for i in range(2, n+1):
        val = prev1 + prev2
        prev1, prev2= val, prev1
    return value
```

    - Recursion is bad, take space in system stack
    - Top down separation: use lots of system stacks/spaces to keep evidence
      - bad for computer, good for human
  - Dynamic Programming
    - Save result in dictionary,
    - O (1) for look up in dictionary
    - Treat space with speed, Fib(n) need n entries in dictionary
  - Math
    - $fib(n) = c_1 * a^n + c^2 * b^n$, a, b, c_1, c_2 are constant
    - Repeat Squaring, O (log n)
  - All Time
    - $O(1.618^n)\ (recursion) \rightarrow O(n) \left(\frac{loop}{DP}\right) \rightarrow O(\log n) optimal\ math$
- Hashing
  - Time: $O(\log\log n)$ due to collision
  - Space: $need\ more\ O(1/e)$
- Sorting/Binary
  - Int Array A of size A, find $A[i] + A[j] = 0$
    - Need two hashing
      - hashing all A[i], O(n)
      - Search/hashing $A[i] = constant - \ A[j]$
- 2D Search n*n matrix
  - Each row and column are sorted in ascend order, time for find v
  - **Binary: Grand Median**
    - Compare center, if less than throw out a quarter
    - Each comparison only limit ¼
    - $T(n^2) = 1 + 3T\left(\frac{n^2}{4}\right) = O\left(n^{\log_3 4}\right)$
  - **O(n) solution**
    - Compare lower left or upper right, each time remove a row or column

7 Level of Cohesion (Worst to Best)

- **Level 1: Coincidental/Trash bin/Garbage bin**
  - Module → actions/features totally irrelevant, not reusable, **unacceptable design**
  - Example: Take exam, eat an apple
- **Level 2: Logical**
  - Share some action in common, multiple separated trash bins for each case
  - Example 1: Take CSC322, Take CSC447, Take ENG101
    - Not totally trash bin, be in general is not reusable, **unacceptable design**
- **Level 3: Temporal**
  - Actions normally happened in time sequence; $p > 0.8$ correlation high
- **Level 4: Procedural**
  - Previous actions are necessary need for later actions, cause & effect
  - Need to identify the cause and effect to prove this through scientific experiment
  - Example 1: Medicine is effective
    - Aspirin is used to cure headache
    - Identify the cause and effect
  - Example 2: Statistical correlation
    - Fully understand linear regression, calls self-expert in machine learning and AI
- **Level 5: Communication**
  - One file/class module that contain all action about the data structure
  - Functions are still public, rarely used due to the existance of the class level
- **Level 6a: Functional**
  - One action/feature/function
  - Non-OOPL Functional programming, Procedural Programming
  - Refinement: don't put too many actions within one module
    - Should be reusable
    - Module/Function should be reasonably small
      - 1000 lines of code → Bad, trash bin is high
      - 10, 20 lines of code → Bad, overhead, reduce efficiency of overall system
    - Note: for databases, classes, and modules, we always want to reduce the size
- **Level 6b: Informational**
  - Well defined class in OOP (3NF)
  - Inheritance (killer function of OOP)
    - Localize features, better develop, test and maintain features
  - Data Hiding → Hiding details (data and logic) of how class implemented
    - Private, Protected, Public
      - Private: unique to object
      - Protected: within the family
      - Public: visible to everyone
    - Auto/Local, Static, Global

                - Auto/local: local variable which can only be used in one class
                - Static: global variable inside a class
                - Global: a variable that can be used throughout a program
- ⇨ **Level 5 VS Level 6b**
    - 5. One data structure, many functional about DS
    - 6b. Member variable, many members function
    - Well-Developed Procedural PL VS OOP (most important crucial/attractive)
- ⇨ Example of determine cohesion level
    - Module A
        - Take 221, Take 322, Take capstone I, Take capstone II, Graduation
        - **Level: Procedural,** b/c there are classes that are prerequisites for other classes
    - Module B
        - Cheat in final exam, Get F in the class
        - **Level**: **Temporal**, b/c the actions coincide, highly related to each other.
            - Not procedural b/c not need to cheat in a final exam in order to get F in the class. You can get F through other means.
    - Module C
        - Quarterback throw football, Receiver runs back and catches ball, Touchdown
        - **Level**: Temporal b/c these actions are typical and can happen often.
            - Not procedural b/c there are different ways to make a touchdown
    - Module D
        - Tall students (> 6'1"), Got A in Prof. Jie's class
        - **Level**: **coincidental** (trash bin)
            - Not temporal because short students can get A too.
    - Module E
        - People with high IQ, Earn lots of $$$
        - **Level**: **Temporal** b/c some correlation, but high IQ not only way to become rich.
    - Module K
        - Type in a username, put in password, Access your account
        - **Level**: **Procedure**
            - Temporal possibly, but need to justify the reasoning
    - Module M
        - In the subway station, slide your card, get in the platform, Ride the train
        - **Level: Procedure** (0.00001% temporal)
    - Module L:
        - Type in a resume to your folder, watch a digital video, conduct a skype meeting
        - **Level: Logical**, computer I/O
            - It's still a trash bin, but it's a classified trash bin

Coupling – worst to better

- **Content Coupling** (worst)
    - Two modules/classes share the same state/private variables
        - one module can access/change the private/state variable of another module
    - Keyword(s) allow private variable sh:
        1. friend (big brother): << >>
- **Common**
    - Two modules/classes share global variable
    - Very close, try to avoid it if possible, try not to use global whenever possible
    - C++: Easy/hard to declare global variable
    - Java: Hard to declare global variables
- **Control**
    - Returned Value of one module decide the logic of the other
    - One module A call another module B, returned value determines different actions of A.
    - Example:
        - Module 1: … return m_val_1
        - Module 2: … value = m1()
            - If value > 1: print("fine")
            - else: print("bad")
    - Not good habit: justify yourself
    - Acceptable coupling, from A, B is already a black box
- **Stamp**
    - Two module/classes: pass parameters but not all of them are useful
    - Example:
        - M1: m2(std1)
        - M2: age=std1.getAge() / gpa=std1.getGPA()
    - Much better, waste system space/bandwidth
- **Data**
    - all parameters are useful, no waste
- **No direct relation**: ideal/optimal
    - Loosest, Remotest
- **Example**:
    - **Goto Statement**
        - Mod A: …. Goto label1 ….
        - Mod B: …. label1….
        - A and B belong to
            - Common → NO global variable share, so not
            - Content → Yes, after label 1, can't distinguish mod A or mod B
        - Dijkstra → goto is back
        - Knuth → goto should be minimized, not too much if hurts understandability
        - destroy the readability/understandability of your code/messy logic

Problem Solving – Algorithm (lower level design)

- Find Min/max: array A of size N
    - Minimal/maximal of efficient: time complexity
    - Guess/Prove check every element at least once n-1
        - O(N), K*N: minimize k, k>= 1
        - N-1: wrong
    - Solution
        - Time complexity: O(2*N)
        - Don't assign min/max $i, j,\ a = i + 2j$
        - Want k = 1.5, O (1.5 *N)
            - Check 2 element 3 times
                - Compare two adjacent elements (get one smaller, one larger)
                - Compare smaller with min, larger with max
    - Fast Fourier Transform (FFT)
        - Fourier Transform: $O(n^2)$
        - 1965: Tukey, Develop algorithm like merge sort: butterfly O(n log n)

```python
Min = A[0]
Max = A[0]
for i in range(1,N):
    if min > A[i]:
        Min = A[i]
    if max < A[i]:
        Max = A[i]
```

- For O(N) time complexity, K*N: K also means a lot
    - Parallel Search:
        - Example: 8 Cups with water in 7, poison in 1
    - Sequential Search (MS interview question):
        - 9-bead problem (3 measurement, not know lighter/heavier of the abnormal)
            - 8 normal, 1 abnormal, different in weight
            - Balance to measure
            - Smart Grouping: 3-3-3, 1-1
    - Example: Find top 3 out of 25 horse, with only 5-lane tracks
        - Local server to do services, combine to form global ranking
        - Strategy
            - 5 groups, lets label R1, R2, R3, R4, R5
                - 5 heat champions for the 5 races
            - No.6 race is heat-champion race
                - Top 1 get the gold medal, but not decide for silver or bronze
            - No.7 race for silver or bronze
                - Let gold horse belong to R1, and No2 at No.6 race in R2
                - Horse in No.7 race
                    - No2, No3 in No.6 race; No2, No3 in R1; No2 in R2
    - Rewrite Merge Sort in Loop
        - Begin: 1-element array is sorted
        - Loop
            - Enlarge N to 2^m, 2^m-1 < L <= 2^n
            - First merge 1 element "arrays"
            - Next: merge 2 element "arrays"
            - Next: merge 4 element "arrays"
            - ….
            - Next: merge 2^n element "arrays"
        - Top Loop: control size of each subarray (O (log n))
        - Lower Loop: merge every neighbor array O(n)

- Edit Distance (Dynamic Programming)
  - Editing Action: delete, insert, replace
  - Example: ‘hte’
    - the (one reversal)
    - hate (one insertion)
    - he (one deletion)
    - hoe (one replaces)
  - Example 2: ‘aet’ –> ‘tea’ too long, need 4
- Greedy Algorithm
  - Best choice at the current step (locally best)
  - Gradient Descent Algorithm
    - Any NN/Deep learning update the gradient by local info
    - Pros: Simple and Fast
    - Cons: Globally Worse
    - Stochastic: simple random selection
- K - Mean

Legal and Security Responsibility

- Privacy problem of programmer
- **ABET**: computational professionals are increasingly **powerful** due to the current pan-computing, universal digitization
- **Universal Computing ERA**
  - Careful with handling sensitive personal data
  - Military → Data must be kept private, tons of security measures
  - Medical → Sign waiver to keep patient data private, else sue (anonymize data)
    - Non-invasive treatment
    - Invasive treatment: insert medical device in the body
    - Deep learning can beat human medical expert
      - Stanford team by Andrew Ng beat medical disease with 10 algorithms
  - Legal/wall street profession: Stocks
  - Math/Stat Model
    - Bayesian reasoning: Foundation of Machine learning/Deep learning
    - model + algorithm + code (must know calculus + statistics + probability)
  - Never Cheating
    - GITHUB, acknowledge the author, don’t claim if not yours
  - As Developer
    - Be careful, serious, competent, and responsible,
    - violations could cause overwhelming results

Compression

- ➪ Shannon's Entropy Axiom
  - ○ Lower bound is entropy
- ➪ Huffman entropy encoding
  - ○ based on Shannon's great work on entropy limit
- ➪ Look UP Table (LUT)
  - ○ If short thing to compress, then after compression will take large space
- ➪ Dictionary-Based Compression
  - ○ Gzip, zip, winzip
  - ○ Zeevi: 5 lines
    - ■ Assign short code/index to popular term, to reduce size
    - ■ Example: keep write 'cuny'. 4 ascii – 4bytes
      - ● Give index to cuny, ex 257
      - ● Index for more appearance word
  - ○ Example in Detect Cheating
    - ■ Creator (Jane): asmt1.py
    - ■ Cheater (Jed) : asmtl1.py
    - ■ Gzip:
      - ● Zip creator file → result 1.gz (let it with size 100k)
      - ● Zip (creator file + cheater file) → result 1.gz (let it with size 105k)
      - ● Small difference, so cheating found
- ➪ **Music Compression**
  - ○ Wav (avi) to mp3
  - ○ Mp3: statistical redundancy, exploit human ear's weakness
    - ■ Listen loud sound, eardrum will down
      - ● Drop small volume of sound
- ➪ **Color Compression**
  - ○ Suppose a picture is 256 * 256 in gray scale
    - ■ In gray scale: each pixel is one byte (8-bit: 0 ~ 255)
    - ■ $256^{256*256}$ possible image
    - ■ Natural Image is small subset of all images
    - ■ Human face small subset of Natural Image
      - ● A lot of structures (spatial redundancy)
    - ■ Human skin color: small subset of digital colors
- ➪ **Image Compression**
  - ○ **Lossless Compression**
    - ■ Usually 70% size of original, used in medical/legal
    - ■ Example: gzip/winzip, jpeg, png/tiff, medical image
  - ○ **Lossy Compression**
    - ■ Usually 25% size of original, used in CS/EE/physics/math
    - ■ Good Cheater: success cheating human eyes (visual system)
    - ■ Example: jpg, mpeg, jpg2000
      - ● jpg (based FFT) reality better
      - ● jpg2000 (based Wavelets) physic better

- **Lossy Compression Quantization**
  - Human eyes → nonlinear recognition
    - Need 24 bits for each color image pixel, 256^3, 16.7 million colors
    - Image Lossy
      - Compress to less bits to cheat 99.9% human, ex:16 bits
  - **Scalar quantization**: 1 number -> another number
  - **Vector quantization**: (r, g, b) -> (r1, g1, b1)
    - Bad quantization example
      - Too different: $(10, 0, 0) \rightarrow (3,3,4)$
      - Too Similar: $(200,100, 100) \rightarrow (195,105,90)$
    - Transmittal Page: with ink, without ink
    - Binarize the page: 0 → background; 1 → foreground
      - 3D → 1D
        - Gray Scale: gray = (.33*red+.33*green+.33blue)
      - Low-Pass Filtering
        - Human eye → color importance
          - green > red >> blue
          - Different weight for each color
            - Example: .35, .45, .2
          - Traffic light (b/c background conflict, tree)
            - red > green, yellow

⇨ Compression Examples

- Lossless compression Example: 1110011111000000011110000011101 1
  - Fax 1000*800: 800k → 1MB per page
  - Telephone: 56k → 5k per sec
    - 10 min to send one fax
  - Simple Algorithm: totally change the fax industry
    - Impossible to transmit to original binary stream
    - Data redundancies: many ones/zeros are adjacent each other
    - Exploit the stat feature: reduce drastically
    - Run-length encoding **(RLE)**
      - ‘1’, 20, 45, 98, 21, 100
      - ‘1’ in front means start number
        - Then follow count of 0’s and 1’s
  - **RLE:** enable algorithm to render **fax machine** possible (useable) in practice
- Lossless compression: Natural image/human face: lots of spatial redundancy
  - Example color: 200, 203, 199, 198, 201, 20, 21, 25
  - **Difference Pulse Coding Modulation** (**DPCM**) – Simple and universal
    - Only encode different, Lossless, used in jpg/mp4/mp3/gif
    - 200, 3, -4, -1, 3, 20,1 5
      - If difference to large, use new beginning
      - Difference ∈ [-16, 16], 5 bits store for difference
      - 1 pixel → 8-bits
- **Newton Idea in Calculus**
  - Use linear function to represent general curve

- **Fourier Transform: (ODE/PDE)**
  - Fast: O (n log n) 1965
  - Key idea in FT power
    - Compress Data
    - Use sum of (small # of) sine/cosine curves to represent original function/curve
    - Orthogonal function generalizes FT (GFT)
- **Jpeg**
  - Joint picture expert group
  - DPCM + FT + Quantization + Huffman/Entropy
  - 20% of original image
- **Mpeg**
  - Motion picture expert group
  - For video: temporal red
  - Use difference between two frames
  - 3% of original video
- Color → binary image (math morphological operator): object of interest

Clock Problem → Angle between hour hand and minute hand, Divide and Conquer

- ⇨ Need a based line, Example 0^o compute the angles of min & hour against this
  - Fix a foundation: theta = theta_m - theta_h
- ⇨ Divide (both hands are moving, so divide)
  - hour contribution HB: hour hand with B (baseline)
  - minute contribution MB: minute hand with B (baseline)

- ⇨ Conquer
  - Solve HourMin first (theta_m)
    - theta_m: range is 0 to 59
    - theta_m = M / 60 * 360
  - Solve dh
    - theta_h = contrib(h) + contrib(m)
    - f(h) only
      - 12 hours, (h/12) * 360
    - f(m) contribute of m in hour
      - Every 60mins, hour hand will move 30 degree
      - m/60*30 = 0.5m
    - theta_h = 30h+0.5m

System Phase

- **Testing**:
  - Passive Testing: By observations / code reading
  - Active Testing: By experimentation, Design Test Cases/Expected Results
  - Top – Down Testing
    - Top level features till finally all basic/leaf
    - More prefer, used >90%
      - Overall feature/functionality is much more important than local
      - Limit time to test all of low-level features
    - Higher Level: drivers
    - Lower Level: stub
  - Bottom – Up Testing
    - Solid footing
  - Lower-Level Testing
    - Black-box: Partition cases according to natures
      - Typical cases (using random number)
        - **Randomized** testing is crucial, more test the better
      - Boundary cases
      - Know nothing inside the system, just input and output
    - White-box: (structural testing) (high level???)
      - Test **all** branches/scenario (flowchart, petri, class, FSM)
        - Know which case test which component
      - See through system (inside)
    - More popular: Black-box: case partitioning/observe
      - White-box: rigorous/pain
      - Black-box: not too much pains
  - Alpha/Beta Testing: Version of different testing phases
    - Alpha → internal testing group
    - Beta → after alpha testing, test by experts, friend, public for free
  - **Debugging VS Testing**
    - Debug by self: belong to implementation phase
      - More subjective and generous
    - Test by other: testing phase
      - More objective and destructive (want to find more bugs)
    - Developer vs tester
  -
- **Delivered** Product
- **Maintain**: customer service
  - Corrective
  - Perfective → Not required/ideal
  - Adaptive → Fit for new environment
  - Legacy system: develop by old DS or logic/algorithm/PL/OS
- Software Engineering: Forward SE
  - Start from specification → Planning → Design → … → System
  - Difficult for visualization, mutual misunderstanding

- ⇨ **Software re-engineering**:
  - rewrite legacy using new data structure/ logic/algorithm/programming lanuage/OS
  - Examples:
    - Cobol -> C++
    - C++ -> Java
  - Easier than Software Engineering
    - Software re-engineering is easier: running well system
- ⇨ **Reverse (Software) Engineering**: more difficult/AI/Classified
  - Reverse Forward SE
    - Specification ⟵Planning ⟵ Design ⟵ … ⟵ System
  - Steps
    - Start from working product (binary/encrypted code): debugger
    - Find source
    - Find design
    - Find specification
    - Find theory/equation/math/logic
  - Get design and specification from binary code (e.g. .exe, .o)
    - Goal: recover algorithm/math/logic using binary code
    - Need: program understanding
  - Companies use this to keep up to date about new feature
    - Copy each other by doing reverse engineering

Final Note

- ⇨ Write programs in final exam
  - Understandable: comment, meaningful variable/function name
- ⇨ Python
  - Vectorized coding, more efficient code
  - NumPy: vectorized computing