- Part 1
  - Chapter 1: Intro, 1.1
  - Chapter 2: Intro, 2.1, 2.2, 2.3
  - Chapter 3: Intro, 3.1, 3.2, 3.3
  - Chapter 4: Intro, 4.1
  - Chapter 8: General
  - Chapter 9: Intro, 9.1
- Part 2
  - Chapter 11: 11.1, 11.3, 11.4
  - Chapter 12: 12.1, 1.2, 12.4
- Part 3
  - Chapter 16: Intro, 16.2, 16.3, 16.4
  - Chapter 20: Intro, 20.1, 20.2, 20.3, 20.4
- Part 4
  - Chapter 23: Intro, 23.2, 23.3, 23.4
  - Chapter 25: Intro, 25.2, 25.3, 23.4

Logistics

Theoretical course (3 credits)
Mandatory attendance (grading: 10%, min 67)
13 exercises - **mandatory** (grading: 20%, min 55)
N.B. justified missing one or two exercises won't affect the grading, there won't be late submissions except for special cases like duty service or giving birth
Final test (70%, min 55)

Mini project course (1.5 credits)
1st assignment – mandatory, w/o grade (pass / no pass)
7 assignments (consequent stages) – mandatory, grading: 10 pt each assignment
2 mini-projects (on top of the previous stages), grading: 1st mini-project – 10pt, 2nd mini-project – 20 pt
Total grade - min 60

Pass both to get full credits

Attendance = 10% of grade

13 quizzes – try to do all of them – magen for the test

- If we take all of them
- On moodle, every week – quiz but not the first one
- 1 hour for each quiz, can use material
  - Try not to use

Final test is closed test

Q1: What is the software?
Q2: What are the attributes of good software?
Q3: What is the S/W engineering?
Q4: What are the fundamentals of S/W Eng. Activities?
Q5: What is the difference between Comp.Sc. And S/W Eng.?
Q6: What is the difference between S/W Eng. and System Eng.?
Q7: What are the key challenges of S/W Eng.?
Q8: What are the costs of S/W Eng.?
Q9: What are the best S/W Eng. Techniques and methods?

1 cod with documentation

Customer product – product that customer comes to company and wants them to develop software that does 1,2,3

- Requirements are set by customer

Generic product

- Startup
- Some idea without customer
  - Bring requirements and develop product according to plan

Good software

- Secure
- Fast
- No bugs
- User friendly
- No errors
- Low maintenance
- Easily developed – expanded

Customers opinion

s/w engineering

- Creating / coding software from scratch
    - Design , architecture
    - What systems to use?
        - Linux, windows
        - This cpu
        - Gpu
        - Programming language
    - From start to end

Coding = CS

- Developing software after design and thinking and constraints

S/w engineering = from scratch

Fundamentals of s/w activities

- Start from requirements
- Testing
- Maintaining software
- And cost of software

Cost of s/w engineering

- Salaries of workers = the major thing – related to development time
    - Major challenge is the time
- Buying hardware/software
- Courses for workers

Time to market

- Time is money
- Have aproduct to develop
    - Bring to market asap b/c competitors exist

s/w vs system

code (s/w) – vs whole system

systems

- Electronics
- Develop a car

Challenge of s/w = time and money

Best techniques and methods

- Different techniques suitable for diff types of systems
    - Airspace vs game engineering

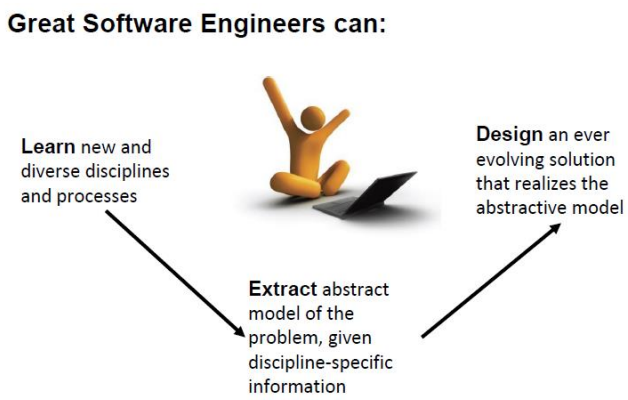
Taking idea -> develop to working system

Learn how to learn new things

From model to working product

- Learn and use knowledge

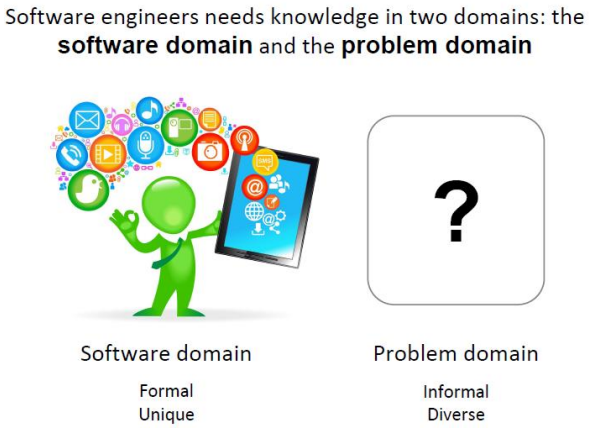
Example:



- Iron dome
    - Physics , chemistry, math, algorithms
    - Gravity
    - Engine
    - Wind
    - Volume
    - Route
        - calc
    - Track
        - missile
    - Comms etc

  Identify rocket in air, alert and calculate trajectory
- Alert systems
- Software engineering is about modeling the physical world and finding abstractions

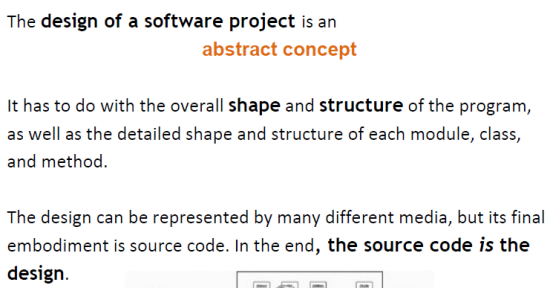
  Problem domain
- What we need to learn
    - Fly an airship to space



Bring solution towards the software domain

Design

- Looks
- Connections
- Classes
- Interfaces
- Functions
- Structure



Structure -> source code (final product design)

Order product from web – example

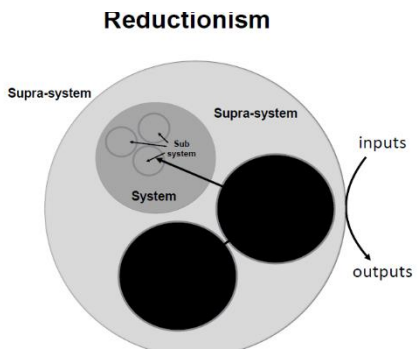
- Know the order fow, how processes work
- Order -> customer DB -> verify CC -> ship to customer

Data flow diagram

- Entity = db

Go from DB to code

- Order fn
- Verify fn

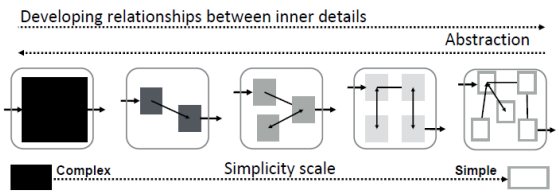


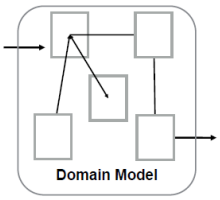
UML is another output

- Class diagram
- Unified modeling language
- Book class , ISBN , title, author class – uses the book
- Bookitem … etc.

Software engineering requires thinking at multiple levels of abstraction

- How it looks
- System components
- Processes in system





Static structure of a system shows internal components and their interactions

    - Program
- Make code/design
-
- Black box with inputs and outputs

Iron dome

Input is enemy rocket

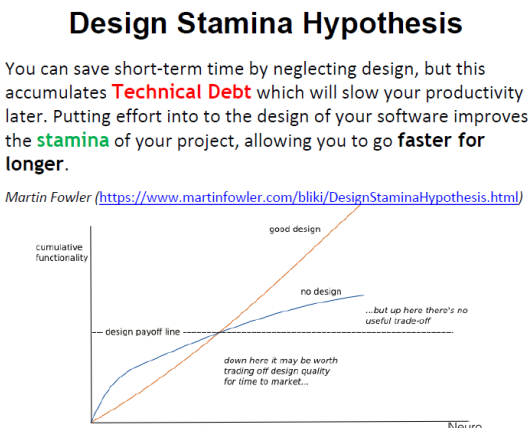
Output is our rocket

Inside we need radar, rocket launcher, how to identify rocket, cacl route etc

Black box -> components -> design -> code

Why not start writing code before designing and thinking ?

- Screw up
- Break down to small problems
- blue is code w/o design
- Orange = code with good design
- Faster results at start
    - But slows down
        - Add something, then need to fix more issues
- https://www.martinfowler.com/bliki/DesignStaminaHypothesis.html



Good design = slower at start

- But after some time, we want to add more code or features… its easier

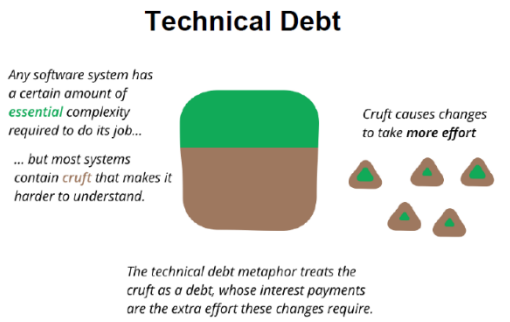
No design = mess = hard to design more

design smells:

First, the system design is a **vital image** in your mind. Then, the clarity of it make it to the first release.

But then something goes wrong –the software starts to **rot**.

Festering **sores** and **boils** accumulate in the code, making it more and more difficult to maintain.



Eventually, the sheer effort required to make even the **simplest of changes** becomes **onerous.**

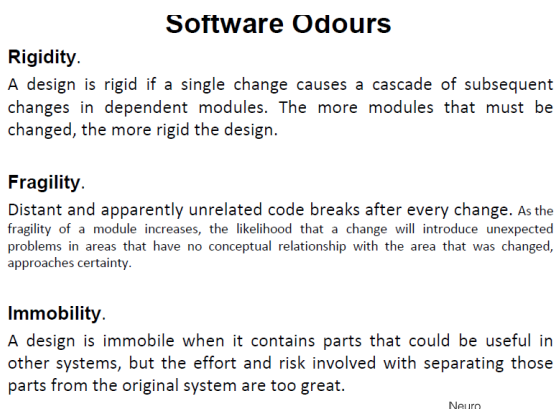
Hard to maintain – becomes a pain – even for simple changes

1 change = many code changes to accommodate

Add a small function which causes code to break in unrelated place

s/w development best thing is mobility – re-use

- Take code, class, system
    - THEN USE THEM in our product
    - Use iron dome radar for planes or arrow missiles



If we want to take a system and use it elsewhere but we need to take unrelated code immobility = bad code

Design is wrong, or many dependencies

Many classes not few

Math class with calcs

- But we made 2 o3r more classes

Complexity = **design**

- Not using classes or functions or constructors
  - Heavy design

Needless repetition = **code** = copy and paste not making functions

- No copy paste
- Params, args, return values

Dry – don't repeat yourself

Opacity

- Missing variable names
- Code not clear
- Unrelated names
- Fn not known what it is doing
- If it works, don't touch it

system dependent = machine

assembly language = mips

- Several types
- \

Intepreded before high

4gl

- Wix
- Shopify
- Wixywig

Object oriented design

Encap – hide functionality- black box

Inherit – code re-use

- Below

Viscosity.
Difficulty at which a developer can add design-preserving code to a system. Changes can be design-preserving or not (hacks). When the design is viscous it is easier to do the "wrong" then the "right" thing. "Behaving badly is the most attractive alternative"

Needless complexity.
By preparing for many contingencies, design can become littered with constructs that are never used. The design then carries the weight of these unused design elements, making it complex and difficult to understand.

Needless repetition.
When the same code appears over and over again, in slightly different forms, the developers are missing an abstraction.

Opacity.
In a fresh module, the code may seem clear to the developers. After all, they have immersed themselves in it and understand it at an intimate level. Later, after the intimacy has worn off, they may return to that module and wonder how they could have written anything so awful. Code should be written in a clear and expressive manner.

Few words on the software domain

'rogramming languages can be categorized into four groups:

Machine language. CPU dependent. Expressed as a series of binary digits.

Assembly language. Replaced binary digits with readable English-like words.

High-level languages Expressed in English-like phrases. Readable and intuitive. Java, C++.

Fourth-generation languages . 4GL followed the 3GL in an upward trend toward higher abstraction and statement power. Might include automatic code generation or interaction with a data base.

Encapsulation
The creation of black boxes where the internal mechanism is hidden. Promotes modular, readable and debugging oriented program.

Inheritance
Hierarchical design. Promoted Reusable, more efficient program.
Really, more efficient?

Polymorphism
Allows values of different data types to be handled using a uniform interface.

Polymorph

- Making 2 or 3 objects behave like 1 via interfaces with different functionality
- Vector, list
- Student, major, professor
    - Calculate salary
        - Interface fo student
        - Virtual functions

Fns and know how to identify problem

iterator – with c++

Solution to problem or need

Iterate over collection but we don't want the client (user of the collection or class)

- Clients needs to iterate over collection
- Client = class or fn
- Client shouldn't know inner structure of collection
    - Don't care if uses array, list, stack, queue
    - And maybe replace collection
        - Uses heap or trie

getNext

hasMore bool

functions that our iterator has

most of the time it's a UML diagram but can be a code or anything else

algoetc

influential

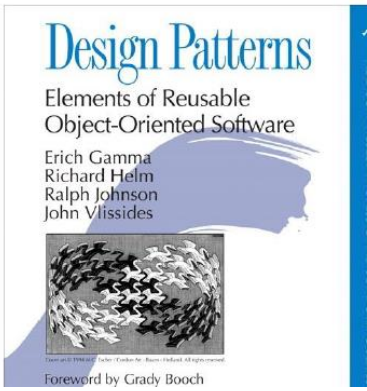
Key features of OOD include:

Design Patterns

- A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in **software design**.
- It is a description or template for how to solve a problem that can be used in many different situations.
- Patterns are formalized **best practices** that the programmer can use to solve common problems in software development.
- **Object-oriented** design patterns typically show relationships and **interactions** between classes or objects, without specifying the final applications
- Design patterns may be viewed as a **structured approach** to computer programming **intermediate** between the levels of a **programming paradigm** and a concrete **algorithm**.

Design Patterns

Design patterns gained popularity in computer science after the book Design *Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 by the so-called "Gang of Four" (**GoF**)



Types of Design Patterns

- **Creational** – Class / Object creation
- **Structural** – Class / Object composition
- **Behavioral** – Class / Object communication

Scope \ Purpose	Creational	Structural	Behavioral
Class	Factory Method	Adapter/Wrapper, Marker Interface	Template Me Interprete
Object	Singleton, Builder, Prototype, Abstract Factory Method, Dependency Injection, Resource Acquisition is Initialization (RAII)	Adapter/Wrapper, Composite, Delegation, Decorator, Façade, Bridge, Flyweight, Proxy	Observe Iterator, Mediator Commanc State, Mementc Strategy, Visitor, Chain of Respor

Singleton

- One instance only of a class

Builder

- Construction is in a certain order
- All params given

Structural

- Connections between classes
- Compose
- Negation
- Class in other class

Behavioral

- How classes behave like iterators

Functional

- Pointer to function
- Event handler
    - C sharp

Concurrency

- Threads
- OS stuff
- Critical sections
- File is shared
    - 3 or 4 programs write to program
        - Lock file while writing to it

Architect

- C sharp
- Interface, logic, db level

- **Functional** – Functional Programming (Closure, Functor, Generator, ...)
- **Concurrency** – Concurrent execution (Active Object, Lock, Monitor, Reactor, Thread Pool, Thread-Specific Storage, ...)
- **Architectural** – Subsystems (Client-Server, MVA/MVC/MVP/MVVM, Multi-Tier, P2P, REST, SOA, Microservices...)

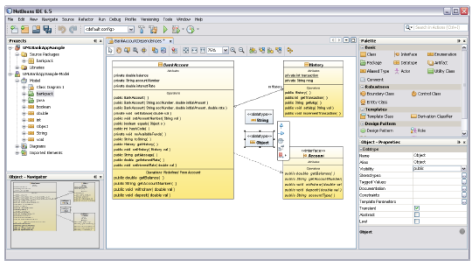
Design is code

Diagrams and planning comes to code

What is Software design?, by Jack Reeves, 1992

The **design** of a software system is documented primarily by its **source code**, that diagrams representing the source code are **ancillary** to the design and **are not** the design itself.

Jack's article was a harbinger of **agile development**.



What the Design Patterns consist of?

Example: Iterator

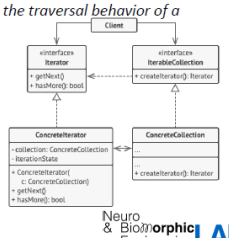
- **Intent briefly** describes both the problem and the solution

  Iterator *is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).*

- **Motivation further** explains the problem and the solution the pattern makes possible

  Problem: *Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects. Most collections store their elements...*
  Solution: *The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator. In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, ...*

- **Structure** of classes shows each part of the pattern and how they are related, usually by UML



- **Code Example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern

Dr. Elishai Ezra Tsur & Dan Zilberstein

Agile

Software engineering intro 10/27/2022

Waterfall

- How to design software
- First take and listen, requirements
- Font, colors, etc
- Takes time
- Design after all requirements
- From top to bottom
- 6 months
    - Then just implement
        - No test, run, etc
- Then test it
    - Write unit tests
    - See inputs, outputs , scenarios
- Then deploy program
- Then maintain – fix problems, add features

bad

- Takes a long time
- Didn't test anything
- Redo structure = start over
- Changes after
    - Bad requirements
        - Bad ideas
-

The Agile Alliance

On February 11-13, 2001, at The Lodge at Snowbird ski resort in the Wasatch mountains of Utah, seventeen people met to talk, ski, relax, and try to find common ground and of course, to eat.
What emerged was the **Agile Software Development Manifesto**. Representatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others sympathetic to the need for an **alternative to documentation driven, heavyweight software development** processes convened.

Good

- Send airshuttle to space = waterfall
    - Cant fix a crash

Agile

-
- Ken beck
    -
    - Tesr driven development
    - J unit dev
    - 4 principles
        -
        - Teamwork
            - People over tools
            - Comments and opinions
            - Interact and communicate with each other
        - Working software trumps documentation
            - Instead of a whole book on design and looks
                - Start programming , small design , document, don't think about all requirements and processes from beginning
- Design testing deploy planning defining
    - Then circle

Kent Beck

- One of the 17 original signatories of the Agile Manifesto and the creator of Extreme Programming, a software development methodology which **eschews rigid formal specification for a collaborative and iterative design process**
- Beck popularized CRC cards with Ward Cunningham, the inventor of the wiki [brainstorming tool used in the design of object-oriented software originally proposed as a teaching tool]
- Beck and Cunningham pioneered software design patterns (1987)
- Beck pioneered the commercial application of Smalltalk. He wrote the SUnit unit testing framework for Smalltalk, which spawned the xUnit series of frameworks, notably JUnit for Java, which Beck wrote with Erich Gamma
- Extreme and Agile methods are closely associated with Test Driven Development, of which Beck is perhaps the leading proponent

https://agilemanifesto.org/principles.html
https://agilemanifesto.org/history.html

Vs waterfall

Reqs , design, implement, test, maintain

Software engineering intro 10/27/2022

Work with customer to get better product

talk with customer, work with them

don't plan it all from beginning

see market, customers, reqs

Team work

share knowledge, communicate,

people over process

- What employees want to develop and no structures
    - More commitment
        - Worker wants to do A rather than telling them do A,b,c

Do document, but concise

- What we need now = design
- Code itself shows the design and structure

Don't mess with tiny things

Allow customer change, don't tell him to renegotiate

React, don't monitor

Flexible, modifiable software

Plan for the near future. Don't spend time planning

the final product

- Zoom
    - Lucky as they existed already
    - Then competitors copied them
    - No one cares as we all used zoom
- **Plan for near future**
    - 1 month to a year
        - Ideas wont last longer than that so much

Set of decisions

- Radar , rocket launcher,alarm communicate w/ each other
- Subsystems
    - Classes, interfaces, fns
- Realtime system – perf matters

Autonomous car – idea

Mini project = makes graphics in CGI

You can create a

world

without ever leaving your

Working software over Documentation

- Concise documentation
- System rational and structure should be reflected from the code itself (improve synchronization, eliminate huge documentation files)
- Training by working
- Progress is measured by working software

Collaboration over negotiation

- Software is prone to changes (schedule, cost) - contracts are problematic
- Constant interaction with the customer

Operating Points

Software Architecture

Design Fundamentals

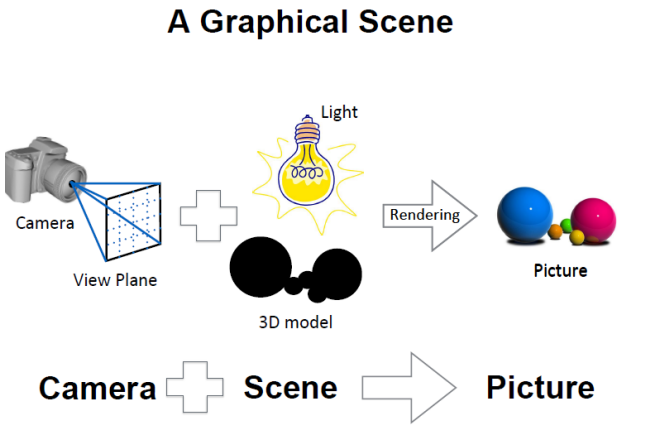
Software Architecture

A set of **significant decisions** about the **organization** of a **software system** including:

- **Structural elements** and their **interfaces** by which the system is composed;
- **Behavior** as specified in the **collaboration** among structural elements;
- **Composition** of structural and behavioral elements into larger **subsystems**;
- **Architectural style** that guides this organization.

Software architecture also involves **functionality**, **usability**, **resilience**, **performance**, **reuse**, **comprehensibility**, **economic** and **technology constraints**, **tradeoffs** and **aesthetic concerns**

home



Movie = frames = fps

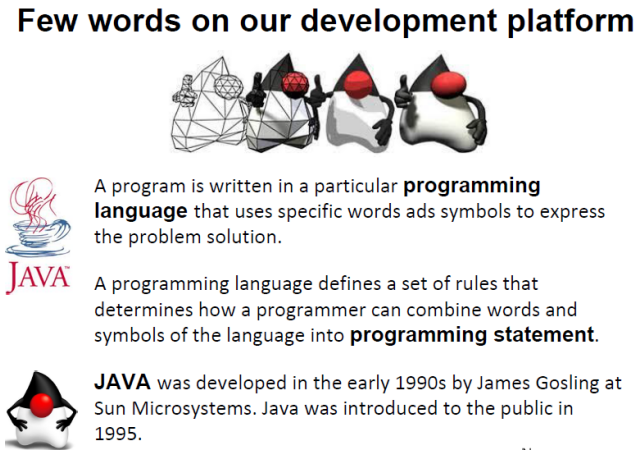
Light = colors of objects

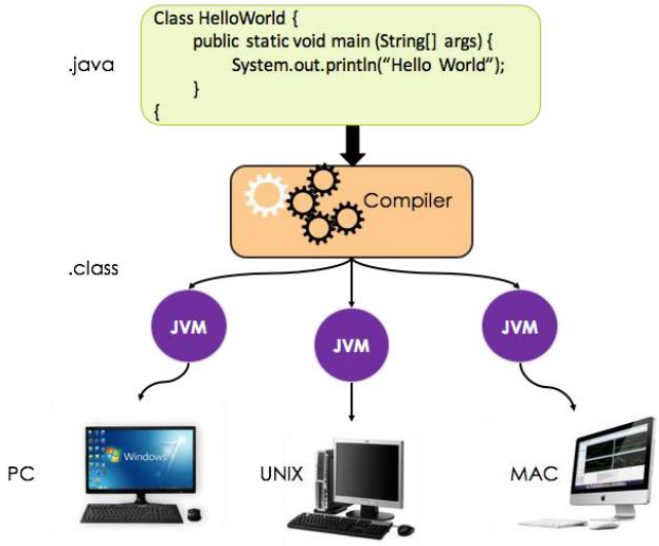
And add camera

- So same picture from different angles
- Scene and light
    - Then render the entire camera

java is OOP

From c and c++



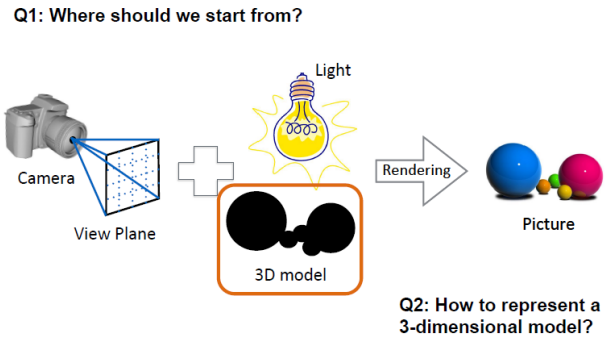


C needs to compile again for all systems (win,apple, linux)

Not Java

.class file run on VM

- Same vm on every computer
- Then runs the file on machine

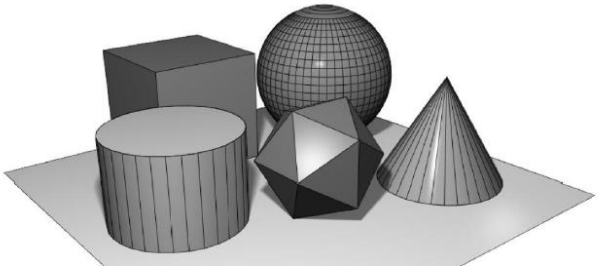


Build a scene from objecf

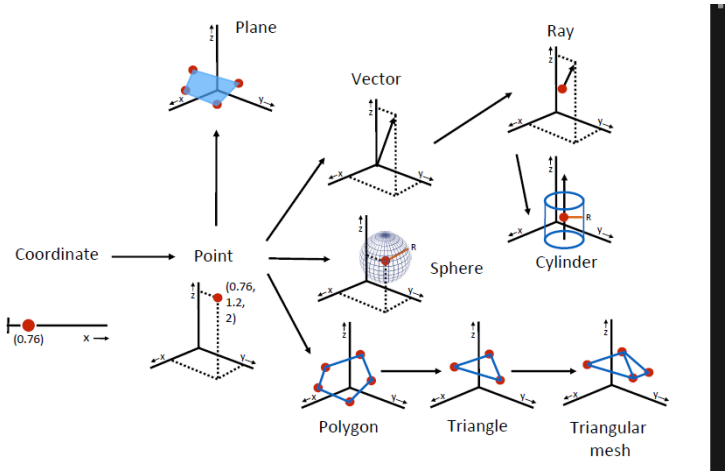
Represent a 3d model in 2d

Start from space

Representing geometrical primitives



Create shapes we want with triangles/rectangles



X,y,z basic 3d coordinates

Make a point

- Then make a vector
  - 3 points
- Rays
  - Starts from anywhere in space

Points -> polygons -> triangles -> mesh

Sphere

- Points with same radius

Planes

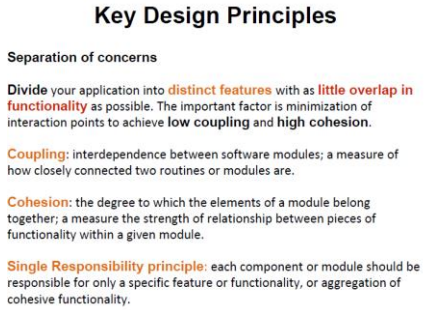
- Infinite plane

Cyllinders have rays

- Triangles doesn't (3 points)

Planes have vectors

- 3 points
- Or a point and vector



Coupling

- Connect 2 parts, objects
- How connected?
- Minimum level of coupling
- So 2 objects aren't co-dependent

Math module for sqrt , etc

And object that uses the math module, don't want circular dependencies

- Know as little about the other object as possible
  - Black box

High cohesion

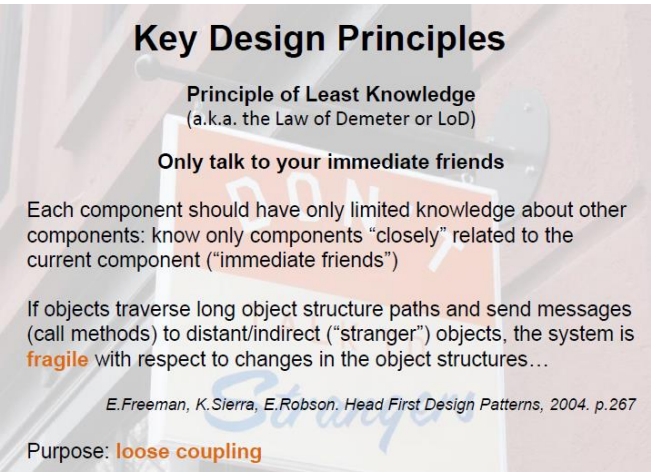
- Inside object, do as much as possible things related to each other
  - Math module has math stuff
    - 1 class not 2 r 3

Single responsibility

- Each component only responsible for specific functionality

Does a lot of things but only need 1 knife at a time

Law of demeter

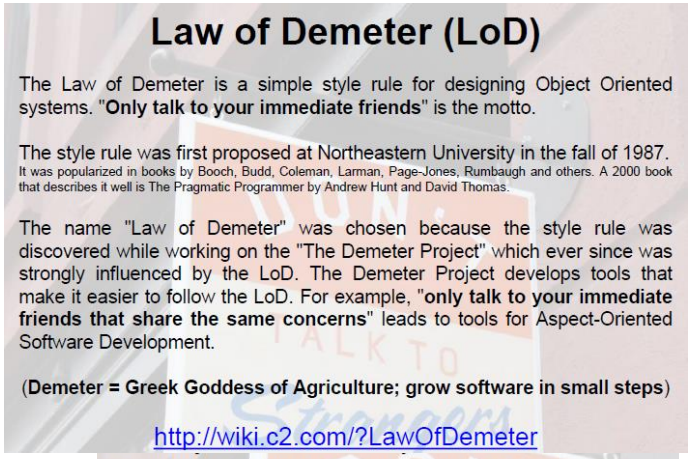


- Project names of Greek gods

Louse coupling

- Class should only talk with another class
  - Not a 3rd class … or talk directly and not through 2nd class

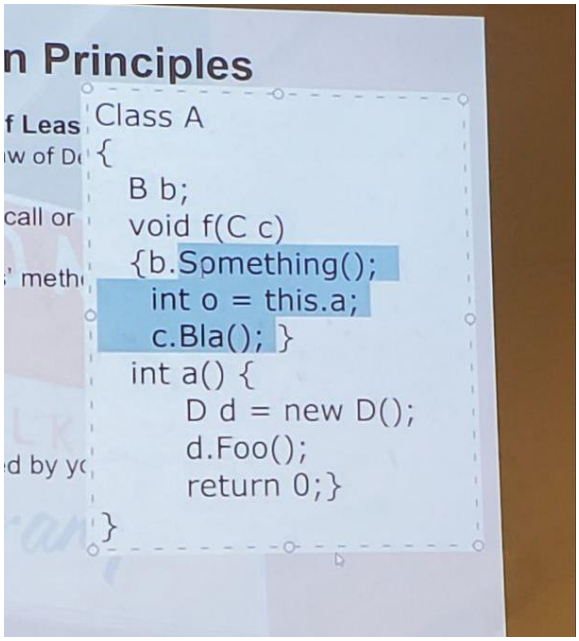
Class a has object of b and another function void f which gets some object c



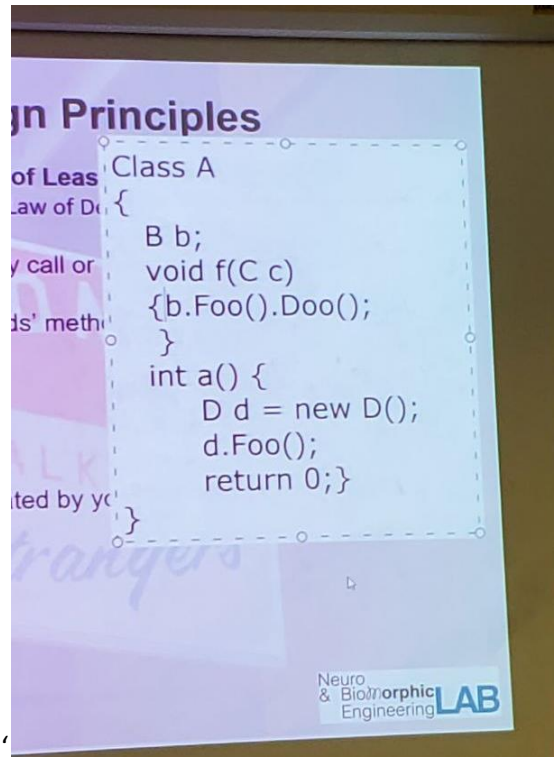
- Your fields and your fields' methods
- Other your methods
- Methods of parameters
- Methods of objects created by you

Cannot access methods of objects created by other objects

Can do:



Cant do:



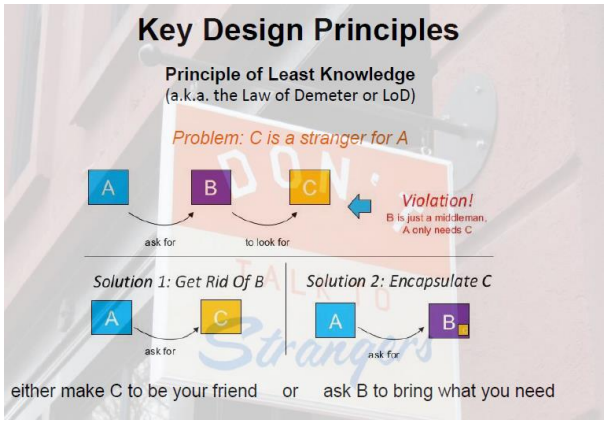
Or F f = b.Foo();

f.something

we can do this

F f1 = new();

F1.something();



B's responsibility to perform that fn

C is a field of a or passed as a parameter

Or created object of C in function of A

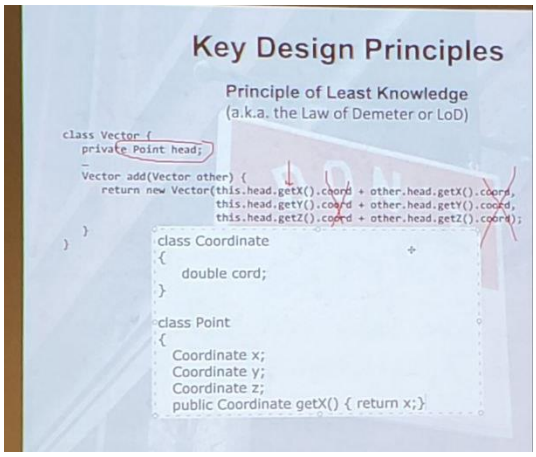
Causes fragility

Change something in C which causes error

Separation of concerns

Point is class

Vector class using point

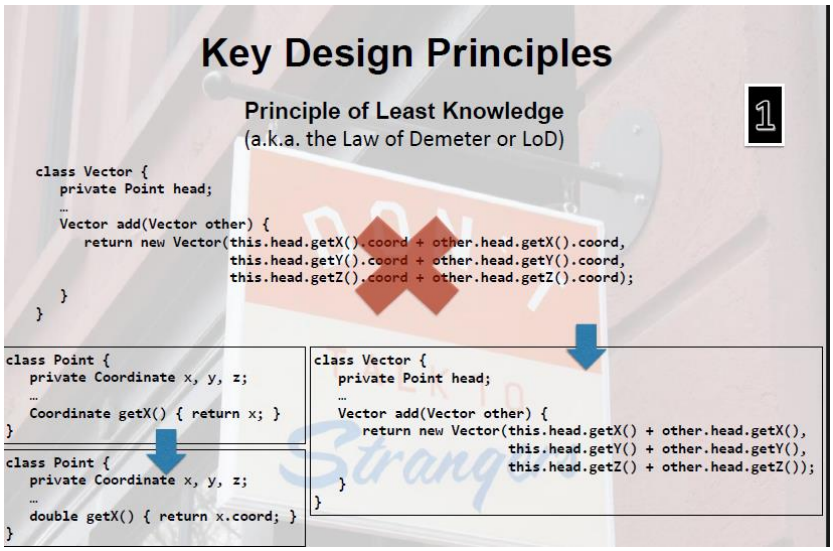


Sumcoords and other

- Getting cords from other objects and getting fields
- How to fix?
  - Or make vector class – use coordinate class and get rid of Point
    - 3 coords not point
  - Or take double field – function and put in point
    - Return double not coordinate
      - Add double to double

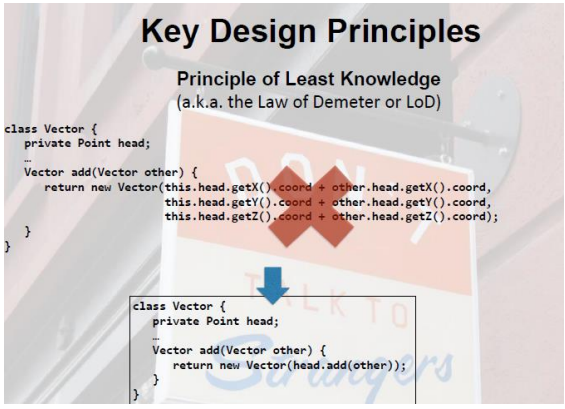
Public double getX() {return this.x.coord;}

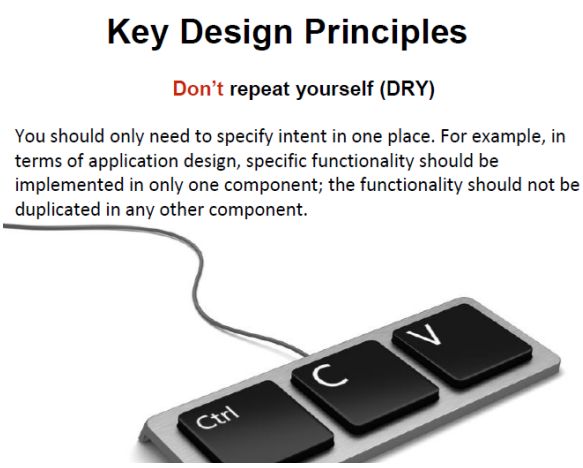
- 2nd solution
- Take double from coord class and put in the point class
-



Or put add fn in point

- Moving point to another location
- Same functionality
- Use add fn of point to return point – add type point
- Get vector and move point in vector direction and length
- Add 1 vect to another = same calc as add vector to point
    - Don't repeat yourself





Class Point{

Public Point add(Vector v){

Return new Point(this.x+ v.getx()

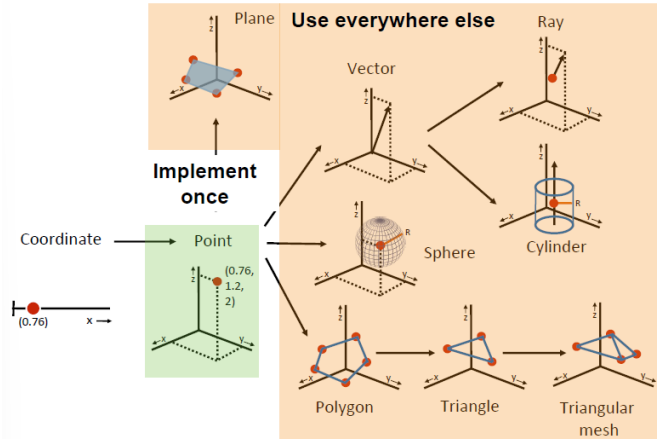
Head returns a point, access field of my class

Change one things, turns to hundreds

Hardcoding

- Int [] a = new int[50];
- For(int i=0; I <100; i++)

- For (int i=0; I < 100; i++)
- Save it in variable
- Code is fragile with hardcoding



Using functions instead of copying

Calc distance between 2 points

Area of triang;e

- Distance fn
  - In point class not triangle class

Responsibility

- Point calculate distance between 2 points

Public double distance(Point p){

Distance between this and point

Who knows the future, if we need distance, then implement it.. wait till we need it to implement it

Helps avoid software hardness

- Wrote something, code changes, then need to write it elsewhere

Kiss

- Avoid 1 long line
- Do it in 3 lines
- Simple

Primitive geometry

Point, coord, vector, arraythen go to bigger geometry

start at coordinate -> point (uses coord) -> triangle, plane, ray, etc which use point

Key Design Principles

Design what's necessary

When the cost of development or a failure in the design is expensive, you may require upfront comprehensive design and testing. In other cases, especially for **agile development**, you can avoid big design upfront (BDUF). If there is a possibility of the design evolving over time, avoid making a large design effort prematurely. This principle is sometimes known as YAGNI ("You ain't gonna need it").

Key Design Principles

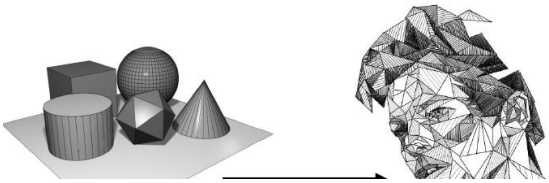
Keep It Simple, Stupid (KISS)

First noted by the US Navy in 1960, the KISS principle states that most systems work best if they are kept simple rather than made complicated. Therefore, **simplicity** should be a key goal in design, and **unnecessary complexity** should be avoided.

Two alternative forms for the abbreviation (for more delicate people):

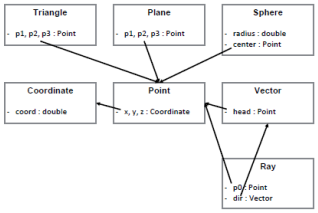
Keep It Short and Simple
Keep It Simple and Straightforward

We are starting with **primitives**, slowly advancing into more complicated models



Initial Design

Build a class diagram for a sphere, plane, triangle, cylinder, coordinate, Point, vector and a ray



Triangle 3 pts – bounded by those

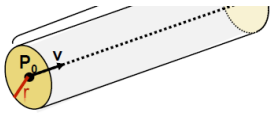
Plane 3 pts – just to represent plane

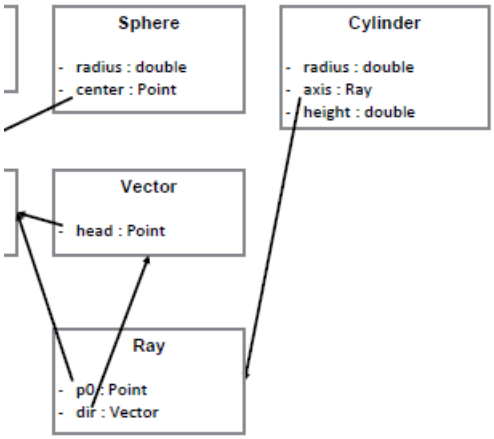
Vector – end of point – head

- Starts from beginning of axis

Ray = vector and point = start of ray

Add a cylinder



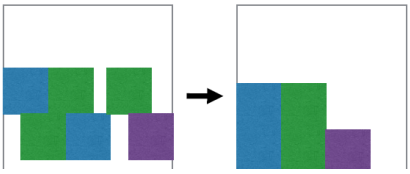


collection finite of 2 points from radius along ray

Code refactoring

The process of restructuring existing code – changing the factoring – without changing its external behavior. Improves software's nonfunctional attributes.

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. *(Martin Fowler, https://refactoring.com/)*



Without changing external behavior

Sort fn implemented by merge sort

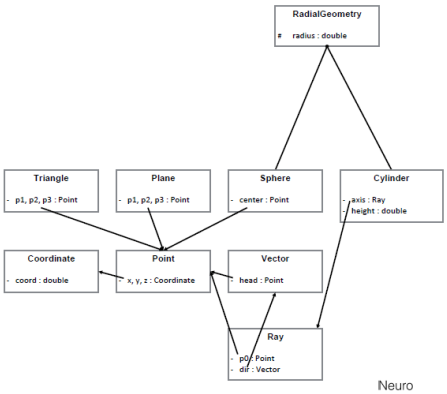
- Now refactor to use bubble sort
  - Merge is n log n
    - Needs another array – recursion
      - Less effective for collections < 30,000
  - Bubble n^2
    - Takes longer

Still have sort fn , get collection and sort it

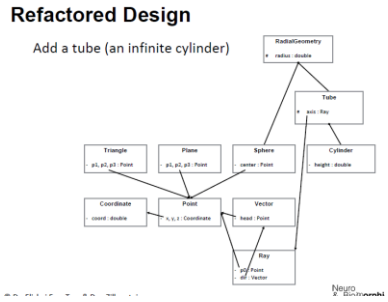
Refactor = reorganize code

- Change var names etc

Refactored Design



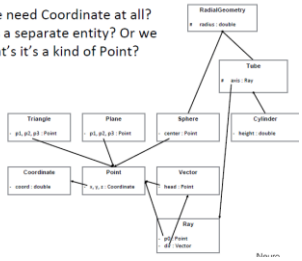
- Neuro
- Remove radius from classes and move it to its own thing
  - Maybe radius calcs
    - 1 central place vs 2 classes



    - Neuro
    - Inherit from tube and add height

Refactored Design

Why do we need Coordinate at all? Is Vector is a separate entity? Or we can tell that's it's a kind of Point?



-
- Coord is a double
  - Not in double class?
  - Add, sum, subtract etc..
  - Vector is a point, why does vector need a point
  - Double can be infinite
    - Computer cant store that
    - 1/3 = 0.33333334 (from mem constraints
    - Double jave 15 decimal digits
      - Our project needs more
        - Calc cords, move them
          - See if we can see object etc…

Taking Constraints in Account

Floating point operations are not accurate: try in an old calculator: input 1, divide by 3, multiply by 3 => 0.999999

In coordinate operations we can be caught in situations when a point we find on a surface is actually calculated to be near the surface – which can cause us some problems

It can also cause some problems in tests when the result is correct, but it is not precisely the same.

In Java, double has 15 decimal digits of accuracy. We can also look at Java math framework function Math.ulp(double).

java.lang.Double		
Modifier and Type	Constant Field	Value
public static final double	MAX_VALUE	1.7976931348623157E308
public static final double	MIN_NORMAL	2.2250738585072014E-308
public static final double	MIN_VALUE	4.9E-324
public static final double	NEGATIVE_INFINITY	-1d/0d
public static final double	POSITIVE_INFINITY	1d/0d
public static final int	SIZE	64

Taking Constraints in Account

There are also positive and negative zeroes (+0.0/-0.0).

Please see how the function equals for wrapper type Double works.

Pay attention that (+0.0 == -0.0) has the value **true**

equals

```
public boolean equals(Object obj)
```

Compares this object against the specified object. The result is `true` if and only if the argument is not `null` and is a `Double` object that represents a `double` that has the same value as the `double` represented by this object. For this purpose, two `double` values are considered to be the same if and only if the method `doubleToLongBits(double)` returns the identical `long` value when applied to each.

Note that in most cases, for two instances of class `Double`, `d1` and `d2`, the value of `d1.equals(d2)` is `true` if and only if

```
d1.doubleValue() == d2.doubleValue()
```

also has the value `true`. However, there are two exceptions:

- If `d1` and `d2` both represent `Double.NaN`, then the `equals` method returns `true`, even though `Double.NaN==Double.NaN` has the value `false`.
- If `d1` represents `+0.0` while `d2` represents `-0.0`, or vice versa, the `equal` test has the value `false`, even though `+0.0==-0.0` has the value `true`.

- +0 or -0 for derivative

Taking Constraints in Account

Utility Class

```
public class Util {
// It is binary, equivalent to ~1/1,000,000 in decimal (6 digits)
  private static final int ACCURACY = -40;

  // double store format: seee eeee eeee (1.)mmmm … mmmm
  // 1 bit sign, 11 bits exponent, 53 bits (52 stored) normalized mantissa
  private static int getExp(double num) {
    return (int)((Double.doubleToRawLongBits(num) >> 52) & 0x7FFL) - 1023;
  }

  public static boolean isZero(double number) {
    return getExp(number) < ACCURACY;
  }
  public static double alignZero(double number) {
    return getExp(number) < ACCURACY ? 0.0 : number;
  }
}
```

- Make a utility class