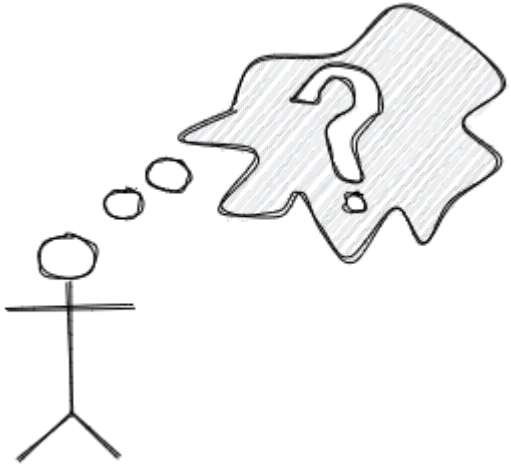
Object Oriented Programming - Day 1

Looking Back



- Who remembers …
    - Structural vs Behavioral Modelling?
    - State Machines?

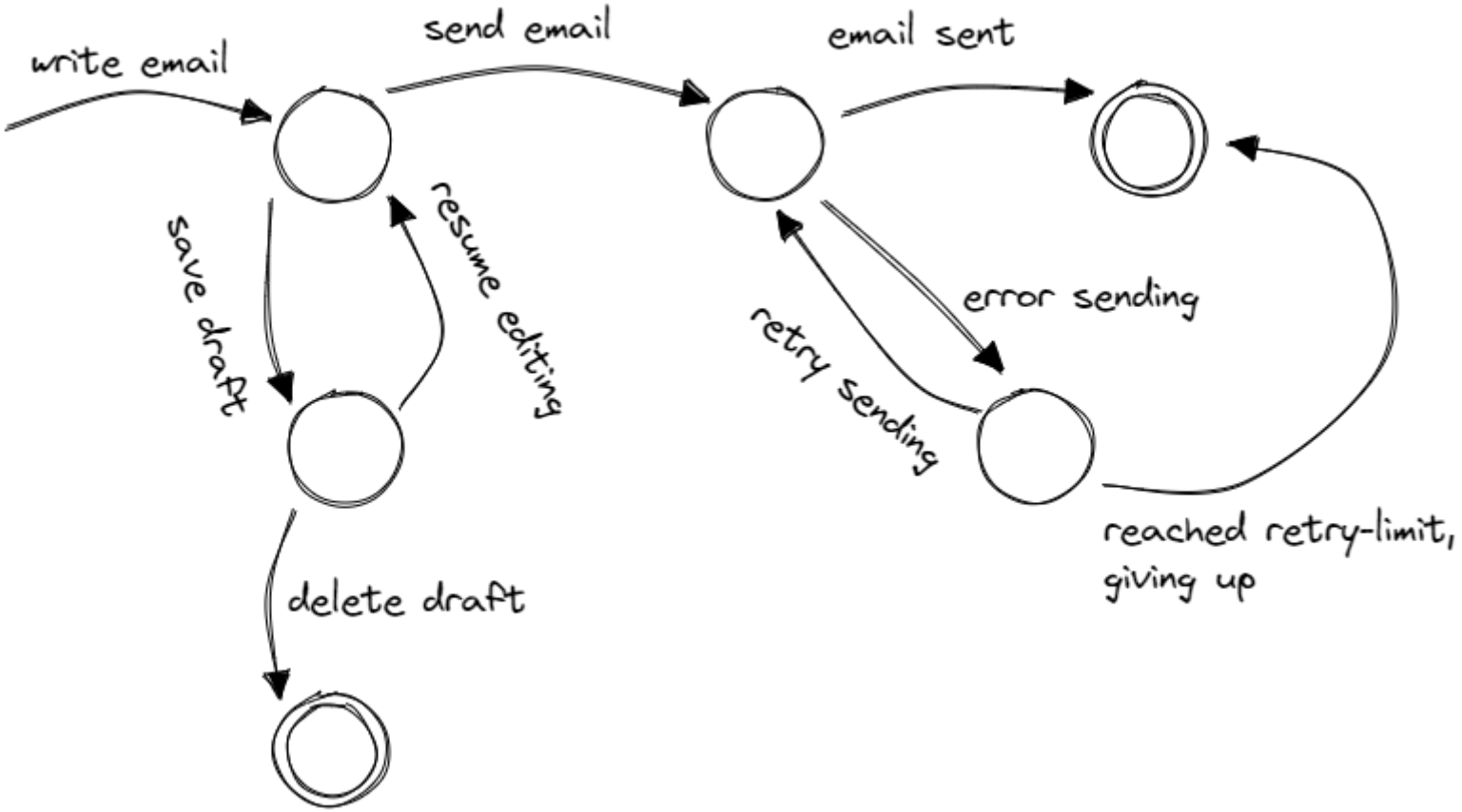
Structural vs Behavioral Modelling

- Structural
    - Class Diagrams
    - What talks to what?

    - Easier to get started on
- Behavioral
    - State Machines / Flow Charts
    - Actual decisions to make to get to output
    - Harder

State Machines

- Two basic elements
    - States and Transitions
    - Labels on transition denote *available* inputs
    - End states have double circle



Movng State Machines to Code

- Label the states with their properties
- Check inputs for valid state transitions

Module: Best practices

Software Engineering

Engineering:

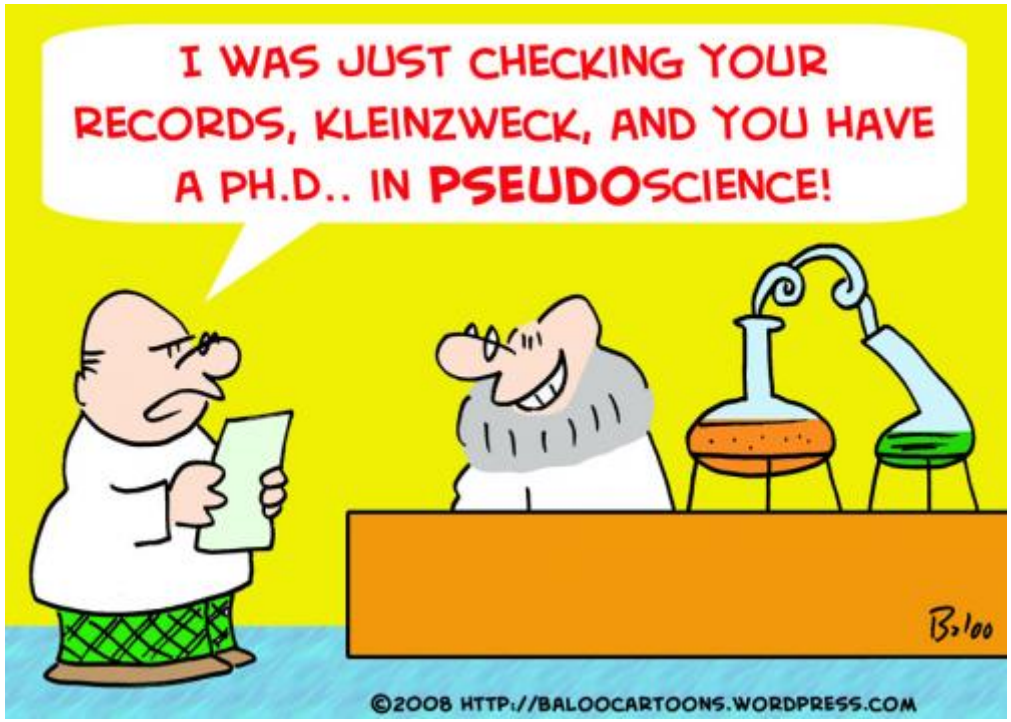
> Engineering is the use of **scientific principles** to design and build machines, structures, and other items, including bridges, tunnels, roads, vehicles, and buildings.

Engineer:

> professionals who invent, design, analyze, build and test machines, complex systems, structures, gadgets and materials to fulfill functional objectives and requirements while considering the limitations imposed by practicality, regulation, safety and cost.

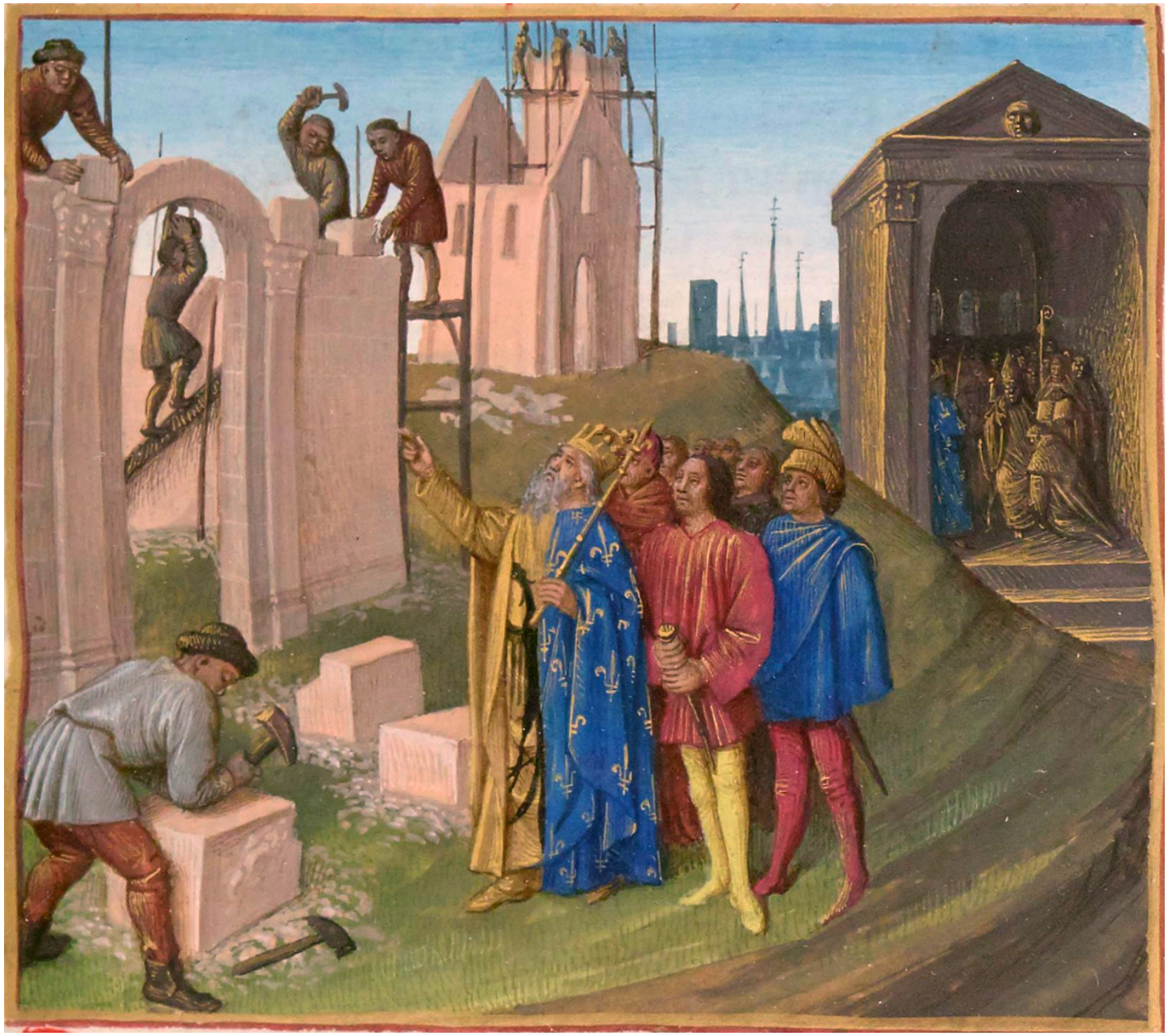
Software Engineering

Is it really "engineering"?



Software Engineering

Like building cathedrals in the 16th century



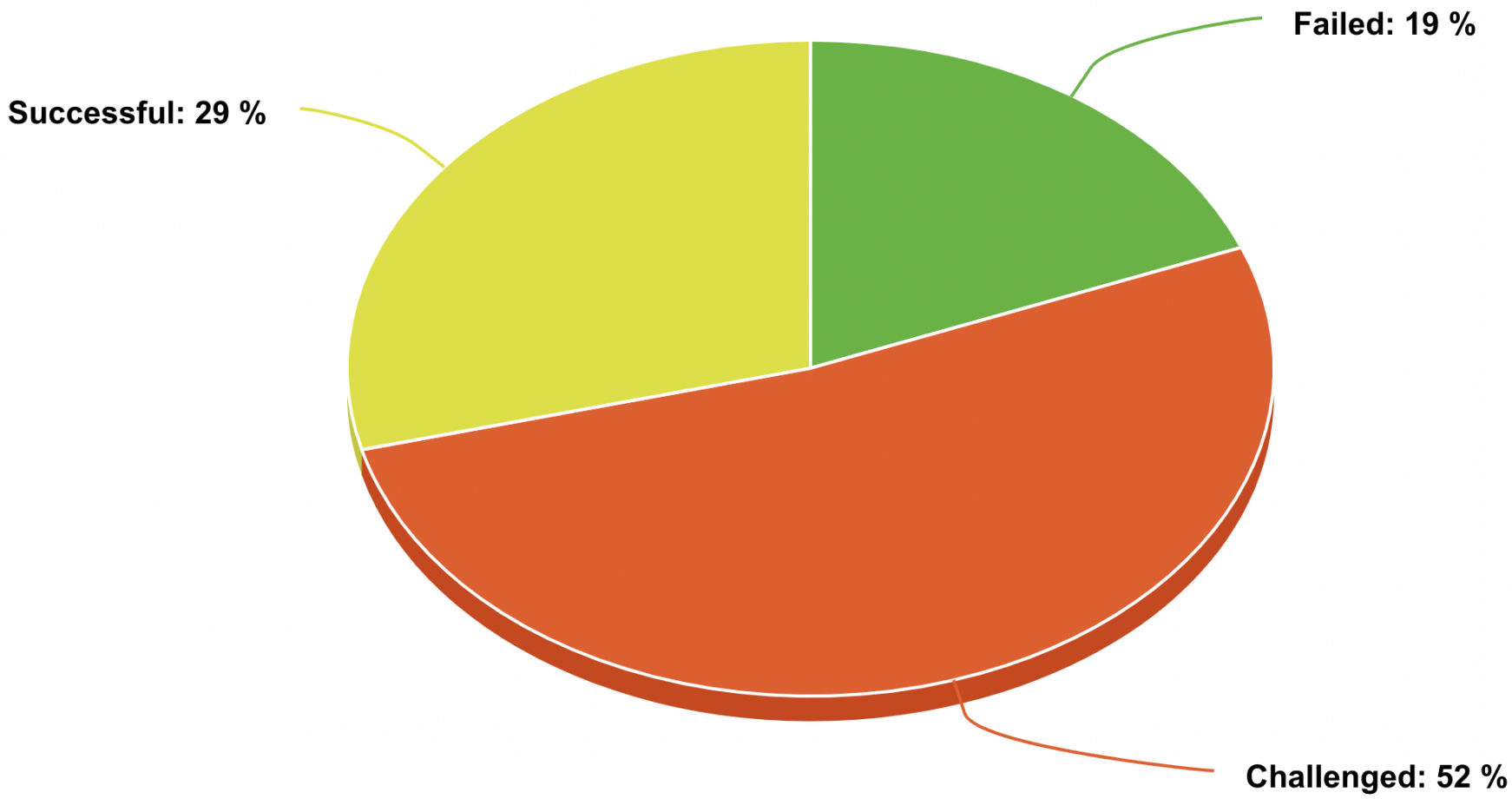
(Picture of a 16th century stakeholder, checking up on the project team)

Software Engineer

Sometimes ad-hoc measures to prevent the structure from collapsing are required

Success rate

Both 16th century cathedrals and 21th century software projects tend to fail often



Best practices

Clever acronyms:

- DRY (Don’t Repeat Yourself)
- KISS (Keep It Simple, Stupid/Silly/Simon)
- YAGNI (You Aren’t Gonna Need It)

Best practices

Principles:

- The single responsibility principle
- The open/closed principle
- The Liskov substitution principle
- The interface segregation principle
- The dependency inversion principle
- The principle of least astonishment
- The principle of least knowledge (the "Law Of Demeter")

Design patterns

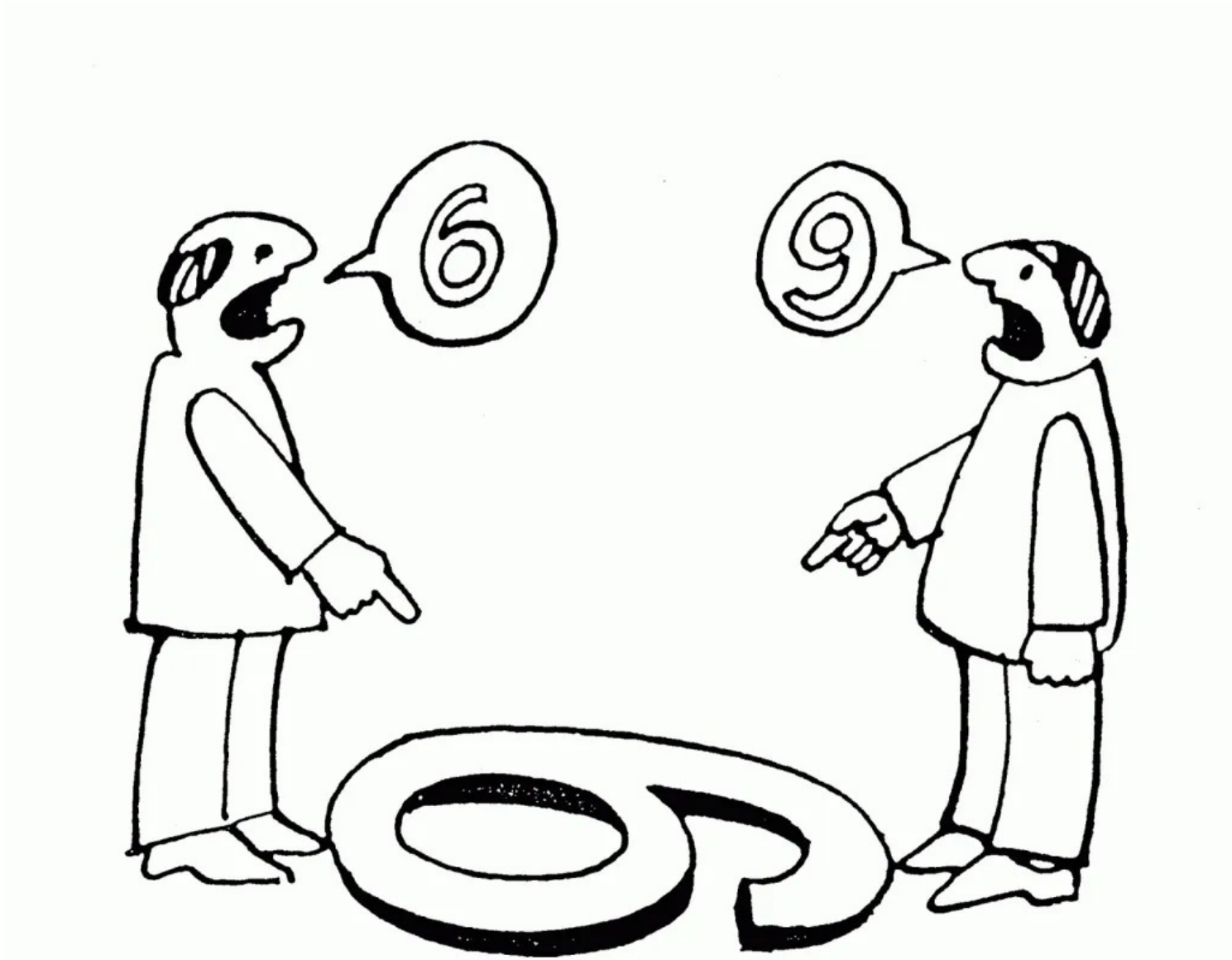
Reusable solutions to common problems in software design

- Factory
- Decorator
- Visitor
- Bridge
- Composite
- …

Best practices

Sometimes people just don’t agree what is "best"

- TDD
- Functional Programming
- Singleton Pattern


6
9

Disclaimer

Every "best practice" can be a "bad practice" in certain situations. Don’t follow too blindly!

DRY

"Don’t Repeat Yourself"

```
class App {
    public static void main(String[] args) {
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
        System.out.println("I will not repeat myself!");
    }
}
```

DRY

DRY is about removing repeated expressions/statements:

```
double input = 36;

// The square root of any number is always bigger than zero:
double guessTooLow = 0;

// The quare root of any number is always smaller than that number itself:
```

```
double guessTooHigh = input;

// Our best guess is right in the middle of those two:
double bestGuess = (guessTooLow + guessTooHigh) / 2;

// Check if we are done:
boolean foundSquareRoot = bestGuess * bestGuess == input;

while (!foundSquareRoot) {

    // Print our best guess:
    System.out.println("Best guess: " + bestGuess);

    if (bestGuess * bestGuess < input) {
        guessTooLow = bestGuess;
    }

    if (bestGuess * bestGuess > input) {
        guessTooHigh = bestGuess;
    }

    // Improve our guess:
    bestGuess = (guessTooLow + guessTooHigh) / 2;

    // Check if we are done now:
    foundSquareRoot = bestGuess * bestGuess == input;
}
System.out.println("Found: " + bestGuess);
```

DRY

We decided to introduce a function and use that everywhere the expression `(guessTooLow + guessTooHigh) / 2` appears:

```
double midPoint(double x, double y) {
    return (x + y) / 2;
}
```

DRY

```
double input = 36;

// The square root of any number is always bigger than zero:
double guessTooLow = 0;

// The quare root of any number is always smaller than that number itself:
double guessTooHigh = input;

// Our best guess is right in the middle of those two:
double bestGuess = midPoint(guessTooLow, guessTooHigh);

// Check if we are done:
boolean foundSquareRoot = bestGuess * bestGuess == input;

while (!foundSquareRoot) {

    // Print our best guess:
    System.out.println("Best guess: " + bestGuess);

    if (bestGuess * bestGuess < input) {
        guessTooLow = bestGuess;
    }

    if (bestGuess * bestGuess > input) {
        guessTooHigh = bestGuess;
    }
```

```
    // Improve our guess:
    bestGuess = midPoint(guessTooLow, guessTooHigh);

    // Check if we are done now:
    foundSquareRoot = bestGuess * bestGuess == input;
}
System.out.println("Found: " + bestGuess);
```

DRY

First 10,000 decimals of $\sqrt{2}$

1.
4142135623 7309504880 1688724209 6980785696 7187537694 8073176679 7379907324 7846210703 8850387534 3276415727 3501384623 0912297024 9248360558
5073721264 4121497099 9358314132 2666592750 5592755799 9505011527 8206057147 0109559971 6059702745 3459686201 4728517418 6408891986 0955232923
0484308714 3214508397 6260362799 5251407989 6872533965 4633180882 9640620615 2583523950 5474575028 7759961729 8355752203 3753185701 1354374603
4084988471 6038689997 0699004815 0305440277 9031645424 7823068492 9369186215 8057846311 1596668713 0130156185 6898723723 5288509264 8612494977
1542183342 0428568606 0146824720 7714358548 7415565706 9677653720 2264854470 1585880162 0758474922 6572260020 8558446652 1458398893 9443709265
9180031138 8246468157 0826301005 9485870400 3186480342 1948972782 9064104507 2636881313 7398552561 1732204024 5091227700 2269411275 7362728049
5738108967 5040183698 6836845072 5799364729 0607629969 4138047565 4823728997 1803268024 7442062926 9124859052 1810044598 4215059112 0249441341
7285314781 0580360337 1077309182 8693147101 7111168391 6581726889 4197587165 8215212822 9518488472 0896946338 6289156288 2765952635 1405422676
5323969461 7511291602 4087155101 3515045538 1287560052 6314680171 2740265396 9470240300 5174953188 6292563138 5188163478 0015693691 7688185237
8684052287 8376293892 1430065586 9568685964 5951555016 4472450983 6896036887 3231143894 1557665104 0883914292 3381132060 5243362948 5317049915
7717562285 4974143899 9188021762 4309652065 6421182731 6726257539 5947172559 3463723863 2261482742 6222086711 5583959992 6521176252 6989175409
8815934864 0083457085 1814722318 1420407042 6509056532 3333984364 5786579679 6519267292 3998753666 1721598257 8860263363 6178274959 9421940377
7753681426 2177387991 9455139723 1274066898 3299898953 8672882285 6378697749 6625199665 8352577619 8939322845 3447356947 9496295216 8891485492
5389047558 2883452609 6524096542 8893945386 4662574492 7556381964 4103169798 3306185201 9379384940 0571563337 2054806854 0575867999 6701213722
3947582142 6306585132 2174088323 8294728761 7393647467 8374319600 0159218880 7347857617 2522118674 9042497736 6929207311 0963697216 0893370866
1156734585 3348332952 5467585164 4710757848 6024636008 3444911481 8587655554 2864551233 1421992631 1332517970 6084365597 0435285641 0087918500
7603610091 5946567067 6883605571 7400767569 0509613671 9401324935 6052401859 9910506210 8163597726 4313806054 6701029356 9971042425 1057817495
3105725593 4984451126 9227803449 1350663756 8747760283 1628296055 3242242695 7534529028 8387684464 2917328277 0888318087 0253398523 3812274999
0812371892 5407264753 6785030482 1591801886 1671089728 6922920119 7599880703 8185433325 3646021108 2299279293 0728717807 9988809917 6741774108
9830608003 2631181642 7988231171 5436386966 1702999934 1616148786 8601804550 5553986913 1151860103 8637532500 4558186044 8040750241 1951843056
7453368361 3674597374 4239885532 8517930896 0373898915 1731958741 3442881784 2125021916 9518755934 4438739618 9314549999 9061075870 4909026088
3517636224 7497578588 5836803745 7931157339 8020999866 2218694992 2595913276 4236194105 9210032802 6149874566 5996888740 6795616739 1859572888
6424734635 8588686449 6822386006 9833526427 9905628316 5613913942 5576490620 6518602164 7263033362 9750756978 7060660685 6498160092 7187092921
5313236828 1356988937 0974165044 7459096053 7472796524 4770940992 4123871061 4470543986 7436473384 7745481910 0872886222 1495895295 9118789214
9179833981 0837882781 5306556231 5810360648 6758730360 1450227320 8829351341 3872276841 7667843690 5294286984 9083845574 4579409598 6260742499
5491680285 3077398938 2960362133 5398753205 0919989360 7513906444 4957684569 9347127636 4507163279 1547015977 3354863893 9423257277 5400382602
7478567417 2580951416 3071595978 4981800944 3560379390 9855901682 7215403458 1581521004 9366629534 4882710729 2396602321 6382382666 1262680350
2572781169 4510353793 7156882336 5932297823 1929860646 7978986409 2085609558 1426143636 3100461559 4332550474 4939759339 9912541953 2300932175
3044765339 6470662761 1661753518 7546462096 7634558738 6164880198 8484974792 6404506544 4896910040 7942118169 2579685756 3784881498 9864168549
9491635761 4484047021 0339892153 4237703723 3353115645 9443897036 5316672194 9049351882 9058063074 0134686264 1672470110 6534634939 1640714628
5567980177 9338144240 4526913706 6609777638 7848662380 0339232437 0474115331 8725319060 1916599645 5381157888 4138084332 3210533767 4618121780
1429609283 2411362752 5408873729 0512940733 9479433061 9439569367 0207942951 5878228349 3219316664 1113015495 9469837897 7674344435 3933770995
7134988407 8908508158 9236607008 8658105470 9497904657 2298888089 2461282816 0131337010 2908029099 9745647849 5815456146 4871551639 0502419857
9061310934 5878330620 0262207372 4716766854 5549990499 4085710809 9257599288 9323661543 8271955005 7816251330 3815314657 7907926868 5008069844
2847915242 4275441026 8057563215 6532206188 5751225113 0639370253 6292716196 8251259192 0252160587 0118959673 2244239267 4237344907 6464672737
5347964598 8191498079 3171800242 3855453886 0383683108 0077918246 6462754117 4442500187 2777951816 4383451463 4612990207 6334301796 8554385631
6677235183 8933666704 2222110939 1449302879 6381283988 9311731308 4300421255 5018549850 6529455637 7660314612 5590910461 1384768282 3595924772
2862904264 2736163264 5854433928 7726386034 3149804896 3973633297 5488592568 1149296836 1267258985 7383321643 6663487023 4773026101 0613050729
8611534129 9488087744 7311122954 2652751653 6659117301 4236062652 5869077198 2170370981 0464436047 7226739282 9874152593 0695620638 4710827408
2184906737 2330587430 2970924289 9481739244 0786937528 4401044399 0485208788 5191419354 1512900681 7351703069 3869705900 4742515765 5248078447
3621441050 1620084544 4122255956 2029847259 4035280190 6798068098 3003964539 8568593045 8625260637 7974535599 2774729906 4888745451 2424960763
7801086390 0191058092 8747647207 5110923860 5950195432 2816020887 9621516233 8521612875 2285180252 9287618325 7037172857 4067639449 0982546442
2184654308 8066105802 0158472840 6712630254 5937989065 0816857137 1656685941 3005331970 3659640337 6674146104 9563765103 0836613489 3109478026
8129355733 1890551970 5201845150 3996909866 3152512411 6111925940 5528085649 8931958983 4562331983 6834948808 0617156243 9112866312 7978483719
7895336901 5277600549 8055166350 1978555711 0140555297 6338412750 4468604647 6631832661 1651820675 0120476699 1098721910 4447440326 8943641595
9427921994 4235537187 0429955924 0314091712 8481585438 6600538571 3583639816 3094524075 5700932516 8243441682 4083619792 7337282521 5462246961
5332170268 2995097908 9034594858 8783494396 1620435842 2497397187 1139589273 0509219705 4917176961 6004455808 9942787888 0369169432 8945951472
2672292612 4850696173 1638094108 2186004528 6102696547 5763043102 5602715231 3969482135 5198214097 1654909731 9992834925 6740974903 9229712634
8693414574 9331980417 1807611196 3902278664 0759224341 6776246623 6238913110 2704333045 7636814112 8321326308 5822394562 1959808661 2939996201
2341561763 1817431242 0089014983 8485604808 7986460839 3596492366 5142968125 7731432291 4568716827 6219961182 7826953157 4983802624 6517590541
0397618128 7604216386 1345022132 6272775661 2441133610 7751955577 4950865636 0673786650 6231856406 9912280187 5741785494 6612532759 9769796059
7760590756 4891066610 1583841720 2818530432 1190446577 5255427754 3798726054 8817361982 6758168628 3295260789 9322266836 0283851351 2281059318
5910286415 0815705631 9717315183 1362502435 9041463212 2392176633 9826893682 5315053005 9891547029 0953719326 6207341123 4947433678 8469020139
0497842852 1634144292 1458955828 7847669394 6464267812 2190497856 3635526336 8278051860 0986992489 3778600239 8769169807 6566219438 9854437080
5946433362 3338105874 5816235475 6001365924 3524265714 3083465545 7680023708 1467573252 5470255074 7637471635 0678515991 7369379325 1032682760
6286459146 1820472148 6370370771 9269268236 2333472037 9245964691 8105261391 5308628029 1440965482 5638730927 3042654466 2929045896 0637519187
1146934536 1973324789 5727070315 3093090192 1199199993 6157650035 0398405406 7425387927 5279227247 3356677060 7837911384 4889362613 6765706026
3600315132 9520953952 0285489738 4486256134 9244147086 0708660267 6349978793 4208758361 2194711699 4223848482 5959143045 2810706260 1508969135
3030177200 6271705440 2090669514 9152745977 1970594769 5474095210 2878725578 5688002219 3717743558 1107939308 8338455864 8277291008 6295545661
4130672123 0848740227 1210586863 2338823741 3884428938 1554446471 0575565146 8435702946 6350628938 7356986868 8376480326 5195284146 5351739530
2736120137 4203009867 3983851432 1900436028 9826982935 2939941412 9230580384 5650227072 1681516194 1011449826 3013649008 7704839848 8386090653
3685990545 8389520318 5648041493 2721423908 6516499943 1659207965 9535694307 2311291162 9286797517 1566889054 3932203569 1293324570 2080671944
4049730494 3981408227 8296027994 2454108316 6675921424 8351827238 1720504103 9274288801 5562233807 9614751243 3514731021 2845459448 9944499600
0752437519 5701166834 1744749079 5882099517 8367680232 3651767497 2301487457 7427259947 6096219843 2714835298 6111902728 7358490521 7975908374
1974860267 0605374623 1530039375 2123678677 5284869219 5857137554 2696848278 3631786110 9933680143 9159059748 4285805451 6130230143 9790570161
0889862777 9610750673 3326760486 5492925139 9781390535 8822768937 3220494148 3940135560 3565604421 4017612060 5131806891 9899626061 8483185340
1836237821 7266375804 5524719626 6174925422 8528045714 4204857834 2113228008 5287042054 8899234127 8554812367 6153770710 4254469868 5219911228
3542663499 9712748366 0762462418 2073646661 7128394748 4732804744 3040334410 7200428727 1275670279 5675824292 6271945458 0530026664 8996507956
9778178621 9421720052 3716536946 7704195111 9127046248 3605113028 9046437751 1486948878 4961511884 1471910001 2558838366 6067720841 1235153558
8112677895 7155859041 2576261601 0675131535 8021242733 1871000635 8249545040 9957940725 4798900316 8265123731 1905566829 1519430537 0848930786
9197428290 4903860372 3116099283 4243171222 5099454715 0192866648 7871079519 9518005463 3883844315 4817246354 8024451803 0845273431 0006213710
3462573306 0012349737 4435581809 6567846464 1533905146 5691932456 2353140577 9193698988 4236471835 2537580525 7713311200 7971040683 1549266540
2026046806 8183914378 2721476906 3242469517 1286367384 4313983337 1176159418 6999346626 2345373452 3567940124 1680922911 6360956372 1674528391
7099091466 4850739205 1516056047 3787106154 7021699607 4656930979 4426121469 2561593425 6494019122 9895147325 4471518126 3258368897 2822628332
9524035970 0727863364 6045947071 2417472946 8775705958 1573499628 4809956783 9255474240 4489918870 7106967524 2507745201 2293608105 7414265323
4724064162 1410333533 4055110452 1261750359 0284037454 5918645047 2762434207 1770929793 5401021409 6464502836 8341804075 8608100140 7216192477
1798098596 8111540446 4437285689 5928683197 7797786934 6415984697 4513391774 1537904877 8808300220 5833504674 6555323028 5873258351...

DRY

Instead of

```
bestGuess * bestGuess == input
```

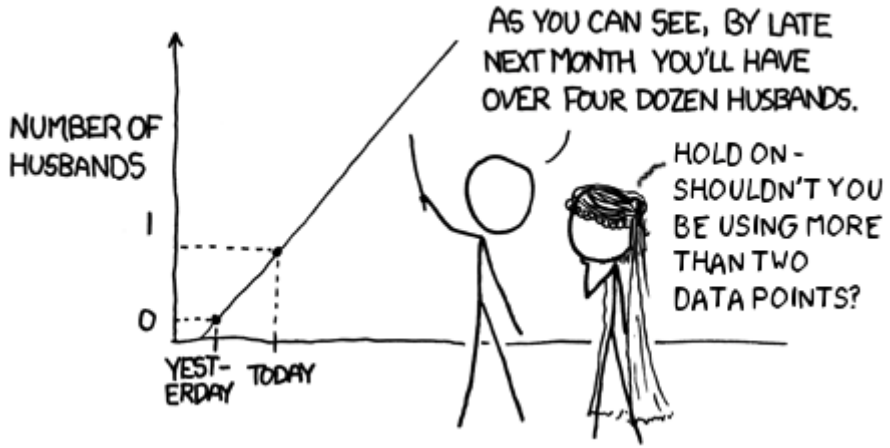
we'd like to use

```
Math.abs(bestGuess * bestGuess - input) < 0.00001
```

We need to make sure to update *both* copies!

Rule Of Three

Don't apply DRY too soon!

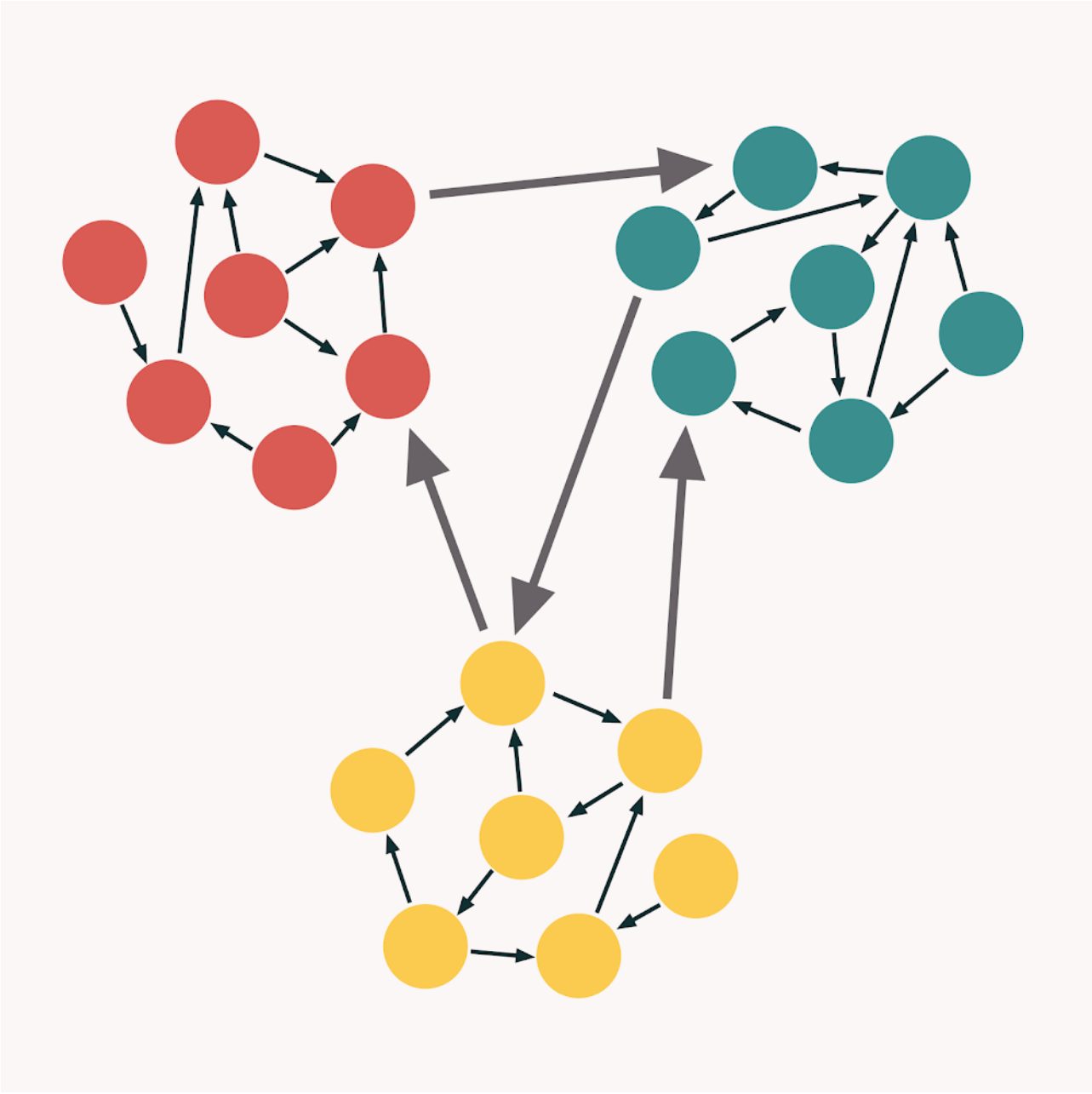


Single Responsibility Principle

Don’t bundle all methods in one class. Example: `Receipt`

- Methods for the store’s customer loyalty program
- Methods to update the inventory
- Methods for accounting
- Methods for marketing
- Methods used by a store designer

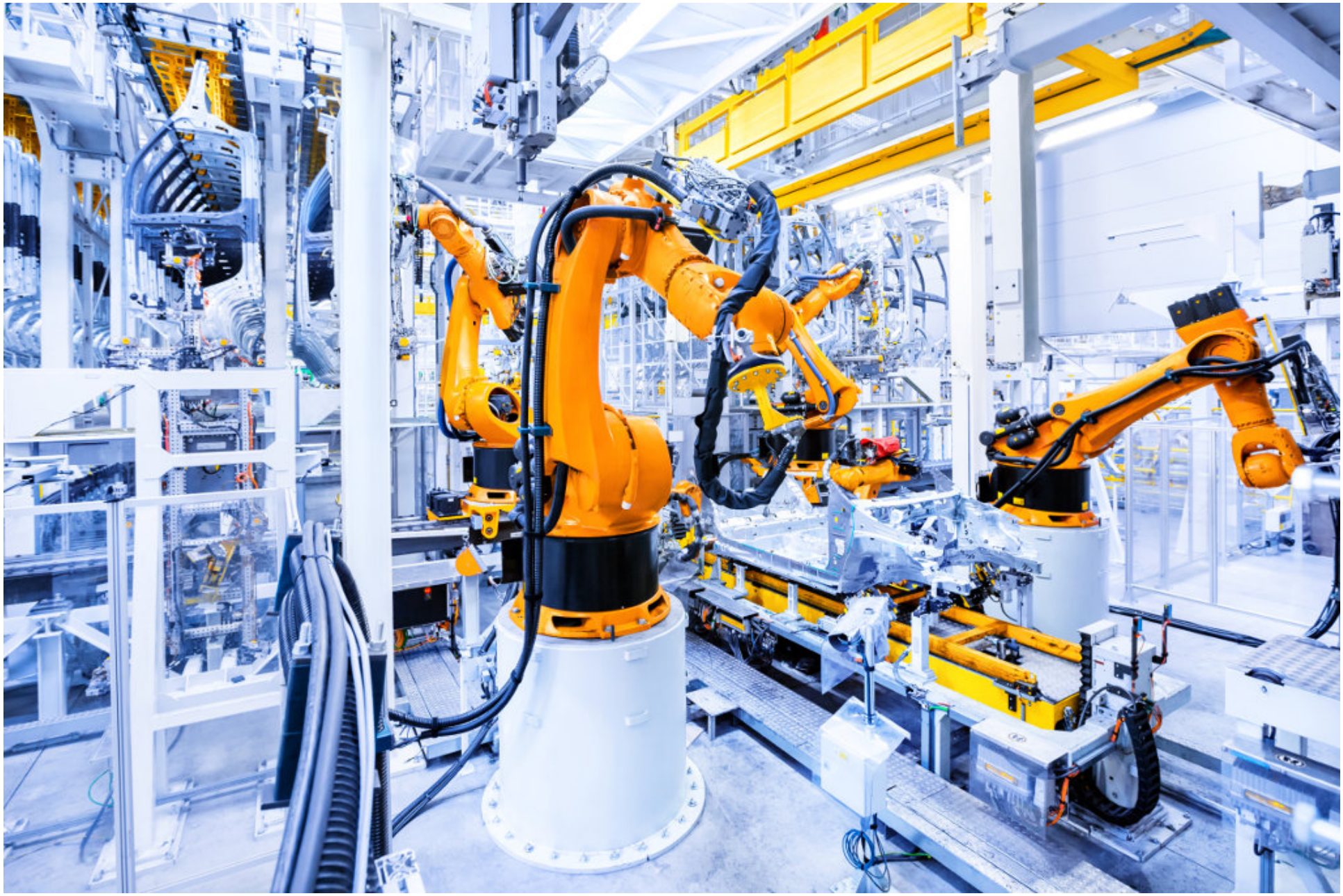
Single Responsibility Principle



Single Responsibilitiy Principle

Don’t take it too far

Factory Pattern



Factory Pattern

Helps stick to the single responsibility principle

- A `PlaneFactory` should produce a `Plane`
- A `BoatFactory` should produce a `Boat`
- A `CarFactory` should produce a `Car` (but may produce a `Mercedes` or a `Tesla`)

Code reviews

Many benefits:

- Check if requirements are satifisfied
- Look for bugs
- Identify code that is hard to understand/maintain
- Show you more elegant/efficient way to implement feature
- Let you know about some handy helper functions

Automated tests

Many benefits:

- Check if code behaves as it should
- See how it "feels" to create an instance of a class you just wrote, and to call its methods


QUALITY
MAX
MAX

Module: Designing classes

- Recap encapsulation
- Decide what to expose
- Immutability
- Visibility and access modifiers
- Reference escapism
- Defensive programming
- Exercises

Recap encapsulation

```
// We want this
double price = purchase.getPrice();

// We don't want this
purchase.price = 0.00; // Now it's free!
```

Recap encapsulation

```
// Instead of this
class Purchase {
    public double price;
}

// We do this
```

```
class Purchase {
    private double price;

    public double getPrice() {
        return price;
    }
}
```

Why?

Your class can look like this

```
class Purchase {
    private double price;

    public double getPrice() {
        return price;
    }
}
```

```
double price = purchase.getPrice();
```

Why?

Or like this

```
class Purchase {
    private int priceInCents;
```

```
    public double getPrice() {
        return priceInCents / 100;
    }
}
```

```
double price = purchase.getPrice();
```

Why?

Or even this:

```
class Purchase {
    // No field at all!

    public double getPrice() {
        return callDatabase().getPrice();
    }
}
```

```
double price = purchase.getPrice();
```

Recap encapsulation

- Freedom to change code
- No freedom for clients to mess with things

Decide what to expose

- It’s an art, not a science
- There are rules of thumb:
- Expose methods, not fields
- Expose only what you need, no more
- Favour immutability

Expose methods, not fields

```
class Purchase {
    public int priceInCents;
}
```

vs

```
class Purchase {
    private int priceInCents;

    public int getPriceInCents() {
        return priceInCents;
    }
}
```

Expose methods, not fields

- Move to a different interface
- Remain compatibility

```
class Purchase {
    private double price;

    public double getPrice() {
        return price;
    }

    public int getPriceInCents() {
        return Math.round(price * 100);
    }
}
```

Expose only what you need

- It’s easy to add things
- Its **hard** to remove things

Expose only what you need

Let’s say we have this

```
class Purchase {
    private int priceInEuros;
```

```
    private int priceRemainderInCents;

    public double getPrice() {
        return priceInEuros + (priceRemainderInCents / 100);
    }

    // Let's add these methods! It's easy!
    public int getPriceInEuros() {
        return priceInEuros;
    }

    public int getPriceRemainderInCents() {
        return priceRemainderInCents;
    }
}
```

Expose only what you need

We want to: * change the implementation * remove `getPriceInEuros()` and `getPriceRemainderInCents()`

But wait!

- The Accountancy team uses `getPriceInEuros()` for tax reasons!
- We can’t remove it
- Tax rules change, now they want us to change the implementation
- ¬
- Nooooo

- We have to support it forever

Immutability

Programmer speak for

classes whose internal values you can’t change

Immutability

So, instead of this:

```
class Purchase {
    private double price;

    public double getPrice() {
        return price;
    }

    public double setPrice(double newValue) {
        this.price = newValue;
    }
}
```

Immutability

Do this:

```
class Purchase {
```

```
    private final double price;

    public double getPrice() {
        return price;
    }
}
```

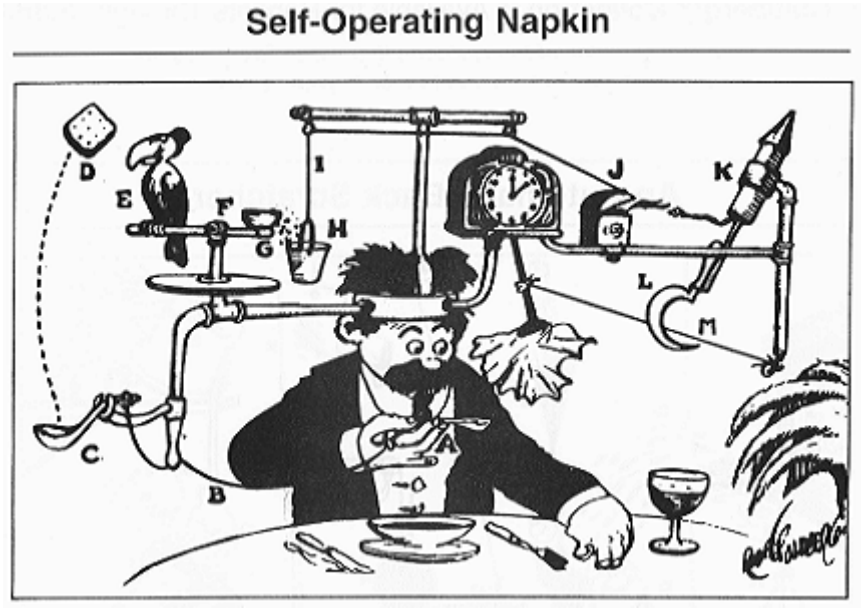
- No setter method
- Field is `final`

Why?

- Only expose what you need
- Easier to understand
    - Takes a while to appreciate
- Also: good for concurrency
    - You can forget about that

Why?

Less moving parts



Visibility and access modifiers

```
// private access
private int getPrice() { return 0; }

// default access
int getPrice() { return 0; }

// protected access
protected int getPrice() { return 0; }

// public access
public int getPrice() { return 0; }
```

Public & private access

- Public access
    - Visible to all classes
- Private access
    - Visible to current class only

Default access

```
// default access
int getPrice() { return 0; }
```

- Visible to all classes in same package
- "package private"
- "package protected"

Default access

```
package nl.yoink.training.domain;

public class Product {
    String getName() { return name; }
}

// ...
```

```
package nl.yoink.training.domain;

class Purchase {
    private final Product product;

    public getProductName() {
        return product.getName(); // yes
    }
}
```

```
package nl.yoink.training.domain;

public class Product {
    String getName() { return name; }
}

// ...

package nl.yoink.training.express;

class ExpressPurchace {
    private final Product product;

    public getProductName() {
        return product.getName(); // no
    }
}
```

Protected access

- Visible to all classes in same package

- Visible to all subclasses

Protected access

```
package nl.yoink.training.domain;

public class Product {
    String getName() { return name; }
}

// ...

package nl.yoink.training.express;

class ExpressPurchace {
    private final Product product;

    public getProductName() {
        return product.getName(); // no
    }
}
```

Protected access

```
package nl.yoink.training.domain;

public class Product {
    String getName() { return name; }
}

// ...
```

```
package nl.yoink.training.express;

class ExpressPurchace extends Purchase {
    private final Product product;

    public getProductName() {
        return product.getName(); // yes
    }
}
```

Default & protected access

- Rarely used in practice

Reference escapism

```
class Receipt {
    private final double[] amounts;

    public Receipt(double[] amounts) {
        this.amounts = amounts;
    }

    public double[] getAmounts() {
        return amounts;
    }
}
```

Reference escapism

```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
```

Call `getAmounts()`:

```
receipt.getAmounts();
// double[3] { 1.0, 2.0, 3.0 }
```

Reference escapism

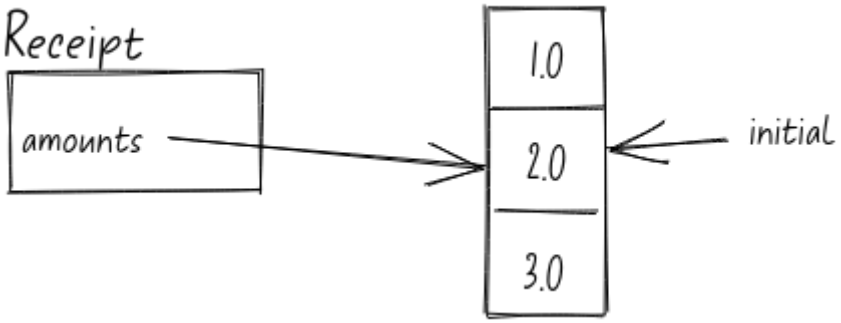
```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
initial[1] = 42.0;
```

Call `getAmounts()`:

```
receipt.getAmounts();
// double[3] { 1.0, 42.0, 3.0 }
```

Reference escapism

What happened?



"The reference has escaped the boundaries of the class"

Reference escapism

```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
double[] observed = receipt.getAmounts();
observed[1] = 42.0;
```

Call `getAmounts()`:

```
receipt.getAmounts();
// double[3] { 1.0, 42.0, 3.0 }
```

Reference escapism

How to prevent this?

- Use primitive values
- Use immutable objects
- Program defensively

Defensive programming

Assume your clients want to attack you

Try to prevent these attacks

Arrays

```
import java.util.Arrays;

class Receipt {
    private final double[] amounts;

    public Receipt(double[] amounts) {
        this.amounts = Arrays.copyOf(amounts, amounts.length);
    }

    public double[] getAmounts() {
        return Arrays.copyOf(amounts, amounts.length);
    }
}
```

Arrays

It works for the constructor:

```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
initial[1] = 42.0;

receipt.getAmounts();
// double[3] { 1.0, 2.0, 3.0 }
```

Arrays

It works for the getter:

```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
double[] observed = receipt.getAmounts();
observed[1] = 42.0;

receipt.getAmounts();
// double[3] { 1.0, 2.0, 3.0 }
```

Cloneable

Some (mutable) types implement `Cloneable`:

```
GregorianCalendar time = new GregorianCalendar();
```

```
GregorianCalendar copy = time.clone();
```

Cost

Copying large objects is expensive

- memory
- time

Wrapped types

Some (mutable) types can be wrapped:

```
List<String> strings = new ArrayList<>();
List<String> copy = Collections.umodifiableList(copy);
```

What does wrapping mean?

```
// Real implementation is different
public class UnmodifiableList<T> implements List<T> {
    private final List<T> underlying;

    public UnmodifiableList(List<T> underlying) {
        this.underlying = underlying;
    }

    public T get(int index) {
        return underlying.get(index);
```

```
    }

    public void add(T element) {
        throws new UnsupportedOperationException();
    }
}
```

Cost

Wrapping objects is cheap

- memory
- time

Exercises

Fill in access modifiers in the following classes, and explain your choices.

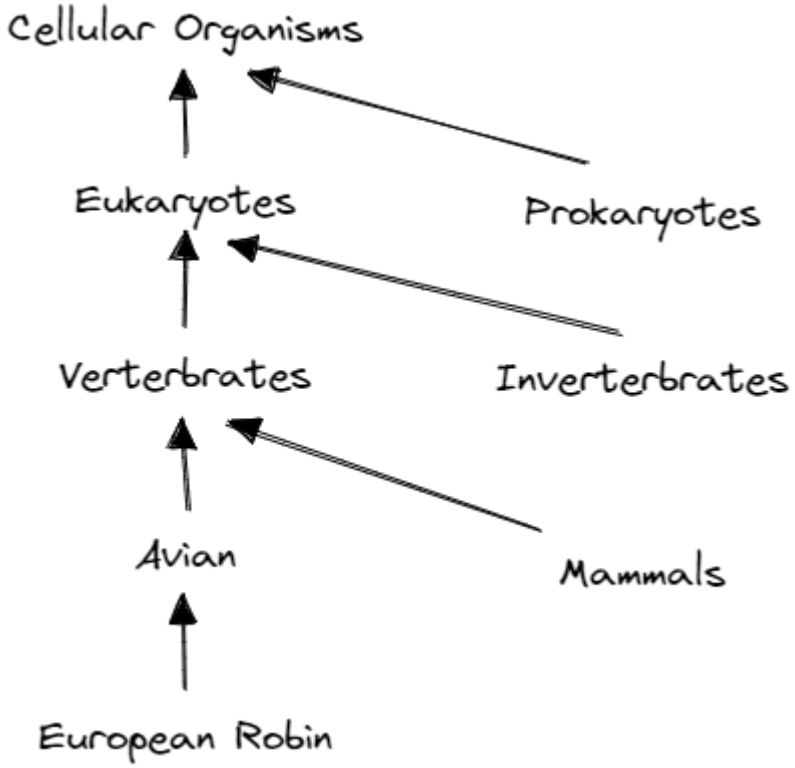
```
class Person {
    final String name;
    final int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    String getName() {
        return name;
    }
```

```
    int getAge() {
        return age;
    }
}
```

Exercises

- Prove by writing a Java program that reference escapism really happens.
- Prove by writing a Java program that with primitive values, reference escapism doesn't happen.
- Can we have reference escapism with `java.util.GregorianCalendar`? And with `java.time.LocalDateTime`? Write programs to prove it.
- Fix the vulnerable programs above using defensive programming techniques.

Module: inheritance (part 1)



Animals or shapes?

Shapes!

They have

- position
- area

- ability to be drawn on a canvas

Rectangle

```
public class Rectangle {
    private double x;
    private double y;
    private double width;
    private double height;

    public Rectangle(double x, double y, double width, double height) {
        this.x = x;
        this.y = y
        this.width = width;
        this.height = height;
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }

    public double area() {
        return edge * edge;
    }

    public void drawOn(Canvas canvas) {
        // cool rectangle drawing logic comes here
    }
```

```
}
```

Circle

```
public class Circle {
    private double x;
    private double y;
    private double radius;

    public Circle(double x, double y, double radius) {
        this.x = x;
        this.y = y
        this.radius = radius;
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public double getRadius() { return radius; }
    public void setRadius(double radius) { this.radius = radius; }

    public double area() {
        return Math.PI * radius * radius;
    }

    public void drawOn(Canvas canvas) {
        // cool circle drawing logic comes here
    }
}
```

Triangle

```
public class Triangle {
    private double x;
    private double y;
    private double base;
    private double height;

    public Triangle(double x, double y, double base, double height) {
        this.x = x;
        this.y = y
        this.base = base;
        this.height = height;
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public double getBase() { return base; }
    public void setBase(double base) { this.base = base; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }

    public double area() {
        return base * height / 2;
    }

    public void drawOn(Canvas canvas) {
        // cool triangle drawing logic comes here
    }
}
```

Similarities

What similarities did you see?

Sharing behaviour

```
public abstract class Shape {
    private double x;
    private double y;

    public Shape(double x, double y) {
        this.x = x;
        this.y = y
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }

    public abstract double area();

    public abstract void drawOn(Canvas canvas);
}
```

What did we see?

- `abstract` class
    - We can’t call the constructor anymore

- `abstract` methods
    - `;` instead of code

Extending Shape

```
public class Rectangle extends Shape {
    private double edge;

    public Rectangle(double x, double y, double width, double height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }

    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }

    @Override
    public double area() {
        return edge * edge;
    }

    @Override
    public void drawOn(Canvas canvas) {
        // cool rectangle drawing logic comes here
    }
}
```

Super and sub

- Shape is the *superclass* of Rectangle
    - A class can have only 1 superclass
- Rectangle is a *subclass* of Shape
    - A class can have many subclasses

What did we see?

- class `extends Shape`
- Constructor calls super constructor with `super(x, y);`
    - Required
    - Must be first in method
- Abstract methods now get body
    - @Override

What's more

`x` and `y` are private in Shape

Rectangle must use `getX()` and `getY()`

Circle

```
public class Circle extends Shape {
    private double radius;

    public Circle(double x, double y, double radius) {
        super(x, y);
        this.radius = radius;
    }

    public double getRadius() { return radius; }
    public void setRadius(double radius) { this.radius = radius; }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }

    @Override
    public void drawOn(Canvas canvas) {
        // cool circle drawing logic comes here
    }
}
```

Extending classes

Let’s define Square

Is it a Rectangle?

Square

```
public class Square extends Rectangle {
    private double edge;

    public Square(double x, double y, double edge) {
        super(x, y, edge, edge);
    }

    public double getEdge() { return edge; }
    public void setEdge(double edge) { this.edge = edge; }
}
```

What did we see?

- You can extend a class that is not abstract
- You can extend a class that already extends something else
    - All Java classes extend Object
- Constructor params don't need to be identical
    - You still have to call `super()`
- Square has everything that Rectangle has
    - `getWidth()`, `getHeight()`, `drawOn()`

But wait!

`setEdge()` doesn't update width or height!

```
public void setEdge(double edge) {
    this.edge = edge;
    setWidth(edge);
    setHeight(edge);
}
```

Now it does

But wait!

We need to do the same with `setWidth()` and `setHeight()`:

```
@Override
public void setWidth(double width) {
    setWidth(width);
    setHeight(width);
    setEdge(width);
}

@Override
public void setHeight(double width) {
    setHeight(width);
    setWidth(width);
    setEdge(width);
}
```

But wait!

`setWidth()` and `setHeight()` call each other:

infinite loop!

Other options

Can you think of some?

Other options

We can’t remove the methods from the class in Java

Is Square still a Rectangle if we could?

Other options

Make `width` and `height` fields protected

Breaks encapsulation?

Other options

Make `setWidth()` and `setHeight()` throw exceptions

Ugly

Other options

Make all Shapes immutable

Requires re-architecting the code

Other options

Make Square extend Shape directly

Square is no longer a rectangle

Other options

Which is the best?

It depends

Polymorphism

Is a way to allow for change and extensibility

Canvas

What is it?

- HTML canvas
- GUI canvas
- Headless canvas

Canvas

```
public abstract class Canvas {
    public abstract void drawPixel(double x, double y);

    public void drawLine(double x1, double y1, double x2, double y2) {
        // calculate all pixels in the line, and draw each of those pixels
    }

    public void drawEllipse(double centerX, double centerY, double width, double height) {
        // calculate all pixels on the ellipse, and draw each of those pixels
    }
}

public class HtmlCanvas extends Canvas {
    @Override
    public void drawPixel(double x, double y) {
        // HTML-specific pixel drawing
    }
}

public class GuiCanvas extends Canvas {
    @Override
    public void drawPixel(double x, double y) {
        // native pixel drawing
    }
}
```

Canvas

Shape doesn’t care what kind of canvas it is

As long as `drawPixel()`, `drawLine()` and `drawEllipse()` are there

Drawing

```
@Override
public void drawOn(Canvas canvas) {
    canvas.drawLine(x, y, x, y + height);
    canvas.drawLine(x, y + height, x + width, y + height);
    canvas.drawLine(x + width, y + height, x + width, y);
    canvas.drawLine(x + width, y, x, y);
}
```

Polymorphism

- We ask for a Canvas
- We get HtmlCanvas, or GuiCanvas, or some other Canvas
- We don’t care which one we get

Composition over inheritance

- X is-a Y
    - extending
    - inheritance
- X has-a Y
    - fields/properties

- composition

is-a vs has-a

- Rectangle is-a Shape
- Circle is-a Shape
- Shape has-a x coordinate and a y coordinate
- Rectangle has-a width and a height

We get it wrong sometimes

Sometimes we use is-a

when has-a would be better

Example

Payroll system entities:

- Person
- Employee
- Manager

Example

Employee is-a Person

Manager is-a Person

Example

But a Manager is also an Employee!

Example

Composition to the rescue!

Example

```
class Person {
    private String name;
    private int age;
}

class Employee {
    private Person person;
    private double salary;
}

class Manager {
    private Person person;
    private Employee[] minions;
```

```
}
```

Best practice

Prefer composition ('has-a') over inheritance ('is-a')

Exercise

Write down the implementation of Triangle when it extends Shape.

Exercise

Try to add a function that calculates the circumference of all Shapes. What problems do you run into?

Exercise

Can you think of a way to take advantage of the polymorphism of Shapes? I.e., what could you do with Shapes where it doesn’t matter whether they are Rectangles, Circles, or Triangles?

Exercise

Does it make sense to have Manager extend Employee?

Exercise

In the introduction I mentioned one of two common inheritance examples: shapes. The other one, of course, is animals! Come up with an inheritance hierarchy of animals. Each animal has a sound it can make, a number of legs, and a type of food it likes. Try to implement this in Java. For inspiration:

think cats, dogs, tigers, giraffes, butterflies, crickets, salmon, and cephalopods, and add whatever else you can think of! Do fish make sounds? What other properties can you come up with?