“If it stinks, you have to change it.”

Code Smells

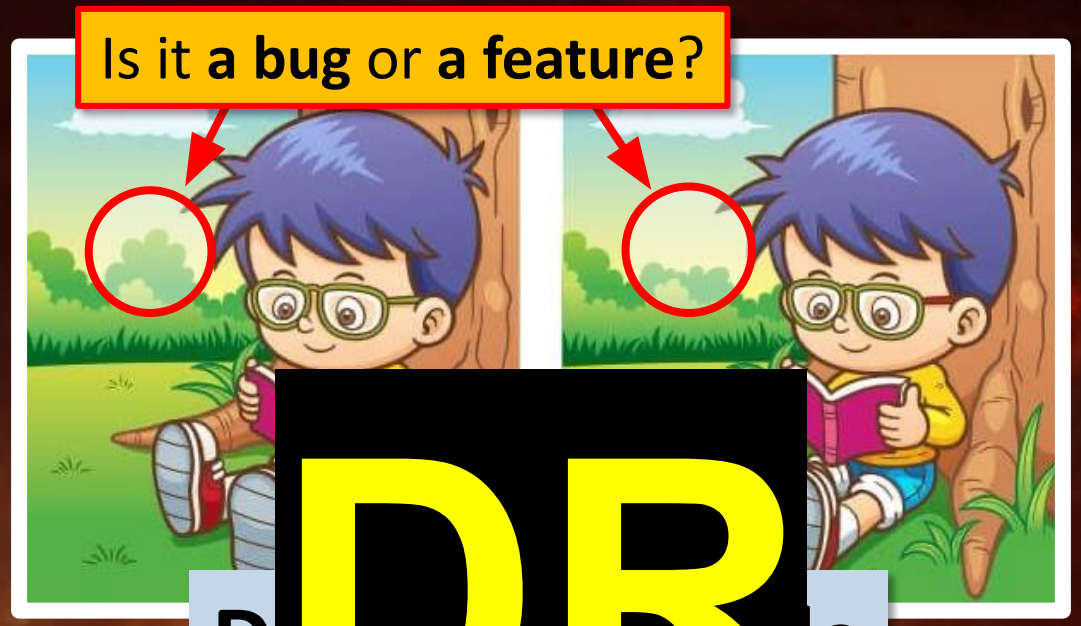
= code patterns that hurt you later

Copy-Paste Programming

DRY

Don't Repeat

This is code copy-pasted 2 years ago



DRY

D e

becomes horrible when it changes

Don't

Code Smells

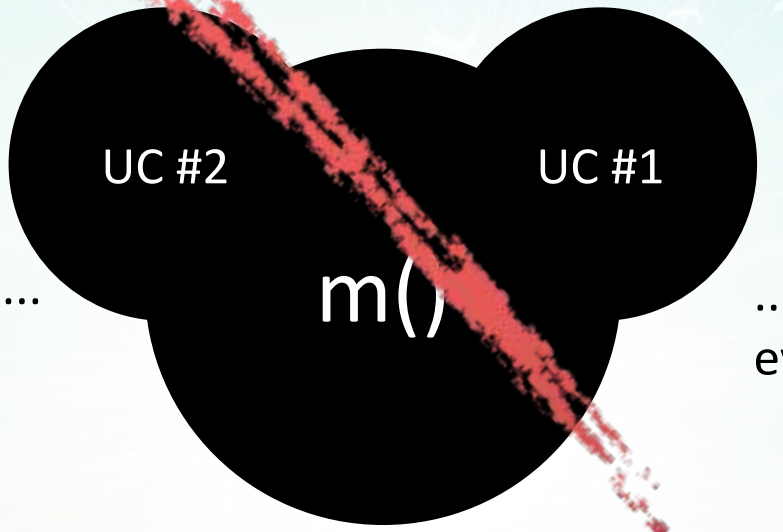
DRY =

Don't have any duplicated code – not the goal ❌

Do NOT duplicate knowledge ✅

eg the same biz rule implemented in 2 places

Once upon a time, there was a method...



... that grew big, and nasty trying to do everyting for everyone

Flag Parameters

eg **`boolean`**

SRP violation

Divergent Changes

Code serving too many use-cases

"Scope Creep"

There are only two things hard in programming:

1) **cache** invalidation

2) **naming** things

A good name proves it does one clear thing

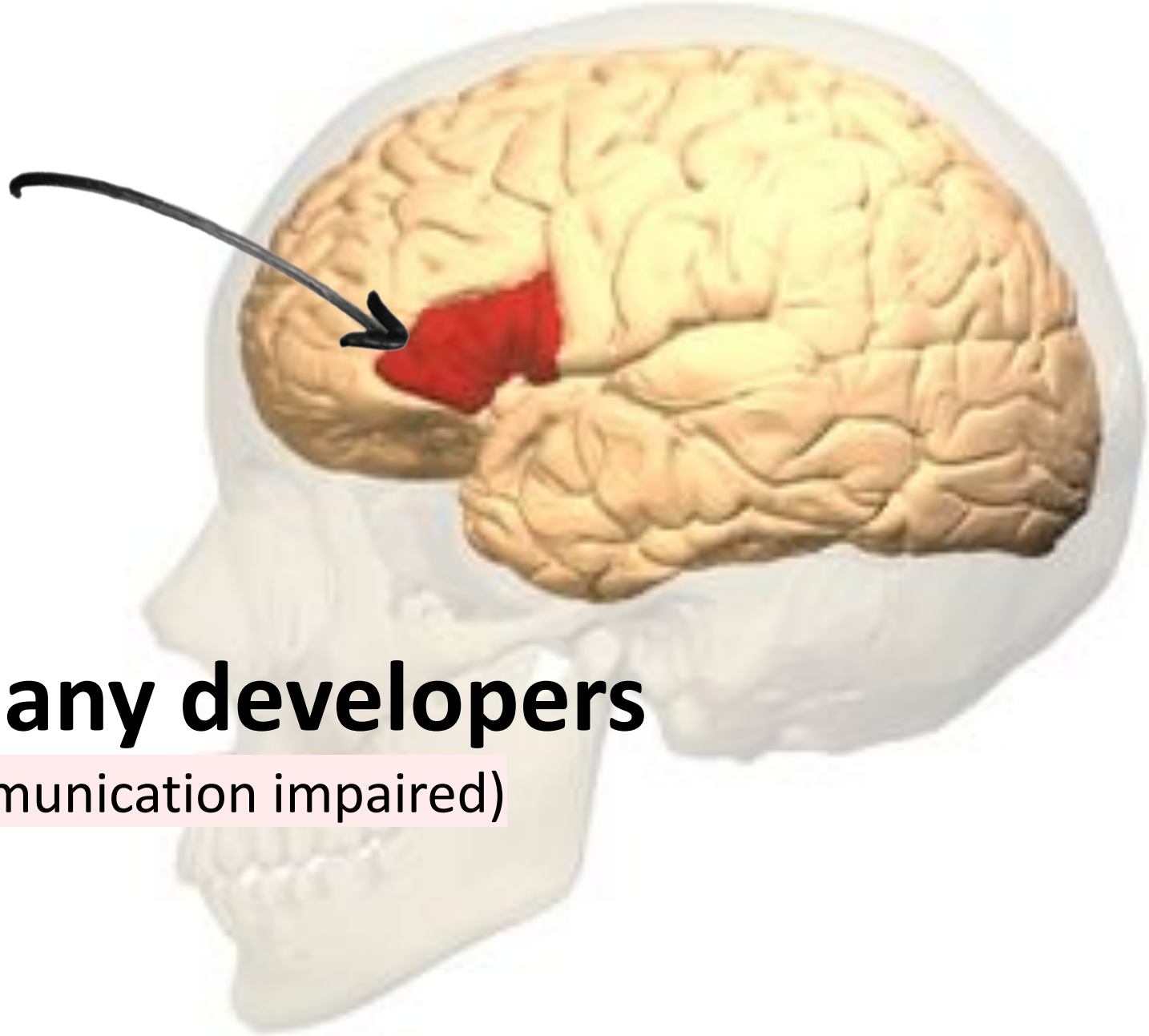
3) and off by one errors

(for the C++ nostalgics)

SRP

Broca’s Area

manages speech function



is atrophic in many developers

(many of us are communication impaired)

Bad Names

Bad Names

```
dr // mysterious acronym
```

```
-> { ... 20 lines of code } // anonymous logic
```

```
incrementPurse(player, ..); // purely technical, lacks intent
```

```
MovieType priceCode; // synonyms confuse😵‍💫
```

```
checkCustomer(customer); // updating customer
```

Command-Query Separation Violation

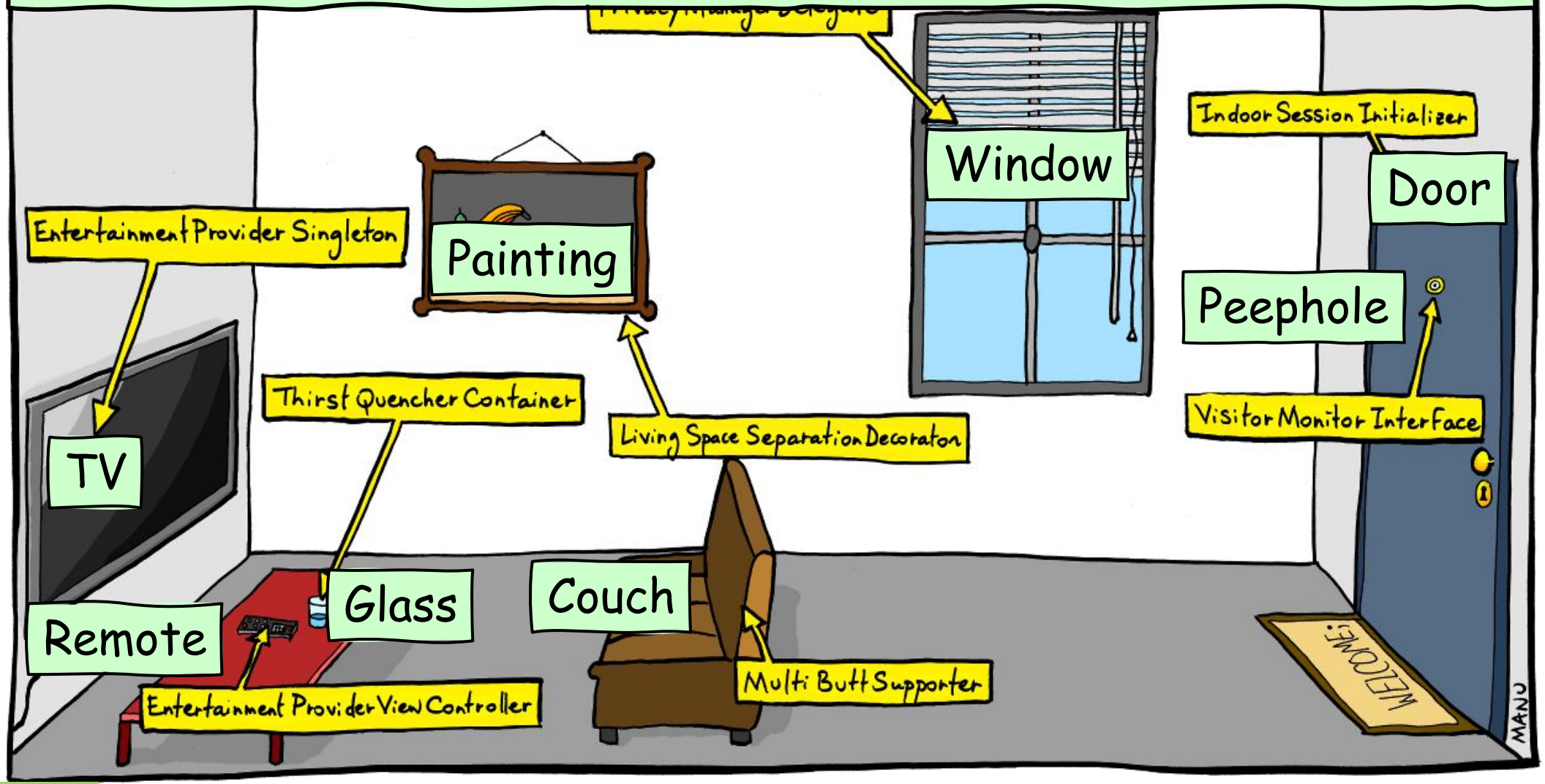
```
price = computePrice(..); // INSERT in DB?!
```

Names should be..

- **Consistent** with conventions: *get-, fetch-, find-*?
- **Unique**: synonyms are confusing: *buyer*, *client*, *customer*?
- **Close to domain**, in a ubiquitous language spoken by entire team
- **Short powerful names** are easier to remember
- **DRY**: Customer.~~customer~~Name
- **Functions** name start with a **verb**, unless getter/pure: offer.id()
- **Hide implementation details**: tell WHAT it does, not HOW or WHY
  When naming stuff you need to put yourself in the shoes of your client.
- Private methods use longer **explanatory names** (documentation)

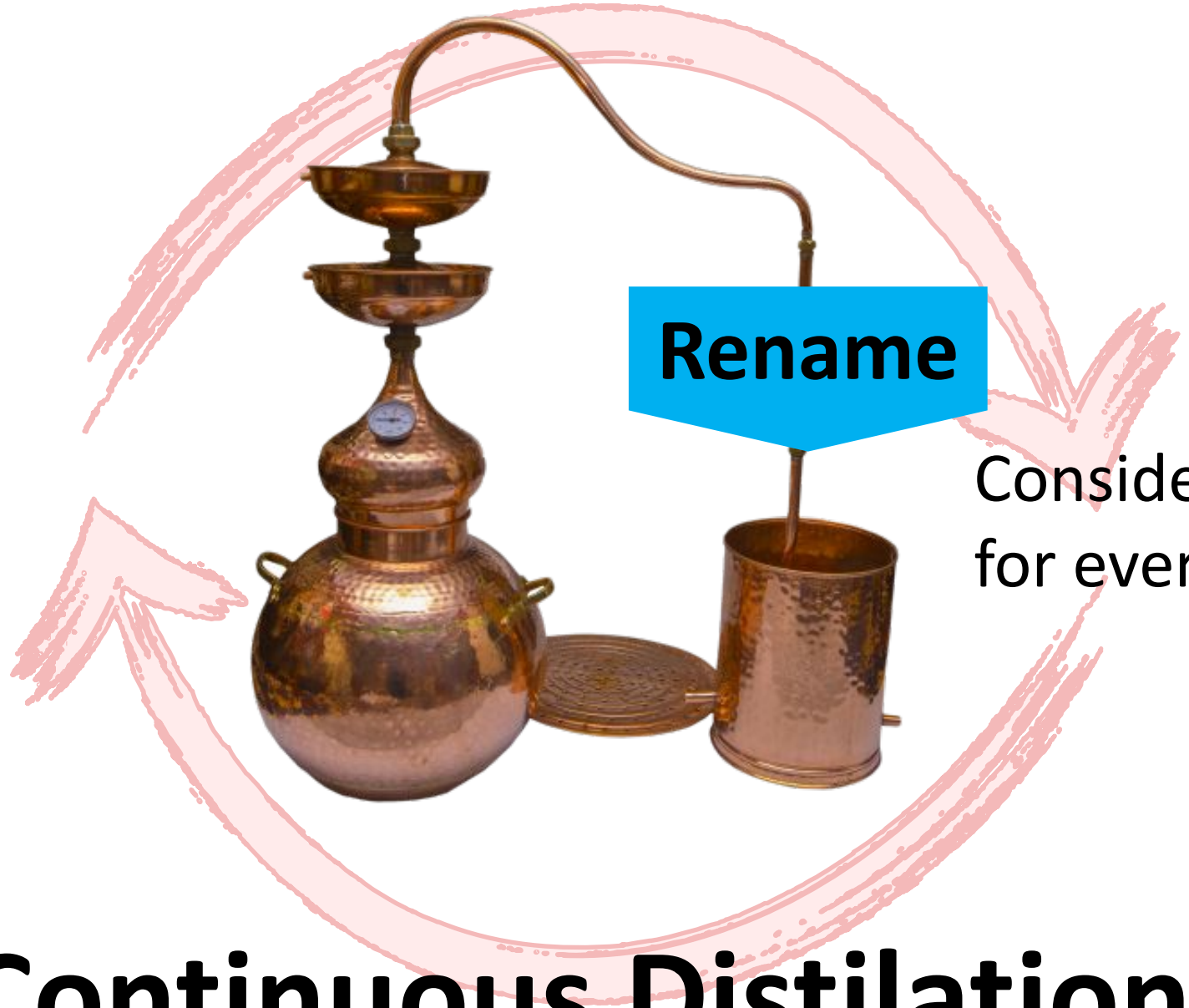
Names with **bad Signal/Noise** ratio

Find **powerful memorable domain terms**



Also see: FizzBuzz Enterpise Edition

👆 Influence of libraries and frameworks surrounding us?



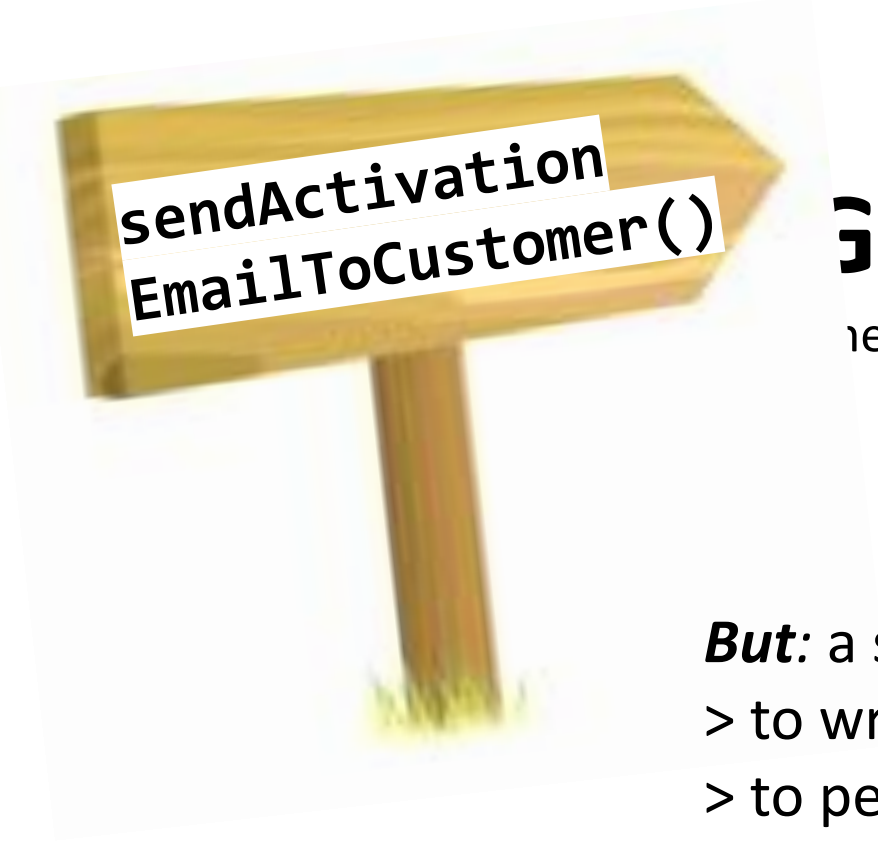
Consider 2-3 options
for every *fresh* name

Continuous Distilation

EXTRACT METHOD with long descriptive names

= 90% of what Clean Code means

The foundation of *Everything*



Guide your Readers

ıember: code is read 10x more than writter

Clean Code = Care for Others

But: a single large function might be easier:

> to write

> to performance tune

(when you need to load it up 100% in 🧠)

Stuff That Starts Small...

- 5 lines of code 😇

  then 7, 10, 15, 25, 45...

Bloaters

Monster Method

> 20 lines⛔

depth > 3⛔

The more complex, the shorter

Single Level of Abstraction (SLAb)

God Class

> 200 lines⛔

☐ Split in High / Low level

☐ Split in Flow A / Flow B

Many Parameters

> 4 ⛔

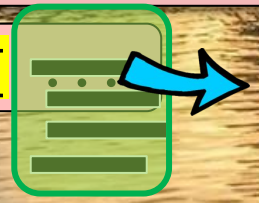
☐ Split method by SRP

☐ Extract **reusable** Parameter Object

also: Generics<A<B>,U,S,E>

Complex Lambda

-> {

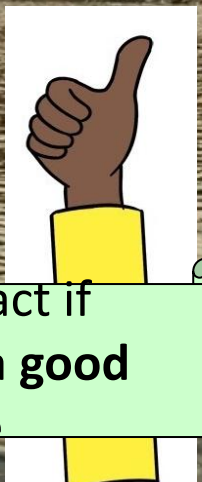


The magic number 7±2

Number of things humans can keep in working memory (in 4±1 in 2000s (pre-TikTok) ~~1956~~)

≤ 7 concepts / row
≤ 7 ideas / method
≤ 7 methods / class

Code Smells



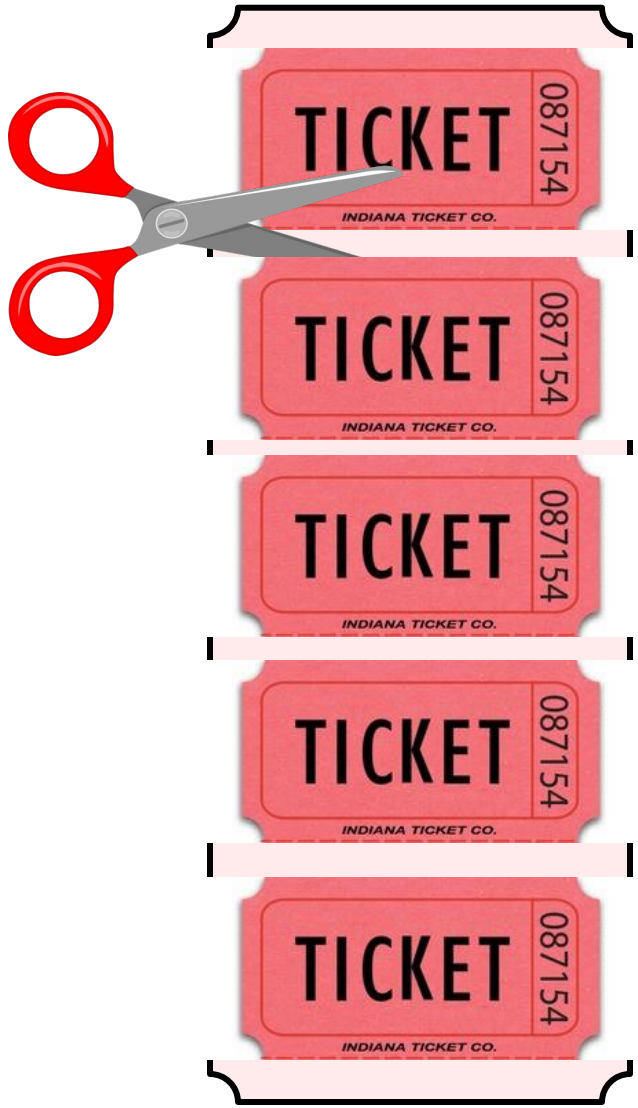
Only extract if you found a **good name**

Function Length

Function Depth

Flat Functions

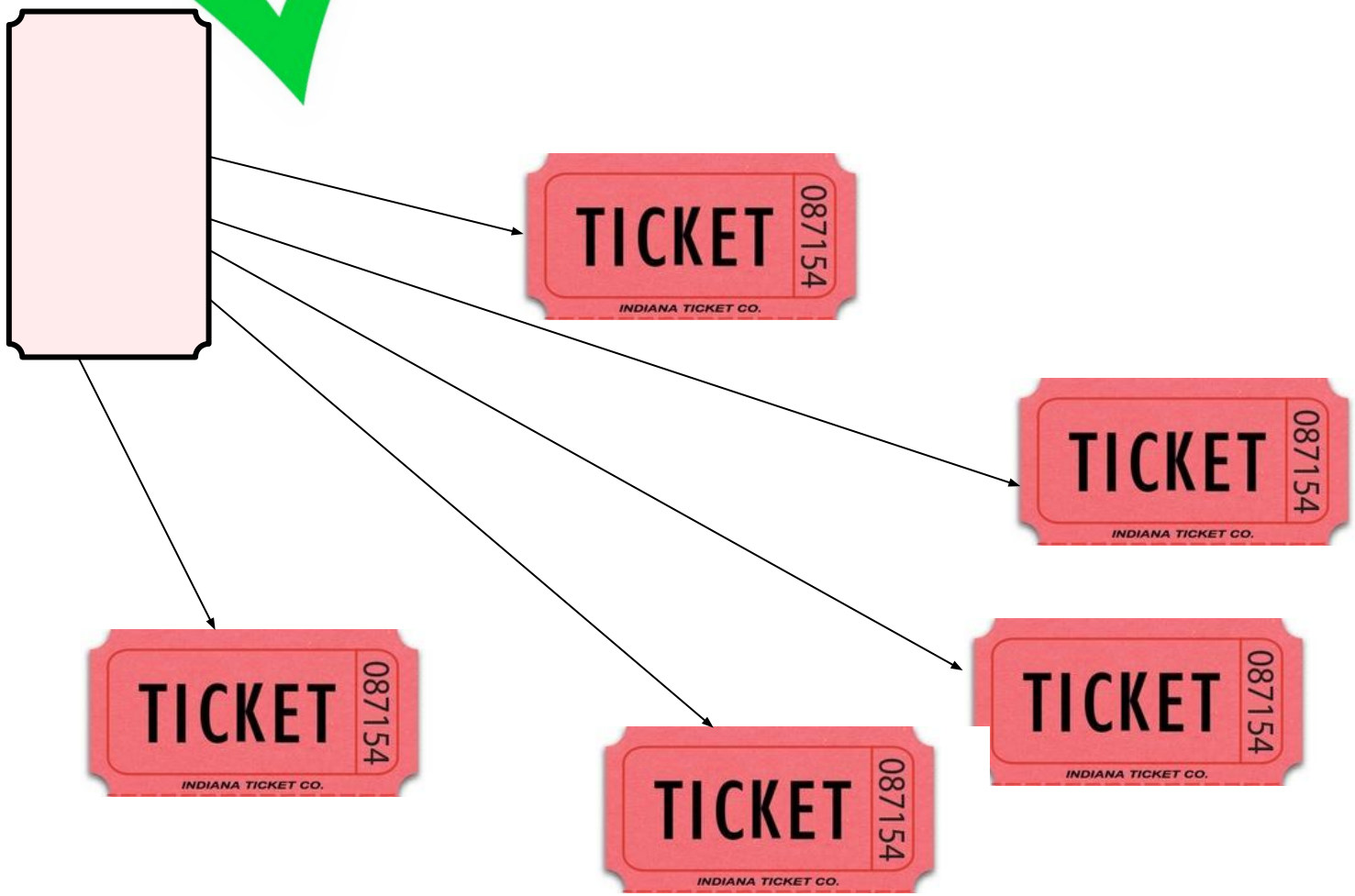
(linear code with no indentation)



depth ≤ 3

❤️

Flat Functions ✓ Are Easy to Break into Pieces



Flat Functions

Deeply ions

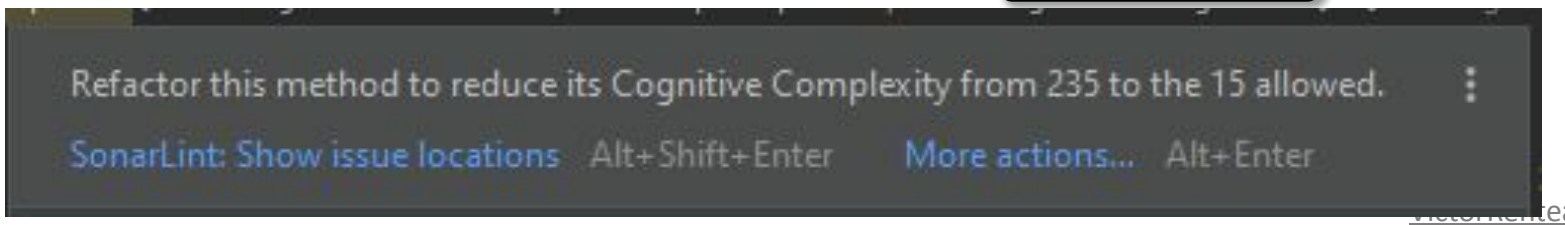
Flatten Code

The Pyramic of Doom

Most Changes occur here

Huge Cyclomatic/Cognitive Complexity

Sonar metric



Flatten Functions

- **`if`**
  - Anemic **`else`** □ flip **`if`** □ Guard Clause:
  - Superficial **`if`** with heavy branches □ Split function:
- **`for`**
  - □ Split Loop □ FP pipeline (eg filter/map...)
  - □ Extract body of for in separate function
- **`switch`**
  - □ keep alone in a function, with one-line / **`case`**
- **`try`** – extra noisy
  - **`catch`** □ Use a top-level exception handler: eg. @RestControllerAdvice
  - **`finally`** □ Close resources with try(x){ } or Loan Pattern (FP)

```
if (param == null) return/throw;
```

```
f(b) {
  if (b) {
    foo 40%
  } else {
    bar 60%
  }
}
```

```
foo() {
  ...
}
```

```
bar() {
  ...
}
```

When in Doubt,

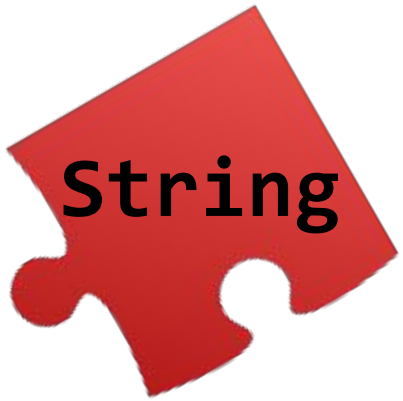
Extract a Method

“A class with too much code is a prime breeding ground for duplicated code, chaos, and death”

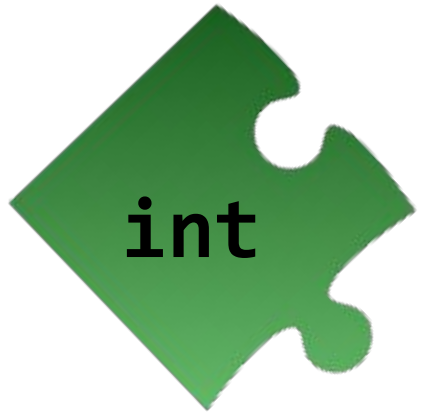
REFACTORING: IMPROVING THE DESIGN OF EXISTING CODE

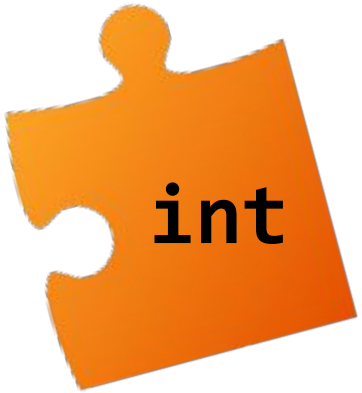
- slide created by a participant to a workshop I lead whose group summed up >300 years of experience

Missing Type



related data elements
moving around together





Missing Type

aka: Data Clumps | Missing Abstraction

Group data in new types

Small Immutable "**Value Objects**" lacking identity (PK)

`Interval(start,end)`

int, int → Interval

{"amount":10, "cur": "EUR"} → Money{amount,currency}

Tuple4<String, Long, Long, Date> → PricedProduct

Map<Long, List<Long>> → List<CustomerOrders>

ArrayGeddon

What fields are there?

😱

IDE Refactoring = unsafe!

js, ts, php, py..

Tangled Tuples

Lightweight structures

Java: @Data/@Value, record

Kotlin: data class

Scala: case class

C#: [readonly] record [struct]

TS: interface, type, or class

Py: class, @dataclass

New Types

Simpler Code

Less Parameters
`in(x, Interval)`

Shrink Entities
`carModel.years:Interval`

If **Object** 👑 (as in OOP)

Host Bits of Behavior
`interval.intersects(..)`

Enforce
Constraints
`start<end`

Util Class or Hellper

Move Logic in Entities

create a Rich Domain Model

❌ `OrderUtil.canReturn(order):bool`

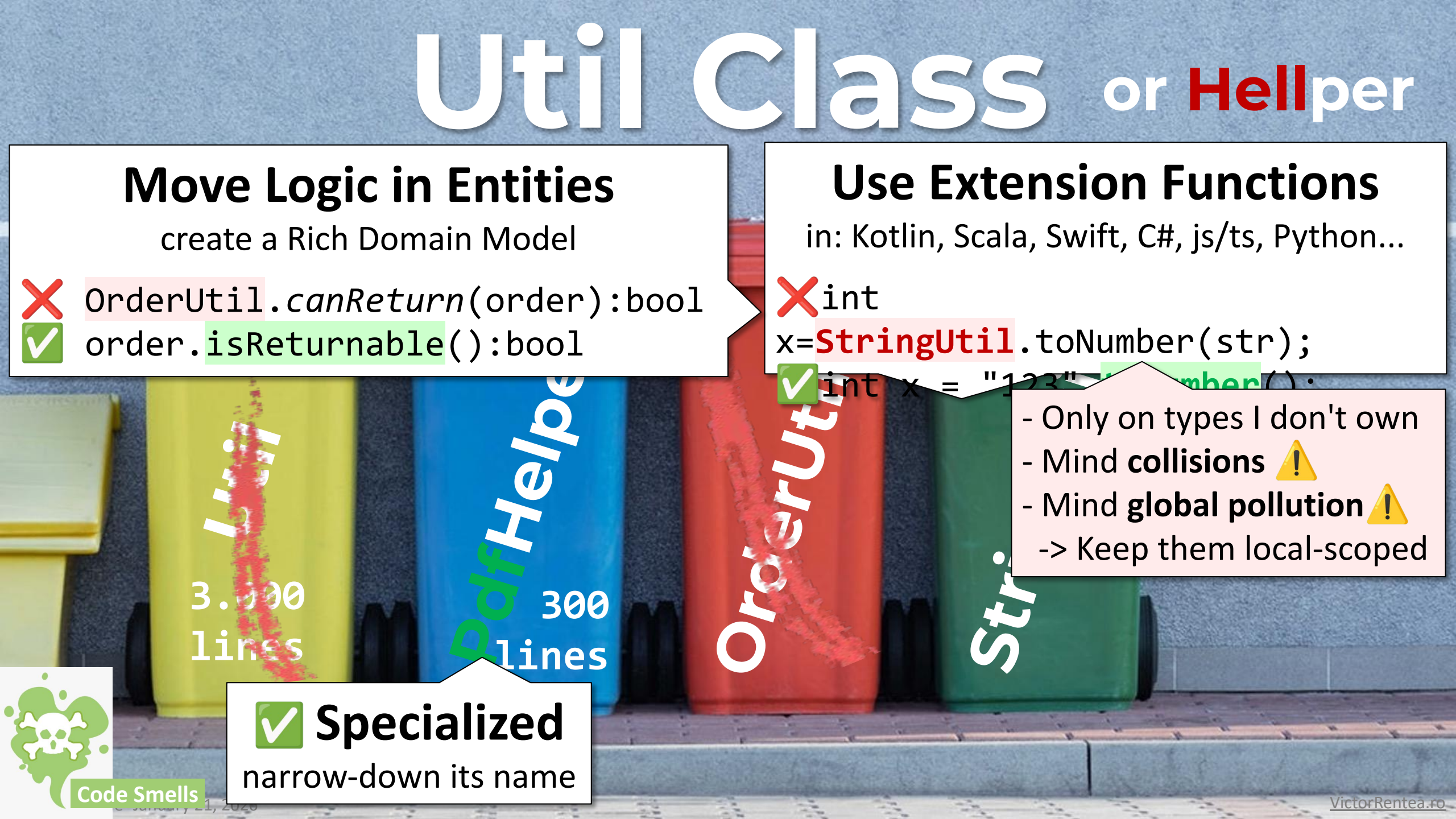
✅ `order.isReturnable():bool`

Use Extension Functions

in: Kotlin, Scala, Swift, C#, js/ts, Python...

❌
```
int x=StringUtil.toNumber(str);
```
✅ `int x = "123"...mber();`

- Only on types I don't own
- Mind **collisions** ⚠️
- Mind **global pollution** ⚠️
  -> Keep them local-scoped



✅ Specialized

narrow-down its name

Code Smells

Discover Value Objects

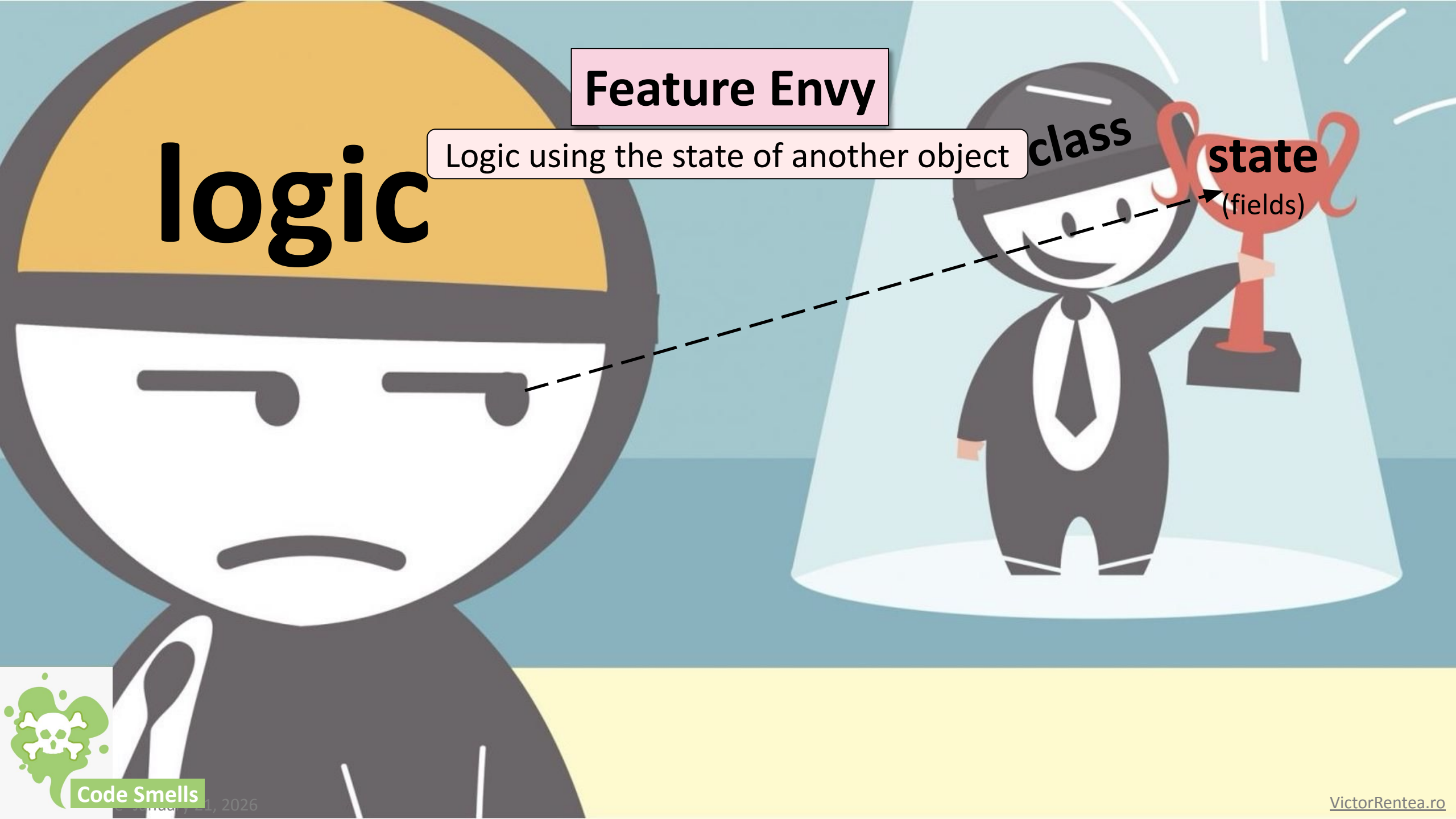
Deeper Model

Group Data in new classes, if it:

- **Moves Together**
  - parameters Parameter Object
- **Changes Together**
  - updatedBy, updatedTime update: UserAction{user, time}
    create: UserAction..
- **Conceptual Whole**
  - moneyAmount, moneyCurrency -> Money{amount, currency}
- **Screen Parts **
  -   Customer{ billingInfo: BillingInfo{a,b,c,d} }

Feature Envy

Logic using the state of another object



Feature Envy

Logic using the state of another object

```
...
int place = player.getPlace() + roll;
if (place ⩾ 12) {
  place -= 12;
}
player.setPlace(place);
...
```

```
player.advance(roll);
```

Keep behavior next to state (OOP)

Data Classes

Anemic type with data, but no behavior

💖**Rich Objects** with logic & constraints can help simplify core logic

@Data-mania
(Lombok in Java)

Methods over **attributes** access
= uniform interface over fields, derived values and logic

Behavior in Data Structures (OOP) 🤔

See Architecture Course for pros/cons, debates□

```
= carModel.getYearInterval().getStart();
```

violates the **Law of DemeTER**

Alternative (a while ago):

```
= carModel.getStartYear();
```

add +1 method:

```
int getStartYear() {
  return yearInterval.getStart();
}
```

NP

```
= carModel.getYearInterval().getStart();
```

violates the ~~Law of Demeter~~

👍 It's ok to depend on ***your*** Domain Model

Middle Man

= indirection without abstraction

Today: AVOID: (unless you are building some framework)

```
= carModel.getStartYear();
```

add +1 method:

Clutters Entities

```
int getStartYear() {
  return yearInterval.getStart();
}
```

Middle Man

aka ***Indirection without abstraction***
= doesn't add more meaning to code, it's just +1 hop

👍

Better **signal/noise ratio**

In Domain Model (Entity) or in a mandated layer:

```
int startYear() {
  return yearInterval.start();
}
```

NOISE

```
= carModel.startYear();
= carModel.yearInterval().start();
```

👎

coupling + noise in clients

Traversing ***your*** Domain Model is ok 👍

it's the backbone of your codebase

Also in a mandated architecture layer: overengineering🤔

one extra hop

Middle Man

simplification

Indirection without abstraction

NOISE

```
int startYear() {
  return yearInterval.start();
}
```

```
public Customer findById(id) {
  return repo.findById(id);
}
```

Inline Method… ⌥⌘N

DON'T REFACTOR WITH GenAI

GenAI Hallucination

(aka TAB-driven Bugs)

```
//     if (user.contact().isPresent()) {
//       email.getCc().add(user.contact().get());
//     }
```

suggestion: `user.email().ifPresent(email.getCc()::add);`

by GitHub Copilot on 2025-01-30

correct: `user.asContact().ifPresent(email.getCc()::add);`

Primitive Obsession

=abuse of **strings** & **numbers** w/o meaning

What's wrong with these lines?

Declaration:

```
void redeemCoupon(Long couponId, Long customerId, String email){
```

Call site:

```
redeemCoupon(dto.cust, dto.coupon, dto.userId);
```

What do these mean? 😨

```
Map<Long, List<String>> customerIdToRegions
```

```
can be: "EMEA","NA","LATA" or "APAC".
```

Fight the Primitive Obsession by creating

Micro-Types

```
void redeemCoupon(Long couponId, Long customerId, String email){
```

Type-safe Semantics

```
redeemCoupon(CouponId coupon, CustomerId customer, Email email)
```

```
class CouponId {
  private final long id;
  +ctor/get/hash/equals
}
```

```
@Value // Lombok
class CustomerId {
  long id;
}
```

```
record Email(String value) {
  String getDomain() {..}
}
```

```
Map<Long, List<String>> map
```

```
Map<CustomerId, Set<Region>> map
```

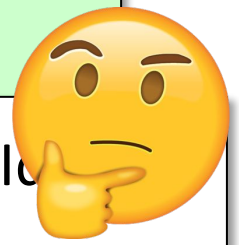
Restrict values

```
enum Region {
   EMEA, NA, LATAM, APAC
} // MUST HAVE!!!!
```

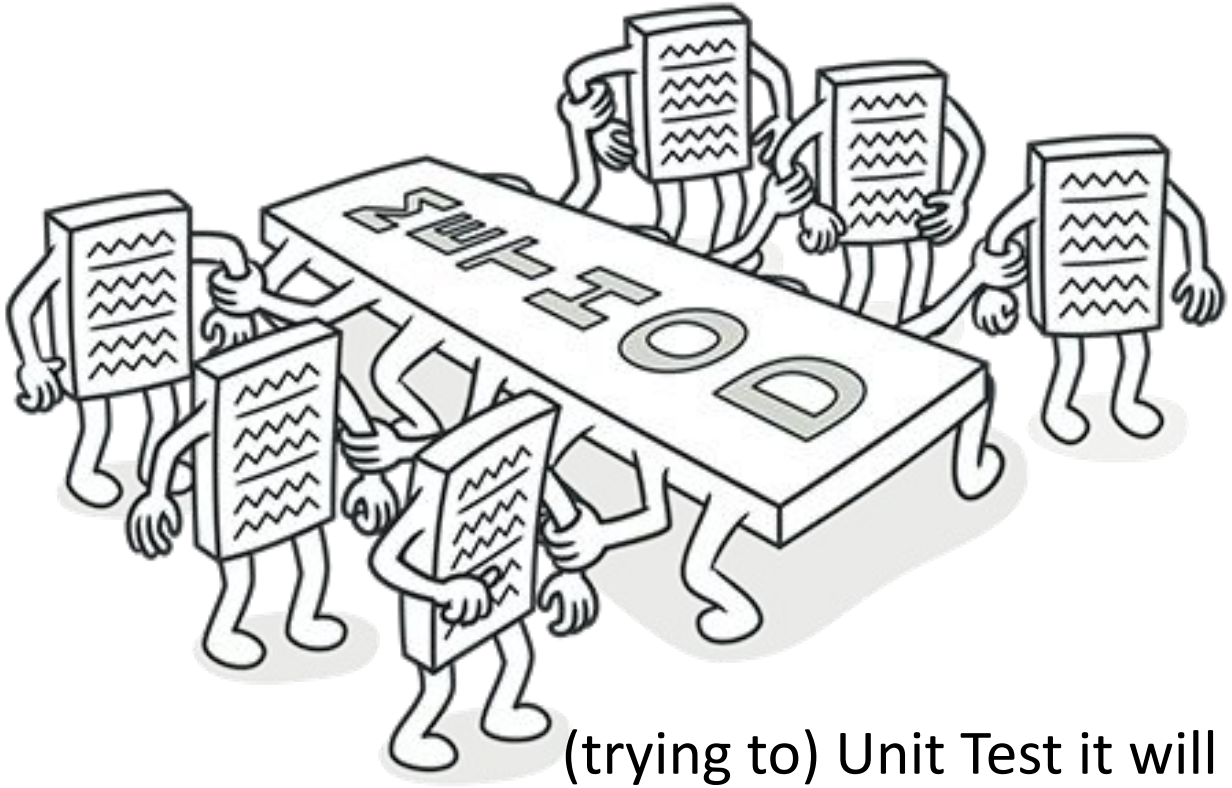
- Use for widespread **Critical IDs**:
  IBAN, SSN, DNI, SwiftCode, OrderId
  especially if having constraints/checksums
- Hard to adopt late in a project
- Can pressure memory

Fix2: recycle in-mem instances: OrderId.*of*(1L) == OrderId.*of*(1L)

Fix2: Project Valhalla value types



Inappropriate Intimacy



(trying to) Unit Test it will point out the problem

Image: https://refactoring.guru/smells/inappropriate-intimacy

Temporary Field

a field computed from other fields

```
private List<Player> players;
private int currentPlayerIndex;
private Player currentPlayer;
               //=players[currentPlayerIndex]

void endTurn() {
  this.currentPlayerIndex ++;
  this.currentPlayer = players[this.index];
}
```

≅ caching 😱

☹ confusing
☹ inconsistency

```
Player currentPlayer() {
  return players[this.index];
}
```

It *should* work faster...

Did you measure ?

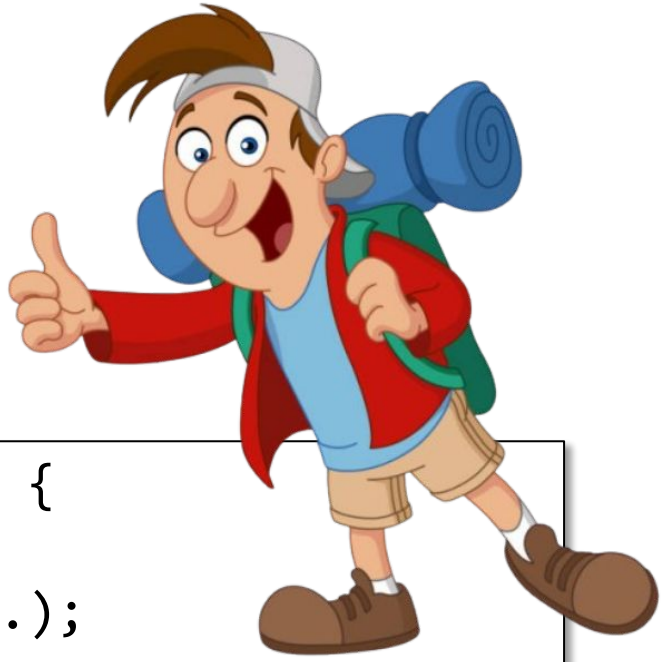
```
  // later...
  this.currentPlayer().f();
```

precomputed field

🙄

Temporary Field

abusing an object to move some data



```
flow(id) {
  customer=repo.byId(id)
  evaluateRisk(customer);
  export(customer);
}
```

```
evaluateRisk(Customer customer) {
  ...
  customer.risk = computeRisk(..);
          // added +1 field ❌
}

export(Customer customer) {
  doStuff(customer.risk); // I need the risk here
}
```

Adding a new field is a code smell if:

1. field does NOT **belong** to customer
2. field is **null** in all other flows

Functional Programming

victorrentea.ro

Functional Programming

Functional Programming

in our daily life

1) Functions as first-class citizens

Syntax sugar to ease passing behavior around

```
f(x → ...)
f(new Consumer<X>() { void accept(X x) {...} })
```

2) Simpler to work with collections

declaratively, without mutating them (less need to ~~add, remove, put..~~)

```
newList =
list.stream().filter(...).map(...).toList();
```

Functional Programming

design philosophy

= Programming without side effects

Functions should be **pure**

- No **Side Effects** (changes to outside state)
- Same **input** → Same **output**

Data structures should be **deep-immutable** (read-only after instantiation)

In OOP we encapsulate the moving parts;
In FP we eliminate the moving parts. - Michael Feather

Functional Programming – random ideas

functions are **first-class citizens (**not just attached to some parent object)

syntax sugar: `→, ⇒, {it`

immutable objects (deep graph, not just read-only attributes)

transforming collections (instead of changing them)

chaining .filter.map.filter.flatMap (read: Functors and Monads)

declarative code rather than an imperative for adding/removing

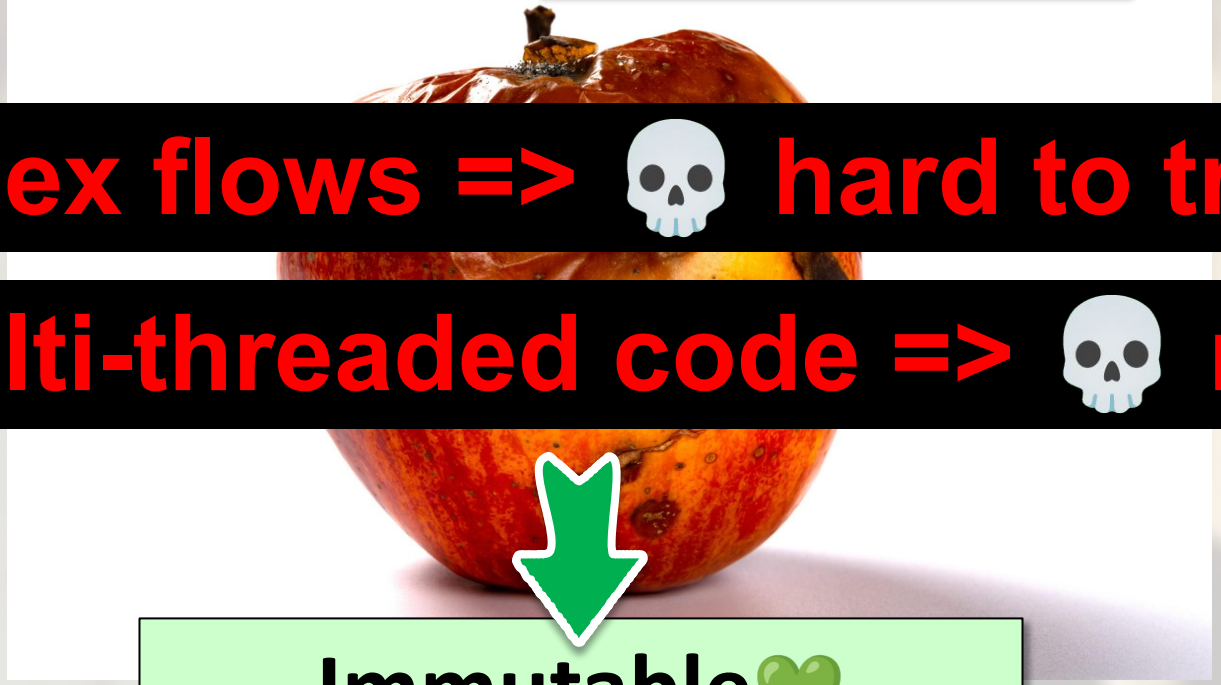
pure functions without side effects that compute a deterministic result

move operations to big data (Spark/Hadoop) = map/reduce pattern

easy to test (with no mocks): params -> result -> assert

is a must-have for parallel processing: no mutation = no race bugs

Mutable Data



in complex flows => 💀 hard to track changes

in multi-threaded code => 💀 race bugs

Immutable💚 Objects

⚠️ Not on ORM @Entity:

Cumbersome

If designed too large:

Fragmented Immutable

If cloned in a loop:

Memory Churn

https://gist.github.com/victorrentea/67f923dca33c7e737c867f60c727c604

2.1 MB

417.33 MB

```
val list10K = (1..10000).toList()
K.fold(listOf<Int>()
    list, e -> list + e }
```

immutab
clone+
1

Code Smells

Cloning collections in a loop can lead to:

Memory Churn

mutable

2.1 MB

417.33 MB

```
val list10K = (1..10000).toList()
list10K.fold(listOf<Int>())
    { list, e -> list + e }
```

immutable list

clone+1larger

https://gist.github.com/victorrentea/67f923dca33c7e737c867f60c727c604

Fragmented Immutable >> Builder

```
x = new X(1, 2, 3, false, -1, null, "X"); // lose track which is which
```

☺ **Builder can name the constructor parameters**

```
x = X.newBuilder().x(1).y(2)...build();
```

```
x = X(a=1,b=2,c=3,...)
```

Other languages: Kotlin,Scala..

☹ **.toBuilder() allows arbitrary changes**

```
var copy = old.toBuilder().x(1).y(2)...build();
```

```
data.copy(a=1, b=2)
```

☺ **Semantic methods**

```
builder.withXY(old.x()+1, old.y()+1)... // hand-crafted builder
var copy = old.moveBy(1, 1); // or semantic "with-er"
```

☺ **Discover new types**

```
var copy = old.moveTo(new Point(2, 3));
```

Confused Variable

This code lies to us!

Recycled variable

```
1 double avg = 0;
2 for (Employee e : employees) {
3   avg += e.getSalary();
4 }
  Log.debug("Average:" + avg);
  avg = avg / employees.size();
```

sum

sum

sum

Here avg means "sum"

Define a new **var**

≠ 0

Don't reassign local variables.

IntelliJ can automatically add **final** everywhere possible: link

final
const
val

IDE warns you 👇

```
long averageEmpAge = emplo
```

Reassigned local variable

Error Prone can fail compilation unless it's annotated with @Var

Confused Variable

Different meaning throughout its scop

```
handle(Form form) {
  form = validate(form);
  form = normalize(form);
  ...
  form = process(form);
  audit(form);
}
```

immutable 😏

Reordering lines produces bugs

```
handle(Form form) {
  var valid = validate(form);
  var normal = normalize(valid);
  ...
  var withId = process(normal);
  audit(withId);
}
```

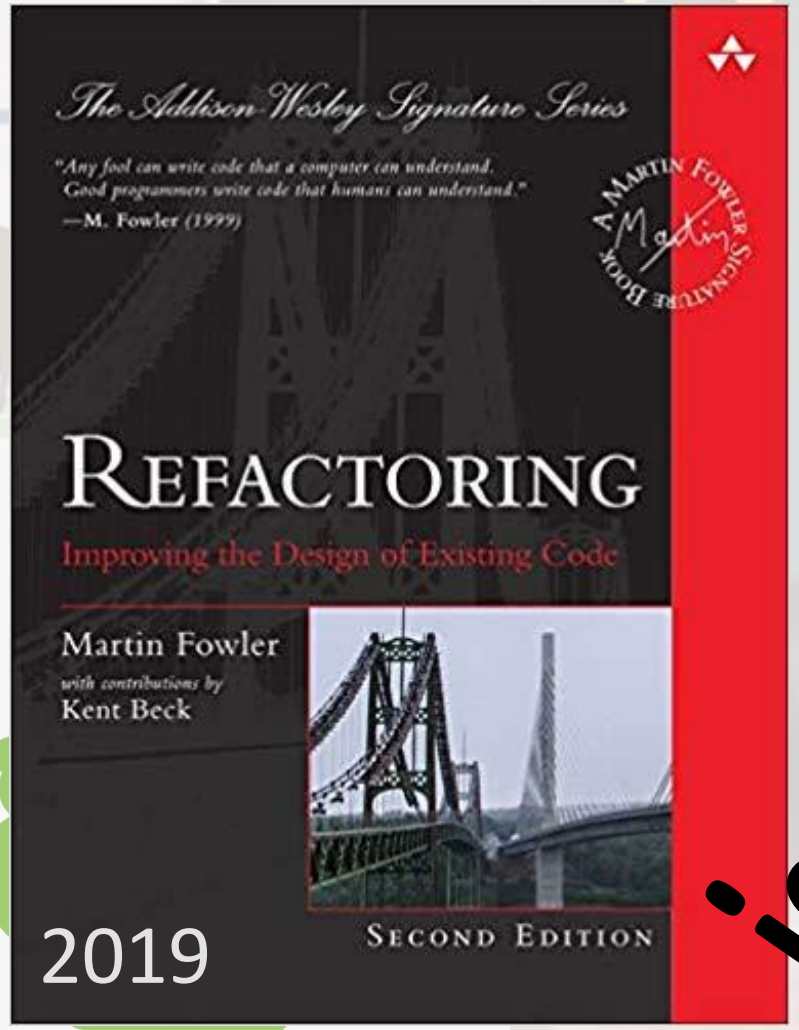
Don't reassign local variables

Temporal Coupling

the order of operations matters for mysterious reasons


for
while
2×2=
3×3=
4×4=
5×5=
6×6=36
7×7=
EUROPE

in Chap 3: Bad Smells in Code



Loop

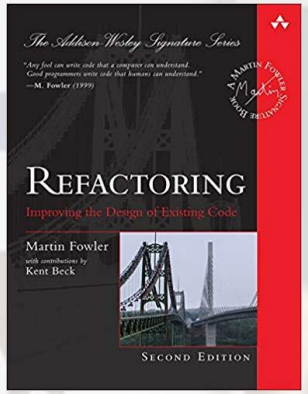
for

while

is a

Code Smells

- A large loop might break SRP
- A FP pipeline is simpler: .filter, .map…



Complex Loop

doing unrelated things

```
for (e : list) {


  results.add(...)
  total += ...
  sideEffect(e);
}
```

Split Loop

```
f(...){
...}
```

Accumulator Loop

gathering data via a loop

```
var results = new ArrayList();
for (e : list) {
  results.add( f(... )
}                    )
```

```
var results =
list.stream()...collect(...)
```

```
int total = 0;
for (e : list) {total += ...}
```

```
var total = list.stream()...sum()
```

```
for (e : list) sideEffect(e);
```

```
list.forEach(e -> sideEffect(e));
```

⚠ NO performance hit (see next slide)

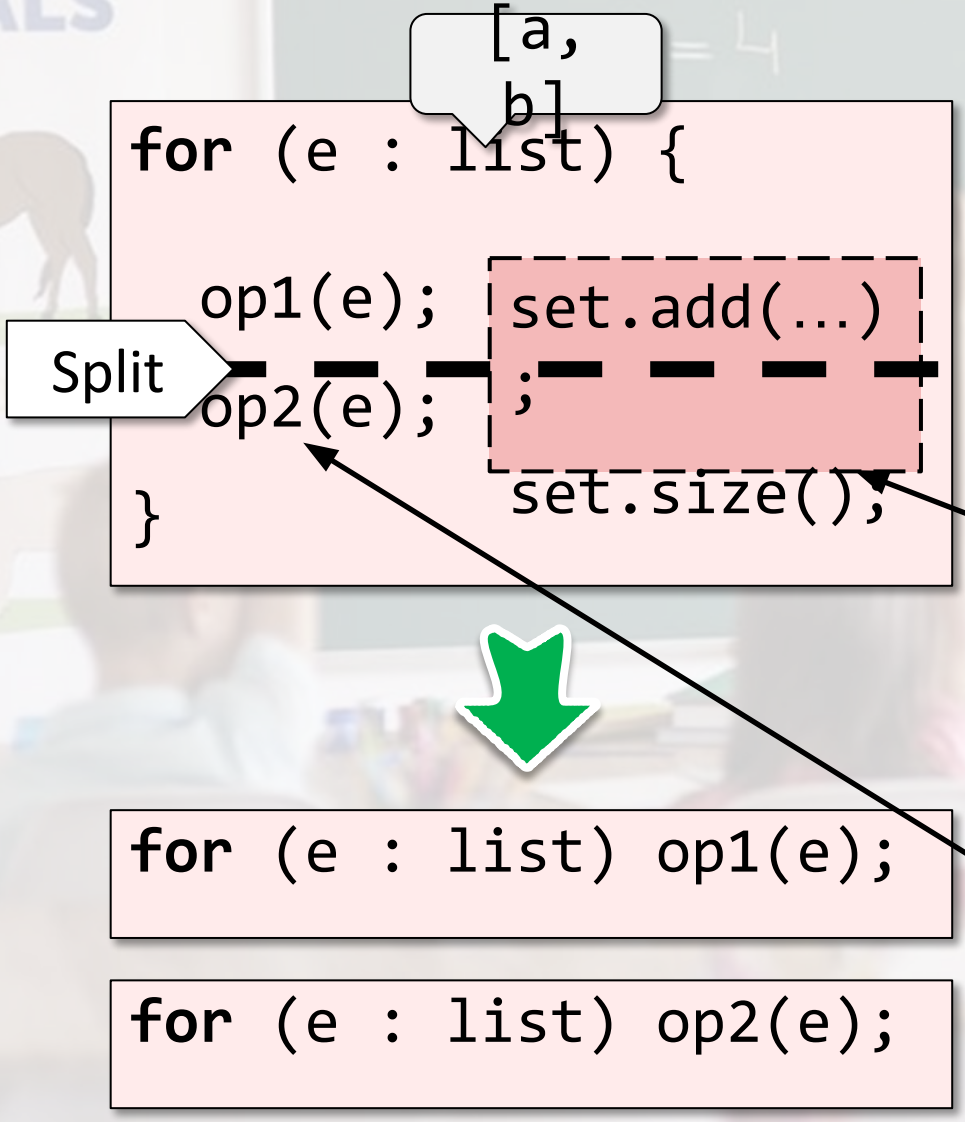
⚠ Order of side-effects CAN

Myth: One extra for will hurt performance

- You loop on a **large** collection of **objects** you have in **memory**
- The **loop does 2+ unrelated things** => split the for by SRP?
- Will it run slower? 😬
- Where did the large collection come from?
  - a) Remote machine: DB query, API call, S3 file ... <⭐usually
    - Overhead of +1 for = neglectable compared to time to download dataset
  - b) File on local disk (rare) -> measure
  - c) Memory (~cache)
    - In 300+ projects in 150 companies, I only met **two projects** that proved a production perf impact:

      In the hot path of HFT stock trading systems having response times of 20-40 µs 99%th (20 x 10^{-6}s)
      Note: in those flows you were not allowed to allocate any memory
      Note: their automatic test pipeline ran benchmarks on each commit

Victor's Benchmarks project: github.com/victorrentea/performance-jmh



Split Loop can cause Bugs

The order of operations changes ⚠️

before: op1(a)-op2(a)-op1(b)-op2(b)

after: op1(a)-op1(b)-op2(a)-op2(b)

Also, mind the **flow control breakers** ⚠️

`break`, `continue`, `return`, `throw`

Mutant Pipeline

avoidable **side-effects** in a FP pipe

compilation fails

lambdas cannot change variables on thread stack (Java Language limitation)

```
// TODO: ∑total active orders price
int total = 0;
orders.stream()
   .filter(order → order.isActive())
   .forEach(order → {
       total += order.getPrice();
   });
```

Order::isActive 😎

Hack

move mutable state on heap

```
atomic.incrementAndGet(pri
total[0] += price;
ref.total += price;
this.total += price;
```

Correct

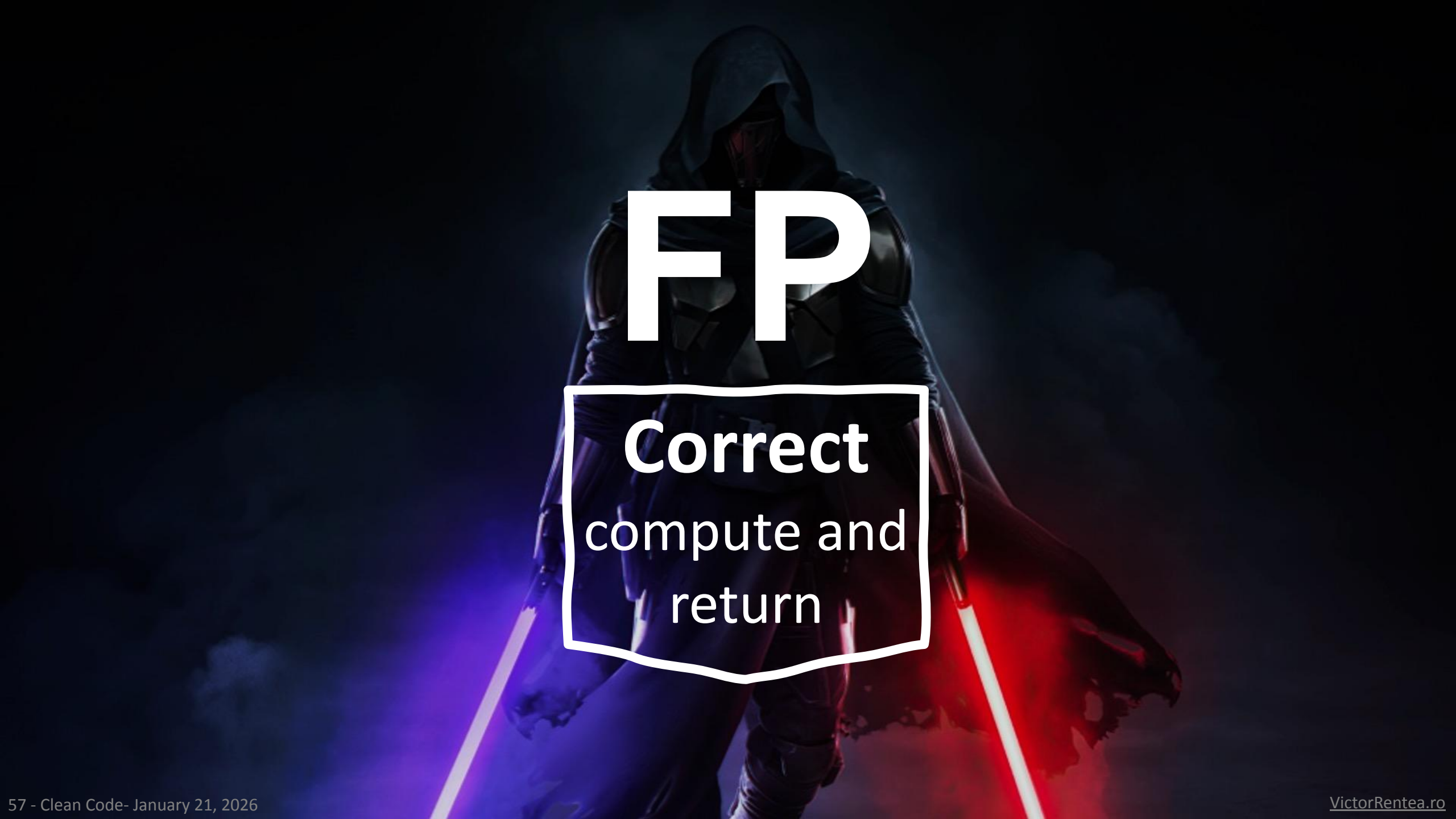
compute and return

```
sum += order.getTotalPrice();
```

Avoid mutation using Stream API 'sum()' operation

```
int total = orders.stream()
    .filter(Order::isActive)
    .mapToInt(Order::getPrice)
    .sum();✅
```

```
    .reduce(0, Integer::sum); ❌
```


FP
Correct
compute and
return

Mutant Pipeline

```
stream.forEach(e → ...):void
optional.ifPresent(e → ...):void
```

❌ **Code smells** when used to accumulate data:

❌ `.forEach(e → map.put(e.id(), e));` ☐ `.collect(toMap()):Map`

❌ `.forEach(e → atomic.increment(e));` ☐ `.sum():int`

❌ `.ifPresent(e → list.add(e));` ☐ `.flatMap(Optional::stream)`

✅ **OK** for external side effects:

✅ `.forEach(m → sender.send(m));` ☐ `sender.sendAll(list)`

✅ `.forEach(e → e.setStartedAt(now()));`

✅ `.ifPresent(e → repo.save(e));` ☐ `repo::save`

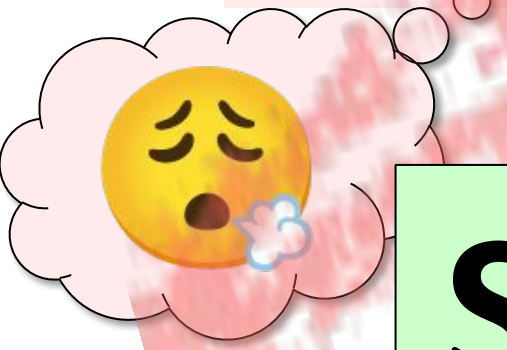
Key Point

Avoid side-effects

Meanwhile:

```
.peek(list::add)
```

```
.map(e -> {list.add(e); return e;})
```



Side-effects are Evil

(but often necessary)

Functional Programming

<<**Misuse** and **Abuse**>>

Functional Chainsa

```
List<Product> streamWreck(List<Order> orders) {
  return orders.stream()
    .filter(o → o.getCreationDate().isAfter(now().minusYears(1)))
    .flatMap(o → o.getOrderLines().stream())
    .collect(groupingBy(OrderLine::getProduct,
      summingInt(OrderLine::getItemCount)))
    .entrySet()
    .stream()
    .filter(e → e.getValue() ⩾ 10)
    .map(Entry::getKey)
    .filter(p → !p.isDeleted())
    .filter(p → !productRepo.findByHiddenTrue().contains(p))
    .collect(toList());
```

Order::isRecent

var frequentProducts =

Complexity obscures a performance issue

extract explanatory **variables** and **functions** after every ~ 3-4 operators ("dots")

Expression Functions 💖

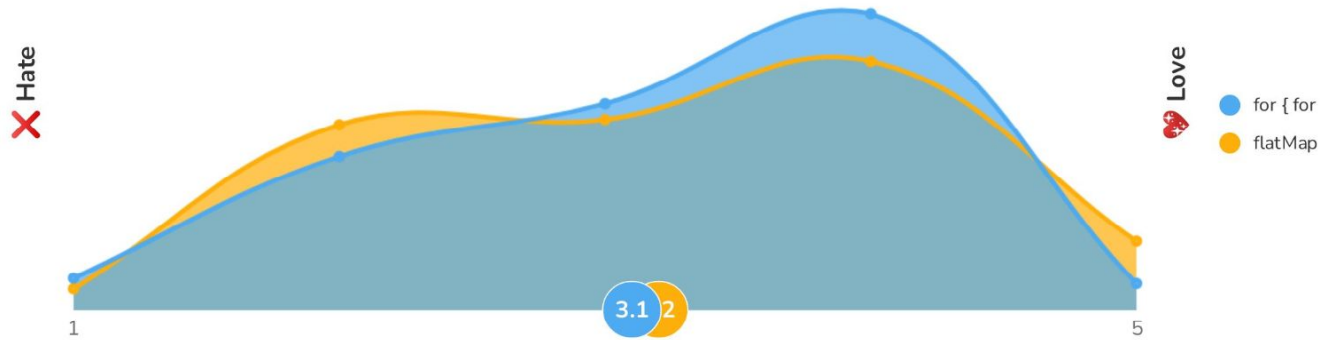
entire body is a single expression

```
fun x(...) = <expr> .kt
x = (...) => <expr> .ts
```

1. Cross-Product

for { for { add

```
var results = new ArrayList<Pair>();
for (String s1 : list1) {
  for (String s2 : list2) {
    results.add(new Pair(s1, s2));
  }
}
return results;
```



flatMap

```
return list1.stream()
    .flatMap( String s1 → list2.stream()
        .map( String s2 → new Pair(s1, s2)))
    .toList();
```

☠ Reactive Programming

☠

```
@Bean
public Function<Flux<LikeEvent>, Flux<LikedPosts>> onLikeEvent() {
  return flux → flux
    .doOnNext(event → postLikes.put(event.postId(),event.likes()))
    .doOnNext(event → eventSink.tryEmitNext(event))
    .map(LikeEvent::postId)
    .buffer(ofSeconds(1))
    .flatMap(ids → postRepo.findAllById(ids)
          .map(Post::title)
          .collectList())
    .map(LikedPosts::new)
    .onErrorContinue((x,e)→log.error(STR."Ignore \{x} for \{e}"))
    .doOnNext(message → log.info("Sending: " + message));
```

Mandatory Chaining
to ensure exactly one subscriber

Goodbye Variables
of type Publisher (same reason)

Never Throw
(only return)

Master 100+ operators
(forget everything you knew!)

WOW!!!

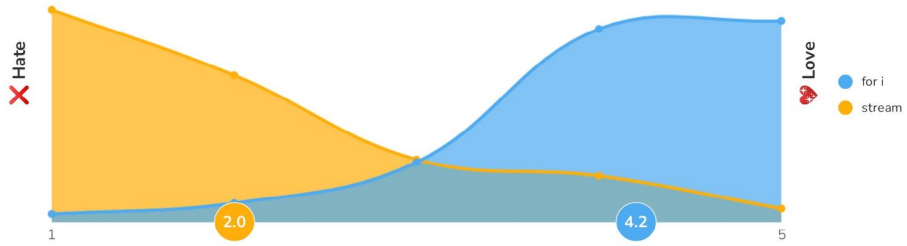
* there are some coding survival best practices, that I teach in my Reactive Programming class.

```
// Reactive Programming with rxJava/Reactor (2017)
public Mono<ABC> abc(int id) {
    return api.a(id) // Mono from webClient.get()...
          .flatMap(a -> api.b(a).flatMap(
                  b -> api.c(a, b).map(
                          c -> new ABC(a, b, c))));
}
```

```
// Virtual Threads with Java 21+
public ABC abc(int id) {
  var a = api.a(id);
  var b = api.b(a);
  var c = api.c(a, b);
  return new ABC(a, b, c);
}
```

2. Zip with Index

2. Zip With Index



a) for i

```
var words : String[] = PATTERN.split(line);
for (int i = 0; i < words.length; i++) {
  String word = words[i];
  if (word.length() > 13) {
    return i + ":" + word;
  }
}
throw new NoSuchElementException();
```

b) stream

```
record Indexed<T>(int index, T word) {}
AtomicInteger atomicInteger = new AtomicInteger();
return PATTERN.splitAsStream(line)
    .map( String word → new Indexed<>(atomicInteger.getAndIncrement(), word)) // (a)
    //.map(new ZipWithIndex<>()) // (b) encapsulated in a stateful transformation
    //.gather(zipWithIndex()) // (c) parallelStream-safe in Java 24+
    .filter( Indexed<String> indexed → indexed.word.length() > 13)
    .map( Indexed<String> indexed → indexed.index + ":" + indexed.word)
    .findFirst()
    .orElseThrow(NoSuchElementException::new);
```

3. Add One More

loop + list.add

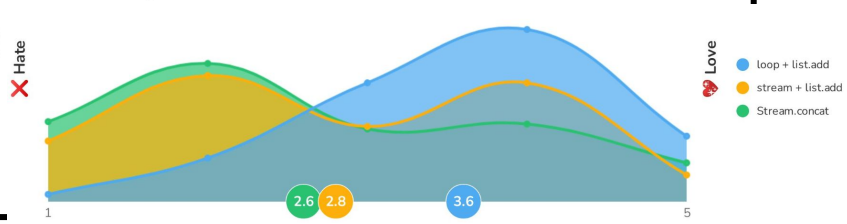
```
List<LocalDate> list = new ArrayList<>();
for (Order order : orders) {
  list.add(order.creationDate());
}
list.add(parse( text: "2025-03-03"));
return list;
```

stream + list.add

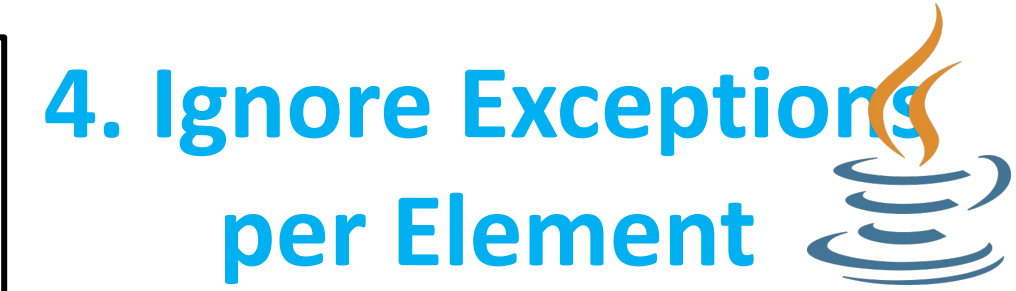
```
List<LocalDate> dates = orders.stream()
    .map(Order::creationDate)
    .collect(toList()); // returns mutable ArrayList
dates.add(parse( text: "2025-03-03"));
return dates;
```

Stream.concat

```
return Stream.concat(
    orders.stream().map(Order::creationDate),
    Stream.of(parse( text: "2025-03-03"))
).toList();
```

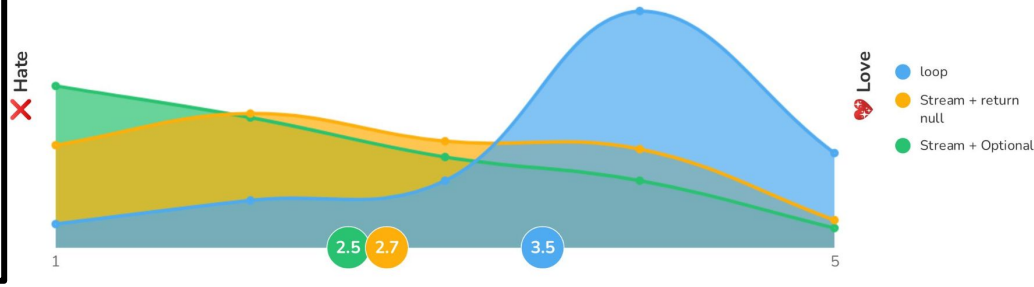


4. Ignore Exceptions per Element



loop

```
String line = reader.readLine();
while (line ≠ null) {
  try {
    transfers.add(Transfer.of(line));
  } catch (NumberFormatException _) {
  }
  line = reader.readLine();
}
```



Stream + return null

```
List<Transfer> transfers = Files.lines(path)
    .map( String csvLine → {
      try {
        return Transfer.of(csvLine);
      } catch (NumberFormatException _) {
        return null;
      }
    })
    .filter(Predicate.not(Objects::isNull))
    .toList();
```

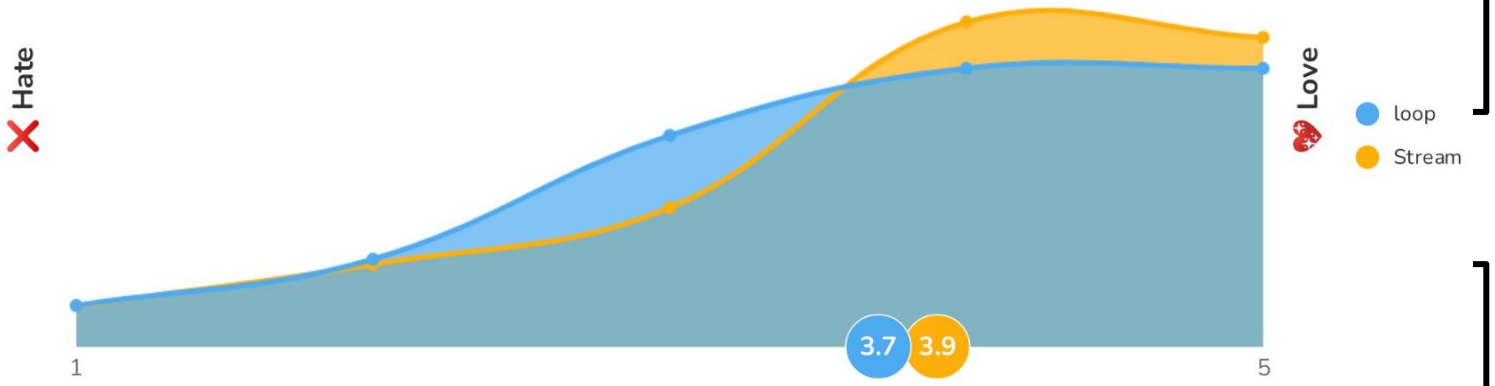
Stream + Optional

```
  return Optional.of(Transfer.of(csvLine));

  return Optional.<Transfer>empty();


.flatMap(Optional::stream)
```

5. Aggregate values by key

loop

```
Map<User, Double> incomePerUser = new HashMap<>();
for (Transfer transfer : transfers) {
  incomePerUser.merge(transfer.recipient(), transfer.amount(), Double::sum);
}
```



Stream

```
Map<User, Double> incomePerUser = transfers.stream()
    .collect(groupingBy(Transfer::recipient, summingDouble(Transfer::amount)));
```

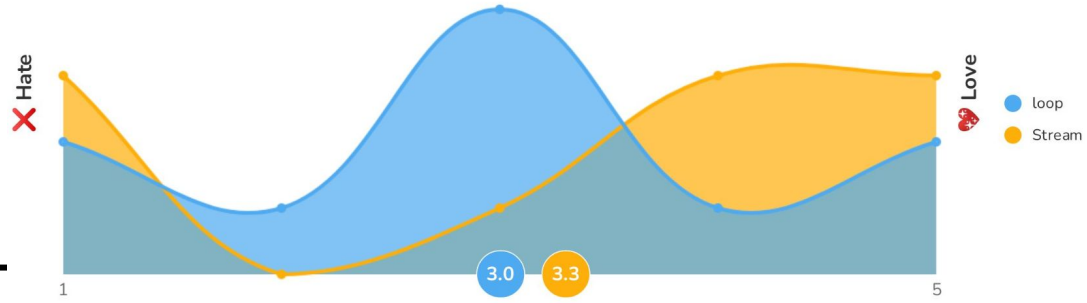
6. Group map keys by values

a) loop

```
TreeMap<Double, List<User>> usersPerIncome = new TreeMap<>();
for (Map.Entry<User, Double> entry : incomePerUser.entrySet()) {
  List<User> users = usersPerIncome.computeIfAbsent(entry.getValue(), Double k → new ArrayList<>());
  users.add(entry.getKey());
}
```

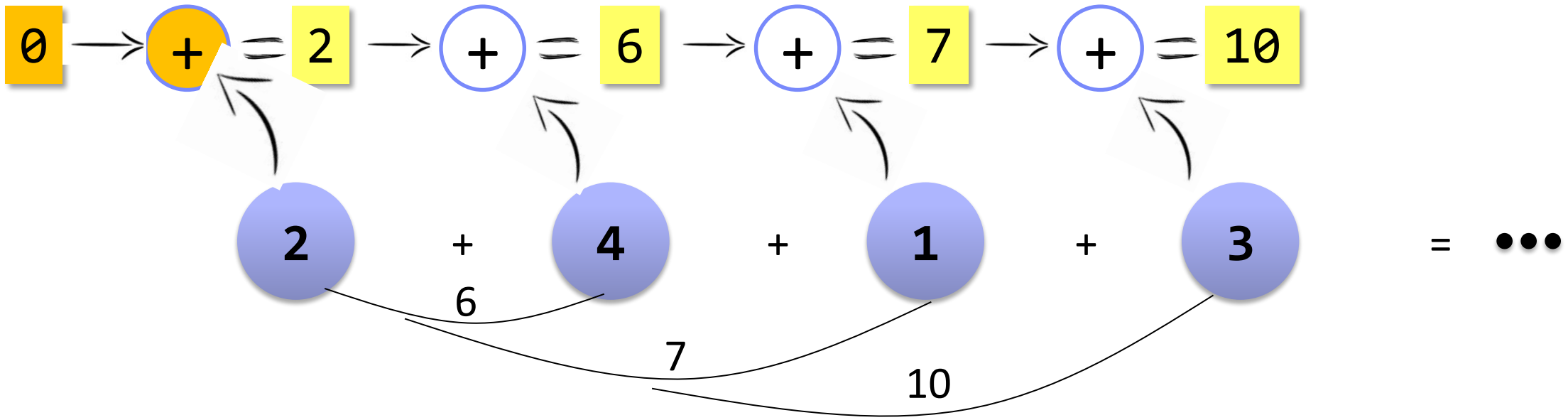
b) Stream

```
TreeMap<Double, List<User>> usersPerIncome = incomePerUser.entrySet().stream()
        .collect(Collectors.groupingBy(
            Map.Entry::getValue,
            TreeMap::new,
            Collectors.mapping(Map.Entry::getKey,
                Collectors.toList())));
```



Folding

```
.reduce(0, +)
```



Reduce Abuse

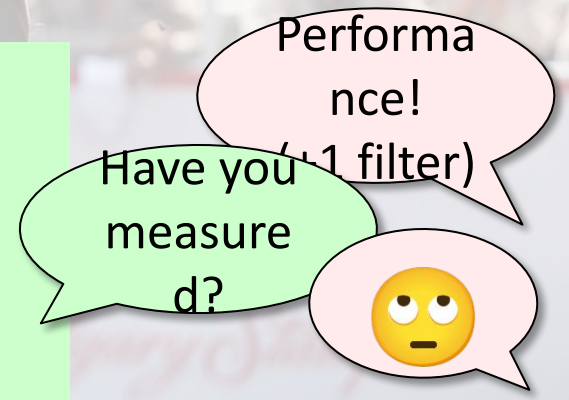
Complex fold expression

```
return list.reduce((prev, e) =>                    // TypeScript
  new Dec(prev?.price || 0).gte(e.price || 0) ? prev : e, undefined);
```

optional chaining | default | ≥ | ternary operator

✅ Simplify the reduce expression

```
const maxByPrice = (a, b) =>
   new Dec(a.price).gte(b.price) ? a : b;

return list.filter(order => order.price)
   .reduce(maxByPrice);
```



Reduce Abuse

```
return orders.stream().reduce(null, (previous, order) ->
    order.orderLines().stream().anyMatch(line -> line.product().isPremium()) &&
    (previous == null || order.creationDate().isAfter(prev.creationDate()))
    ? order : previous
);
```

❌ Instead of a complex .reduce expression 👆:

✅ **Transform the collection** with filter, map, ...

✅ **Use collectors:** toList, toMap, sum, max, average, anyMatch, findFirst...

```
return orders.stream()
    .filter(Order::hasPremiumProduct)
    .max(comparing(Order::creationDate))
    .orElse(null);
```

Calculating Multiple Results in One Pass

```
TotalBalance balance = transactionRepo.streamAll() // fetch as Stream<Tx>
 .reduce(
    new TotalBalance(0, 0, 0),
    (agg, tx) -> new TotalBalance( // +malloc
       agg.totalCredit() + tx.credit(),
       agg.totalDebit() + tx.debit(),
       agg.count() + 1),
    (agg1, agg2) -> new TotalBalance( // +malloc
       agg1.totalCredit() + agg2.totalCredit(),
       agg1.totalDebit()  + agg2.totalDebit(),
       agg1.count()       + agg2.count())
);
```

```
+ OOP =
.reduce(
   TotalBalance.empty(),
   TotalBalance::addToTotal,
   TotalBalance::merge)
```

Alternatives:

a) small collection => 3 x list.stream()...sum();

b) large collection => 💖 `SELECT SUM(T.CREDIT), SUM(T.DEBIT), COUNT(T.ID)`

Credit to Emanuel Trandafir

.reduce as a Last Resort

- **Unusual accumulation**
  - .reduce(BigDecimal.*ZERO*, BigDecimal::add);
  - maximum average over a sliding window ± reactive flows
- **Aggregate multiple results** in one pass over **data streamed on-the-fly** from SQL/Spark
  - ⚠️ If iterating over an in-memory collection, repeating the iteration it won't hurt
  - ⚠️ Much faster if computed directly in DB
- In **JavaScript/TypeScript** that lack collectors:
  - but kept simple:arr.reduce(0,(a,b=>a+b)
  - **Performance** (only if benchmarked⚠️) □ Go raw (for/while) or SQL

Double-Edged Return

Returning result OR technical errors

Use to:
- Return the **business** outcome
- Collect many errors in one pass

```
Long boring(Data data) {
  String r = process(data);
  log.info("X");
  saveToDb(r);
  audit(r);
  return r.id();
}
```

throws checked

```
String process(Data data) {
  if (data.name().isBlank())
    throw new Ex("...");
  ...
  return result;
}
```

```
Try<Long> horrorFP(Data data) {
  return process(data) // FP chain
    .onSuccess(e -> log.info("X"))
    .flatMap(repo::save)
    .andThen(id -> audit(data));
}
```

pollutes caller

```
if (result.failed())
  return failed(..);
```

```
Try<String> process(Data data) {
  if (data.name().isBlank())
    return Try.failure(new
Ex("..."));
  ...
  re                    es
}
```

nostalgic 🥺 for Scala, Kotlin, Go

"In FP don't throw"

Optional Use / Abuse

✅ **Return type** of methods, instead of returning null

- Especially methods used in complex logic
- Including getters of core structures, if not 90% optional ☺️

❌ **Optional<collection>** □ Use an empty collection

- Collection variables should never be left null

❌ **Optional parameter** □ Split the function

- f(a,b,c, opt<d>); □ f(a,b,c); optD.ifPresent(d->g(d));

❌ **Optional field** □ Split the structure if they stick together

- Tolerate null inside the class boundary.
- Use optional for fields in **records** or when mandated by some frameworks [1,2]

❌ **Fatal empty()** – when all callers (if known!) do .orElseThrow(..) ?

- □ throw from inside the method could be cleaner: repo.findByIdOrThrow(13L);

Proper use of Optional - answer by Brian Goetz the Java Language Architect

NEW

Function Factory Frenzy

```
<T> Consumer<T> uncheck(ThrowingConsumer<T> f) {
    return x -> {
        try {
            f.accept(x);
        } catch (Throwable e) {
            throw new RuntimeExc
        }
    }; // what does this do ?
}
```

returning behavior is hard:

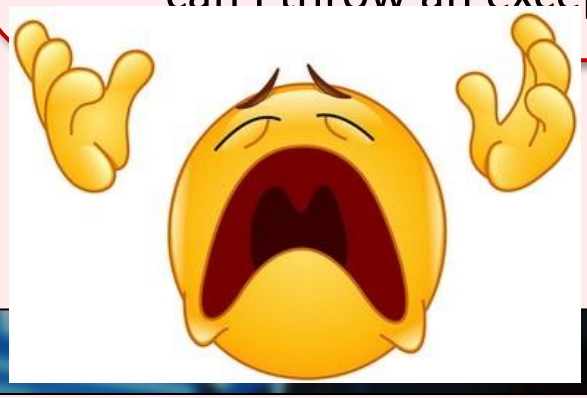
will it run?

how many times?

in what thread(s)?

in what transaction?

can I throw an exception?



```
  .forEach(uncheck(fileWriter::write)); // caller
```

function returning predicate

```
private Optional<OfferInfo> getOfferInfoFromRequest( CalculateTaxesRestRequest restRequest,  1 usage
                                                     FlightOfferTaxInfo offerTaxInfo )
{
  return restRequest.getFlightOfferInfos().stream()
    .filter( isFlightMatching( offerTaxInfo ) )
    .flatMap( FlightOfferInfo i -> i.getOffers().stream() )
    .filter( isOfferMatching( offerTaxInfo ) )
    .findFirst();
}

private Predicate<CalculateTaxesRestRequest.FlightOfferInfo> isFlightMatching( FlightOfferTaxInfo offerTaxInfo )  1 usage
{
  return FlightOfferInfo i -> i.getOrigin().equals( offerTaxInfo.getOrigin() )
    && i.getDestination().equals( offerTaxInfo.getDestination() )
    && i.getFlightNumber() == offerTaxInfo.getFlightNumber()
    && i.getTravelDate().equals( offerTaxInfo.getTravelDate() );
}
```

Less Mocks, More Functions

https://levelup.gitconnected.com/less-mocks-more-functions-860aac67d4a7

Functional Programming Anti-Patterns

1. **Mutable Data**
2. **Fragmented Immutable**, toBuilder()
3. **Mutant Pipeline** – avoid side-effects in a FP pipeline
4. **Functional Chainsaw** (Massacre)
5. **Reduce Rodeo** – highly complex reduce expression
6. **Double-Edged Return** – returning a technical error
7. **Optional Abuse**
8. **Function Factory Frenzy**


Dead ☠ Code
Ever met?
What's kind?

Dead Code

Not Referenced

A parameter, variable, method → Delete

= Negligence / Lack of IDE? 🚩⚠️

Unreachable Code

How to find code never called in production?

- Monitor API calls in prod
- git blame: if recent ☝️ > **Ask the author NOW**!
  (6 month later: *It works? Don't touch it!™*)
- `log.warn("Delete this code after Sep'24 if this message was not logger over the last 13 months")`

```
if (impossible)
{
  lots of code
  lots of code
}
```

Delete it 🥳🍾

Dead ☠️ Code

// Commented Code

```
// KEEP for BF
// < 15-50 lines of code... >
```

in dozens of places

- `if (featureFlags.bf) {`
- Move it to a "RIP branch"

Code Not Referenced

= grayed out by most IDEs

Local Variables

Public methods/params (⚠️ lib clients?)

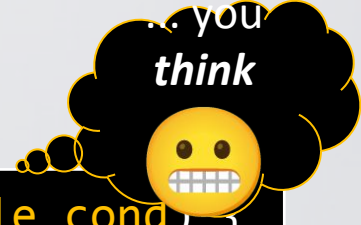
Private Method (⚠️ via reflection?)

Code only used by tests 🥺🥺🥺

Code Smells

Unreachable Code

```
if (impossible_cond) {
  20+ lines of code
}
```



insert here

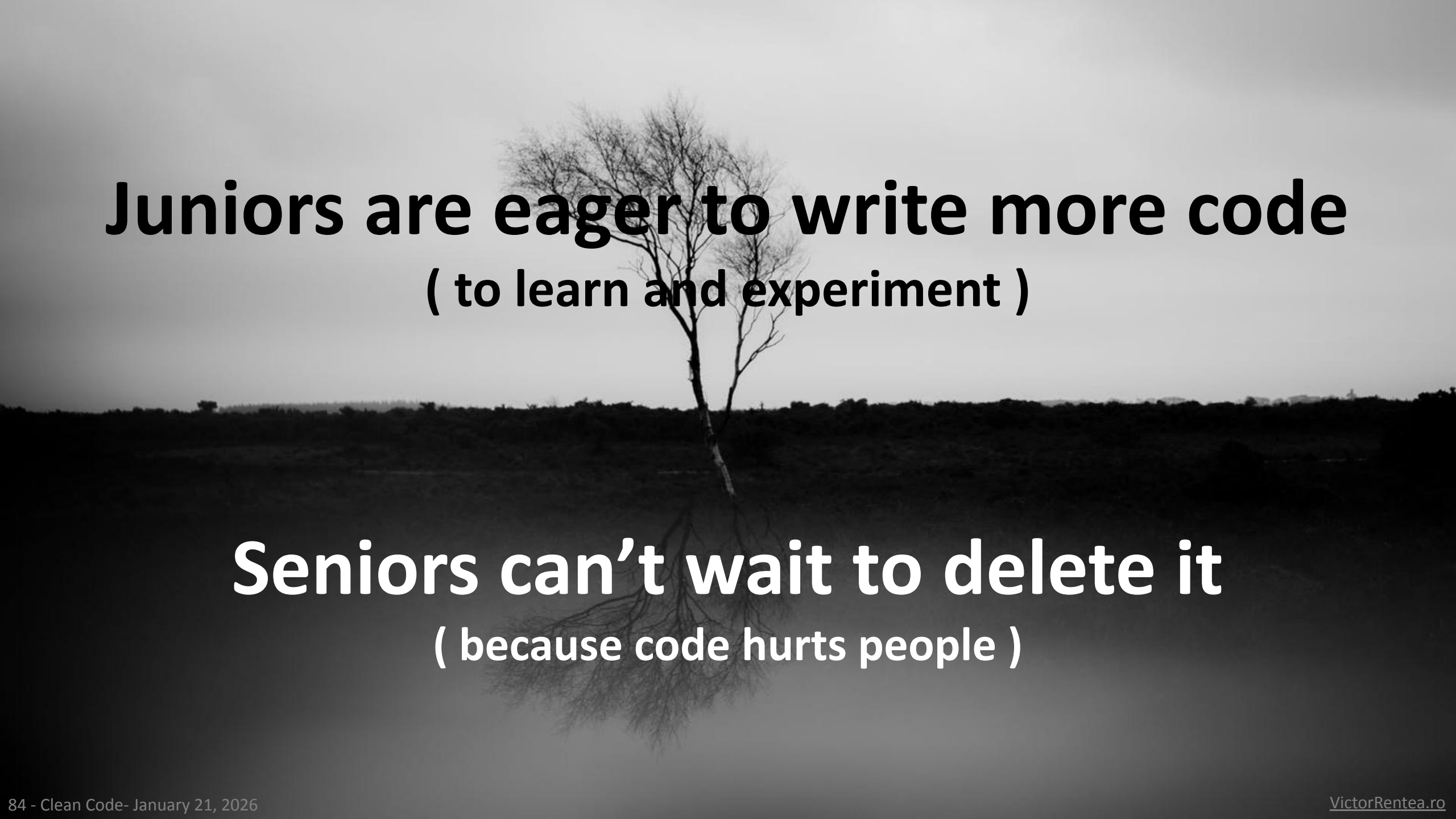
How to find code never executed in production?

Always do:

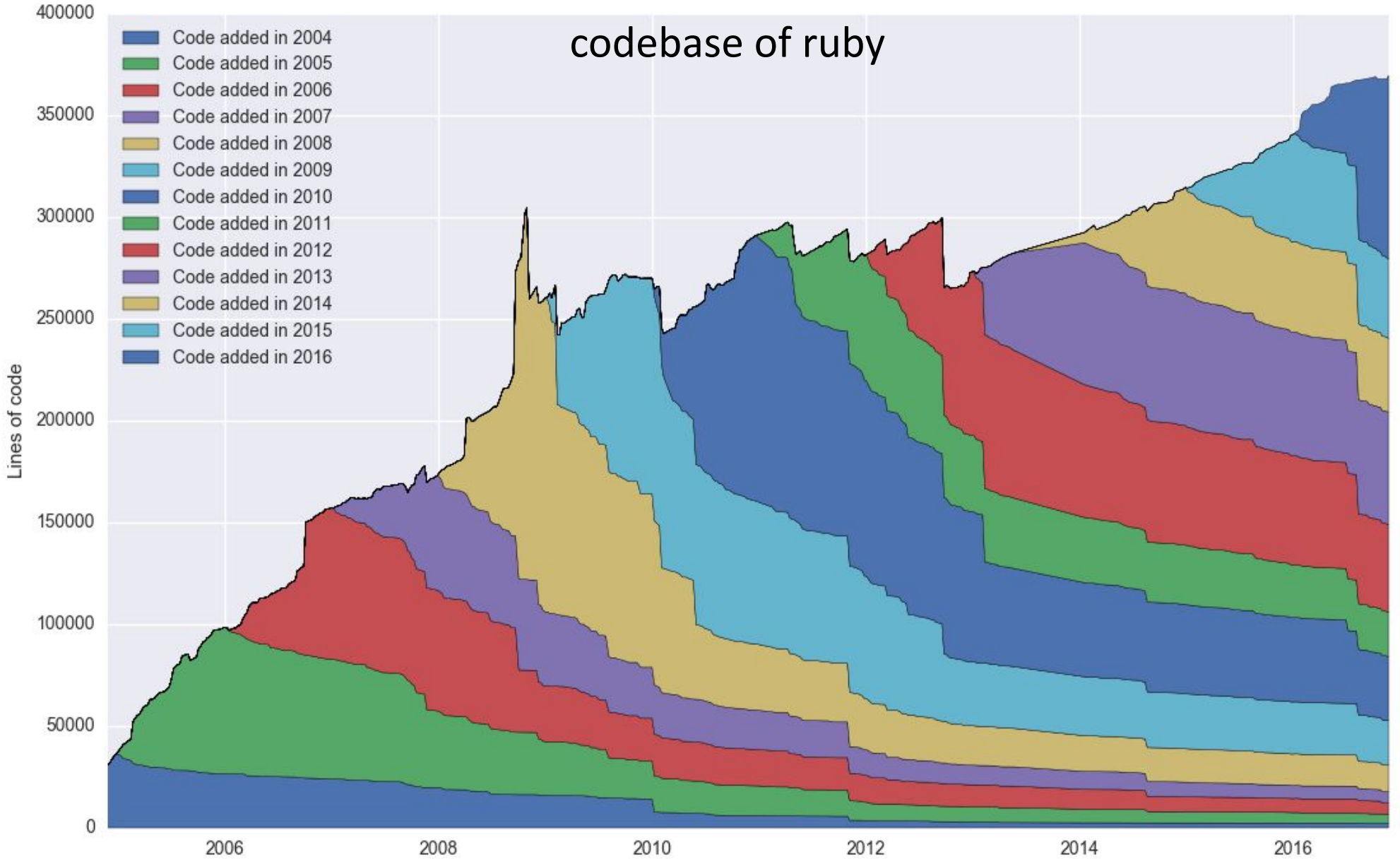
- **git blame** > **Ask author now** if recent ☝️ !
  6 month later: *Don't touch it!™*
- **Remove any feature flags** that are ON/OFF for ages
- Strip to minimum any solutions from SO/AI

Risky:

- **Scream-test**😱**:** delete > deploy + blue/green, canary
- **Record prod API calls** (eg w/ Apache log)
- `log.warn("Delete this section after Oct 2026 if this line was not printed in last 13 months")`
- **Metrics**.counter.increment() if log are not retained
- **Profiler**: ⚠️ can miss very fast / rarely called methods
- **Coverage agent** on 1 prod instance ⚠️ Performance


Juniors are eager to write more code
(to learn and experiment)
Seniors can’t wait to delete it
(because code hurts people)

Code is compressed/removed with time



Render a similar chart for your git using https://github.com/erikbern/git-of-theseus


I need
a bike!
- biz

Ever wrote code
anticipating a future
requirement,
or a **hoping a wider use**
(eg. a shared library)?

👍 Don't complicate today's code for a possible tomorrow

Ever wrote code **anticipating a** future **requirement**, or a **hoping a wider use** (eg. a shared library)?

Yes!

- a bright developer

Speculative Generality

aka " Overengineering "

Keep It Short & Simple (KISS, YAGNI)

But it's fun!!

Pet-proj ect!

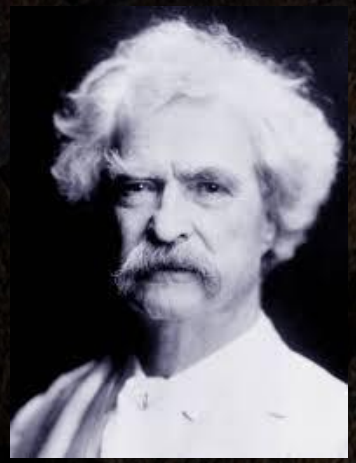
Common causes:
- long release cycle = Waterfall != Agile
- senior with better vision that business

Ship First Version

First attempt to a non-trivial problem is far from perfect

"I didn't have time to write a shorter letter,
so I wrote a long one instead."

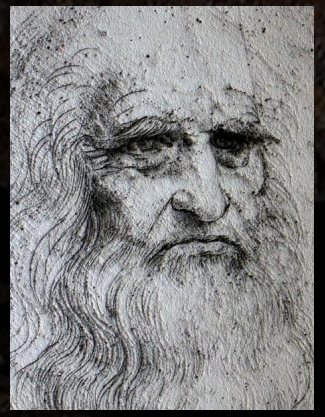
– Mark Twain



"Nothing is more difficult than
finding a simple solution to a complex problem.

Simplicity is the ultimate sophistication."

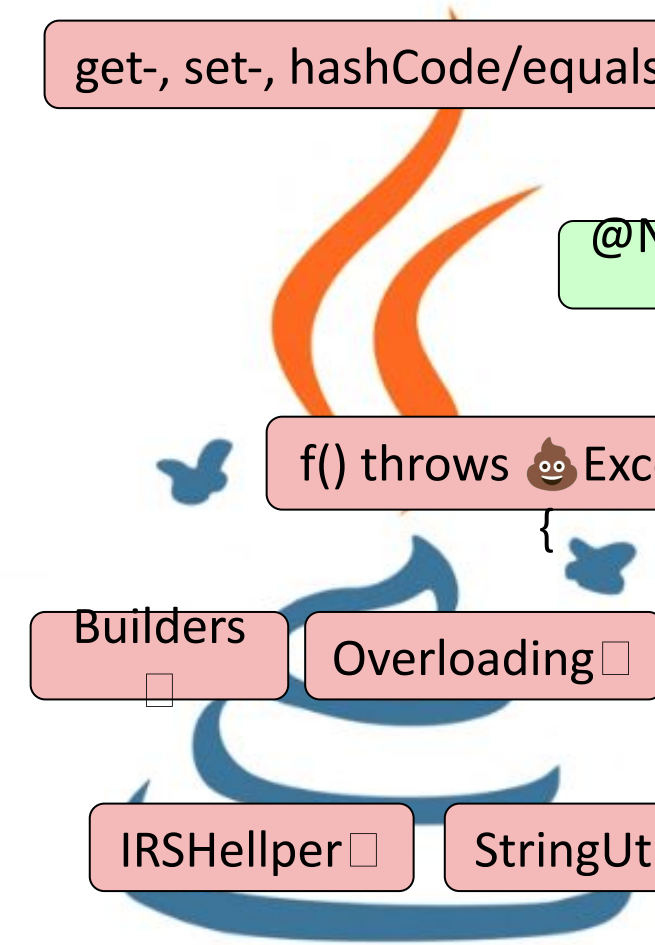
-- Leonardo DaVinci




AWESOME
Since
1995
Aged to
PERFECTION

Java Weaknesses

happening today, in modern languages like Kt, Scala, TS...



Boilerplate Code

get-, set-, hashCode/equals, toString =

Lombok

records [17]

data class

null

@Nullable

Optional<>

s: String?

s=str?.toUpper()??""

Checked Exceptions

f() throws 💩Exception

❤️ Runtime Exceptions

No Named or Default Params

Builders

Overloading

f(y=2, y=3);

f(x:Int=0, y:Int=0) {

No Extension Functions

IRSHellper

StringUtils

fun String.toCamel() = ...

interface MutableList<>
extends List<> (immutable)

Mutable Collections

list.add(e)

unmodifiableList(..)

List.of(..) [11]

ImmutableList (Guava)

.toList(..) [17]

Code Smells

Java got its name by people that had no idea if they were awake or asleep due to coffee abuse.

Despite all that,

Java Rocks

° https://victorrentea.ro/blog/the-clean-switch-rules/

```
foo(m) {
  switch (m.type) {
    case A: ...
    case B: ...
    case C: ...
    default: throw ...
  }
}
```

1° extract method

2° protection against new cases

```
bar(m) {
  switch (m.type) {
    case A: ...
    case B: ...
    case C: ...
    default: throw ...
  }
}
```

case D:

```
gee(m) {
  switch (m.type) {
    case A: ...
    case B: ...
    case C: ...
    default: throw
  }
}
```

RUNTIME EXCEPTION

repeated switches

```
class D implements Letter {
  foo(){} bar(){} gee(){}}
```

vs Polymorphism

enum with **abstract** methods

enum with BiFunction<> fields

return switch(enum)

Java17, Kotlin, Scala, C#8, php8

COMPILATION FAILS
if missing a gee() for D

Functional Programming Patterns with Java 8
(📄article and 📺talk)

Code Smell

AI hallucinations

CORE

- Magic Values
- Bad Names
- Monster Method
- God Class
- Flags
- Many Parameters
- Complex Loop
- Repeated Switch

Dead ☠ Code

OOP

- Missing Type
- Feature Envy
- Anemic Data Class
- Middle-Man
- Temporary Field
- Primitive Obsession

Overengineering 🚲🚀

FP

- Mutable Data
- Confused Variable
- Accumulator Loop
- Mutant Pipeline
- FP Chainsaw
- Reduce Abuse
- Monad Mania

Comments

We aren't saying that people should NOT write comments.

In our olfactory analogy, comments are a sweet smell.

The reason we mention comments as Code Smells is that comments are often used

as a deodorant

- Refactoring 2nd by Martin Flowler,Kent Beck 2019