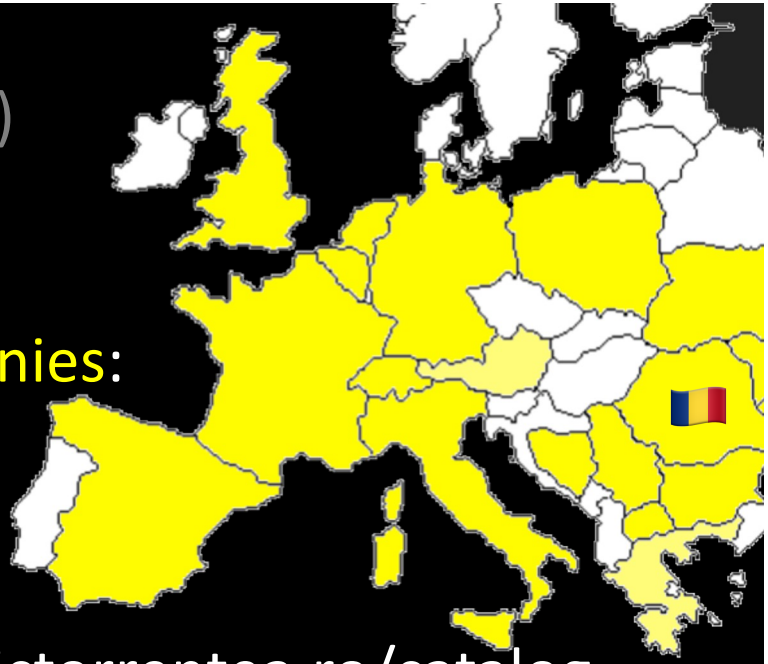
Code Smells

by Victor Rentea

👋 I'm **Victor Rentea** 🇷🇴 Java Champion, PhD(CS)

18 years of **coding**

10 years of **training & consulting** at 130+ companies:



- ❤️ Refactoring, Architecture, Unit Testing
- 🛠️ Spring, Hibernate, Reactive
- ⚡ Java Performance & Secure Coding

👉 victorrentea.ro/catalog

Lead of European Software Crafters (6.900) 👉 victorrentea.ro/community

Meeting online every month from 17^{00} CET

Channel: YouTube.com/vrentea ▶ past events

Life += 👰 + 👧 + 🤦 + 🐈

in 🐦 f

@victorrentea

 Kick-off Quiz

- Method length < 20* lines of code
- Method depth < 3 indentation levels
- File length < 200 lines
- Feeling about //comments in code: not required by good code
- When we see duplicated code: extract a shared method
- To work with collections, instead of a for loop, use: .map / .filter
- Feeling about setters: avoid - prefer immutable objects!
- Formatting: `for ( var e : list )` auto-format by IDE

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

Code Smells

code patterns that hurt you later

Copy-Paste Programming

DRY

Don't Repeat Yourself

Too DRY 

Removing duplication too aggressively can:

- **Increase complexity**: godMethod(a, b, c, d, false, true, -1, (x,y) -> ...)
- **Increase coupling**: method uses 10 dependencies
- **Abuse inheritance**: class Green extends Color extends Base extends ...
- **Accidental Coupling**☣ of unrelated flows with similar code

Only unify code that changes together 👉 ask PO

The rule of 3: **a little** repetition is better than the **wrong abstractions**

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


Bad Names

Bad Names

```
dr // mysterious acronym
```

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

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

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

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

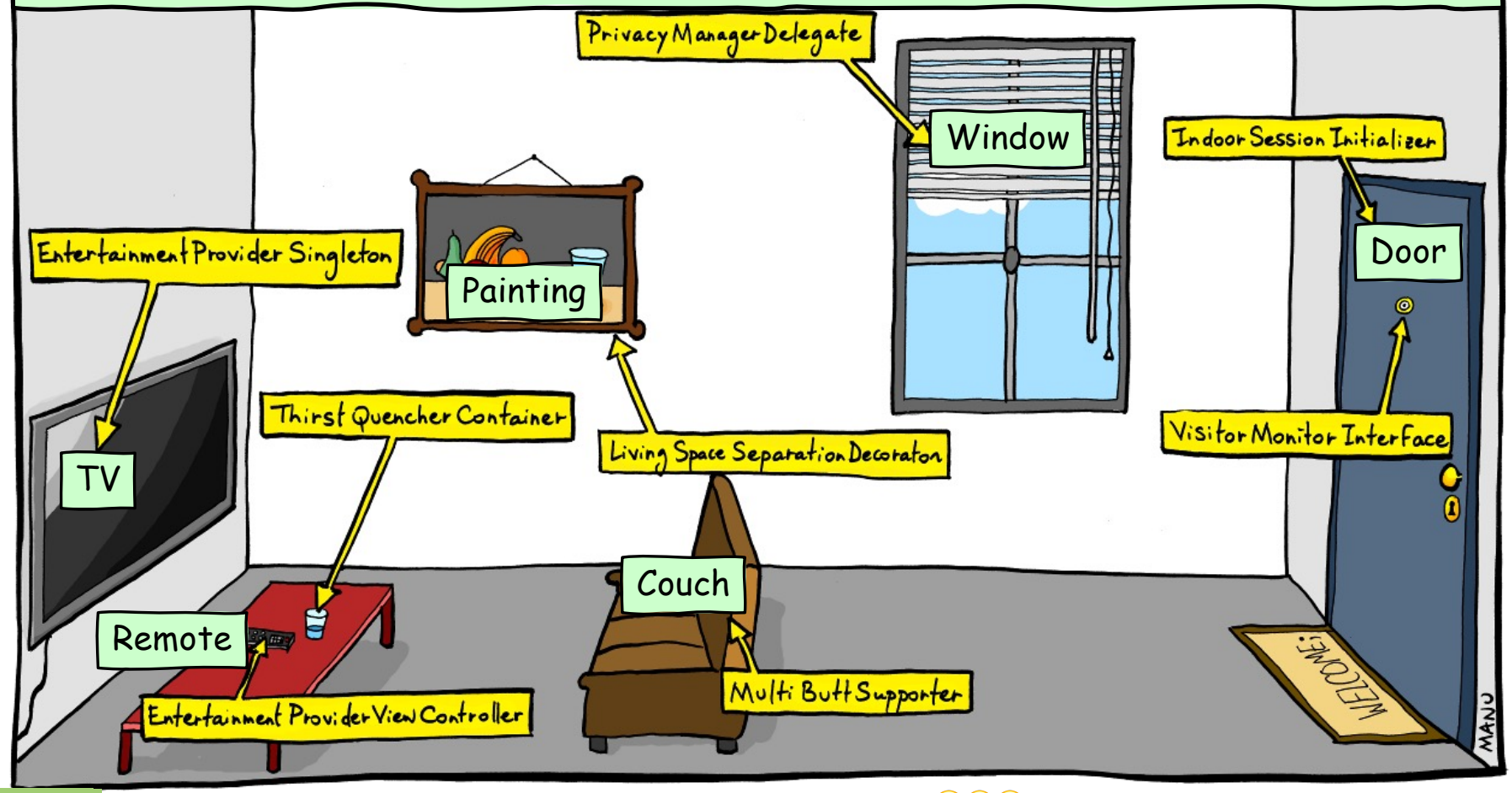
```
price = computePrice(..); // INSERT in DB?!
```

Command-Query Separation Violation

Code Smells

Names with bad Signal/Noise ratio

Use simple memorable names from the problem domain



Also see: [FizzBuzz Enterpise Edition](#) 🤣🤣🤣

Continuous Distilation

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
- Same Level of Abstraction (SLAb)

God Class

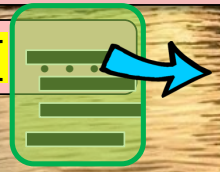
- > 200 lines ⛔
- ➔ Split in High / Low level
- ➔ Split in Flow A / Flow B

Many Parameters

- > 4 ⛔
- ➔ Split method by SRP
- ➔ Extract **reusable** Parameter Object
- also: Generics<A,B,U,S,E>

Complex Lambda

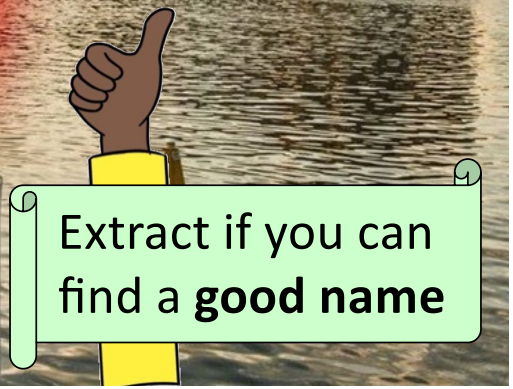
-> {

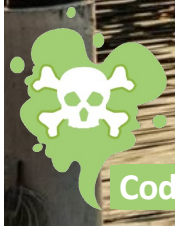


The magic number 7±2

How many things the human brain can keep in working memory (wiki)

max 7 ideas/method
max 7 methods/class

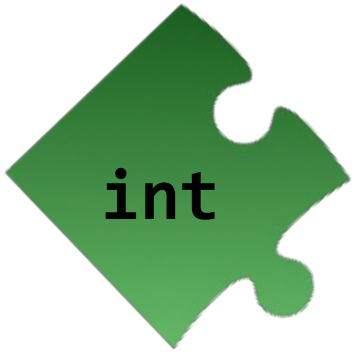


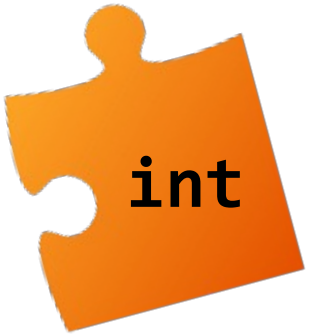


When in Doubt, Extract a Method

Missing Type

related data elements
moving around together

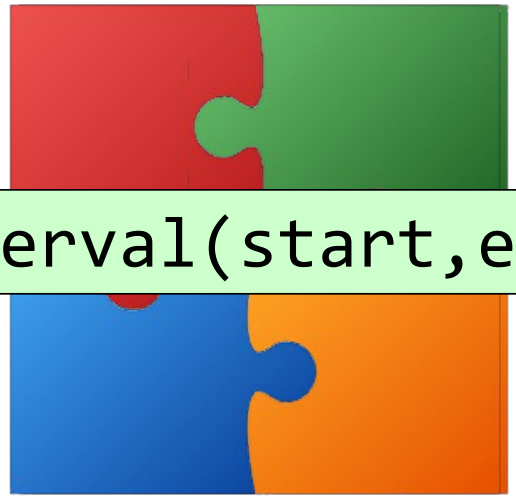




Missing Type

aka: Data Clumps, Missing Abstraction

Create more classes to group data that sticks together



Lightweight structures
Java: @Data/@Value, record
Kotlin: data class
Scala: case class
C#: record class
TS: class, interface, type

ArrayGeddon
What fields I receive?😱
IDE Refactoring = unsafe!

int, int ➔ Interval

js, ts, php, py..

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

Tangled Tuples

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

Map<Long, List<Long>> ➔ List<CustomerOrders>

Discover New Classes

Less Parameters
`in(x, Interval)`

Shrink Entities
`carModel.years:Interval`

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

Guard
Constraints
`start<end`

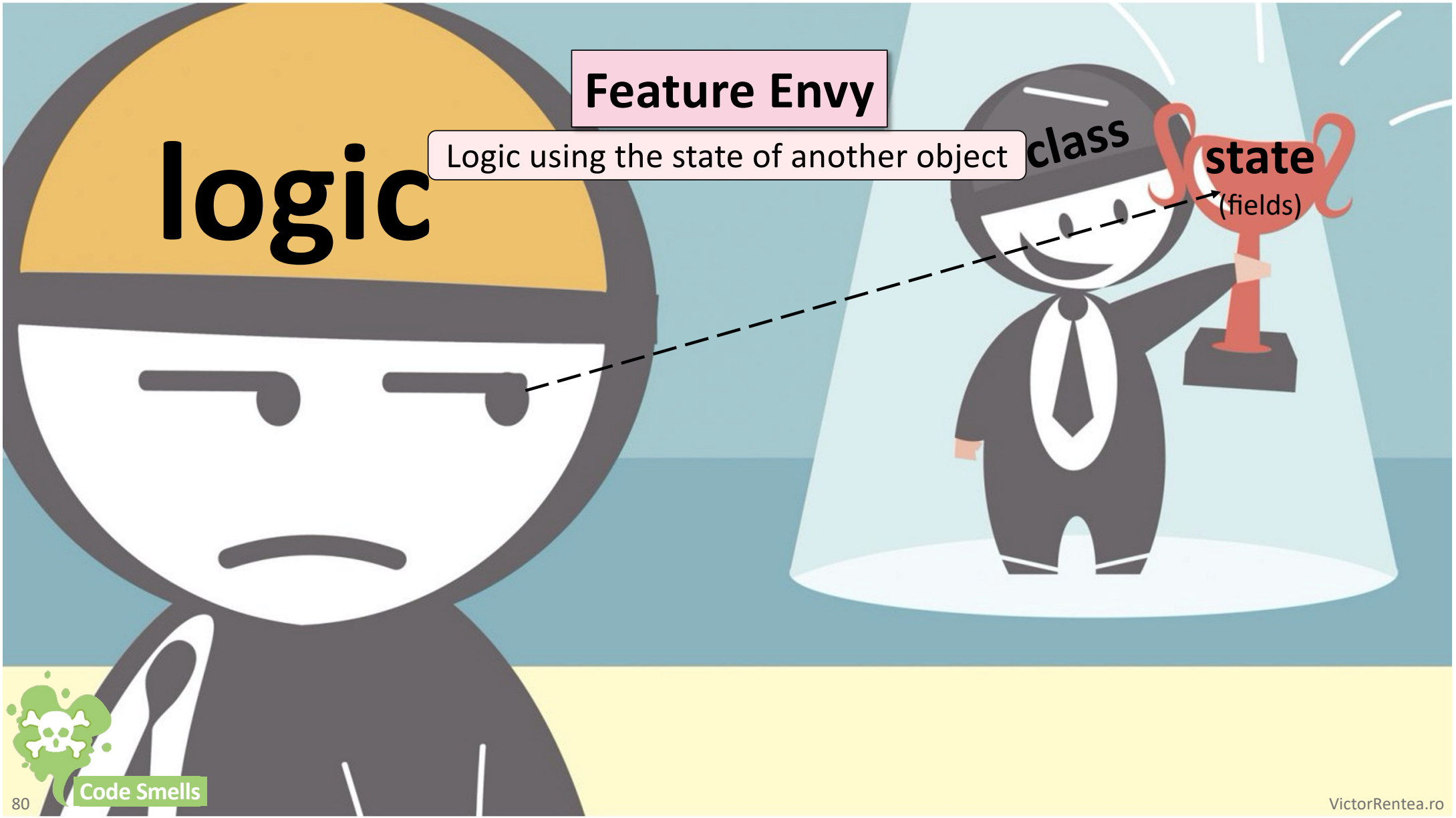
If Objects, not only a Data Structure

Util & Helper

Collect Selectively

Extension Functions

Code Smells


Feature Envy
Logic using the state of another object
logic
class
state
(fields)

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 class with fields, but no behavior

Rich Objects with logic & constraints help simplify core logic

Middle Man

Indirection without abstraction

NOISE

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

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

Inline Method… ⌥⌘N

Primitive Obsession

confused by raw data without meaning

Declaration:

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

Caller:

```
redeemCoupon(dto.cust, dto.coupon, dto.email)
```

```
Tuple3<Long, String, List<Long>>
```

one of: EMEA|NA|LATA|APAC

What do these mean? 😨

```
Map<Long, List<Long>> customerIdToOrderIds
```

Fight the Primitive Obsession by creating

Micro-Types

👍 Use for core IDs

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

Type-safe semantics

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

```
class CustomerId {
  private final Long id;
   ...get/hash/equals
}
```

```
@Value // Lombok
class CouponId {
  Long id;
}
```

```
enum Region {
  EMEA,
  NA,
  LATAM,
  APAC
}
```

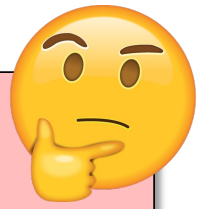
Restrict values

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

Logic inside vs an Util

```
Tuple3<Long, String, List<Long>>
Tuple3<CustomerId, Region, List<OrderId>>
```

```
Map<Long, List<Long>> map
Map<CustomerId, List<OrderId>> orders
```



- early decision
- **surprising**
- memory usage++

Fix: recycle in-memory instances:

ProductId.*of*(1L) == ProductId.*of*(1L)

(until Project Valhalla is done)


Functional
Programming

Functional Programming

In our daily life:

1) Pass behavior around

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

= Functions are first-class citizens

2) Simplify work with collections

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

+ Less mutation: ~~add, remove..~~

Philosophy of

Functional Programming

= Programming without side effects

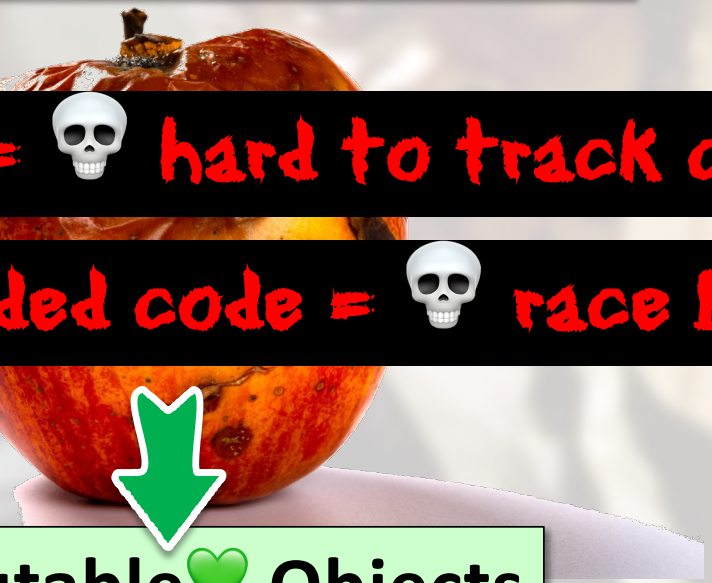
Functions should be *pure*

- No **Side Effects**
- Same **input** → Same **output**

Data should be *immutable*


SAY MUTABLE STATE
ONE MORE TIME

Long-Lived Mutable Data



in complex flows = 💀 hard to track changes

in multi-threaded code = 💀 race bugs

Immutable💚 Objects

immutable

https://gist.github.com/v[illegible]/c867f60c727c604

```
list5000.fold(listOf<Int>())
417.33 MB { list, e -> list + e }
```

if mixed with an ORM:

Cumbersome

if designed too large:

Fragmented Immutable

if cloned repeatedly:

Memory Churn

Fragmented Immutable >> Builder

Ugly large constructor

```
x = new X(1, 2, 3, false, -1, null, "X");
```

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

☹ .toBuilder() allows unrestricted changes

```
var obj2 = obj1.toBuilder().a(1).b(2)...build();
```

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

☺ Meaningful methods, that may guard rules

```
builder.fullName("First", "Last")... // manual builder
var obj2 = obj1.withFullName("First", "Last"); // wither
```

☺ Deeper model (new types)

```
var obj2 = obj1.withFullName(new FullName("F", "L"));
```

Confused Variable

This code lies to you!

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: SO

final
const
val

Mind **IDE warnings**

```
long averageEmpAge = emplo
```

Reassigned local variable

Error Prone fails compilation unless it's annotated with @Var

Code Smells

Confused Variable

Recycled variable

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

Reordering lines produces bugs

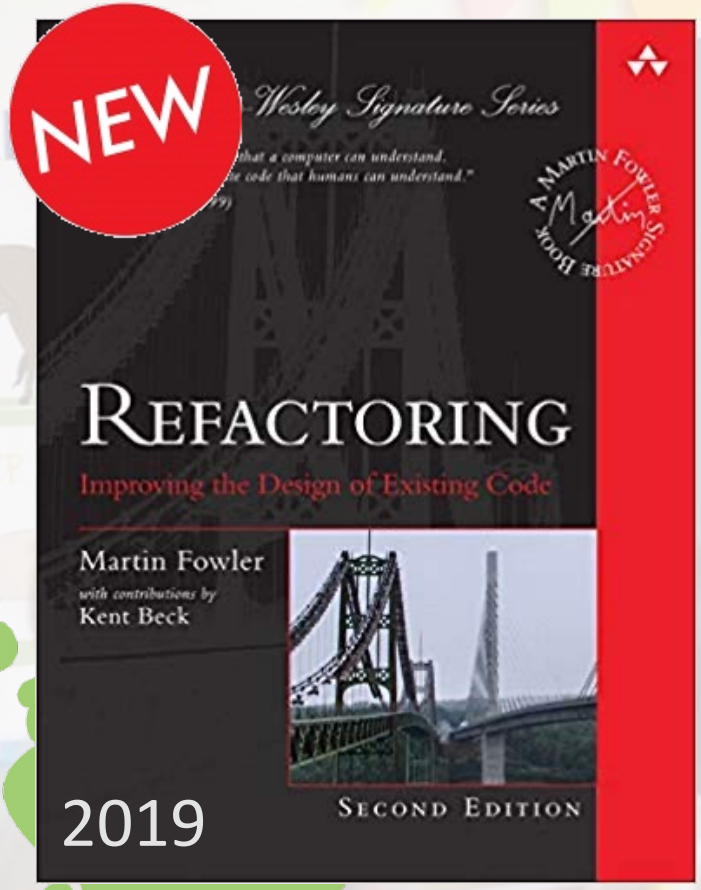
```
handle(Form form) {
  var valid = validate(form);
  var normal = normalize(valid);
  ...
  var withId = process(normal);
  audit(withId);
}
```

Don't reassign local variables

Temporal Coupling

the order of operations matters for mysterious reasons

Code Smells



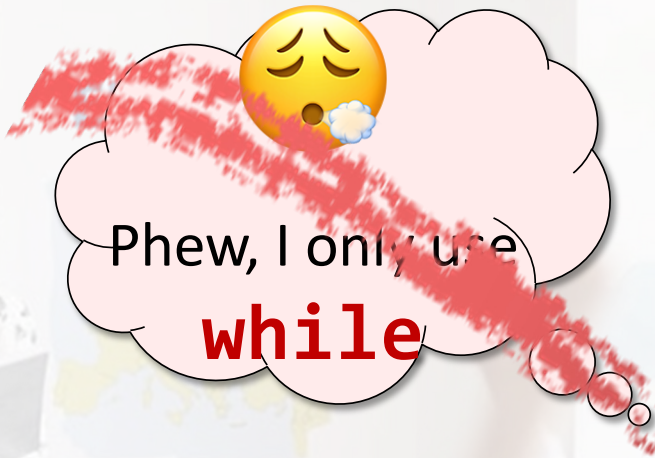
Loop

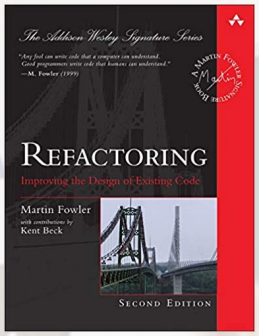
for

is a

Code Smells

- if it does more than one thing
- if it can be replaced with FP: .filter, .map…



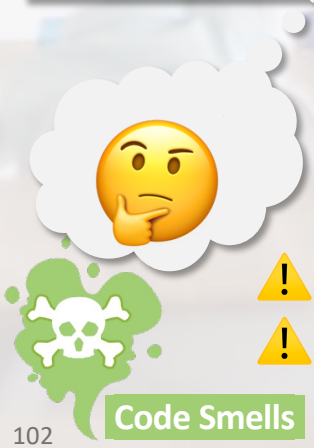


Complex Loop

doing unrelated things

```
for (e : list) {

  results.add(...)

  total += ...

  sideEffect(e);

}
```

Split Loop



⚠ no performance hit for typical BE

⚠ order of side-effects can matter

Code Smells

Accumulator Loop

NEW

gathering data via a loop

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

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

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

```
var total = list.stream()….sum()/reduce
```

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

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

Mutant Pipeline

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

NEW

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

compilation fails

lambdas can't change variables on stack

Hack

move mutable state on heap

```
// AtomicInteger total;
total.incrementAndGet(price);

// int[] sum = {0}; 😐
total[0] += price;

this.total += price;//field🚫
```

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);//avoid
```

Mutant Pipeline

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

```
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 → adder.increment(e));` ➔ `.sum():int`

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

✅ **OK** to do external side effects:

✅ `.forEach(m → mailSender.send(m));` ➔ `mailSender::send` ❤️

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

✅ `.ifPresent(e → repo.save(e));` ➔ `{save(it)} Kt` ❤️

Key Point

Avoid side-effects in `.forEach` (if possible)

Meanwhile:

```
.peek(list::add) // also
.map(e -> {list.add(e); return e;})
```

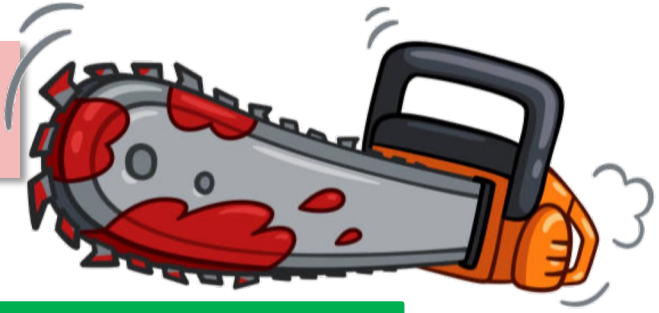
Side-effects are bad

(but often necessary)

Functional Programming

Misuse > Abuse

Functional Chainsaw



```
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

orderedProducts()

frequentProducts =

➔ extract explanatory **variables** and **functions**

after every ~ 3-4 operators

Expression Functions

(entire body is an expression)

```
fun x(…) = <expr>
x = (…) => <expr>
```

Reduce Rodeo

Complex folding

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

✅ Prep the collection

using `.filter()` `.map()` and keep reduce trivial

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

return list.filter(({price}) => !!price).reduce(maxByPrice);
```

✅ Use specialized collectors

`.sum()` `.toList()` `.max()` `.average()`

Use reduce for:

a) **unusual accumulation**
b) **compute multiple results**
eg max+min in a single pass
c) **performance** (measured)
d) **JS/TS**, kept simple⚠️

☠ 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
                   map(Post::title)
                   collectList())
    .map(LikedPosts::new)
    .onErrorContinue((x,e)→log.error(STR."Ignore \{x} for \{e}"))
    .doOnNext(message → log.info("Sending: " + message));
}
```

Chaining is mandatory

Reactive Programming is NOT clean!*

Goodbye variables and exceptions.

And learn 100+ operators.

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

Caught in translatic

```
  ...
  return result;
}
```

Java:
Kotlin: `Result`

Returning technical errors

rn

Use it to:
- Return a **sealed** type carrying the business outcome of an action
- Collect all failed items in a list

```
Try<String> process(Data data) {
  if (data.name().isBlank())
    return Try.failure(new AnException("…"));
  ..throw new ...
  return Try.success(result);
}
```

Just like checked exceptions(🤢) `throws AnException`

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

```
Try<Long> tryingFP(Data data) {
  return process(data)
    .onSuccess(e -> log.info("X"))
    .flatMap(repo::save)
    .andThen(id -> audit(data));
} // feeling zmart? 😎🤓
```

Pollutes caller

```
  }
  return idTry;
}
```

Double-Edged Return

Returning technical errors

Use it to:
- Return a **sealed** type carrying the business outcome of an action
- Collect all failed items in a list

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

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

throws Checked

pollutes caller

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

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

Caught in Transition

nostalgic 😢 for Go,Scala,Kotlin,JS

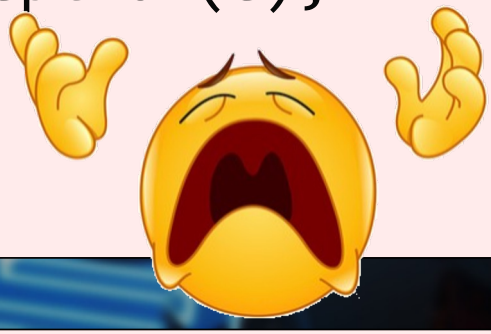
In FP
don't throw

Function Factory Frenzy

NEW

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

avoid taking / **returning** behavior whenever possible



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

Code Smells

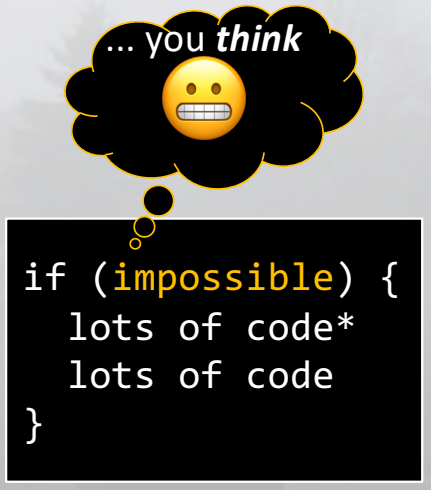

Dead ☠ Code

Dead ☠ Code

Not Referenced

A param, variable, method, unused in IDE

➔ Delete [with IDE] 🥳



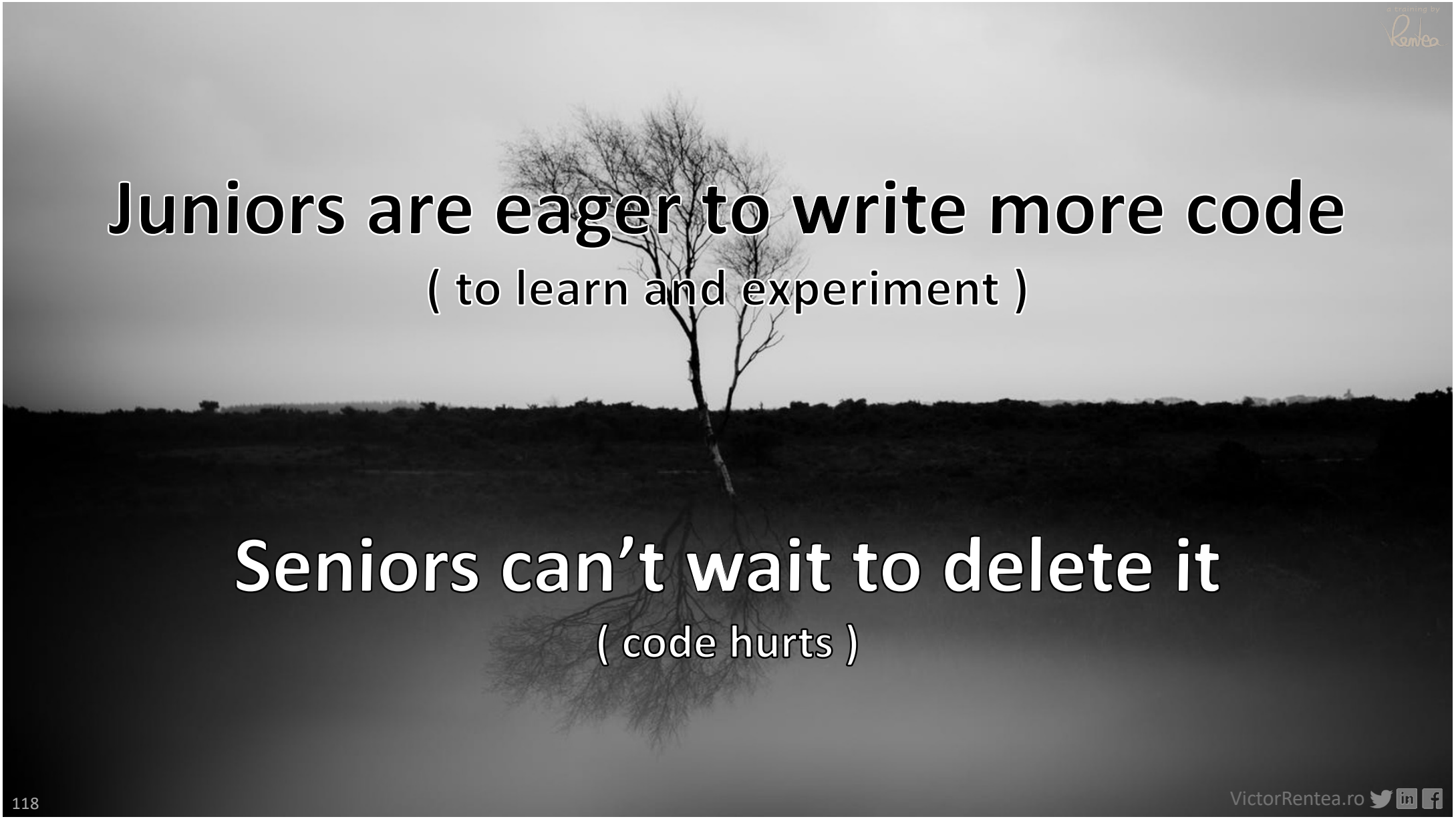
Unreachable Code

How to find code never called in production?

➔ Monitor API calls in prod

➔ git blame: if recent ☝ > **Ask author now**!
(6 month later: *It works? Don't touch it!* ™)

➔ `log.warn("🙏Delete after Sep'24 if this is in not in log of the last 6 months*")`


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


I need
a bike!
- biz
Ever wrote code
anticipating a future requirement,
or a hoping a wider use
(eg. a shared library)?
VictorRentea.ro

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

Yes!
- a bright developer

Overengineering
aka Speculative Generality

Keep It Short & Simple (**KISS**)

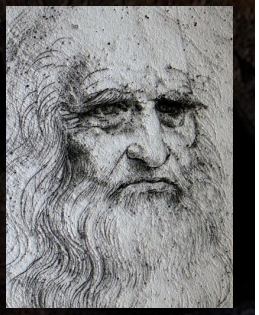
But it's fun!!

Pet-project!

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

**Simplicity is the
ultimate sophistication.**

-- Leonardo DaVinci



Code Smells


AWESOME
Since
1995
Aged to
PERFECTION

Java Weaknesses

getter, setter, hashCode/equals, toString

Boilerplate Code

Lombok

records [17]

`null`

Optional<>

@Nullable

Checked Exceptions

❤️ Runtime Exceptions

Builders

Overloading

No Default / Named Params

Utils

No Extension Functions

list.add(e)

Mutable Collections

unmodifiableList(list)

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.

Code Smells

most common as of 2024

- Bad Names
- Monster Method
- God Class
- Flags
- Many Parameters
- Complex Loop
- Repeated Switches
- **Dead Code**

OOP

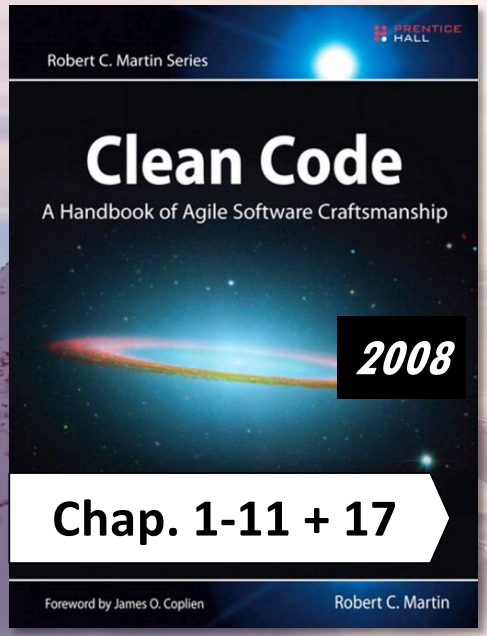
- Missing Abstraction
- Feature Envy
- Data Classes
- Middle Man
- Primitive Obsession
- **Overengineering**

FP

- Accumulators
- Imperative FP
- Heavy Lambda
- Reduce Rodeo
- FP Chainsaw
- Mutable Data
- Temporary Field
- **Confused Variable**

Clean Code - Reading Guide

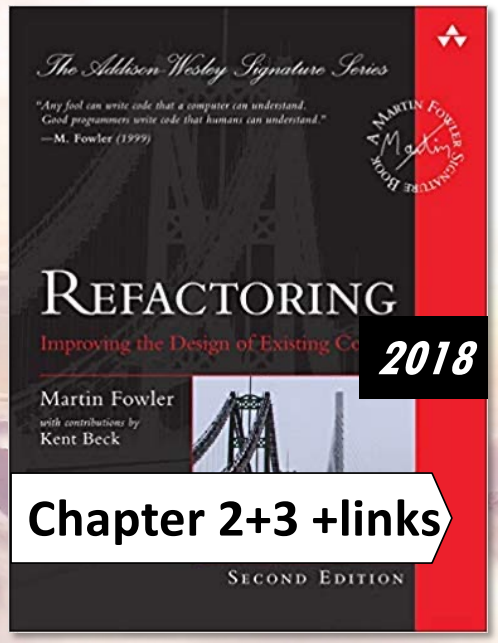
A time-efficient way to get up to speed with Clean Code and Refactoring



Chap. 1-11 + 17

Summary article: link

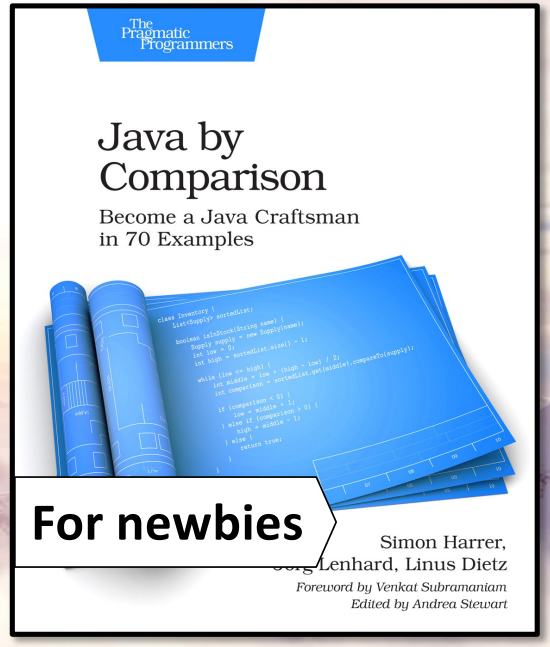
cleancoders.com



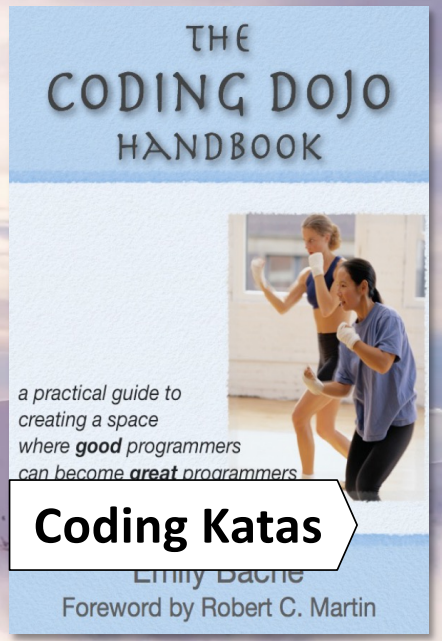
Chapter 2+3 +links

sourcemaking.com/refactoring

refactoring.guru



For newbies



Coding Katas

kata-log.rocks/refactoring

Seed Passion for Clean Code

is this *The End*

European Software Crafters

https://www.meetup.com/european-software-crafters

The largest community in the world on
- crafting **good quality code,**
- **simple design**
- **robust unit tests**
- continuous growth & teamwork

Meeting **every month**
after hours (17^{00} CET)
online on Zoom + YouTube Live

Free Forever!

I was VictorRentea.ro