Design Patterns and Principles

A TRAINING BY VICTOR RENTEA

VictorRentea.ro
victor.rentea@gmail.com
@victorrentea

Hi, I'm **Victor Rentea** 

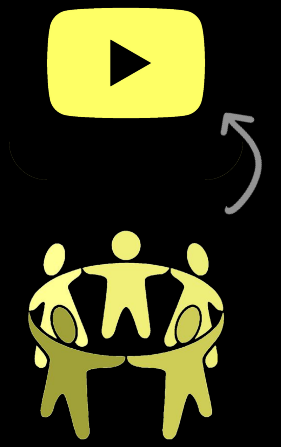
Java Champion, enjoying  since 2006

Trainer – 5000+ developers of 100+ companies in 20+ countries

- Clean Code & Unit Testing
- Spring & JPA
- Architecture, DDD
- Java Performance
- Reactive Programming 😱
- Secure Coding

- **in-house**
- mixed groups
- video classes

YouTube Channel



Speaker at Conferences & Meetups 

Software Craftsmanship Community

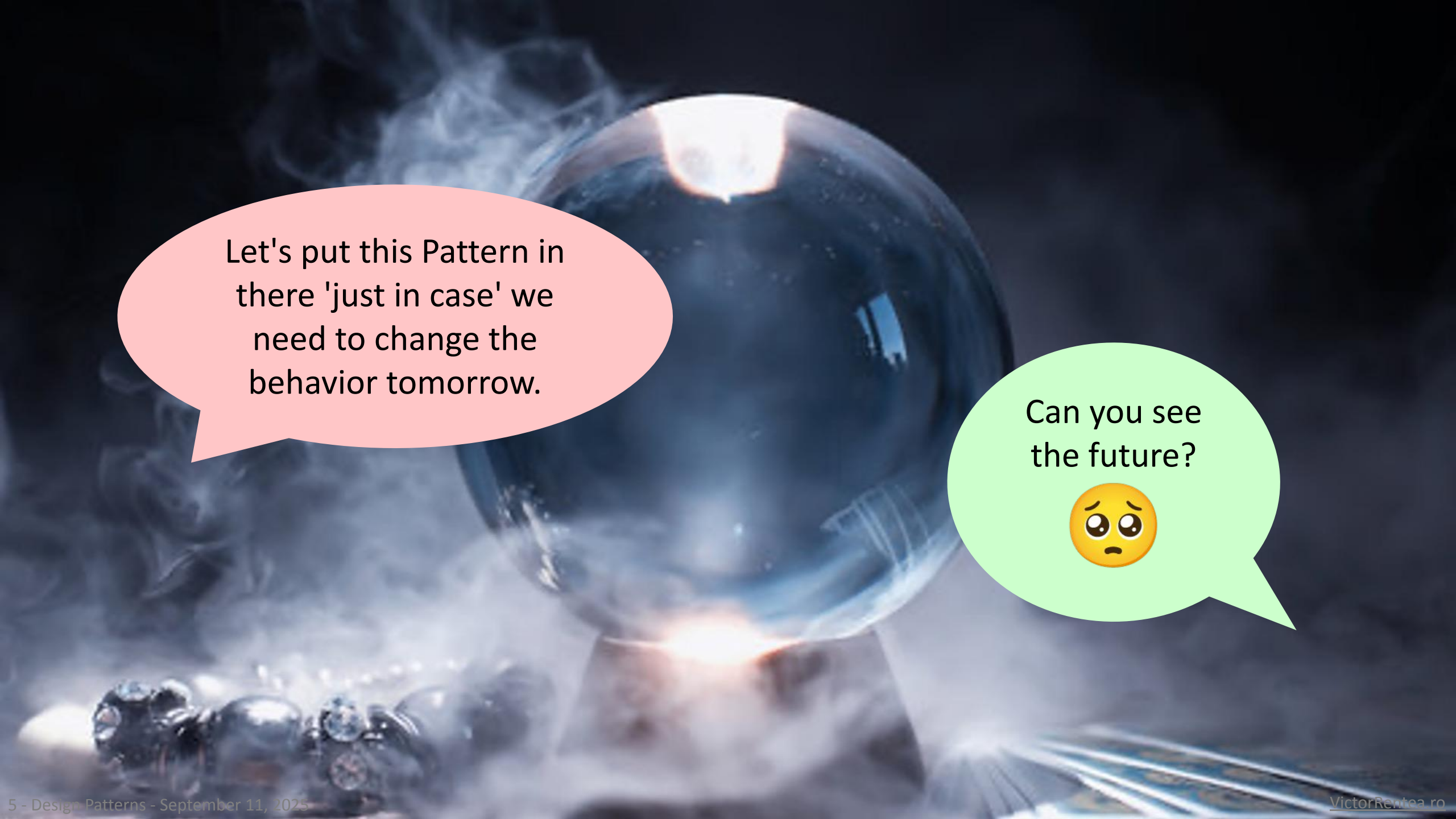
Join at victorrentea.ro/community

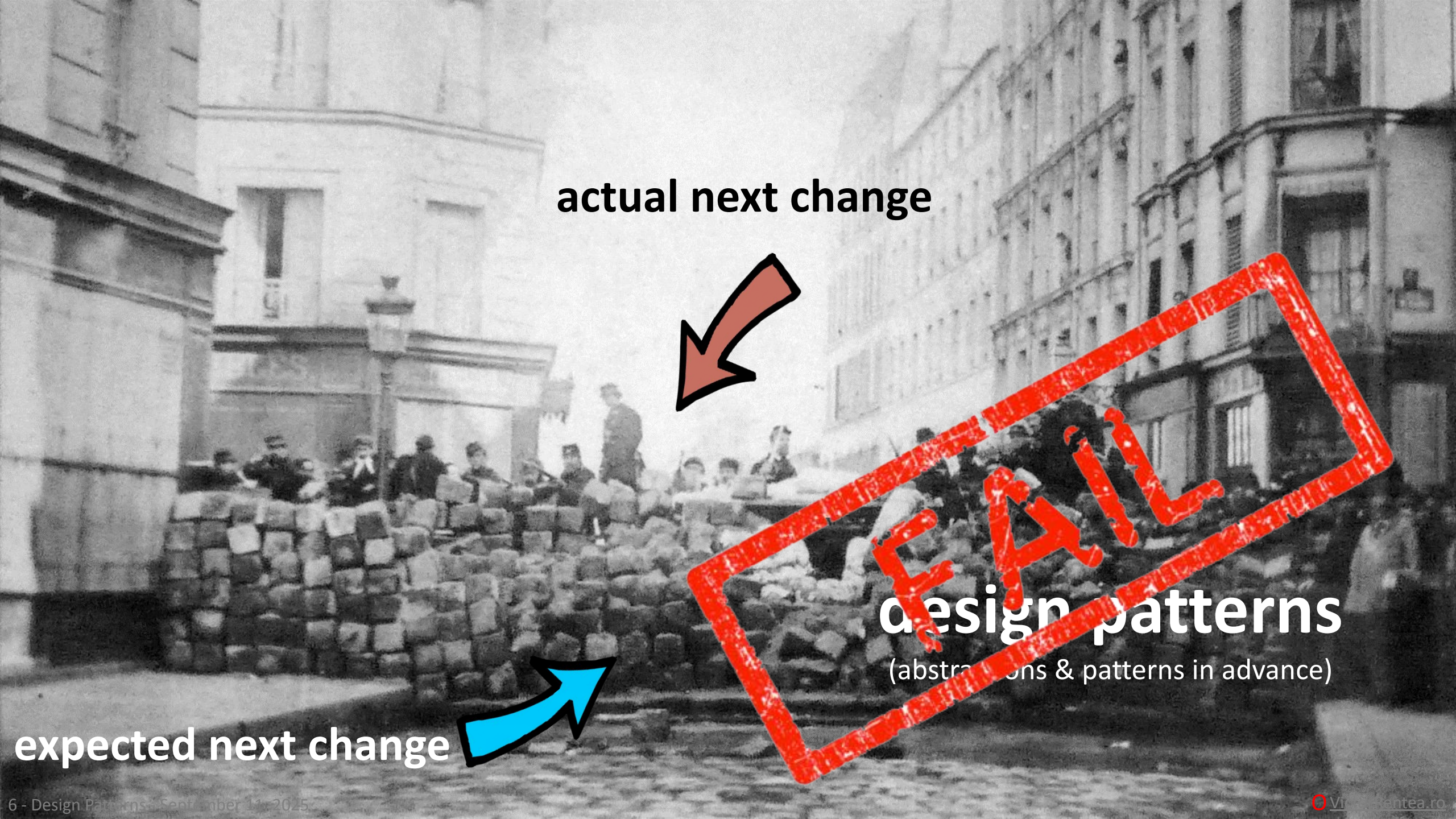
and enjoy 1h/month of free workshop/talk

VictorRentea.ro

@victorrentea


Design Patterns


Let's put this Pattern in there 'just in case' we need to change the behavior tomorrow.
Can you see the future?
🥺

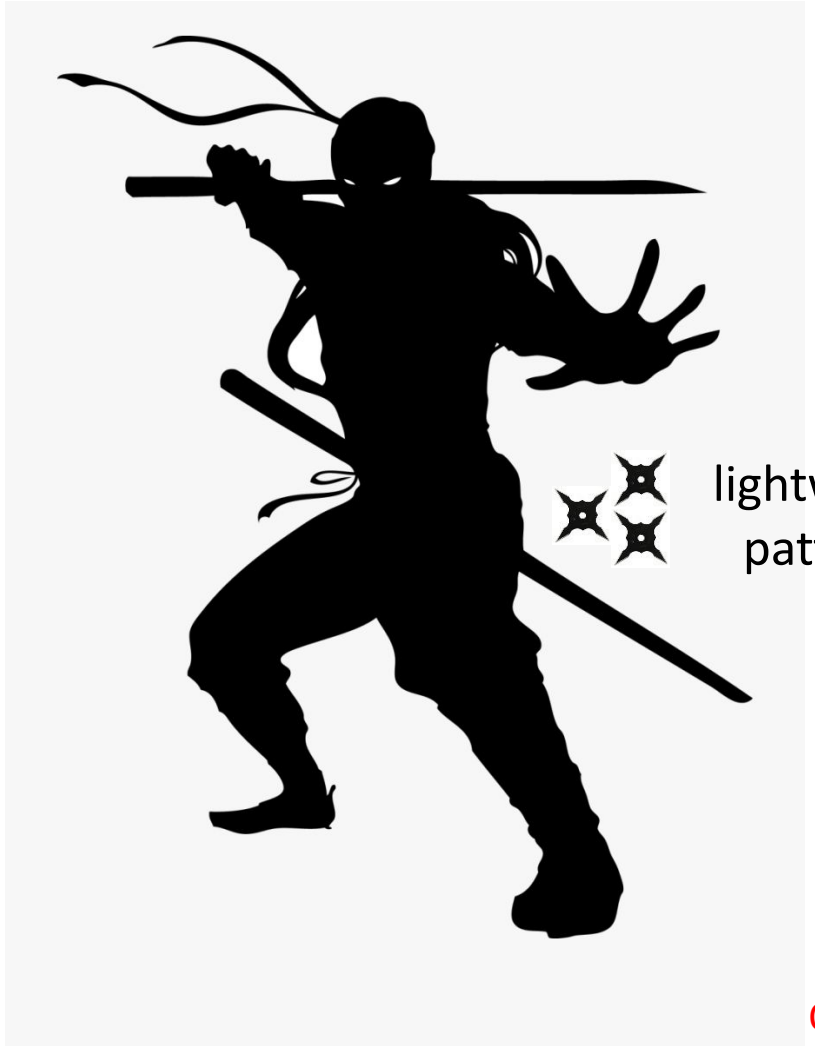

actual next change
FAIL
design patterns
(abstra ons & patterns in advance)
expected next change

Overengineering

Anticipating changes

Simple Code

Flexibility



Design Pattern

Where can I apply this Design Pattern?

Pr le

Premature optimization ***Overengineering*** *is the root of all evil*

– Adam Bien

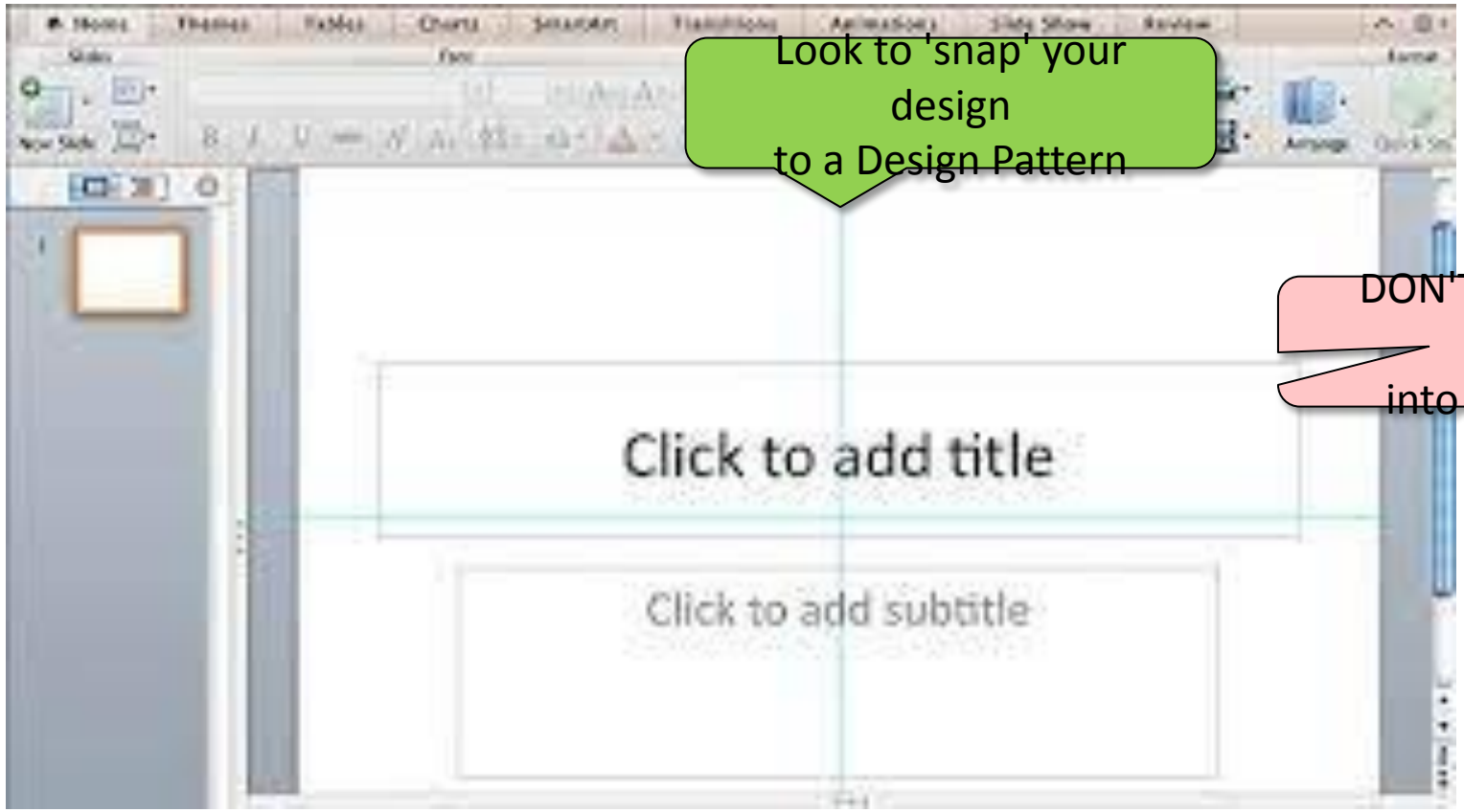
Refactor towards Design Patterns to Simplify code,

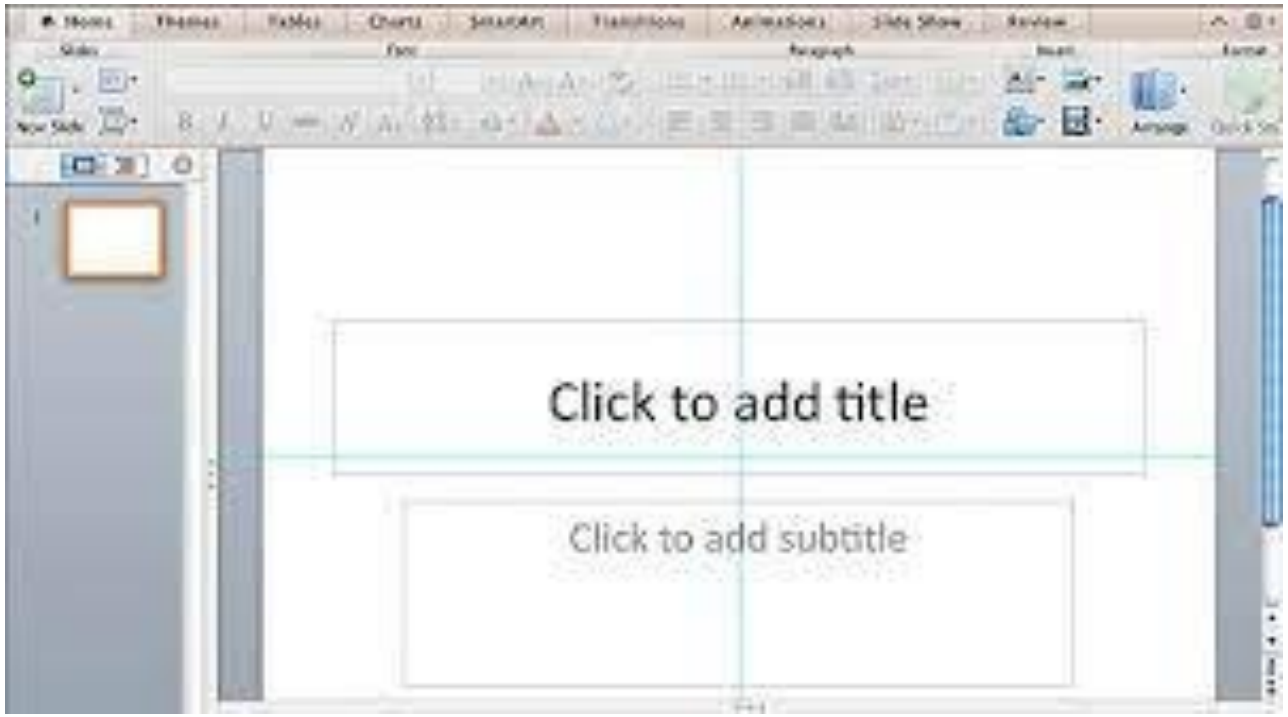
<u>only after</u> you have an (ugly) working solution!

(you understood the problem)

Keep It Short & Simple

"Snap to Patterns"




Click to add title
Click to add subtitle

Agenda

- **Design Principles**: Cohesion, Coupling, SOLID, DRY, KISS
- **System Architecture**: Layers, Onion, Adapter, Facade
- **Intercept Calls**: Decorator, Proxy, AOP, Execute-Around
- **Creating Objects**: Builder, Singleton, DI, Object Pool
- **Propagate Change**: Command vs Events
- **Microservice Patterns**: Orchestration, Saga, Outbox,...
- **Fork Behavior**: Strategy, Filters, Template, State, Visitor

Agenda

1. Factories
2. Builders
3. Singleton
4. Null Object

Factories

Instead of **`new`**

Create via a method

WHY

Hide Instantiated Type

Named constructors

Prefill instances

Control # of instances

LIBRARY CODE

Static Factory Method™

Hide Instantiated Type

The returned type depends on the library detected on classpath

XML `DocumentBuilderFactory.newInstance(): DocumentBuilderFactory`

SQL `DriverManager.getConnection(): Connection`

abstract

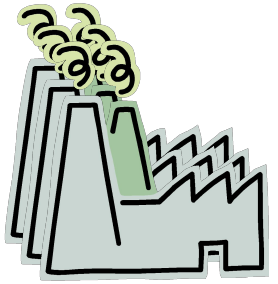
Return different types based on parameter

`Collections.unmodifiableList(list): List`

`getMessageHandler(MessageType): MessageHandler`

LIBRARY CODE

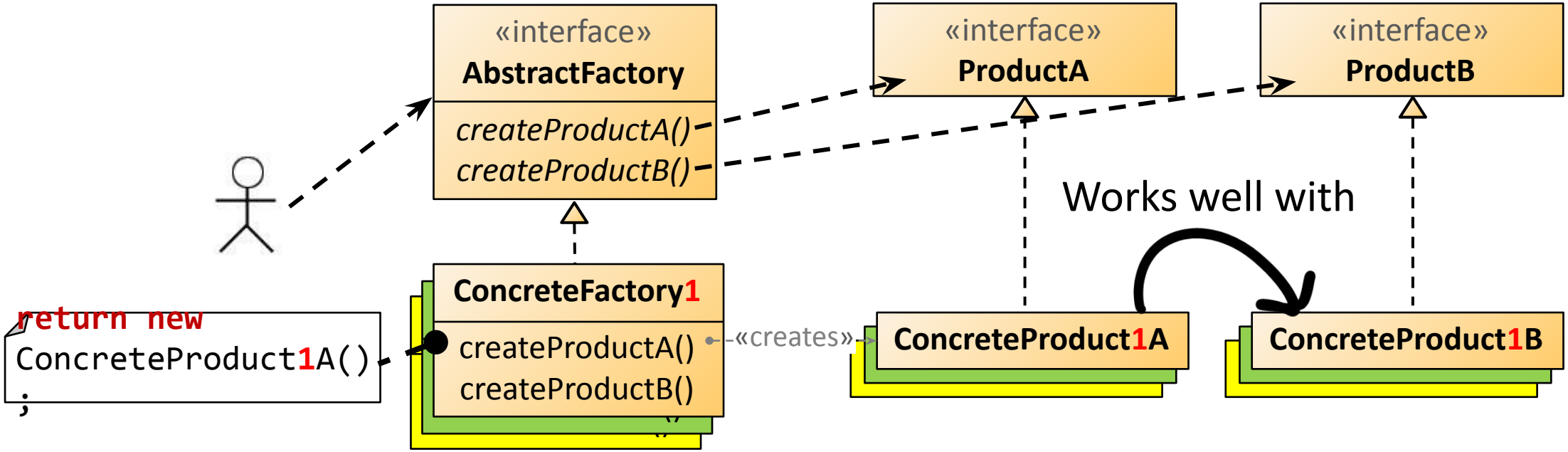
Abstract Factory



define an interface for creating families of objects

but let Factory implementations choose the classes to instantiate

- Examples: UI Look-and-feel, XML DOM[JSE], Eclipse SWT, JMS[JEE], DbProviderFactory[C#]



LIBRARY CODE

Instead of Constructors

Overloaded or with Default Parameters

```
= new OutputFile("anaf");
```

```
public OutputFile(String name) {
  this(name, ".csv");//call another ctor
}
```

If you open it, would you expect this ?

... use Static Factory Method™ as

Named Constructors

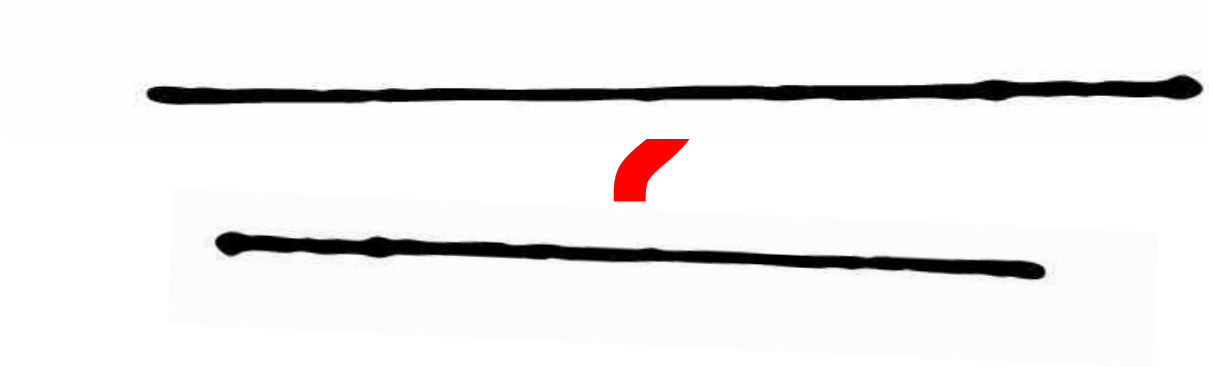
```
= OutputFile.aCsv("anaf");
```

```
public static OutputFile aCsv(String name) {
  return new OutputFile(name, ".csv");
}
```

Instead of **`new`**

Create via a method

WHY



Control # of Instances

Prefill Instances

Factory Method™

Control # of Instances

Resource Pool

```
DataSource.getConnection():Connection
```

May block until a connection is released: `Connection.close()`

Canonical Instances

```
Long.valueOf(...)
```

```
Long l1 = 127L;
Long l2 = 127L;
System.out.println(l1==l2);
```

Save memory by caching immutable instances

Factory Method™

Prefill Instances

Copy-mutators for Immutable Objects

```
fullName.withLastName("LN"):FullName
```

to hide the self constructor call

Test Data Factory

```
aValidCustomer():Customer
```

to reuse 'reasonable' instances in multiple tests

Where do you need a lot of data variants?

Factory Method Pattern

Hide Instantiated Type

Many Types? → **Abstract Factory**

Named constructors

Control # of instances

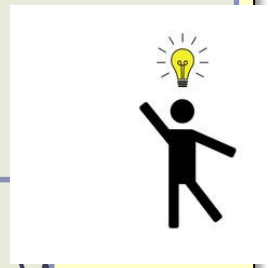
Prefill Instances

Builders

How to populate an instance?

Temporary variables

```
Customer customer = new Customer();
customer.setName("John Doe");
List<String> labels = new
ArrayList<>();
labels.add("Label1");
customer.setLabels(labels);
Address address = new Address();
adress.setStreetName("Viorele");
customer.setAddress(address);
```



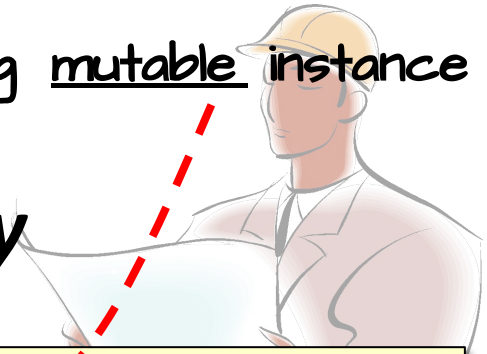
```
Customer customer = new
CustomerBuilder()
    .withName("John Doe")
    .withLabel("Label1")
    .withAddress(new AddressBuilder()
        .withStreetName("Viorele")
        .build())
    .build();
```

Hard to read

Want some fun? Generic Builder: https://stackoverflow.com/questions/31754786/how-to-implement-the-builder-pattern-in-java-8

Fluent Builder

Form: holding mutable instance



create ar tance in a human readable way

Method Chaining

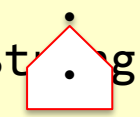
☺ Less stupid variables

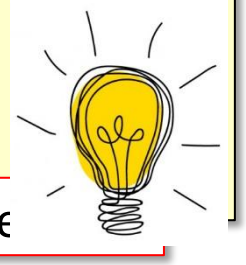
☹ More Code (generate it!)

```
Customer customer = new
CustomerBuilder()
    .withName("John Doe")
    .withLabel("Label1")
    .withAddress(new AddressBuilder()
        .withStre
        .build())
    .build();
```

.withAddress(AddressBuilder)

```
public class CustomerBuilder {
  private Customer customer=new Customer();

  public CustomerBuilder withName(String name)

      tomer.setName(name);
    eturn this;

  public CustomerBuilder withLabel(St g
label) {
    customer.getLabels().add(label);
  public Customer persist() {insert;
  return;}
  }
  ...
  public Customer build() {
    return customer;
  }
}
```





Customization ide

Form 2: Fluent Setters

```
public class Customer {
  private String name;
  private List<String> labels=new ArrayList<>();
  private Address address;

  public Customer setName(String name) {
    this.name = name;
    return this;
  }

  public Customer addLabels(String... labels) {
    this.labels.addAll(asList(labels));
    return this;
  }
}
```

Fluent Setter

(few) extra Test-Helper Method

Form 3: Fluent Setters by Lombok

auto-generated

```
@Entity
public class Customer {
  @Setter @Getter
  private String name;
  private List<String> labels=new ArrayList<>();
  private Address address;

  public Customer setName(String name) {
    this.name = name;
    return this;
  }

  public Customer addLabels(String... labels) {
    this.labels.addAll(asList(label));
    return this;
  }
}
```

src/main/java/lombok.config

```
lombok.accessors.chain=true
```

Somewhere in your tests...

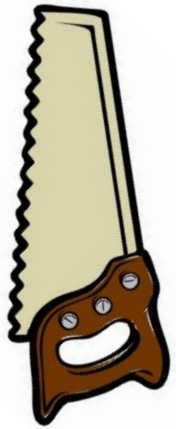
"I need a filled entity... Just like the one in that other test!"

Hm...

`TestData.aValidCustomer():Customer`

(Static) Factory Methods creating 'reasonable' test instances

Obj[illegible]ther F... *(when it couples your tests together in L-XL apps)*



Customer
sub-domain
OrderMother `.aValidCustomer()`

Order
sub-domain
OrderMother `.aValidCustomer()`

https://martinfowler.com/bliki/ObjectMother.html

Can a Customer have a null name?

NO!

Fluent Builder

Form4: hide heavy constructor

aka "DRAFT STATE Builder"

OOP: ***Objects should guard their internal consistency!***

Validate mandatory fields in constructor

```
public class Customer {
  public Customer(String name,
                  List<String> labels, ...) {
    if (name == null)
      throw new IllegalArgEx();
    this.name = requiname;
    this.labels = labels;
    ...
```

All mandatory fields

Too many constructor params

>3, 5...

Yuck!

```
public class CustomerBuilder {
  private String name;
  private List<String> labels=new ArrayList<>();
  private Address address;

  public CustomerBuilder withName(String name) {
    this.name = name;
    return this;
  }
  ...
  public Customer build() {
    return new Customer(name, labels, address,
...)
  }
}
```

Or

```
return new Customer(this);
```

Passing the builder to the Entity Constructor

Real problem: Bloated Entity

Decompose it into smaller pieces

Real Problem:

The BF Entity

De-compose it

ORM


```
public class Customer {
   private Long id;
   private String firstName;
   private String lastName;
   private String namePrefix;
   private String streetNumber;
   private String city;
   <30 more fields>
   <10 more fields>
}
```

@Embedded
`private FullName fullName;`

@Embedded
`private Address address;`

create new classes!

@Embeddable
```
public class FullName {
   private String firstName;
   private String lastName;
   private String namePrefix;
}
```

Best if Immutable

@Embeddable
```
public class Address {
   ...
}
```

BTW, I am only talking about data objects here, not Services

create new classes!

Creating Objects

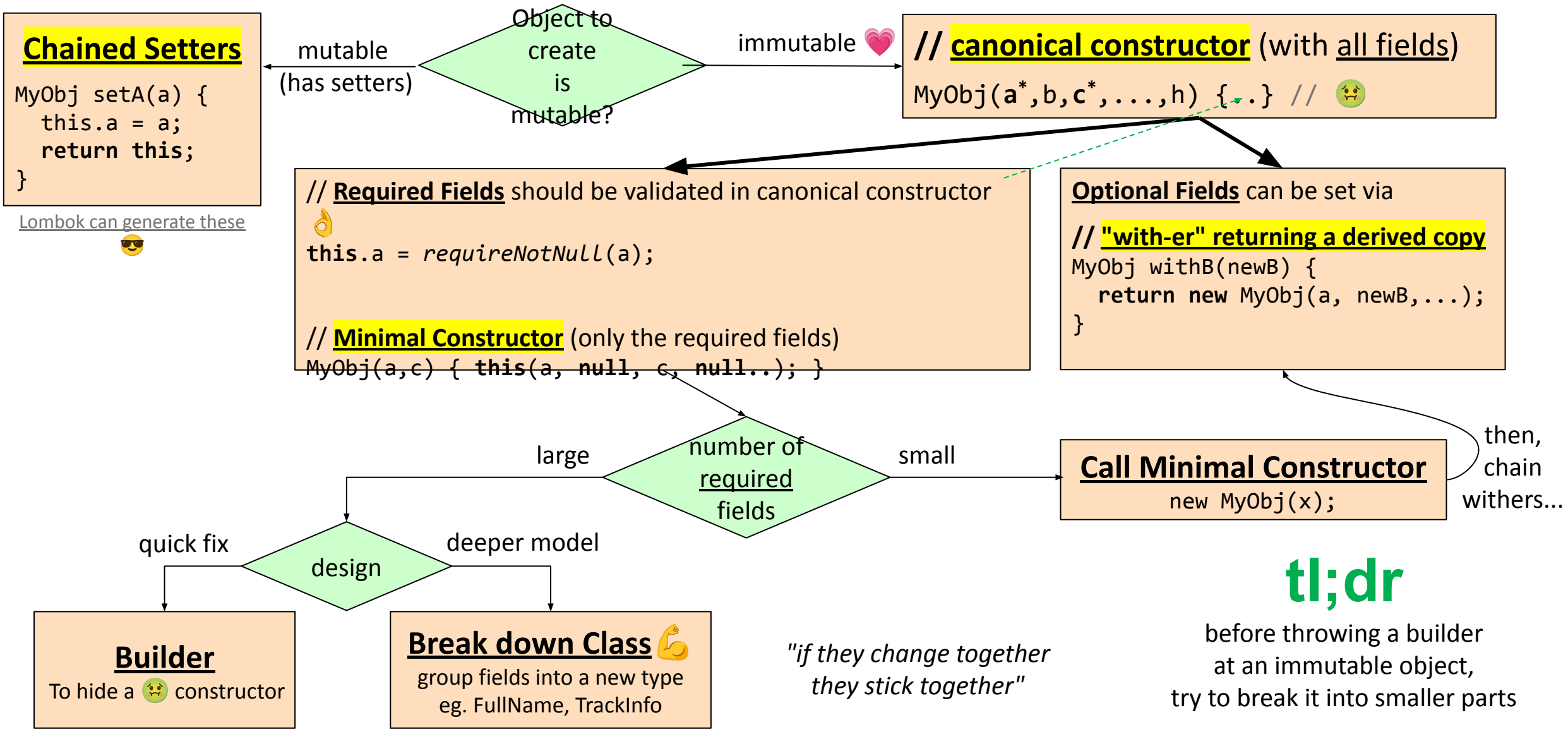
Fancy ways

Builder

Singleton

DI, Pooling

Is Builder an Anti-Pattern?



Builder – Conclusions

- Goal: simplify creation of objects by pulling stupid code in
  - Pays off if used many times to create such objects (eg in tests)
- Goal: hide a constructor with many parameters
- Self-validating Model
  - Long constructor arguments □ decompose the model
- "Step Builder" for building APIs
  - see next slide (horror) 🤓

Form 5: Step Builder

https://www.javacodegeeks.com/2013/05/building-smart-builders.html

GEEK STUFF

Builders return different 'next builders', forcing you to set the correct state

Use for: Unknown developers building complex configuration

```
UserConfiguration config1 = StepBuilder.newBuilder()  StepBuilder.NameStep
    .name("aaa")  StepBuilder.ServerStep
    .onRemotehost("http://localhost:9080")  StepBuilder.CredentialsStep
    .credentials( user: "a",  password: "p").build();

UserConfiguration config2 = StepBuilder.newBuilder()  StepBuilder.NameStep
    .name("aaa")  StepBuilder.ServerStep
    .onLocalhost()  StepBuilder.BuildStep
    .build();
```

```
public interface NameStep {
   ServerStep name(String name);
}

public interface ServerStep {
   BuildStep onLocalhost();
   CredentialsStep onRemotehost(String
}

public interface CredentialsStep {
   BuildStep credentials(String user,
}

public interface BuildStep {
   UserConfiguration build();
}

private static class Steps implements
         NameStep, ServerStep, Credent
     .....
```

~= Builder+State Pattern

Singleton

+other container lifecycles

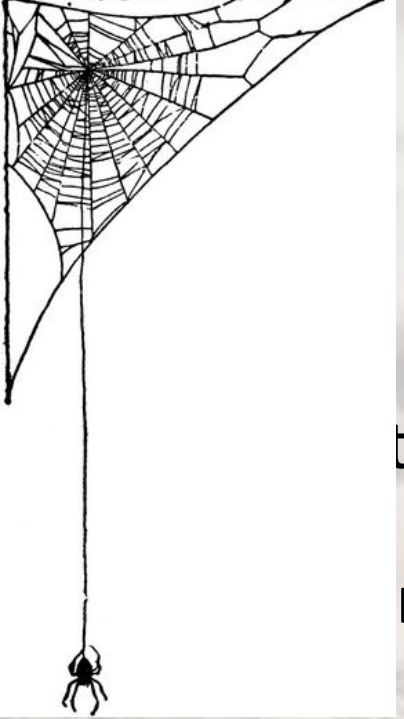
Class **A** needs an instance of class **X** (therefore X has internal state)

- What's wrong with **`new X(...)`**?
  - Not polymorphic: **unable to proxy X**; **unable to mock X** while unit-testing A
  - **Creating X might grow complex:** `new X(new XA(new Foo(...)), new XB(...))`
  - Unable to **control # of instances**. Alternatives:
    - pooling (=reuse for performance): DB Connection Pool, thread pool
    - multi-ton (=1 instance / key): `LoggerFactory.getLogger(...)`, 1 DataSource / tenant
    - singleton
- **`static`** methods
  - `X x = X.getInstance();` = the singleton *idiom* → next slide
    - **Not polymorphic**: no proxies, hard to mock
    - **Multi-threading + order of init =** 😨
  - `ServiceLocator.getService(X.class)` = a central registry managing instances →
- **DI framework** ❤️ injecting in fields / constructor of A (eg: Spring, Guice, ..) →
  - Easy to inject a Proxy or a Mock of X
  - Singleton *lifecycle* → stateless = ⚠️ fields should NOT keep user state in web APIs (for other scopes?)

Dependency Injection Rocks! 🤘

- Takes care of **lifecycle management** for you
- Less effort to **break classes**
- Easier to decouple code in **testing**
- **Proxies** can be transparently injected
- Framework calls you = **Inversion of Control** (IoC)
  - @GetMapping, @EventListener, @MessageListener, @Scheduled, ...
  - The Hollywood principle: "Don't call us, we'll call you."

"Classic" Singleton HOW



Problems

tic code

rolled dependency in Prod

- Concurrent call of getInstance() 😱
- Difficult to decouple in Tests:

m

oc

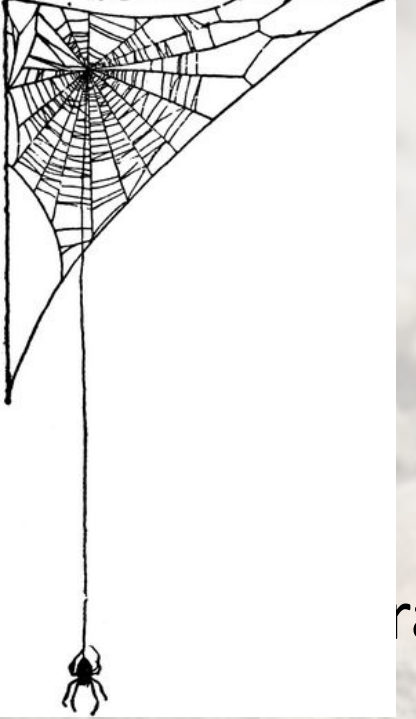
Mock Statics ☹

```
// prod code:
Singleton k =
Singleton.getInstance();
Config config = s      g();
... logic to test using config ...
```

```
public class Singleton {
  private static Singleton INSTANCE;
  private Singleton() {
    config = loadConfig(); /*300 ms*/
  }
  public static Singleton getInstance()
{
    if (INSTANCE == null) {
      INSTANCE = new Singleton();
    }
    return INSTANCE;
  }
  private Config config;
  ...
}
```

?

https://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons



Service Locator HOW

```
// At startup:
ServiceLocator.registerSingleton(Singleton.class);
```

?

Problems

- rash at runtime: not defined
- Testing: easier, buy still ugly
  - requires cleanup

```
public class Singleton {
  public Singleton() {
    config = loadConfig(); /*300 ms*/
  }
  private Config config;
  ...
```

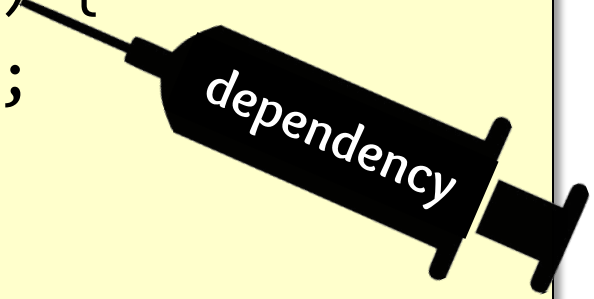
No more lifecycle management

```
Singleton s = ServiceLocator.getInstance(Singleton.class);
Config config = s.getConfig();
... logic to test ...
```

```
//From Tests
ServiceLocator.registerTestDouble(Singleton.class,
singletonMock);
```

Dependency Injection HOW ?

```
@Service  (Spring) or
@ApplicationScoped  (javaEE)
public class A {

  private final B b;

  public A(B b) {
    this.b = b;
  }
  ...
}
```



The container takes care to create only one instance

Obvious Dependencies

"Ask, and it will be given to you"

Container checks Wiring-up at Startup

Problems of Singleton Lifecycle

(however implemented)

Global State

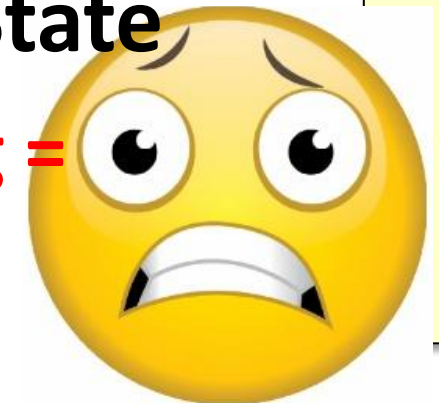
(nothing changes if I add `static`

Changes anything?

May Couple the Unit Tests

Global Mutable State

+ multithreading =



```
// only one instance is created
public class Singleton {
  private Config config;

  public Config getConfig() {...}

  public void setConfig(Config c) {
    this.config = c;
  }
```

Temporal Coupling

```
labelService.load(locale);
labelService.getCountry(iso);
```

Similar:

```
labelService.setCountryRepo(repo)
;
```

Scopes in Spring

Singleton

(default)

Shared globally

⚠️ Multithread changes
(no request data in fields)

Request

One instance per HTTP request

Used for **metadata**
user name, timezone, locale,
tenant id, correlation id..

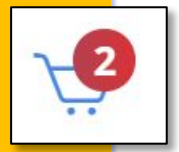
Prototype

Can keep state

Is injected and proxied

⚠️ Ask for a new instance every time

Session



Avoid if exposing REST
(use SecurityContext)

⚠️ Out of memory if many users
⚠️ Tricky if Load-Balanced

Scopes in Java EE™

@Stateless

A pool of instances

No concurrent calls on the same instance

Each call gets to a different instance

@Inject

An instance for each injection point

Get a new one via Provider<>

@Stateful

Can keep state

Dependencies can be injected

Get via initialContext.lookup()

@Singleton
@ApplicationScoped

Shared by everyone
No data in fields
Multithread ?...

CDI

Singleton - Conclusions

- Avoid programmatic style **getInstance**() or **ServiceLocator**
  - □ use Dependency Injection (eg @Autowired / @Inject)
  - Spring's @Component/@Service/... are singleton by default
- State in a singleton
  - Objects referenced by a singleton should be stateless (eg. other singletons)
  - If you need mutable state inside, careful with multithreading □

    a) redesign,

    b) **new** (give up Spring)

    c) Employ Magic: @Scope("prototype"), @Scope("thread")

Factory Class

```
OrderFactory.createOrder(): Order
```

Bloat Danger: "create" can mean anything

300 lines of angry business logic inside

Because it happens when instances are created

Factory Class

Encapsulate complex object creation

```
OrderFactory.createOrder(): Order
```

What's so complex?

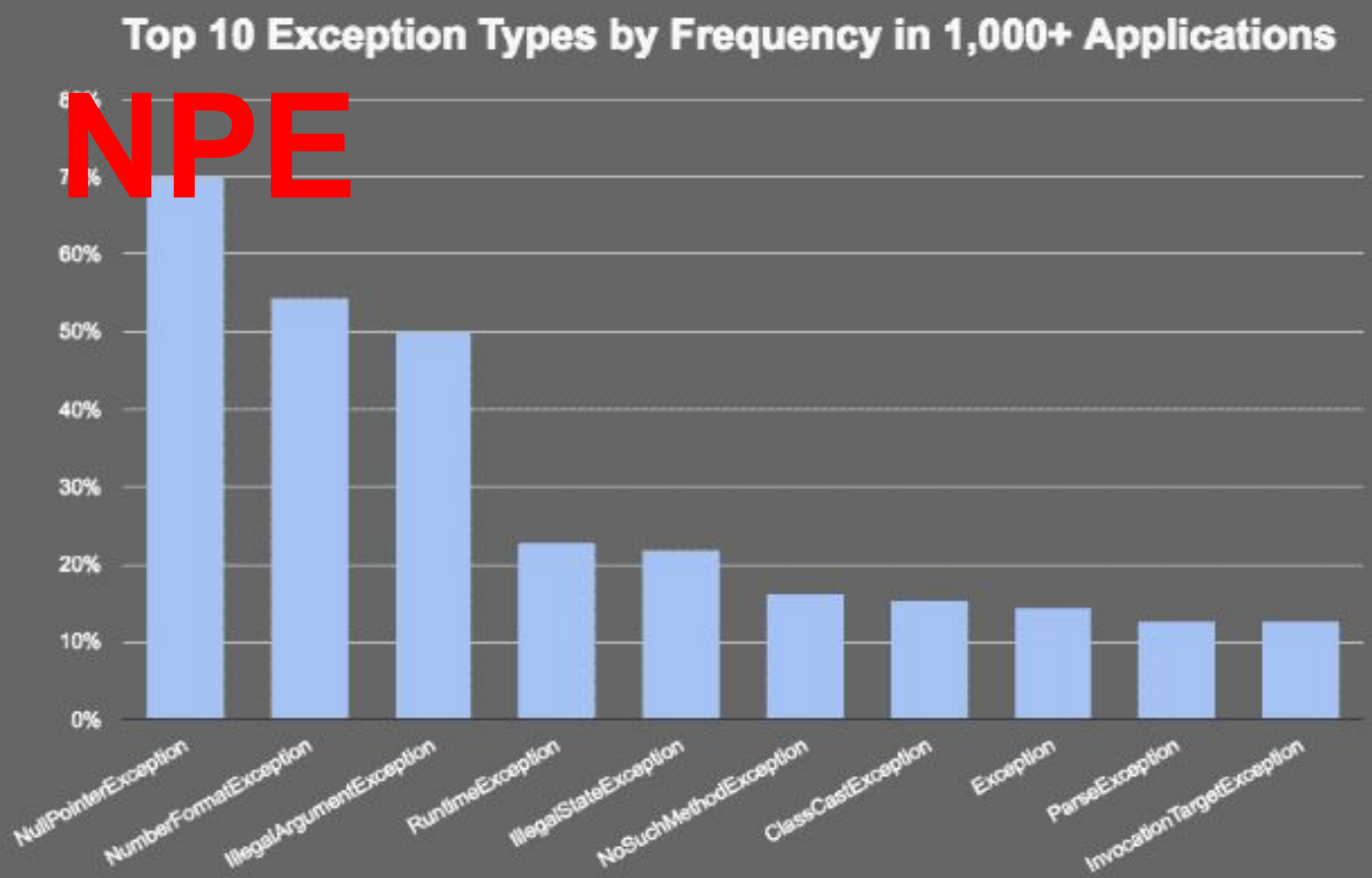
- Get from DB, Files → **Repository**
- Get via REST calls → **Adapter**/**Mapper**

Hide dependencies of created object ✓

Mandatory pre-Initialization before instance is usable ✓

Null Object

☺️ - from Emma



https://blog.overops.com/the-top-10-exceptions-types-in-production-java-applications-based-on-1b-events/

OverOps

Dealing with NULL

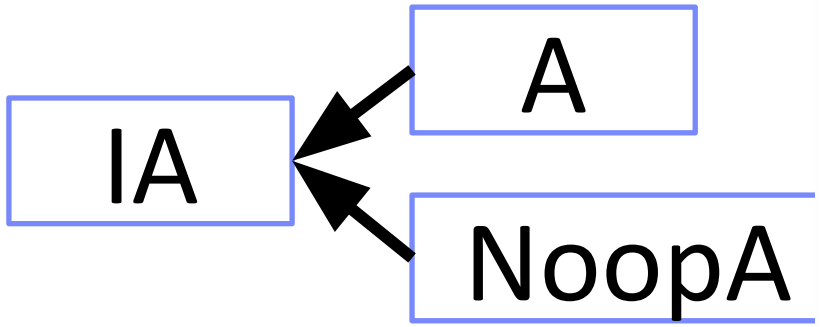
- Passing **`null`** around = **dangerous**
- Throw early for aberrant cases
  - Requires global awareness
- Return Optional<> to signal caller of absence
  - *Their* job to handle it
- Instantiate a Null Object

0 = business valid value

Data-Object: instance with default attributes
eg: MemberCard{**fidelityPoints=0**;~~**email=null;**~~}

Behavior-Object: Null Object Pattern

Null Object Pattern



1) Extract Interface

2) Reimplement to do <nothing>

Examples: dummy authenticator, noop notifier, silent logger = *side effecting object*

1. Ensure max 1 instance gets created
2. Allow to construct types you don't know about
3. Construct test data
4. Instead of multiple overloaded constructors
5. Hide a constructor with too many arguments
6. Share data throughout your entire application
7. Changing an immutable object
8. Hide complex creation or dependencies
9. Avoid nulls
0. Life-cycle: dedicated instance per HTTP request

Abstract Factory

Factory Class

Factory Method

Object Mother

Fluent Builder

Singleton

Request Scoped

Optional

Null Object

Creational Patterns

Summary

Intercepting Calls

Wrap some code around existing methods

Decorator

Proxy

Aspect-Oriented Programming

Execute-Around (FP)

```
code(T t) {
    ..
}
```

Some code
expects an object of type T

```
class T {
  m(..) {..}
}
```

Without editing the code of T,
how can I do more on T's methods?

`code(new T());`

Using **extends** ☐

`code(new TGreen());`

```
class TGreen extends T {
  @Override
  m(..) {
    ..
    super.m(..);
    ..
  }
}
```

Can I "**favor composition over inheritance**"?

But what if I want another variation (TSilent)?
(then TGreenSilent😨?)

```
code(T t) {
    ..
}
```

Some code
expects an object of type T

(T is an interface)

```
interface T {
  m(..);
}
```

```
class TImpl
      implements T{
  m(..) {..}
}
```

Decorator Pattern

Another implementation of an interface
wrapping around the original object

(favor **Composition over inheritance**)

```
code(new TGreen(new TImpl()));
```

Alternative Wrappers

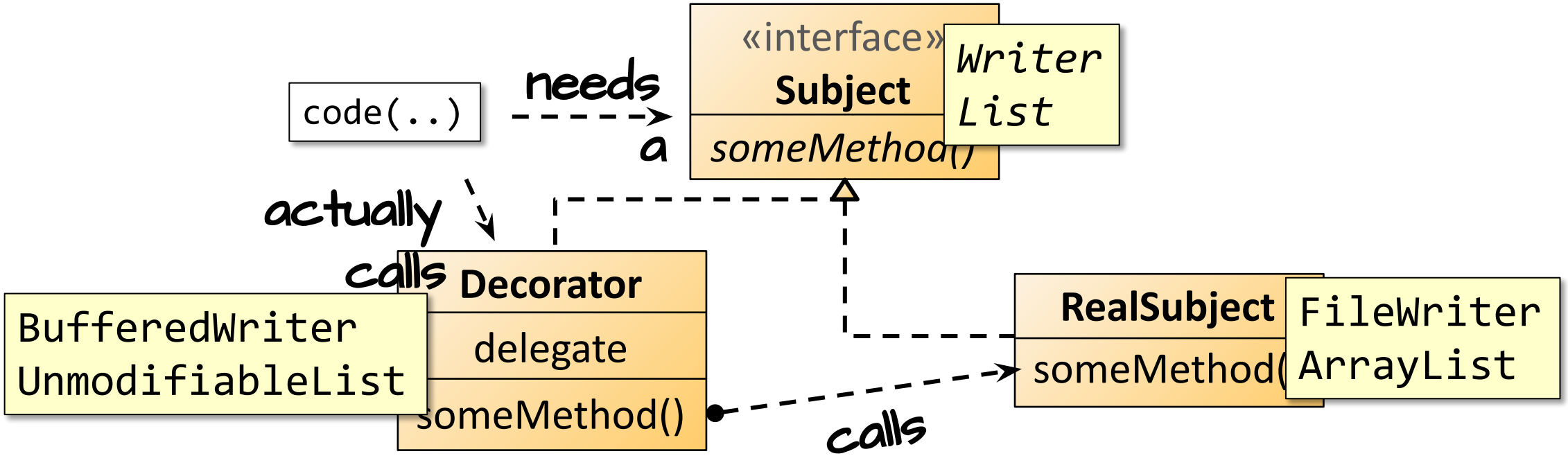
```
class TGreen //decorator
```

```
class TYellow //decoratr
    implements T {
  private T delegate;
  m(..) {
    .. delegate.m(..);..
  }
}
```

Decorator Pattern

Another implementation of an interface wrapping around the original object

to execute code before/after/instead of the original method



A Success Story

- Log the arguments passed to m1()
- The same for m2(), please
- and for m3,m4,...m30
- I want the same for **all use-cases**

You add a log.debug(...) in m1

You add a log.debug(...) in m2

🤔 ?!!...Argh!!... **copy-paste** ...

😨! ... **copy-paste** in 110 methods.

You feel 😔. For 5 minutes.

But then... next task.

Time passes by

A Success Story

Time passes by

I'll put this method here...

- **BUG: a method is not logged !!!%@^!**

- Please change the log format
- **BUG: you missed a place !!!%@^!**

Oh boy!.. 109 changes.

Done!

I ❤️ my job!😝

A Success Story

- Log the arguments passed to m1()
- The same for m2(), please
- and for m3,m4,...m30
- I want the same for **all use-cases**

You add a log.debug(...) in m1

You add a log.debug(...) in m2

🤔 ?!!...Argh!!... **copy-paste** ...

😨! ... **copy-paste** in 110 methods.

You feel 😔. For 5 minutes.

But then... next task.

Time passes by

~~copy-paste~~

D R Y

Don’t Repeat Yourself

<> **W**rite **E**verything **T**wice

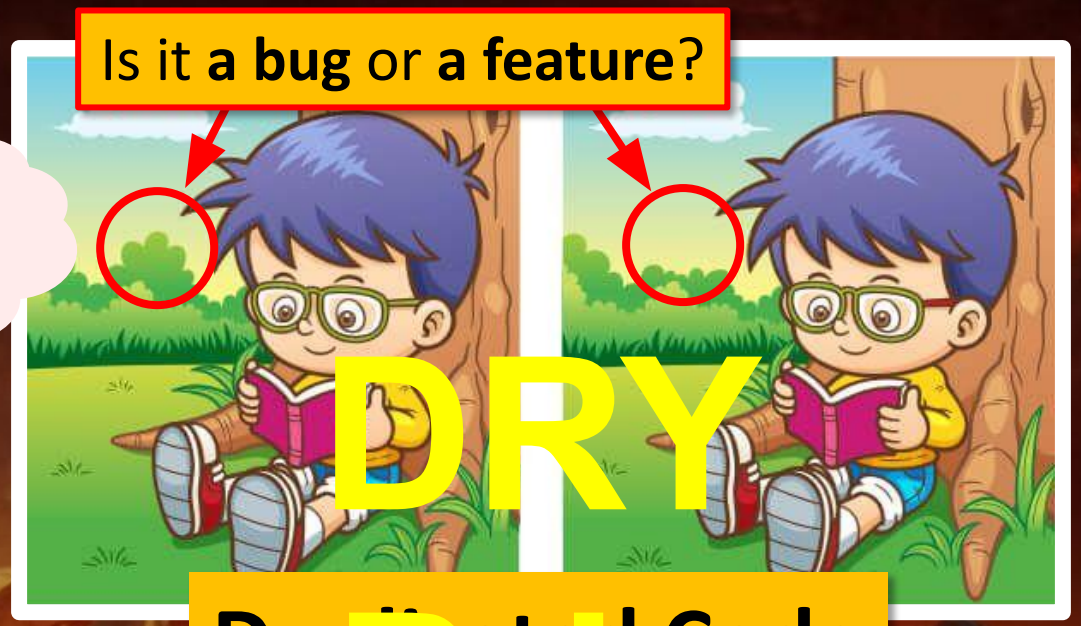
We **E**njoy **T**yping

BTW, this is code copy-pasted 2 years ago

Find 5

differences !

Donno!
Biz forgot,
and
the author
left



DRY

Duplicated Code

becomes terrible when it changes

Prin

cipl

e

In such dark moments, people turned to magic...

Aspect-Oriented Programming

Let's magically intercept method calls...

to implement such

“Cross-cutting concerns”

(idiomatic code repeated in many places)

Logging
Transactions
Access Control... .. ..

Aspect-Oriented Programming

Idiomatic code:

```
start transaction
try {
   <arbitrary code>
   commit transaction
} catch (e) {
   rollback transaction
}
```

Implement the same steps
(cross-cutting concerns)

"around" arbitrary code

without copy-paste

Java with Spring

```
@Transactional
public void f() {
   <arbitrary code>
}
```

The Magic of Spring

Every time you don’t understand how Spring does something…

it's a Proxy

```
@Transactional
@Cacheable
@Valid
@Aspect
@Async
@Secured
@Timed
@Retry @RateLimiter @Bulkhead @CircuitBreaker
@Scope
Mockito
```

Custom Spring Aspect (example)

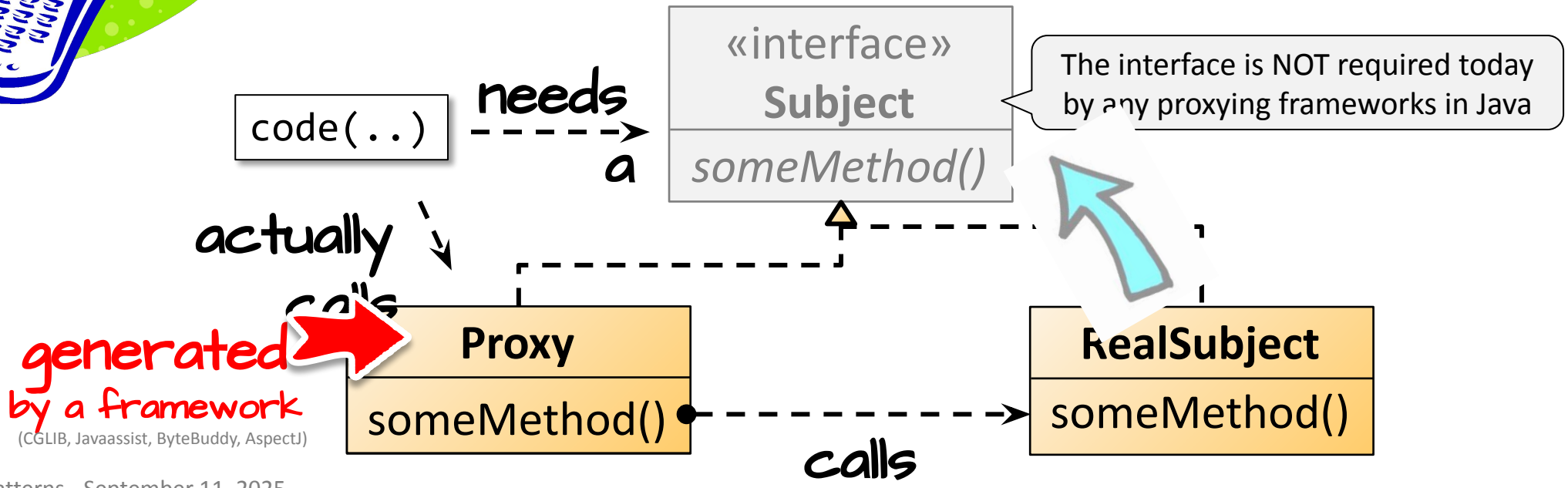
```
@Aspect
@Component
public class LoggingAspect {

   // intercept all methods annotated with @LoggedMethod
   @Around("@annotation(my.annot.LoggedMethod))")
   public Object logAround(ProceedingJoinPoint joinPoint) {
      String methodName = joinPoint...;
      Object[] arguments = joinpoint...;
      log.debug("Invoking {}(): {}", methodName, argList);
      return joinPoint.proceed(); // call intercepted method
   }
}
```

Proxy

Proxy

placeholder intermediating interactions with an object

- The caller is given a Proxy to **intercept its method calls**, to do:
  - (1) Aspect Oriented Programming
  - (2) Remote calls (RMI, Web Services, Feign clients, Spring Data)



Interface Proxy

```
class A {
  private B b;
  ...
}
```

has a

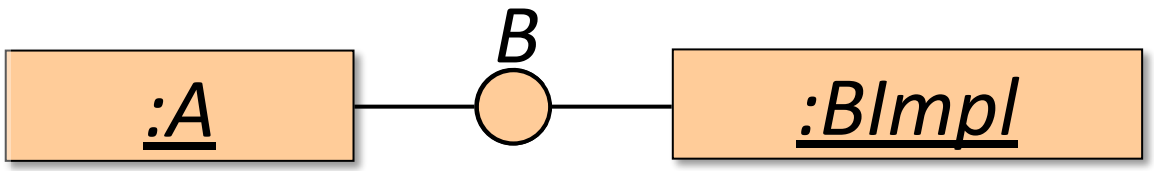
```
interface B {
  m(); ...
}
```

implem

```
class BImpl
  implements B {
  m() {} ...
}
```

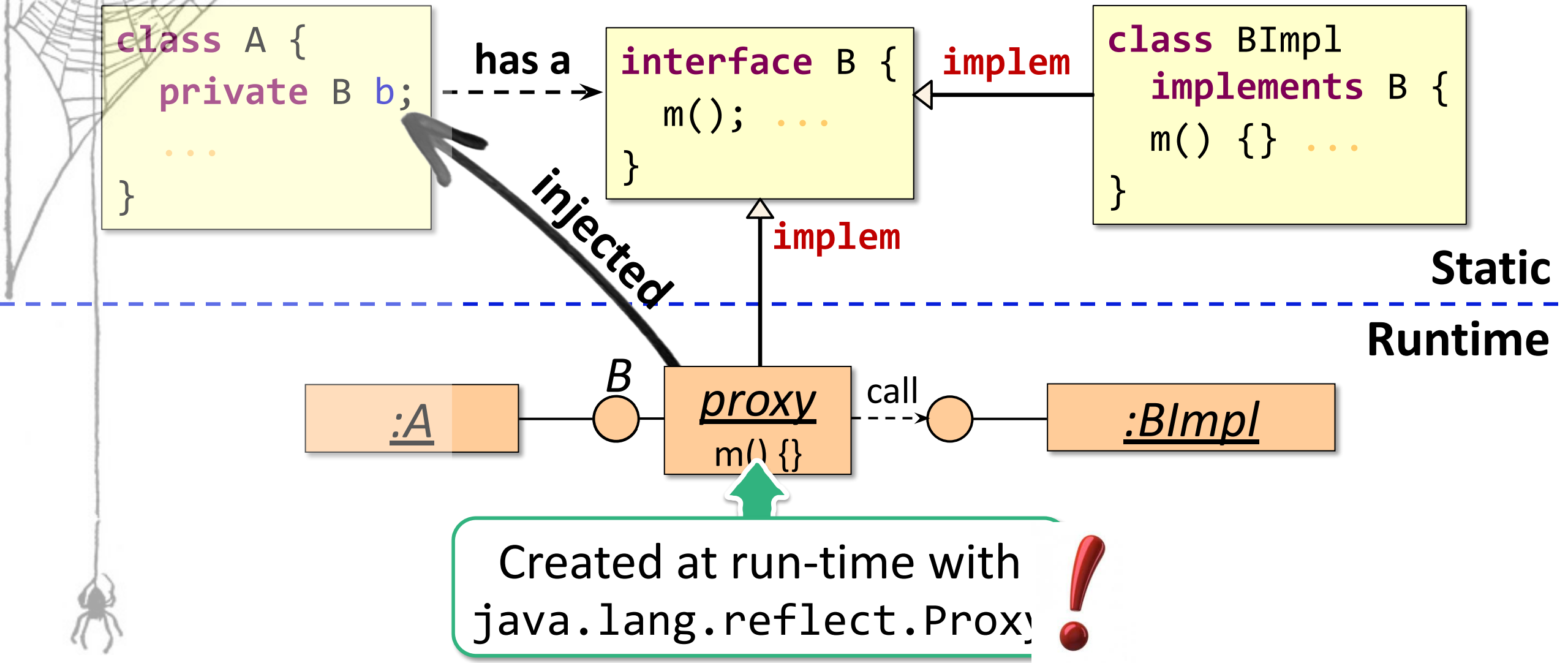
Static

Runtime



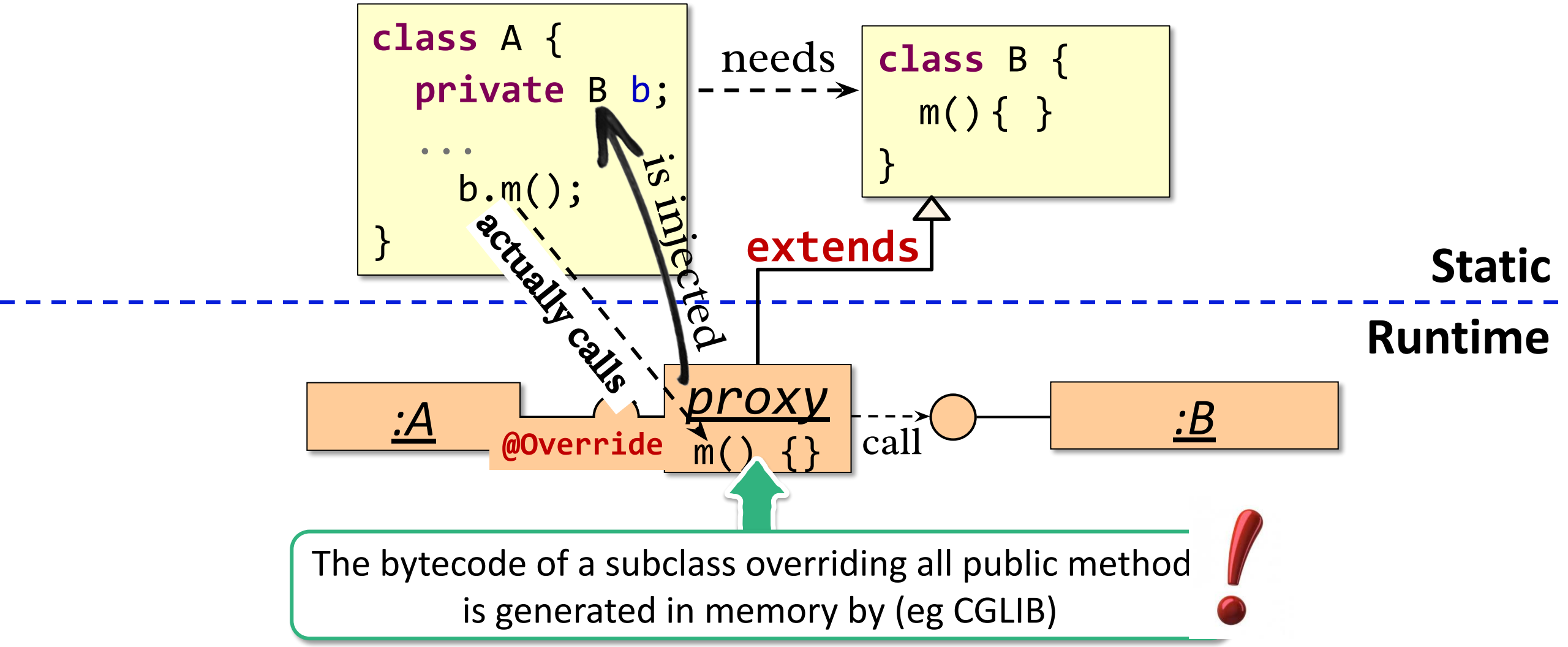
By default, Spring will only use class proxies (see next slide)

Interface Proxy

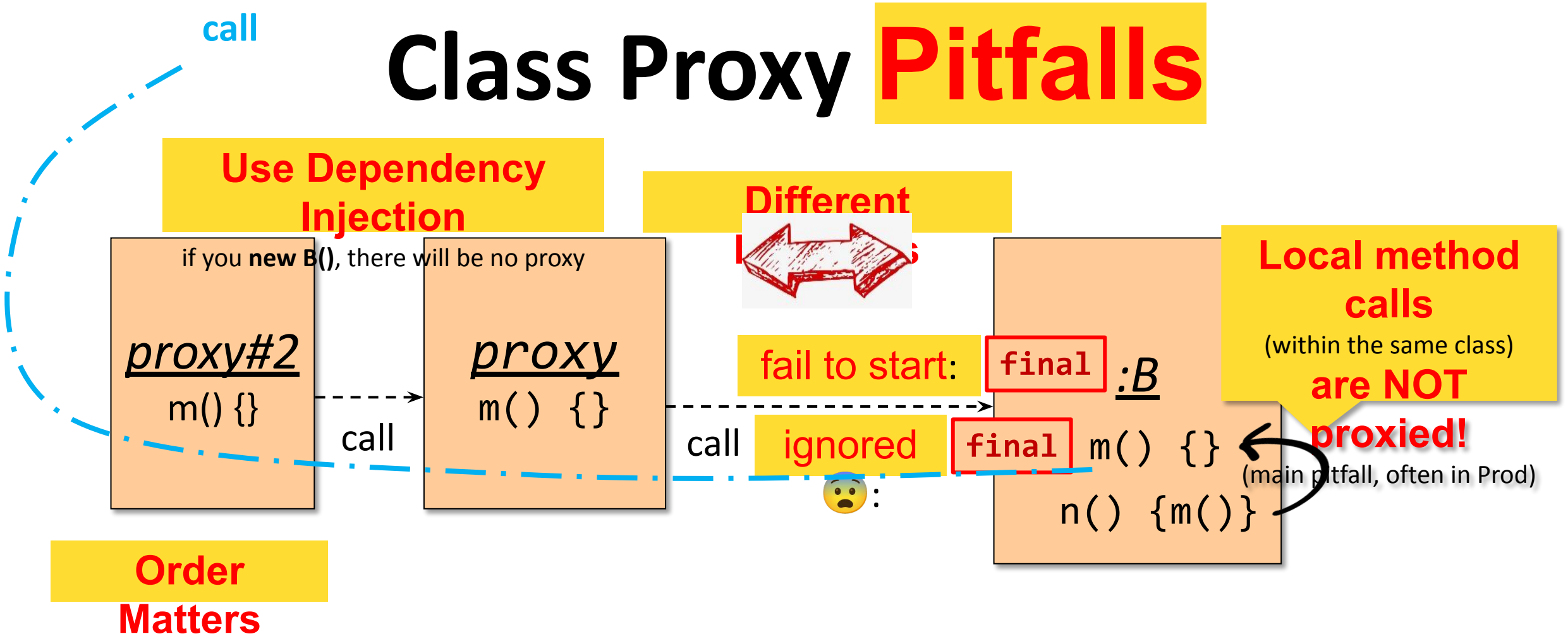


eg: Spring Data Repos, Feign Clients, JAXWS Port interfaces, RMI remote interfaces

Class Proxy


class A {
private B b;
...
b.m();
}
needs
class B {
m(){ }
}
extends
is injected
actually calls
Static
Runtime
:A
@Override
proxy
m() {}
call
:B
The bytecode of a subclass overriding all public method is generated in memory by (eg CGLIB)

Class Proxy Pitfalls



A word to "hackers":
you can actually "fix" all the above problems using black magic (DON'T !):
bytecode instrumentation/enhancement

Want More?

Talk: Proxy Fairy and the Magic of Spring

Due to their invisible nature, AOP is sometimes considered **black magic**


AO


AO
P

the **Functional Programming**
alternative

the Functional Programming alternative

Execute-Around Pattern

```
void measure(Runnable r) {
  long t0 = currentTimeMillis();
  r.run();
  long t1 = currentTimeMillis();
  log.debug("Took {} ms", t1-t0);
} // PS: use @Timed
```

'decorate' existing logic manually

```
measure(() -> savePage(recordsPage));
```

```
runInNewTransaction(() -> saveRequest());
```

```
void runInNewTransaction(Runnable r) {
  start tx
  try {
    r.run();
    commit tx
  } catch (..) {
    rollback tx;
  }
} // PS: use TransactionTemplate
```

+ Simpler, if applied in a few places

+ No (black) magic

- One more level of indentation

Intercepting Calls: Key Points

- Alter the behavior of existing methods without editing them
  - With no impact on their public API

- **Decorator**: allow clients to optionally alter the behavior a class
  - `eg. unmodifiableList(list);  new BufferedReader(new FileReader());`
- **Proxy**: generated by a framework to implement **AOP** (⚠️pitfalls)
  - `eg. @Transactional`
- **Execute-Around**: pass the function into a new one to run it
  - `eg. measure( () -> stuff() )`

	 Adapter	 Proxy Decorator 	Facade 
Purpose	Wrap a foreign component	Intermediate interactions	Higher level API of my system
Exposed API	Nice for <u>my application</u>	Same as Real Subject (fake it)	Nice for <u>my users</u>
Nuances	**Anti-Corruption** Defend the rest of your code	**AOP Proxy** Execute code before/after/instead the actual call	**Emerging Design** Gradually extract what grows

What do you mean ?

How?

Huh!? Then I extract? That's it?

Piece a cake!

Extract when it Grows

When a class grows too big (>~200 lines?) ▯ break it

Look for a good class name that summarizes some of its methods

Exactly!

He-he!☺️

*"There are only two things **hard** in programming: Cache Invalidation and **Naming Things**"*

SRP

applied to reduce complexity

not to generate 10-lines classes

Extract when it Grows

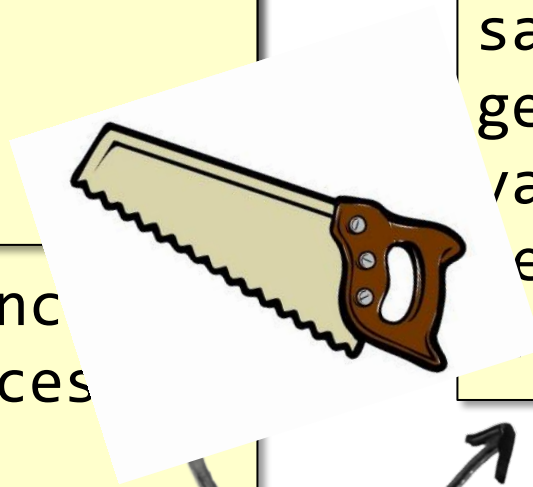
Horizontal Splitting

Control
ler

CustomerOpsFacade

```
saveCustomer()
getCustomer()
searchCustomer()
```

```
saveCustomerPreferenc
getCustomerPreferences
validateAddress()
resetPassword()
checkPassworStrength()
```



CustomerPreferencesFacad
e

```
saveCustomerPreferences()
getCustomerPreferences()
/alidateAddress()
esetPassword()
eckPassworStrength()
```

at the same level of abstraction

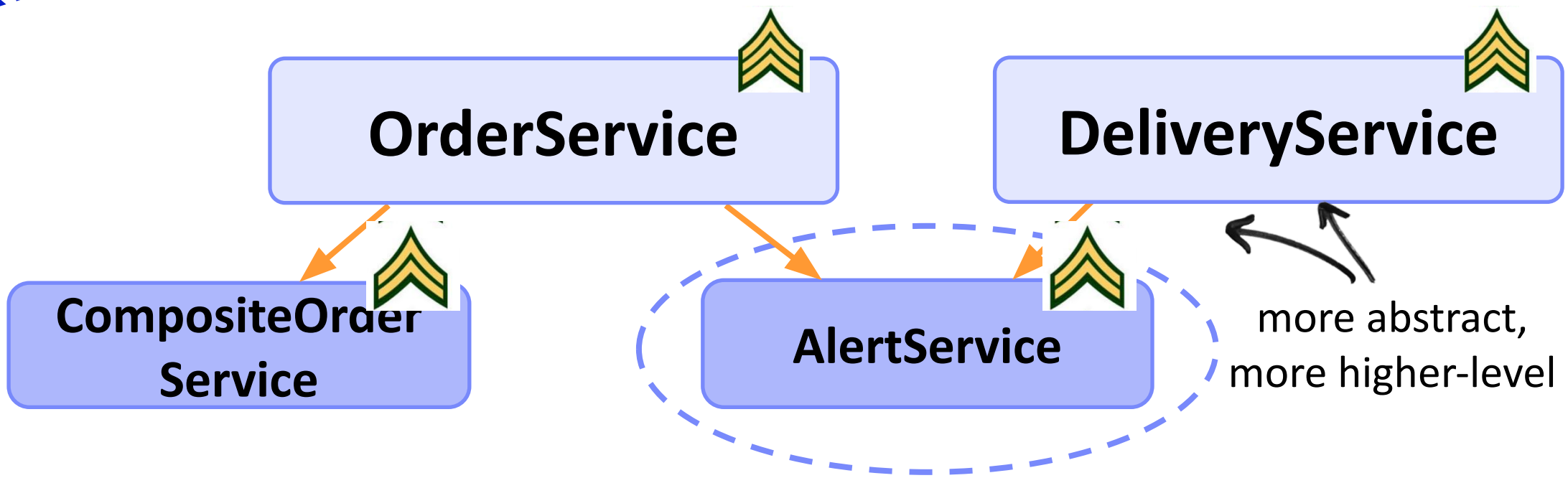
> 300 lines

Extract when it Grows

Vertical Extraction

KISS

DRY



Separation by Layers of Abstraction

Propagating Change

When a method call is not enough

Command (Queues)

Observer (Events)

A Scalability Problem...

https://www.youtube.com/watch?v=3R5J8Je2j_4

Why would a method call() not be enough?

- You want to be able to "undo" / "redo" the action (GUI)
- You want to keep track of each action (robustness)
- Too many concurrent calls might overload a system (throttling)
- Enqueue, merge and optimize actions (Gaming)

Command

encapsulate and handle a call as an object

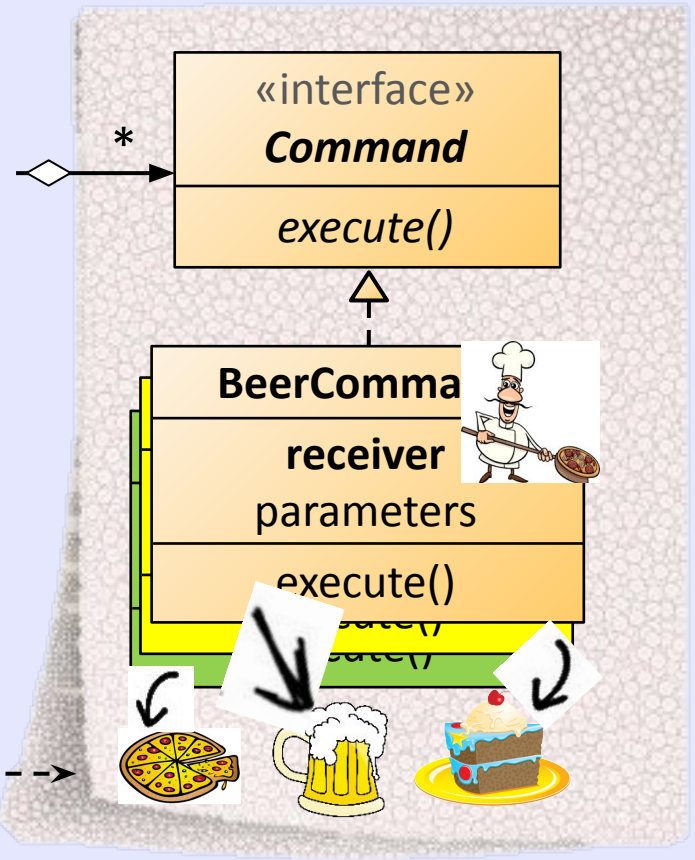
Queues the commands,
and executes them in turn,
maybe on a thread pool
maybe on remote nodes

GUI actions (to provide "undo"/"redo")
Thread Pools (parallelize / throttle)
Messages on Queues (robust) vs REST call

receiver

submit commands

invoker



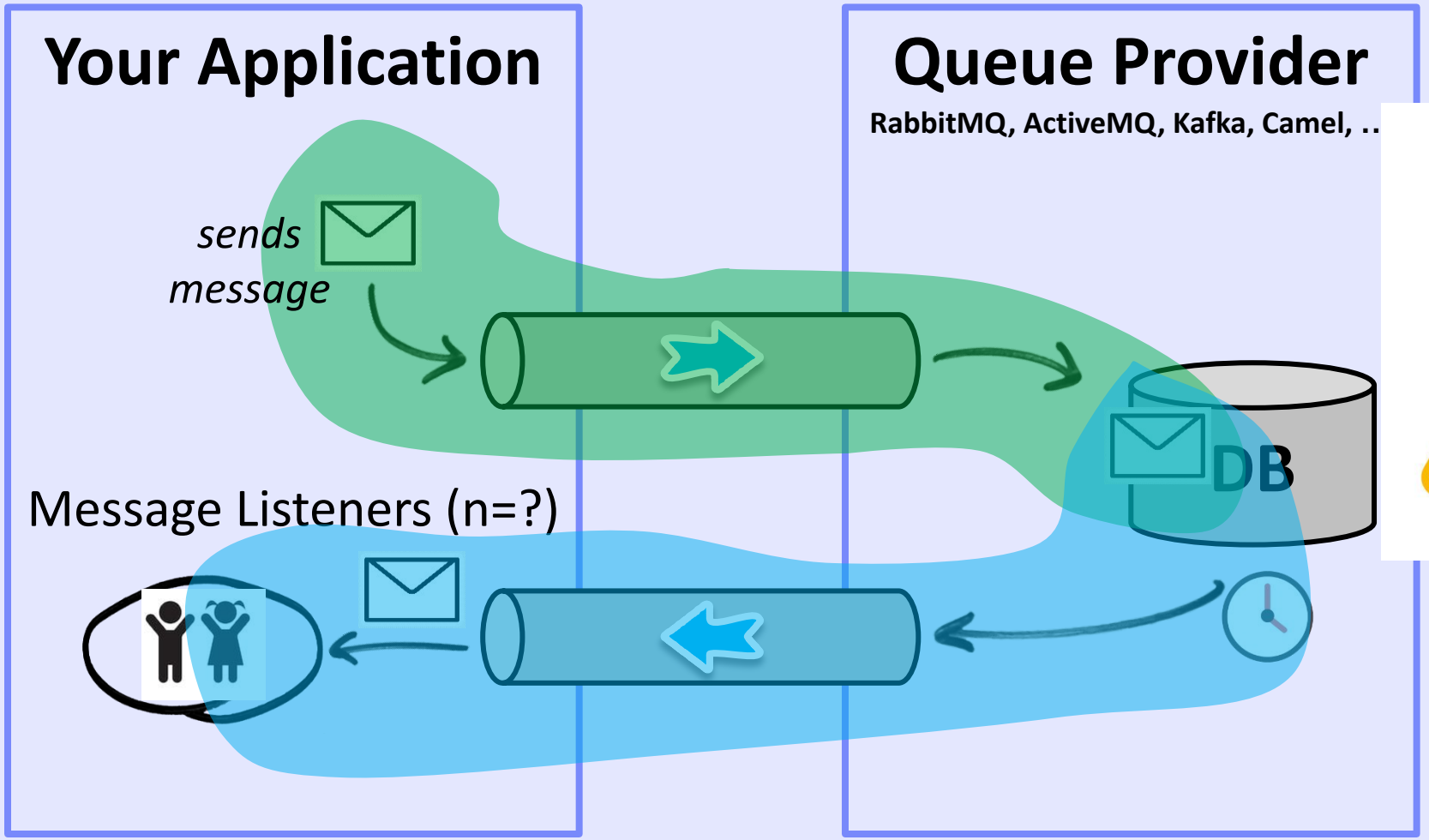
gives command

client

crea
te

Tailing your own queue

Sending Yourself a Command

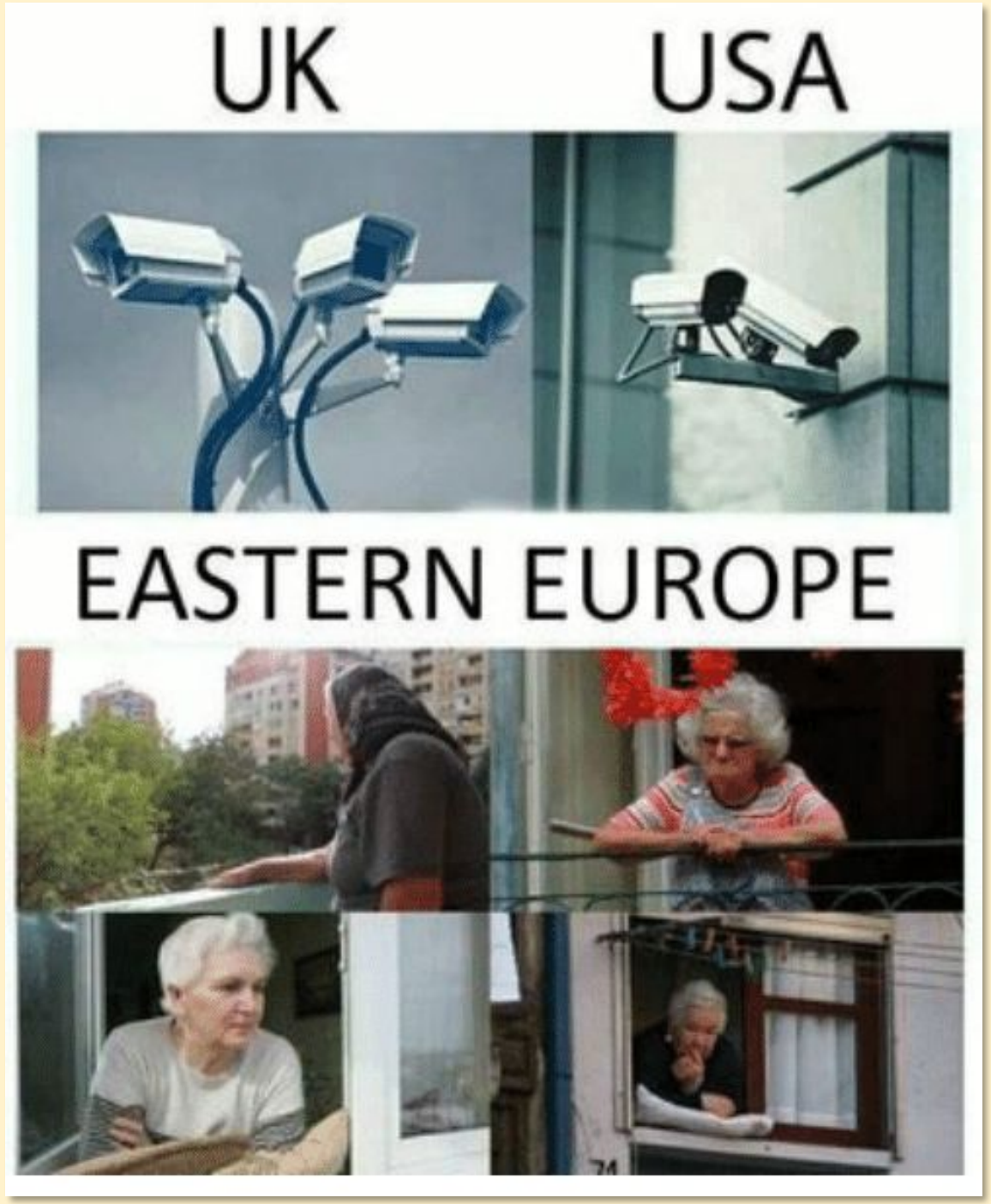


of messages
can be transacted

https://stackoverflow.com/questions/40749877/rabbitmq-sending-message-in-transaction

= Robust Async Calls

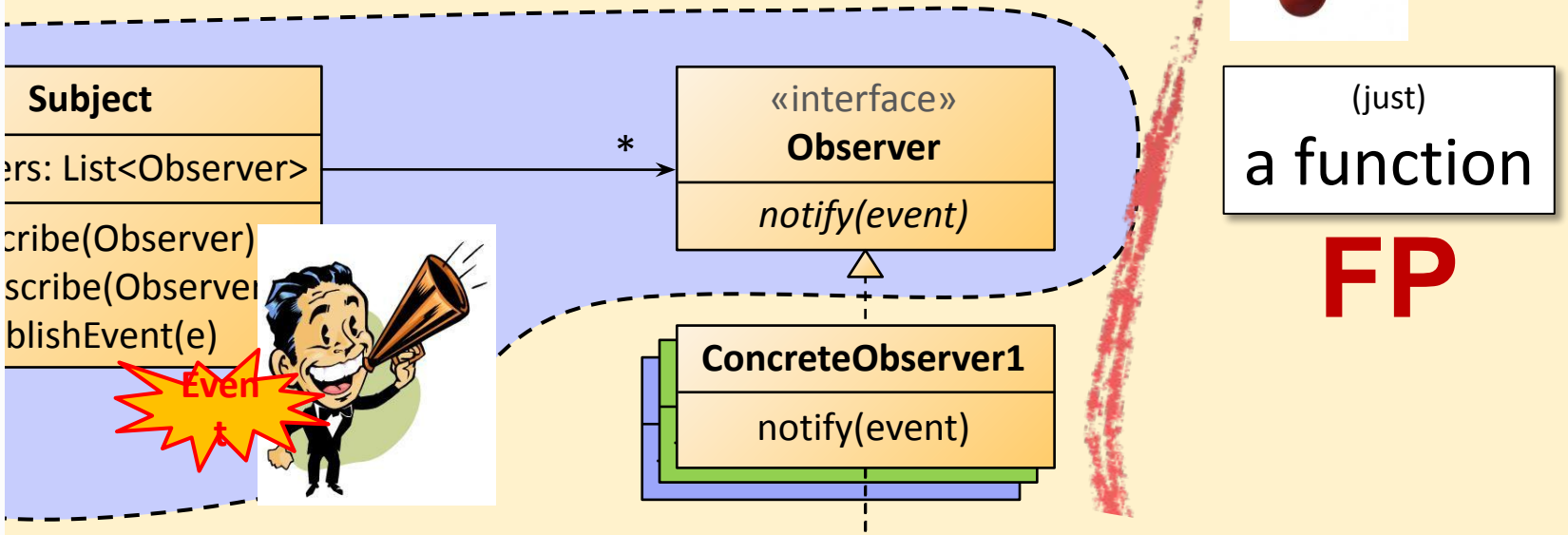
Observer



Observer

notify interested listeners when events occ

Subject doesn't know the concrete **Observer** implementations



(just)
a function

FP

- **UI Event Handler**: `jButton.addActionListener(event -> {...});` `<button onclick="stuff()"/>`
- **Framework Hooks**: `@EventListener(ApplicationStartedEvent) public void onStart() {...}`
- **Message Topic**: "OrderPlacedEvent" ✉ 

Sending Events

notify interested listeners when events occur

Spring Events

extremely loosely coupled Observer for intra-app events

```
@Autowired
private ApplicationEventPublisher publisher;

    ...

    publisher.publishEvent(new
OrderPlaced(13));
```

arbitrary POJO

Event

By default:

- Unspecified handing order (@Order)
- Sequential on same thread (different)
- Same transaction (in a separated Tx)

```
@EventListener
public void audit(OrderPlaced event) {
```

event chaining

```
@EventListener
public OrderInStock handle(OrderPlaced event)
{
    ...
    return new OrderInStock(event.orderId);
}
```

(ab)Using @EventListener with no clear decoupling goal can lead to messy untraceable code (more, later)

Observer: Key Points

- Goal: **decouple** publisher from listeners
  - "I don't want to know what happens because of what I do"
  - Often, listeners are also unaware of the publisher
- Usage: UI, framework hooks, inter-app messaging (topics)
- Multiple listeners can receive the same event
- Order of listeners matters!
  - Chain events
    - Propagate a "Trace ID" over messages for traceback (see Spring Sleuth)

Forking Behavior

Because sometimes we find different ways of doing something

Strategy

Chain of Responsibility (aka **Filters**)

Template Method

Loan Pattern

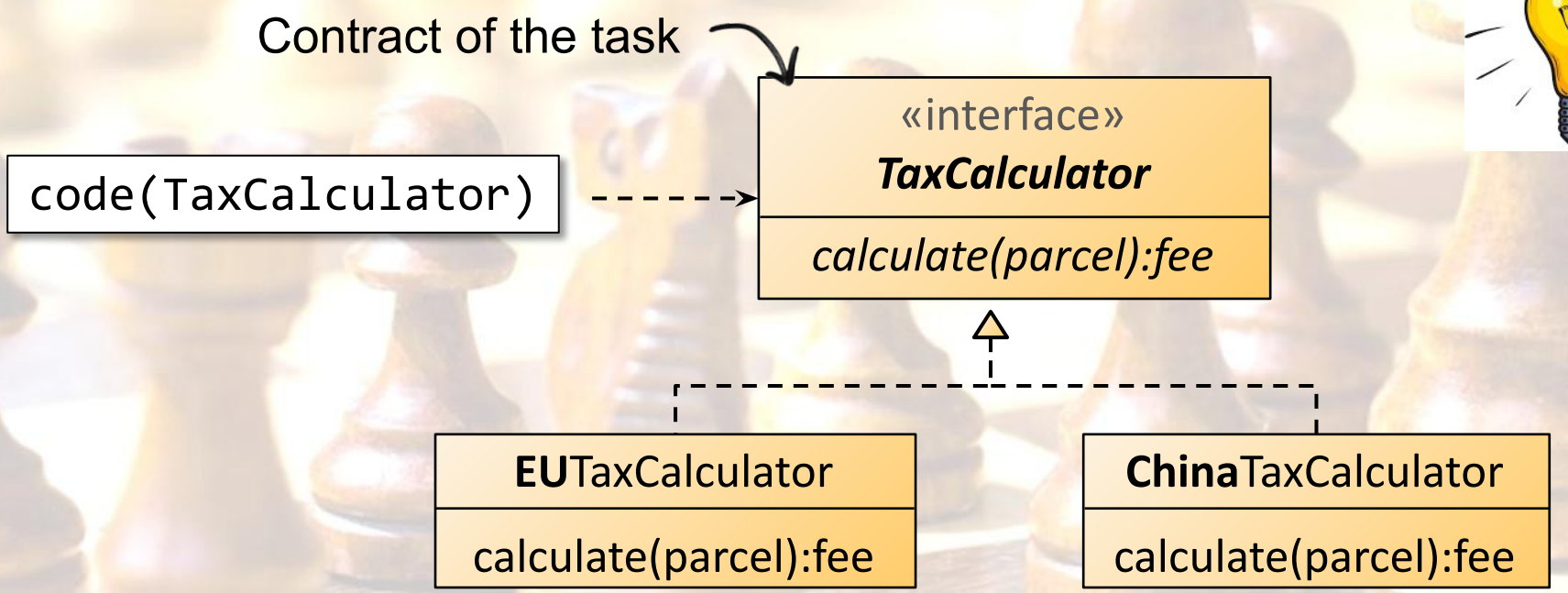
Visitor

State

Strategy

alternative ways of doing a task implement the same interface

eg: sorting, heuristics, handling a message, sending a notification, loading config...



Plugins (**OCP**)

Complex Logic

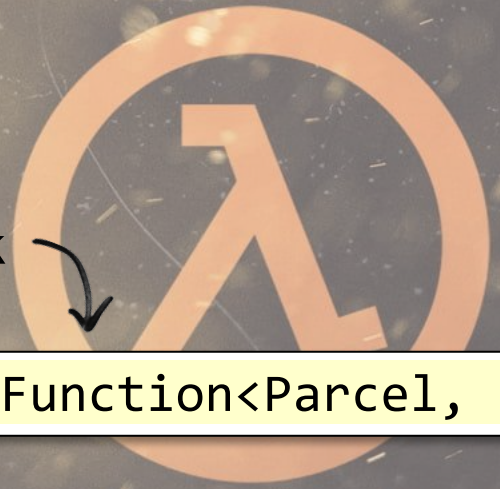
simple?

Functional Programming

functions are first-class citizens

Lightweight Strategy (FP)

pass alternative ways of doing a task as a function



Decision which function/strategy to use is centralized

Contract of the task

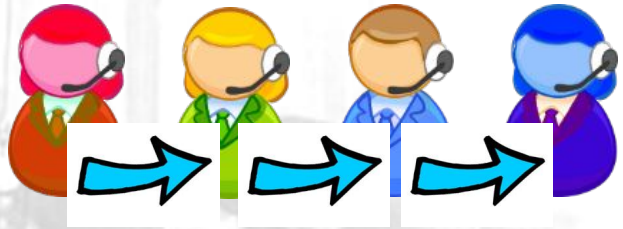
```
code(Function<Parcel, Fee>)
```

calculate**EU**Tax(Parcel):Fee

calculate**China**Tax(Parcel):Fee

Decentralize selection

Chain of Responsibility



complex

each filter can decide to process the request, or not

Examples: Security Filters, HTTP Request/Response Filters on Client/Server, ...

Extremely Low Coupling

A filter should not care who follows them

Multiple filters can execute

☢ Order Matters

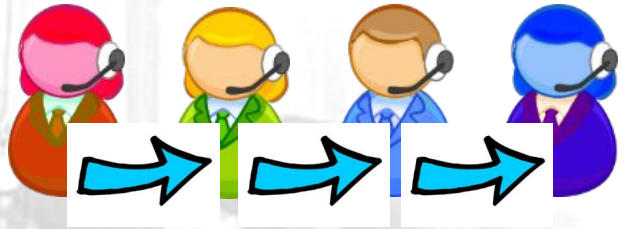
if multiple filters can apply

 filter order is usually set in a config

Decentralize selection

Chain of Responsibility

Implementation Forms



```
interface Filter {
   boolean accepts(Work);
   Result/Work process(Work);
}
```

```
interface EventHandler {
   // event.setDone(true); when done
   void handle(event);
}
```

```
interface Filter { // servlet
   void doFilter(
    request,
    response,
    next: FilterChain);
}
// next.doFilter(req,res)
```

```
interface OptionalFilter {
   // return empty when not supported
   Optional<Result> compute(params);
}
```

```
interface ChainedFilter {
   output doFilter(input, next:Filter);
}
```

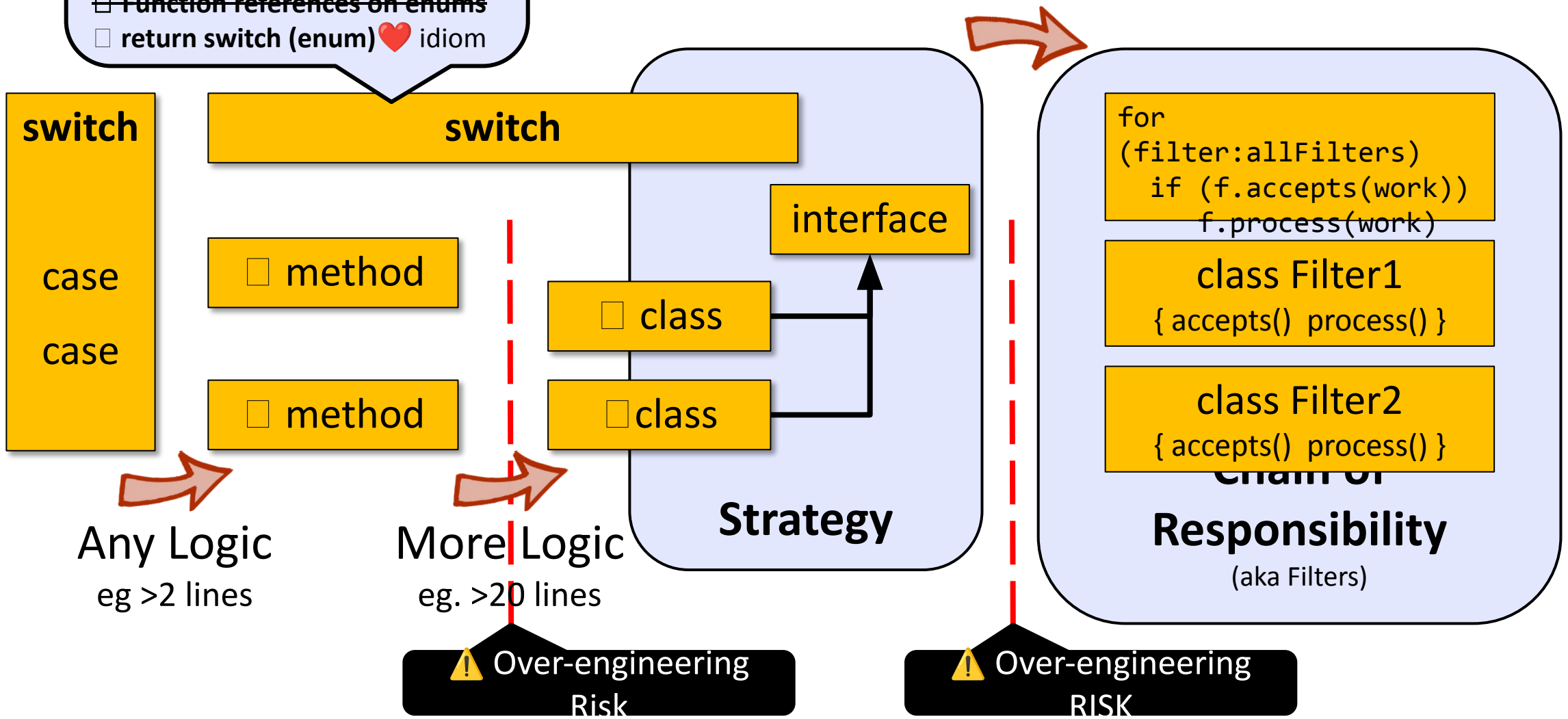
Others? ....

Recap

Large/Volatile Mapping
- **externalize** to .properties/.yaml

Risk of missing a case
- ~~Abstract **functions on enums**~~
- ~~**Function references on enums**~~
- **return switch (enum)**❤️ idiom

Use if
+ decisions to act or not is **complex**
+ **multiple filters** can act
+ add/shift filters without code impact



A way to write code; a hack, not really a pattern

Functions on Enums Idiom

Better languages need less patterns to correct their flaws

pre-java17

= Keep a function in a field of the enum

Why? The compiler checks that each enum value specifies a function

Cons: coupling the enum to a logic class

Trick: to reference instance functions, keep a BiFunction references (see below). Scary

Alternative: abstract methods in enum

- useful for static small functions

```
enum MovieType {
  REGULAR(Prices::regular),
  NEW_RELEASE(Prices::newRelease);

  BiFunction<Prices,Integer,Double> priceFun;

  MovieType(BiFunction<Prices,Integer,Double> f) {
    this.priceFun = f;
  }
}
```

```
@Component
class Prices {
   @Value("${new.release.factor}")
   private int factor;

   double regular(int days) {
      return days + 1;
   }
   double newRelease(int days) {
      return days * factor;
   }
   public double computePrice(
    MovieType type, int days) {
      return type.priceFun.apply(this, days);
   }
}
```

"return switch (enum)" Idiom

In case we add an enum value

```
enum MovieType {
  REGULAR,
  NEW_RELEASE
}
```

```
public double computePrice(MovieType type, int days) {
  return switch (type) {
    case REGULAR → days + 1;
    case NEW_RELEASE → days * factor;
    default → throw new IllegalArgumentException(type);
  };
}
```

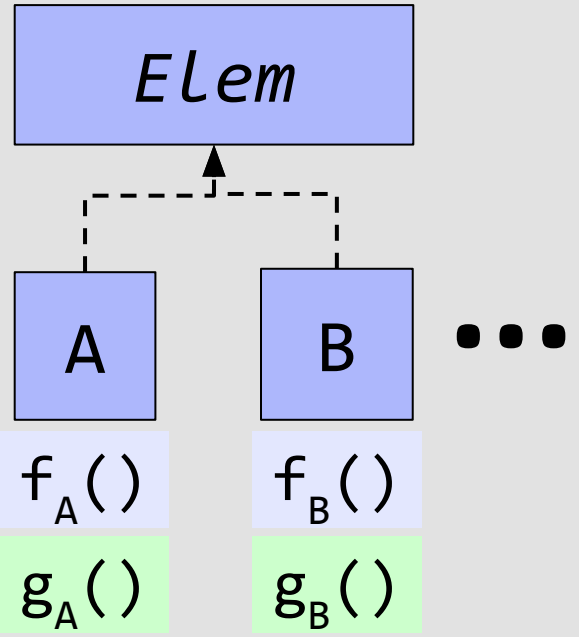
avoid!

A **`switch`** expression (returning a value)

fails compilation if you missed one of the enum values

Available since Java17, Kotlin, Scala, C#8, php8

Forking by Subtype



- You have a hierarchy of elements
  - eg. Types of Contracts, Geo Region>Country>Site,
- You process an *Elem*, depending on its type:
  - eg. compute the price of contracts, exporting Geo nodes in XML

Ideas ?

- OOP solution: an abstract method `f()` in *Elem* 😌

NO! - the architect

- Why **NOT** pushing logic inside the data object:
  - Avoid **polluting the model**: eg formatting as XML in the model (**DIP**)
  - Too **large element classes** (A, B..)
  - Too **heavy logic**: eg. $f_A() + f_B()$ > 100 lines
  - Need to **share code** between $f_A()$ and $f_B()$ □ group logic *by feature* and not *by element type*
  - Support **adding more behaviors** (eg. $g_A()$ and $g_B()$) with minimal impact in code

Then what?

Forking by Subtype

```
if (e instanceof A) {
  fA((A)e);
} else if (e instanceof B) {
  fB((B)e);
} ...
```

Any Risks ?

😨 risk to forget an element type

(C)

Visitor

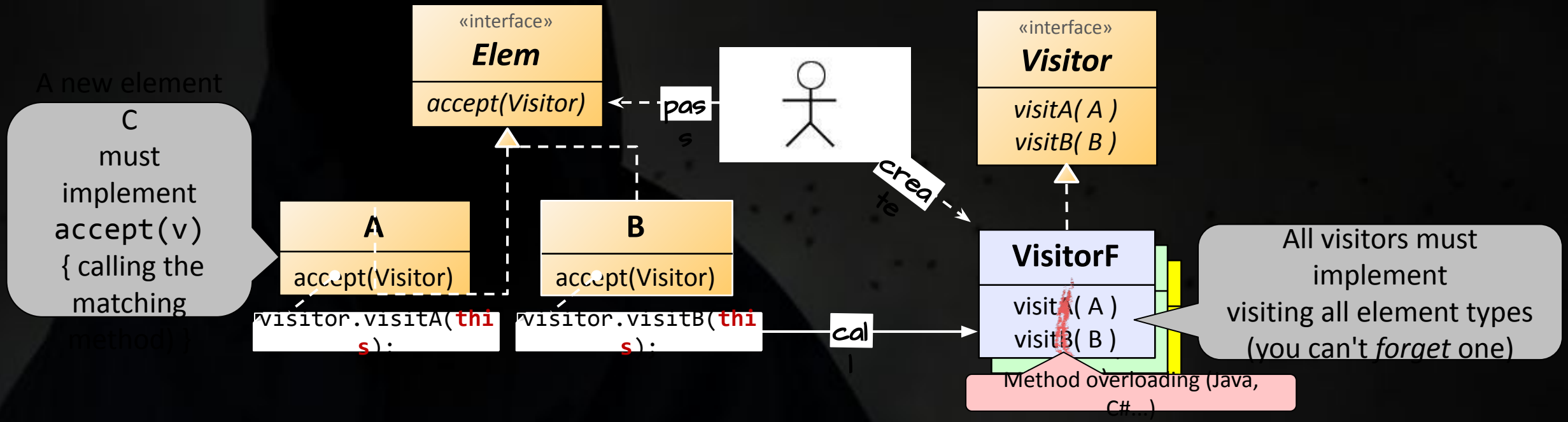
Sealed Classes

(Java 21*, Kotlin, Scala, ..)


Visitor

Visitor

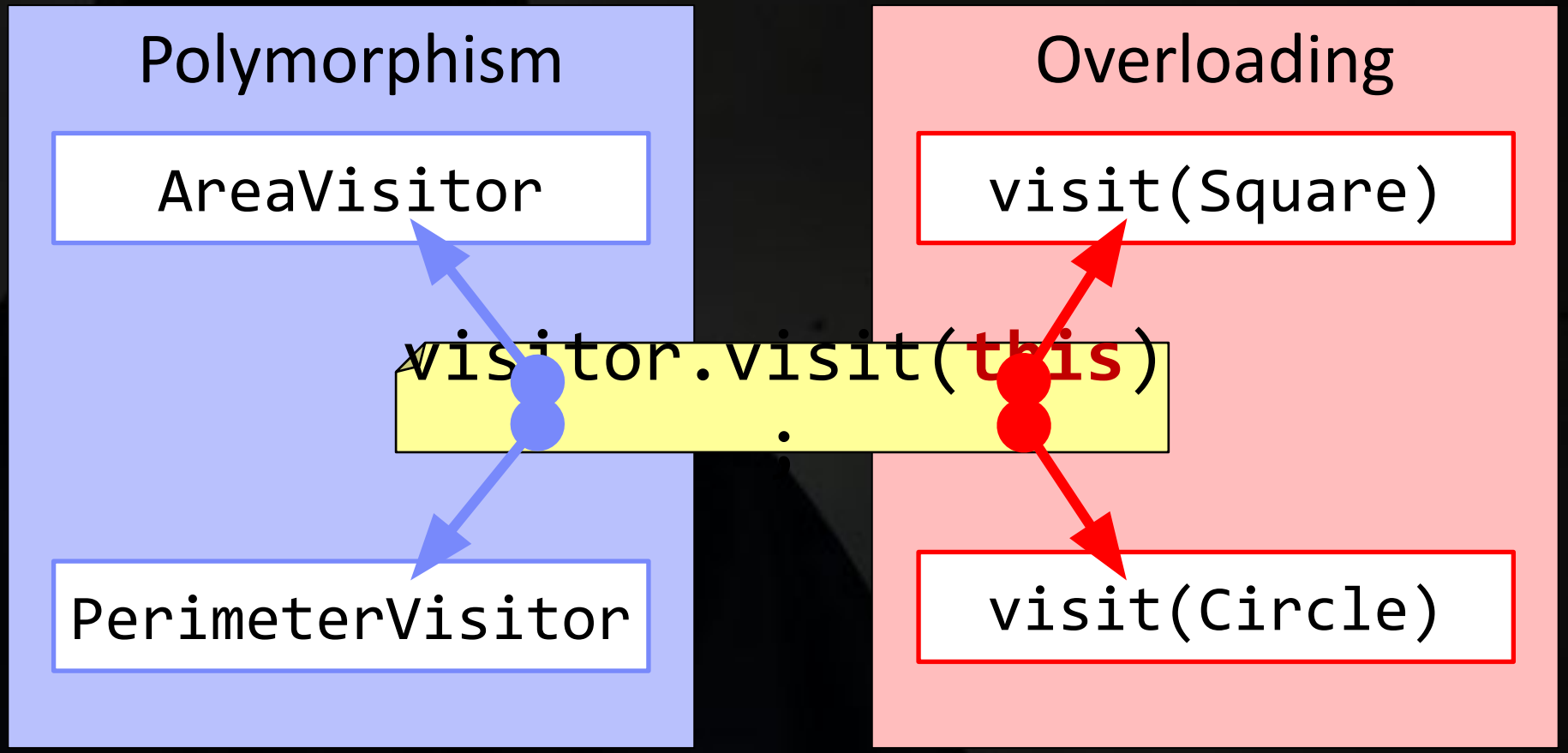
encapsulate an operation on different element types



Allows adding operations on elements without editing their classes

Visitor

Double-Dispatch



Sealed Classes

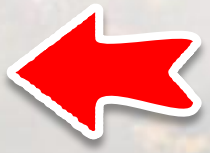
(Java 21*, Kotlin, Scala, ..)

```
sealed interface Shape
   permits Circle, Square
}

record Circle(int radius)
   implements Shape {
   ...
}

record Square(int edge)
   implements Shape {
   ...
}
```

Better languages need less patterns
to correct their flaws

```
double f(Shape shape) {
   double perimeter = switch (shape) {
      case Square s -> s.edge() * 4;
      case Circle c -> c.getRadius() * PI * 2;
                  // no need for default 😲
   }; // compilation fails if you missed a subtype
}
```

```
m(a, b, c, d, ...) {
    ...
     ...
    ...
       ...
              ...
          ...
       ...
      ...
             ...
    ...
} // line 531 😩
```



Loan Pattern

Easier testing

👌

Test with a dummy format
(check the file contains "X")

temporal coupled

```
void export(..., Consumer<Writer> writerFun) {
  try (Writer writer = new FileWriter(...)) {
    writerFun.accept(writer);
  } catch (...) {
    ...
  }
}
```

infrastructure manages a Writer and "loans" it to the writerFun()

resource management

export formats

```
export(..., w -> writeOrdersOn(w));
export(..., w -> writeUsersOn(w));
```

Test using a StringWriter
(check the written string)

```
void writeUsersOn(Writer writer) {
  // work with the "loaned" writer
}
```

What else can you loan: eg. a (Kafka) message to ack after, a transaction, a connection, a ThreadLocal

Loan Pattern

💡 Used since 2001 (with a cumbersome

```
List<User> users = jdbc.query("SELECT id, name FROM users",
    new RowMapper<User>() {
        @Override
        public User mapRow(ResultSet rs, int rowNum) {
            User user = new User();
            user.setId(rs.getLong(1));
            user.setFullName(rs.getString(2));
            return user;
        }
    });
```

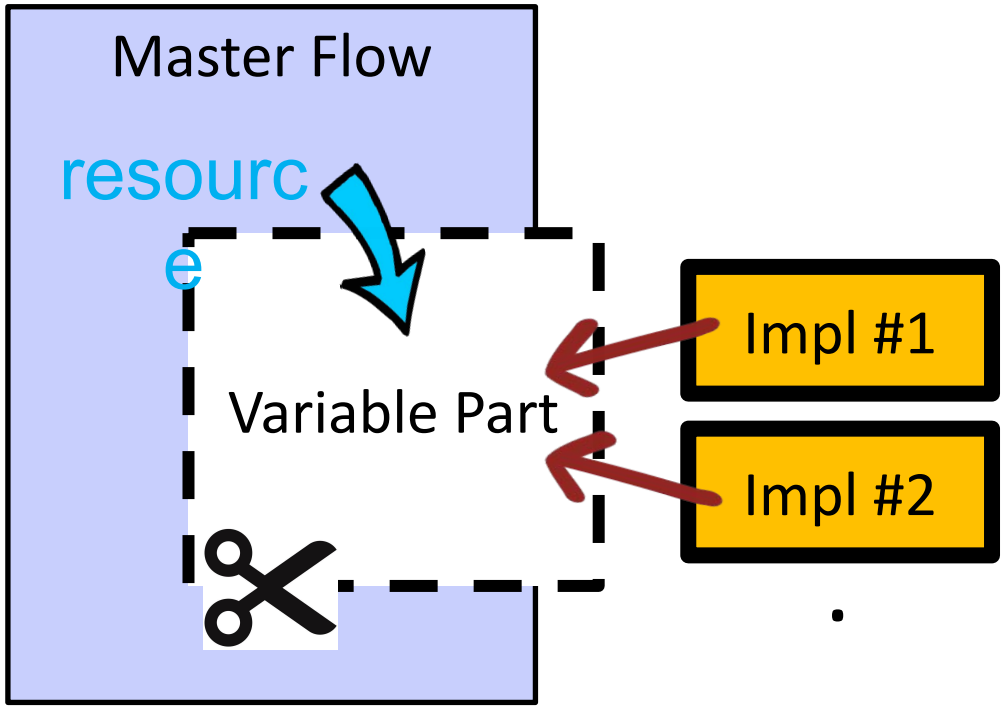
passing a function as an argument

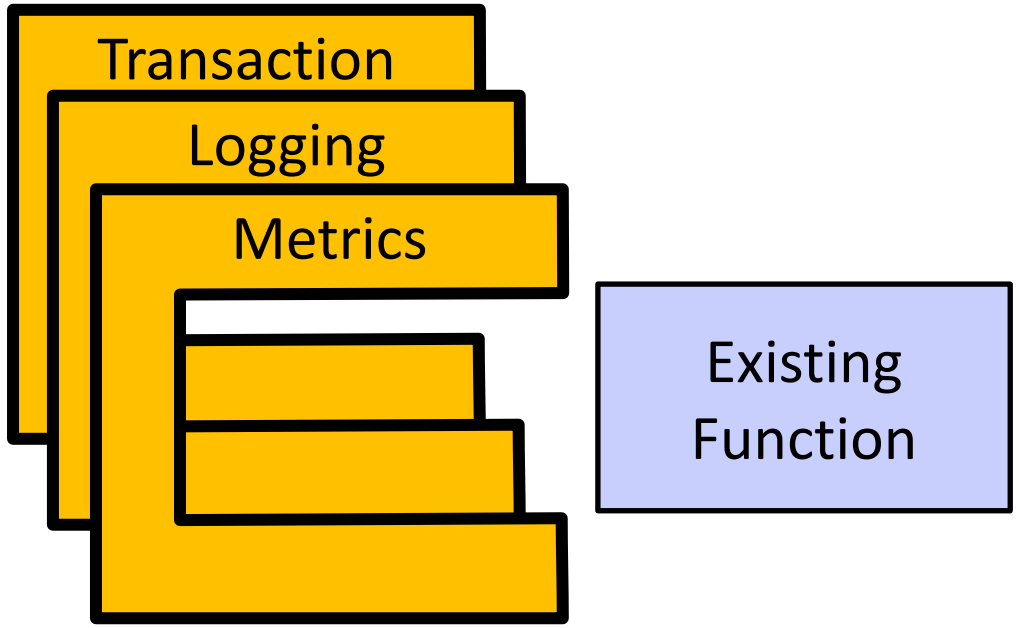
(previously discussed next to AOP and Proxy)

Loan Pattern VS Execute-Around

Parameterize the variable behavior
to run in a specific context

Wrap a new context "around" existing logic






```
export(..., w -> writeUsersOn(w));
```

```
measure(() -> f1(..));
```

Long-long time ago

OOH was KING

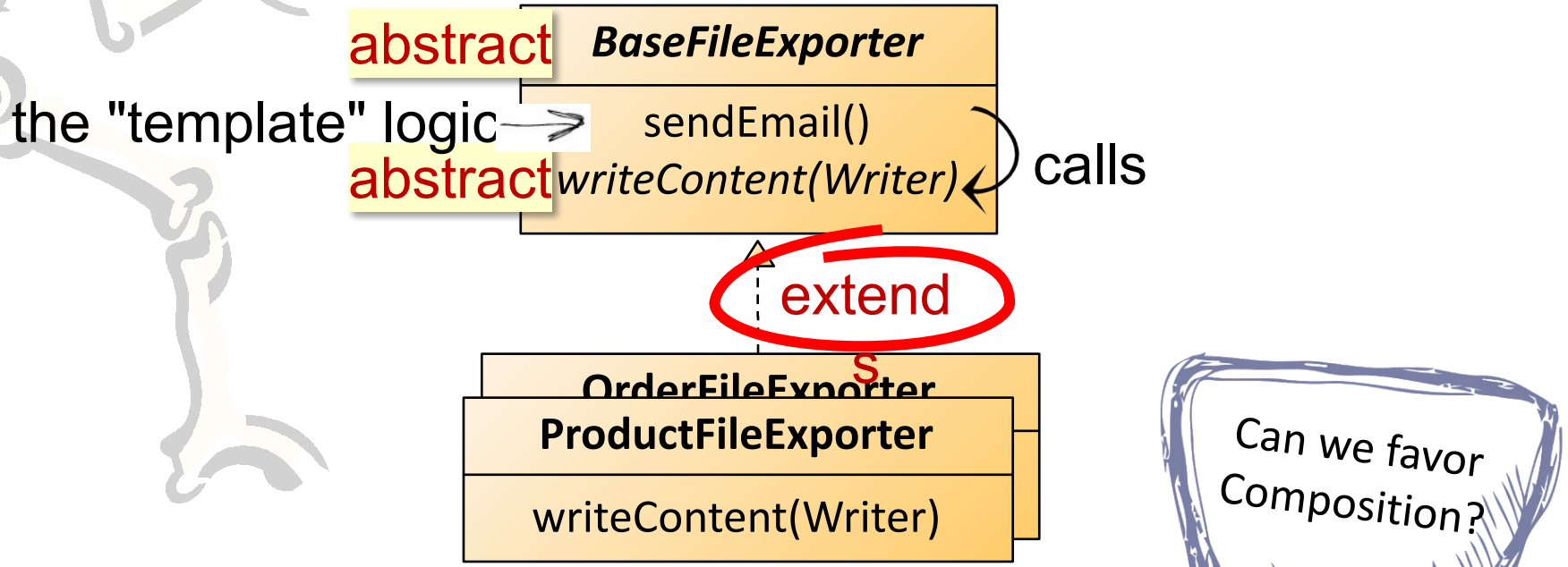
SO THEY SOLVED THIS USING

EXTENDS

Template Method

a complex workflow in a class,

call variable steps implemented in subclasses

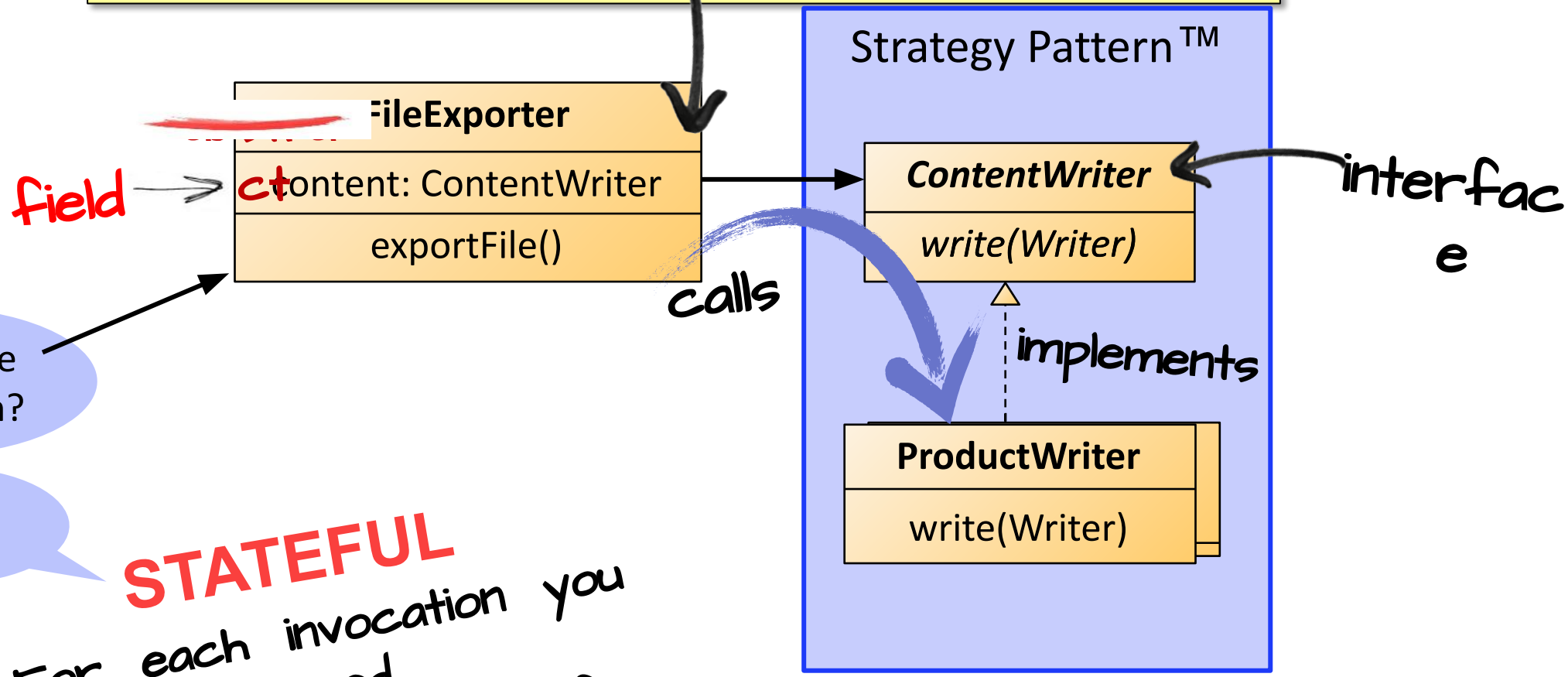


Can we favor Composition?

~~Temp~~ EXTENDS

Composing a Strategy

```
fileExporter.exportFile(new OrderContentWriter());
```

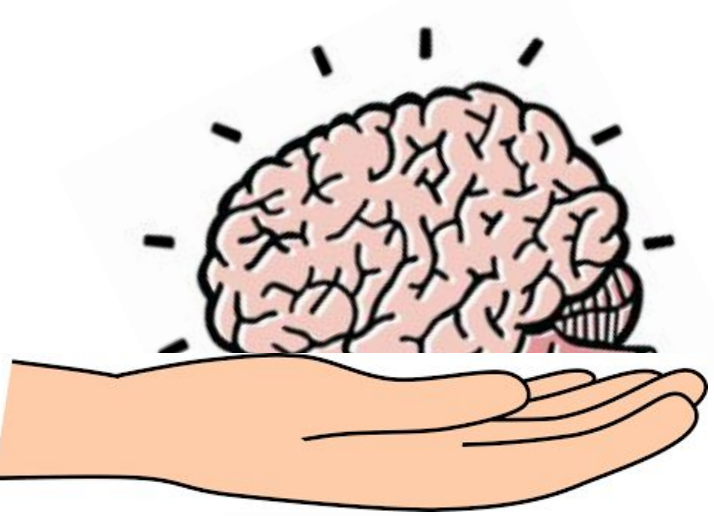


Can this be a singleton?

NO!

STATEFUL

For each invocation you need separate instance of Writer



~~EXTENDS~~

~~STATEFUL~~

Passing a Block

Pass a function

```
fileExporter.exportFile(orderContentWriter::write);
```

replaced inheritance with field

replaced fields with params

extends < composition < parameter

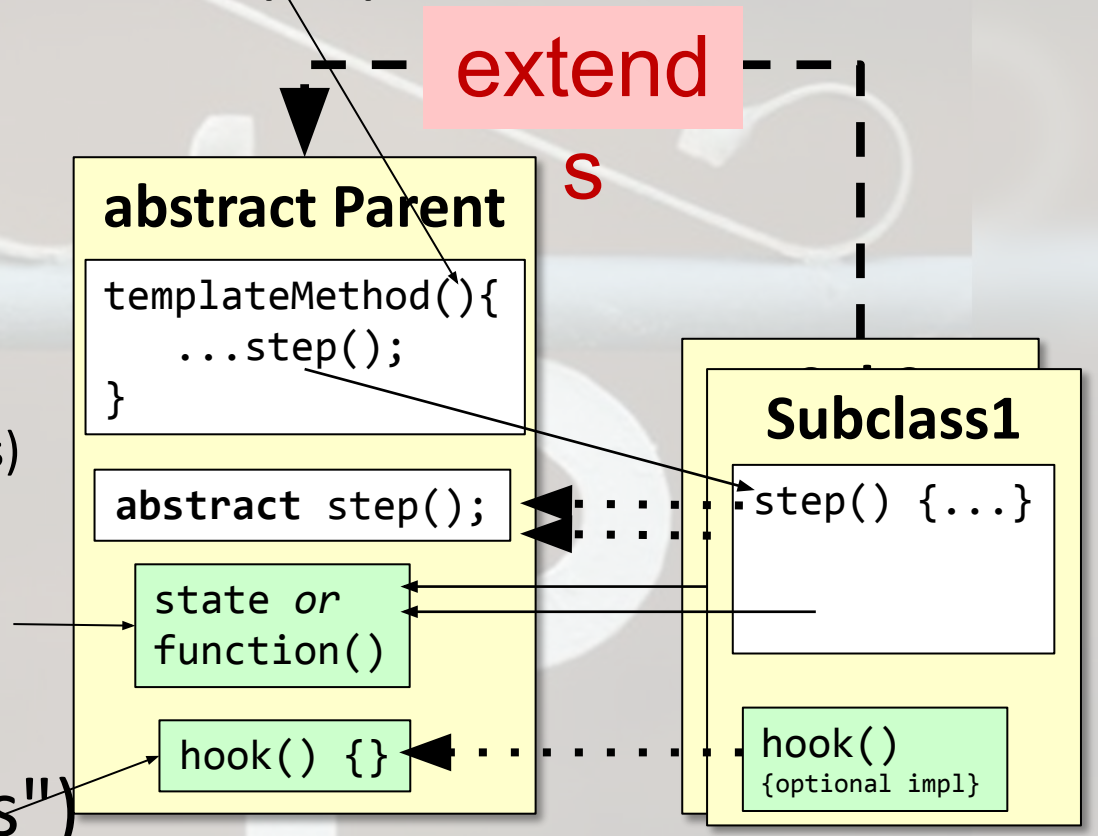
less coupled❤️ □

stateless❤️ □

Template Method

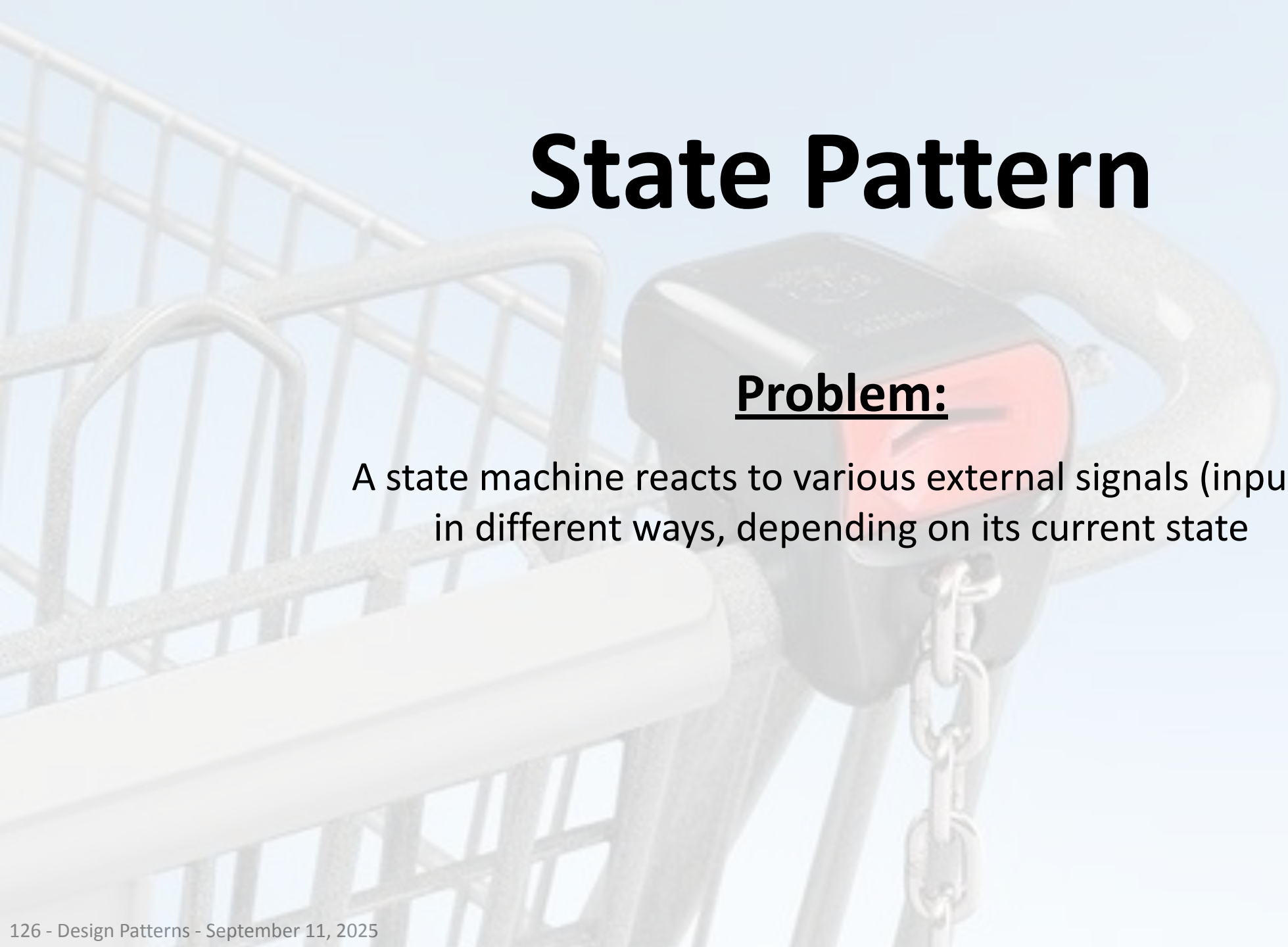
Template Method use-cases

- **One piece** of missing logic **Pass a Function** (FP)
  - More decoupled, testable, smaller code, DI-friendlier ✅
- **Multiple variable steps**:
  - a) take ≥ 2 lambdas 🤢
  - b) group them in a **Strategy** (interface with ≥ 2 methods)
  - c) stick to **Template Method** (with ≥ 2 `abstract` methods)
- **Subclasses use parent** data/methods
  - **Template Method**
- **Optional Steps** (empty "hook methods")
  - Only some children want to override them
  - a) **Template Method** b) **interfaces with default methods**



State Pattern

State Pattern



Problem:

A state machine reacts to various external signals (inputs) in different ways, depending on its current state

initial state

External signal

Side-effect

insert coin/capture coin

locked
state

unlock
ed
state

insert coin/

push in stack
/

push in stack/release coin

A state machine reacts to various
external signals (inputs)
with **different effects**,
depending on its current state

State Machine

For every input bit, output '1' if a '1011' sequence was completed,
or 0 otherwise

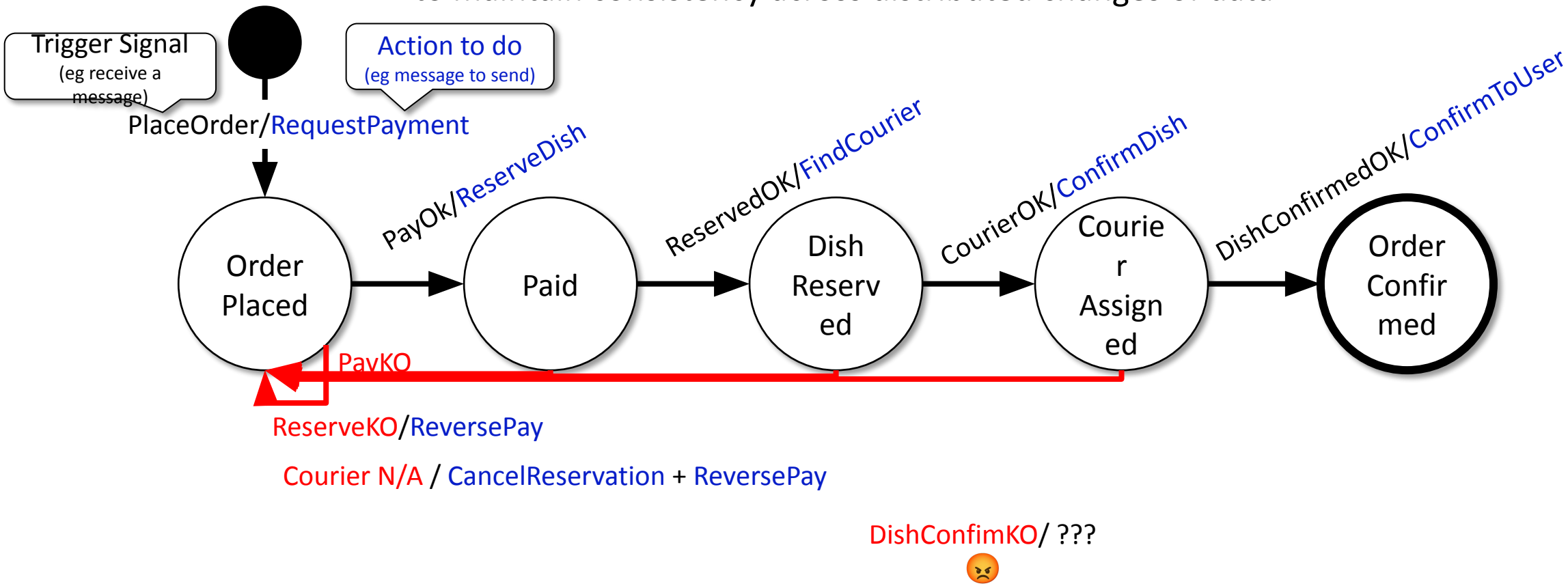

```
def "offered PC matches preferred configuration"() {
  when:
  def pc = shop.buyPc()

  then:
  verifyAll(pc) {
    vendor == "Sunny"
    clockRate >= 2333
    ram >= 406
    os == "Linux"
  }
}
```

Input: 10101101101111

Output:000001001001000

Microservice Saga

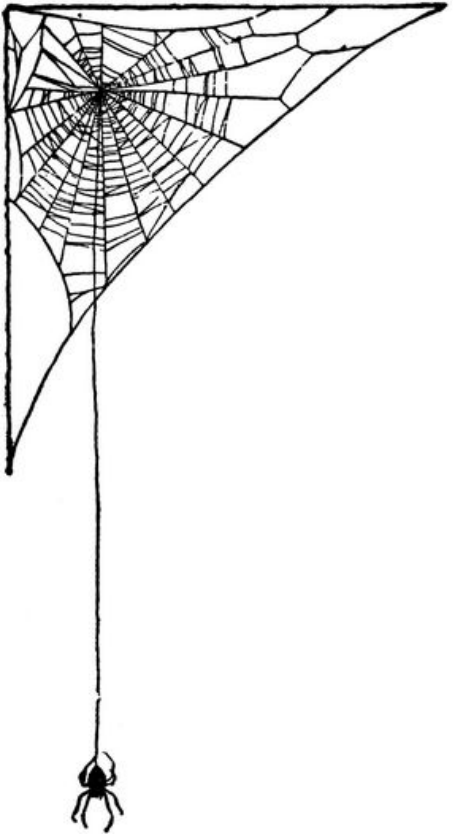
to maintain consistency across distributed changes of data



State Pattern

- Implement State-dependent behavior in different classes
  - Each state reacts to external inputs in their own way
  - A general-purpose 'State Machine' keeps track of the current state and directs inputs to it

- Uses:
  - Parsers (low level)
  - Communication Protocols (eg. Actor model)
  - Business Flows involving finite state machines (also consider a BPM tool)
  - Animations (eg. in Unity)
  - Microservice Saga aiming to achieve eventual consistency of a cross-microservice flow

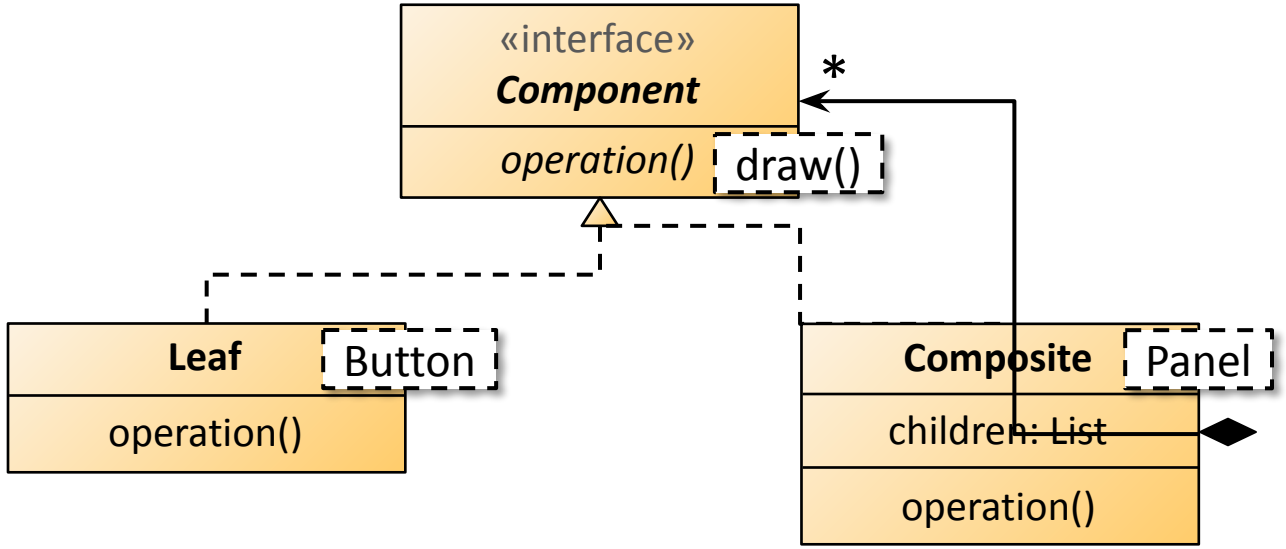


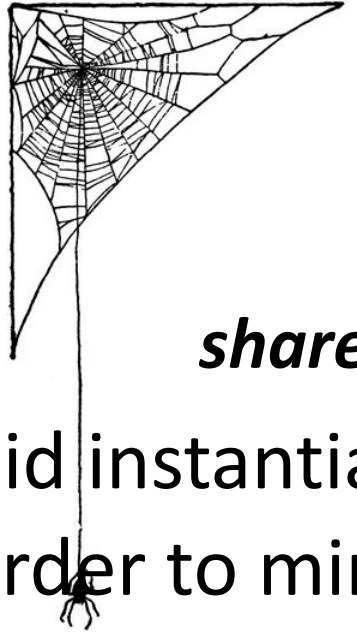
Composite

(for GUI)

compose objects into part-whole hierarchies;
treat atomic and composite objects uniformly

"A Panel holds another Panel and a Button",
"draw this Component"





Flyweight ★

share as much data as possible with other objects

- Avoid instantiating numerous similar objects in order to minimize memory

The quick ***brown*** fox jumps over the lazy dog

How would you implement Word?

a) One instance for each character (think 1M characters/doc☺)

b) Bold(start,end). And then collect state for each character

- .draw(DocumentCharacter, Font)

Singleton | Factory Method | Abstract Factory | Fluent Builder | Facade | Adapter | Decorator | Proxy | Observer (Events) | Command | Strategy | Chain of Responsib | Template Method | Passing a Block (callable)

Practical Use of Patterns

Used in application code (microdesign)

Builder, Adapter/Facade, Passing a Block, Strategy, Factory Method, Decorator

Used by frameworks

Factories, Singleton, Proxy, Chain of Responsibility (Filters), Decorator, Visitor

Used for Architecture (intra-/cross- systems)

Observer/Command, REST vs MQ, Choreography/Orchestration, Saga

Summary of Principles

- **SRP** – Keep your classes/methods focused on one thing; Name > Extract
- **DRY** – Do not duplicate business logic
- **Separation by Layers of Abstraction**: clean up the high-level logic
- **Dependency Inversion Principle** – Agnostic Domain of infrastructure
- **Avoid Inheritance**: favor composition to reuse behavior
  - don't extend concrete classes, don't override concrete methods
  - prefer interfaces
- **Loose Coupling** – easier to reason and test
- **Prefer Stateless Classes**
- **KISS** 💋 **Production**. Practice with design patterns in pet projects first

Do NOT Use	When..
Chain of Resp	When the filters rarely change, or are few, of complex selection criteria
Strategy	When the alternative logic is small or if the API is dramatically different
Adapter	When the data conversion is trivial. Convert via a separate method
Template	If the missing step is a single simple piece of logic pass a function
Proxy	If you need to wrap code just around several methods. *measureTime*(() -> f1(...));
Decorator	If you don't need to swap in/out the added logic transparently
Factories	If the creation is simple. Keep them XXS!
extends	To use behavior or state from another class composition or parameters.
Builder	Aren't fluent setters enough? Immutable > break more the classes.
Object Mother	If it couples your tests to each other
Singleton	If you want request-specific fields (per-call)
Visitor	If you can add polymorphic methods to your (small) model

Anti-Pattern

https://www.amazon.ca/Wenger-16999-Swiss-Knife-Giant/dp/B001DZTJRQ

Big Ball of Mud ~~SRP~~

aka. God Object, The Blob, Swiss Army Knife

Anti-Patterns

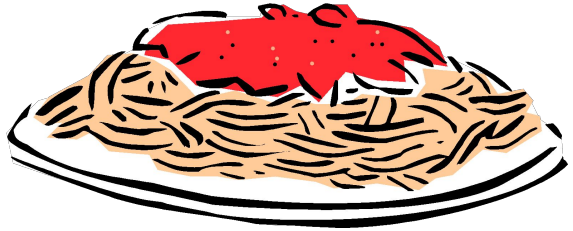
Big Ball of Mud ~~SRP~~

aka. God Object, The Blob, Swiss Army Knife

Spaghetti Code

Too many, unclear, dependencies

Entagled calls, huge call stacks



Spaghetti/Macaroni/Lasagna/Ravioli Code

Circular Dependencies

merge

split in 3+

lower level of abstraction

CsvWriter

Anti-Patterns

Golden Hammer

e a hammer, everything looks like a …

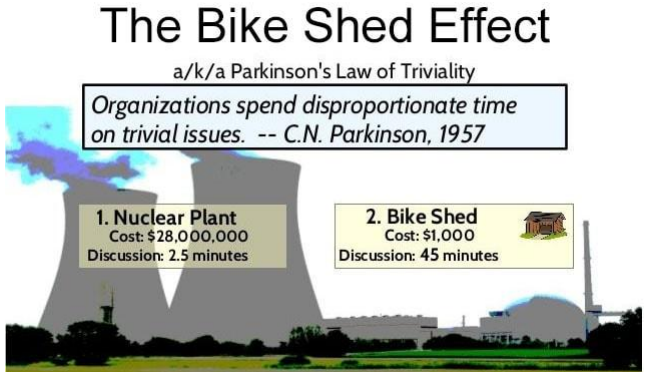
Diaper Pattern

aka. shawarma error-handling

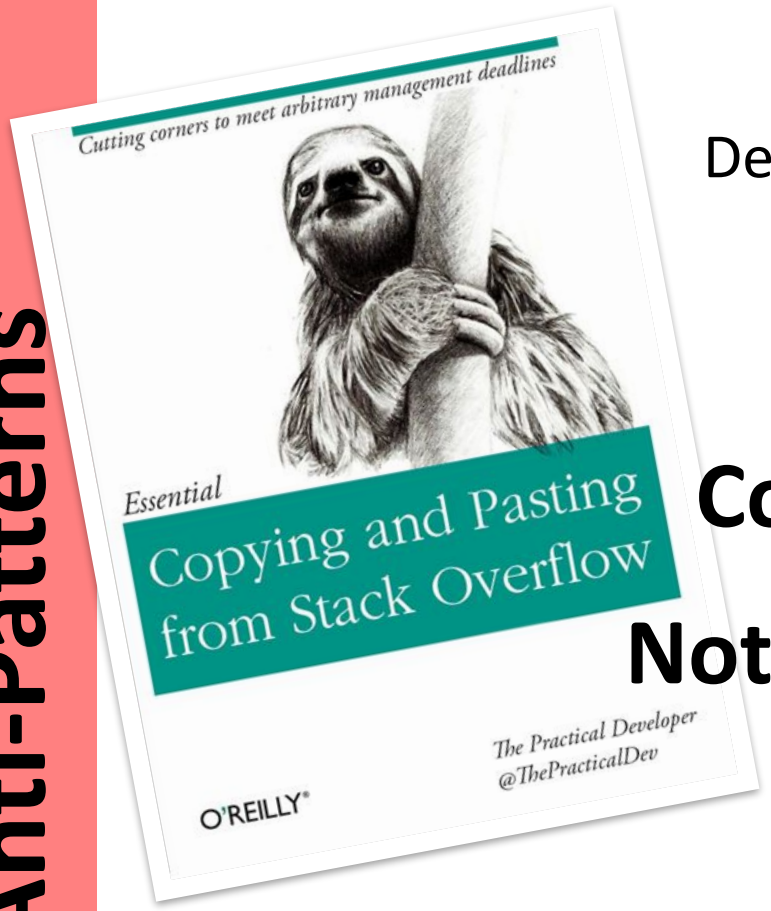
```
} catch (Exception e) {}
```

Gold Plating

Know when to stop! Move on!



Anti-Patterns



Fake Encapsulation

Decent Objects should hide their state

Copy-Paste Programming

vs

Not Invented Here Syndrome

□ open-source rocks!

Lava Flow

Overengineering;
Speculative Generality

Patterns in JDK

- java.lang.**Object#clone()** (

Singleton

- java.lang.**Runtime#getRuntime()**
- java.awt.**Desktop#getDesktop()**

Structural patterns

Adapter

- java.util.**Arrays#asList()**
- java.io.**InputStreamReader(InputStream)** (a Reader)
- java.io.**OutputStreamWriter(OutputStream)** (a Writer)
- javax.xml.bind..**XmlAdapter#marshal()** and **#unmarshal()**

Composite

- java.awt.**Container#add(Component)** - all over Swing

Decorator

- Subclasses of java.io.**InputStream**, **OutputStream**, **Reader** and **Writer** have a constructor taking an instance of same type.
- **Collections#checkedXXX()**, **#synchronizedXXX()** and **#unmodifiableXXX()**
- **HttpServletRequestWrapper** and **HttpServletResponseWrapper**

Proxy

- java.lang.reflect.**Proxy**
- java.**rmi.***, the whole API actually.

Behavioral patterns

Chain of responsibility

=behavioral methods which (indirectly) invoke the same method in *another* implementation of *same* abstract/interface type in a queue)

- java.util.logging.**Logger#log()**
- javax.servlet.**Filter#doFilter()**

Command

- Implementations of java.lang.**Runnable**
- Implementations of javax.swing.**Action**

Iterator

- java.util.**Iterator** (also java.util.**Scanner**!).
- java.util.**Enumeration**

Observer (or Publish/Subscribe)

- java.util.**EventListener** - all over Swing
- javax.servlet.**ServletContextListener**

Strategy

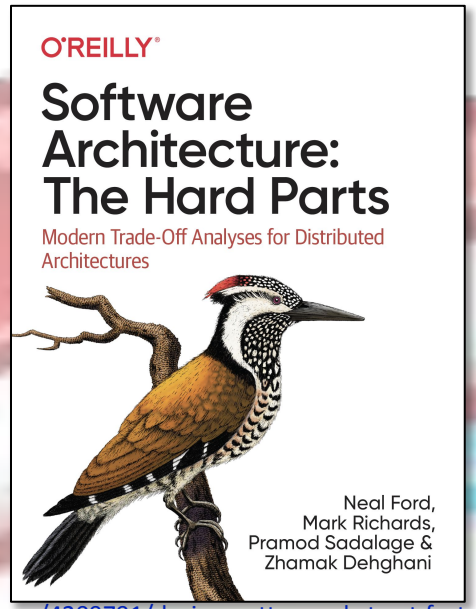
Gang of Four

Image from “Head First – Design Patterns” ISBN: 978-0-596-00712-6

Learning is not Compulsory

Nor is Survival

- **Head First: Design Patterns, 2nd ed**
  - https://www.amazon.de/-/en/Eric-Freeman/dp/149207800X



- **Clean Code**
  Robert C. Martin (aka Uncle Bob)

- **My public talks (recordings):**
  - **http://victorrentea.ro/#talks**
- Abstract Factory vs Static Factory vs… http://stackoverflow.com/questions/4209791/design-patterns-abstract-factory-vs-factory-method
- Patterns discussed: http://en.wikipedia.org/wiki/Design_Patterns (catalog at the bottom of the page)
- SOLID Principles: http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29
  - In images: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/
- SOLID is WRONG: https://speakerdeck.com/tastapod/why-every-element-of-solid-is-wrong
- GRASP: http://en.wikipedia.org/wiki/GRASP_%28object-oriented_design%29
- GoF Design Patterns in Java SDK http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns