Design Pattern Reloaded

Rémi Forax
ToursJUG march 2015

with Java8 inside

https://github.com/forax/design-pattern-reloaded

Me, Myself and I

I'm a Schizophren, so am I

Assistant Prof at Paris East University

Expert for JSR 292, 335 and 376

Open source dev: OpenJDK, ASM, Tatoo, etc

Father of 3

Evolution

Design Pattern, GoF

SOLID, Robert C Martin aka Uncle Bob

Java Lambda, JSR 335 Expert Group

Design Pattern - 1994

Two big principles:

- Program to an interface, not an implementation
- Favor object composition over class inheritance

Side note:

Some GoF patterns do not respect these principles

SOLID principles - 2000

Single Responsability Principle

Open/Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Functional Interface - 2014

Single Responsability Principle

Open/Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

```
@FunctionalInterface
interface DoIt {
  int apply(int val1, int val2);
}
```

a functional interface has only one abstract method

Functional Interface - 2014

```
@FunctionalInterface
interface DoIt {
  int apply(int val1, int val2);
}
```

a functional interface has only one abstract method

Conceptually equivalent to

```
  typedef DoIt = (int, int) → int
```

```
DoIt add = (x, y) -> x + y;
DoIt mul = (a, b) -> a * b;
```

A simple Logger

```
public interface Logger {
  public void log(String message);


  public static void main(String[] args) {
    Logger logger = msg -> System.out.println(msg);
  }
}
```

No parenthesis if one argument

Filtering logs

```
public interface Logger {
  public void log(String message);
}
public interface Filter {
  public boolean accept(String message);
}
```

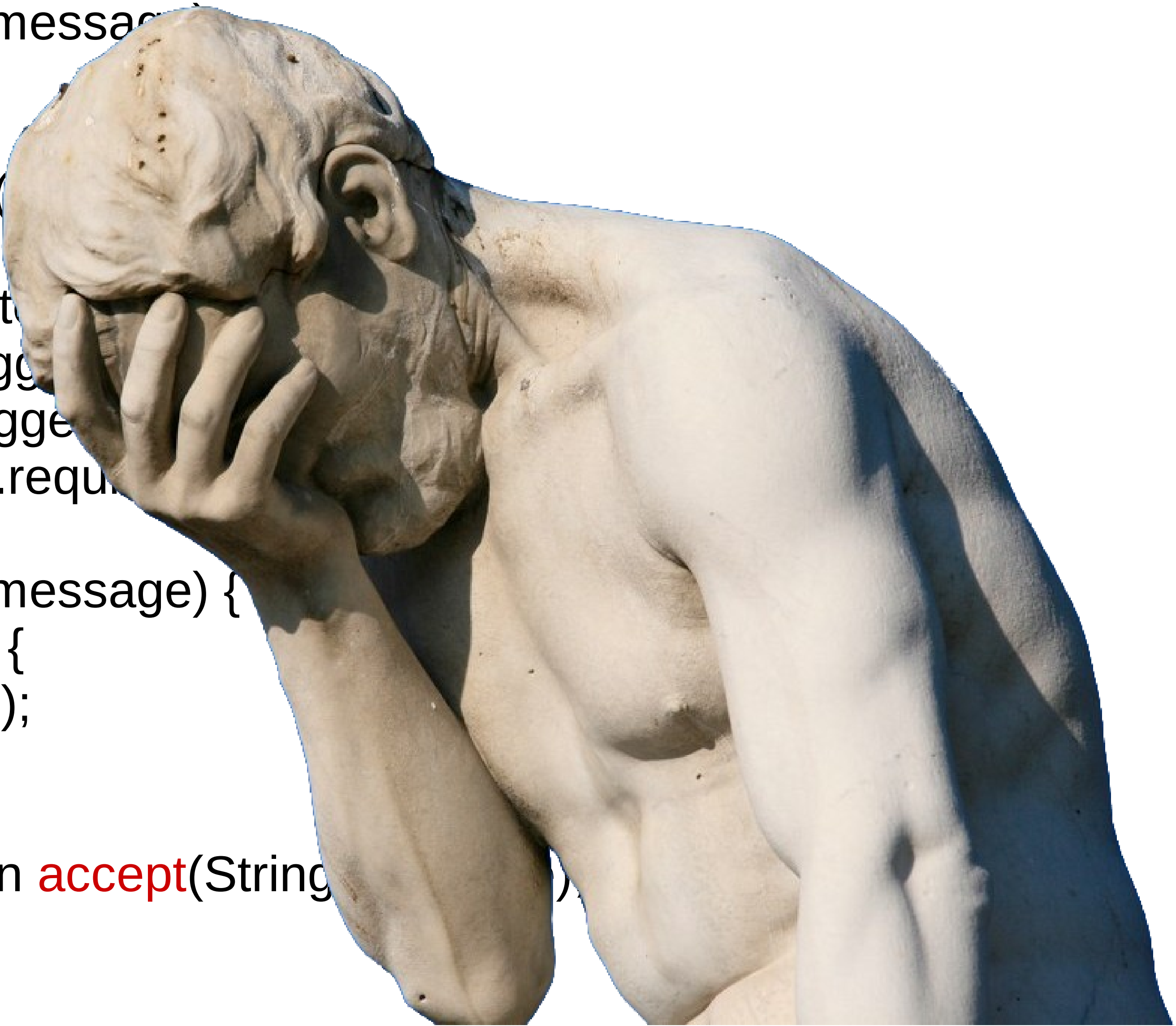
I want a Logger that only log messages that are accepted by a filter

GoF Template Method !

```
public interface Logger {
  public void log(String message);
}
public interface Filter {
  public boolean accept(String message);
}
public abstract class FilterLogger implements Logger, Filter {
  private final Logger logger;
  public FilterLogger(Logger logger) {
    this.logger = Objects.requireNonNull(logger);
  }
  public void log(String message) {
    if (accept(message)) {
      logger.log(message);
    }
  }
  public abstract boolean accept(String message);
}
```

GoF Template Method :(

```
public interface Logger {
  public void log(String messa
}
public interface Filter {
  public boolean accept(
}
public abstract class Filt
  private final Logger logg
  public FilterLogger(Logge
    this.logger = Objects.requ
  }
  public void log(String message) {
    if (accept(message)) {
      logger.log(message);
    }
  }
  public abstract boolean accept(String          );
}
```



GoF Template Method :(

public abstract classes are harmful !

Favor object composition !

```
public class FilterLogger implements Logger {
  private final Logger logger;
  private final Filter filter;

  public FilterLogger(Logger logger, Filter filter) {
    this.logger = Objects.requireNonNull(logger);
    this.filter = Objects.requireNonNull(filter);
  }

  public void log(String message) {
    if (filter.accept(message)) {
      logger.log(message);
    }
  }
}
```

bluish-grey backdrop.
chosen composition.
patterned yellow backdrop.
patterned yellow backdrop for foreground
light source

High order function / function composition

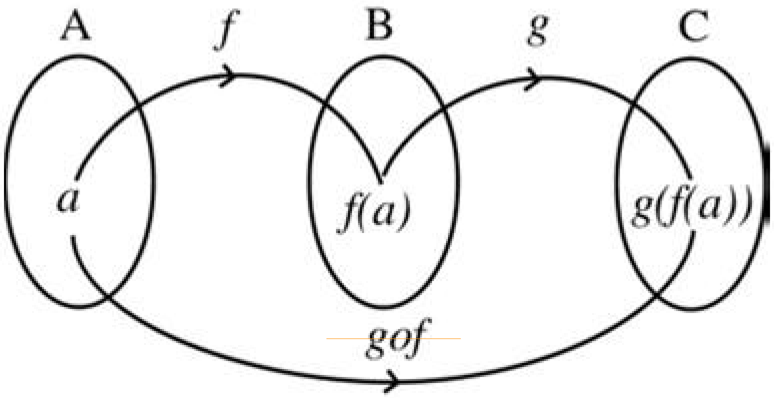
```
public class Loggers {
  public static Logger filterLogger(Logger logger, Filter filter) {
     Objects.requireNonNull(logger);
     Objects.requireNonNull(filter);
     return message -> {
       if (filter.accept(message)) {
         logger.log(message);
       }
     };
  }
}
```

```
Logger logger = msg - > writer.write(msg);
Logger filterLogger =
   Logger.filterLogger(logger, msg -> msg.startsWith("foo"));
```

Function composition using instance (default) method

```
public interface Logger {
  public void log(String message);

  public default Logger filter(Filter filter) {
    Objects.requireNonNull(filter);
    return message -> {
      if (filter.accept(message)) {
        log(message);
      }
     };
  }
}
```



g.filter(f)

```
Logger logger = msg - > writer.write(msg);
Logger filterLogger =
   logger.filter(msg -> msg.startsWith("foo"));
```

With Java 8 predefined interfaces

```
public interface Logger {
  public void log(String message);

  public default Logger filter(Predicate<String> filter) {
    Objects.requireNonNull(filter);
    return message -> {
      if (filter.test(message)) {
        log(message);
      }
     };
  }
}
```

```
package java.util.function;

@FunctionalInterface
public interface Predicate<T> {
  public boolean test(T t);
}
```

FP vs OOP from 10 000 miles

FP can be seen through GoF Principles:

- Program to an interface, not an implementation
- Favor object composition over class inheritance

=> no class, no inheritance, only functions

=> function composition <=> object composition

GoF kind of patterns

Structural

Behavioral

Creational

Structural Patterns

Adapter,
Bridge,
Decorator,
Composite,
Proxy,
Flyweight, etc

most of them derive from

“Favor object composition over class inheritance”

Yet Another Logger

```
public enum Level { WARNING, ERROR }
public interface Logger2 {
  public void log(Level level, String message);
}

Logger2 logger2 = (level, msg) ->
    System.out.println(level + " " + msg);
logger2.log(ERROR, "abort abort !");

// how to adapt the two loggers ?
Logger logger = logger2 ??
```

Partial Application

Set the value of some parameters of a function (also called curryfication)

```
interface DoIt { int apply(int x, int y); }
interface DoIt1 { int apply(int x); }


DoIt add = (x, y) -> x + y;
DoIt1 add1 = x -> add.apply(x, 1);

DoIt mul = (a, b) - > a * b;
DoIt1 mulBy2 = a - > mul.apply(2, a);
```

Partial Application

log(msg) == log2(*level*, msg) with *level* = ERROR

```
public interface Logger {
  public void log(String message);
}

public interface Logger2 {
  public void log(Level level, String message);

  public default Logger error() {
    return msg -> log(ERROR, msg);
  }
}
```

Adapter

```
public interface Logger2 {
  public void log(Level level, String message);

  public default Logger error() {
    return msg -> log(ERROR, msg);
  }
}

Logger2 logger2 = ...
Logger logger = logger2.error();
logger.log("abort abort !");
```

Method Reference in Java 8

The operator :: allows to reference an existing method

```
BiConsumer<PrintStream, Object> cons =
  PrintStream::println;
```

type

```
package java.util.function;
public interface Consumer<T> {
  public void accept(T t);
}
public interface BiConsumer<T, U> {
  public void accept(T t, U u);
}
```

Partial Application & Method Ref.

The operator :: also allows to do partial application on the receiver

```
BiConsumer<PrintStream, Object> consumer =
  PrintStream::println;
```

type

```
PrintStream out = System.out;
Consumer<Object> consumer2 = out::println;
```

instance

Behavioral Patterns

Command
Observer
State
~~Iterator~~ Iteration (Internal vs External)
Visitor

Command

A command is just a function

```
interface Command {
  public void perform();
}

Command command =
  () - > System.out.println("hello command");
```

Sum values of a CSV ?

```
public class SumCSV {
  public static double parseAndSum(Path path) throws … {
    try (Stream<String> lines = Files.lines(path)) {
      return lines
        .flatMap(line -> Arrays.stream(line.split(",")))
        .mapToDouble(token -> Double.parseDouble(token))
        .sum();
    }
  }
}
```

Sum values of a CSV ?
(using method reference)

```
public class SumCSV {
  public static double parseAndSum(Path path) throws … {
    try (Stream<String> lines = Files.lines(path)) {
      return lines
        .flatMap(Pattern.compile(",")::splitAsStream)
        .mapToDouble(Double::parseDouble)
        .sum();
    }
  }
}
```

Partial application

Observer

Decouple work in order to close the CVSParser
(as in Open/Close Principle)

```
public interface Observer {
  public void data(double value);
}

public class CSVParser {
  public static void parse(Path path, Observer observer) throws … {
    ...
  }
}

public class SumCSV {
  public double parseAndSum(Path path) throws … {
    CSVParser.parse(path, ...);
    ...
  }
}
```

Observer – Client side

```
public interface Observer {
  public void data(double value);
}
public class CSVParser {
  public static void parse(Path path, Observer observer) throws … {
    ...
  }
}
```

side effect

```
public class SumCSV {
  private double sum;
  public double parseAndSum(Path path) throws … {
    CSVParser.parse(path, value -> sum += value);
    return sum;
  }
}
```

Observer

The closed module

```
public interface Observer {
  public void data(double value);
}

public class CSVParser {
  public static void parse(Path path, Observer observer) throws … {
    try (Stream<String> lines = Files.lines(path)) {
      lines.flatMap(Pattern.compile(",")::splitAsStream)
        .mapToDouble(Double::parseDouble)
        .forEach(value -> observer.data(value));
    }
  }
}
```

Functional Interfaces conversion

DoubleStream.forEach() takes a DoubleConsumer as parameter

```
public interface Observer {
  public void data(double value);
}

public class CSVParser {
  public static void parse(Path path, Observer observer) throws … {
    try (Stream<String> lines = Files.lines(path)) {
      lines.flatMap(Pattern.compile(",")::splitAsStream)
        .mapToDouble(Double::parseDouble)
        .forEach(value -> observer.data(value));
    }
  }
}
```

DoubleConsumer

Functional Interfaces conversion

:: can be used to do interface to interface conversion

```
public interface Observer {
  public void data(double value);
}

public class CSVParser {
  public static void parse(Path path, Observer observer) throws … {
    try (Stream<String> lines = Files.lines(path)) {
      lines.flatMap(Pattern.compile(",")::splitAsStream)
        .mapToDouble(Double::parseDouble)
        .forEach(observer::data);
    }
  }
}
```

DoubleConsumer

Iteration

2 kinds

- Internal iteration (push == Observer)
- External iteration (pull == Iterator)

```
List<String> list = …
```

Internal:

```
list.forEach(item -> System.out.println(item));
```

External:

```
for(String item: list) {
  System.out.println(item);
}
```

External iteration is harder to write

forEach is easier to write than an Iterator

```
public class ArrayList<E> implements Iterable<E> {
  private E[] elementData;
  private int size;

  public void forEach(Consumer<E> consumer) {
    for(int I = 0; i < size; i++) {
      consumer.accept(elementData[i]);
    }
  }

  public Iterator<E> iterator() {
    return new Iterator<E>() {
      private int i;
      public boolean hasNext() { return i < size; }
      public E next() { return elementData[i++]; }
    };
  }
}
```

internal

external

Internal Iteration is less powerful in Java

No side effect on local variables allowed !

```
List<Double> list = …
```

Internal:

sum is not effectively final

```
double sum = 0;
list.forEach(value -> sum += value);
```

External:

```
for(double value: list) {
  sum += value;
}
```

Visitor ?

Given a hierarchy

```
  public interface Vehicle { … }
  public class Car implements Vehicle { … }
  public class Moto implements Vehicle { … }
```

want to close it but
allow to add new operations and new subtypes !

Destructured Visitor API

API I want

```
Visitor<String> visitor = new Visitor<>();
visitor.when(Car.class, car -> "car")
       .when(Moto.class, moto -> "moto");

Vehicle vehicle = ...
String text = visitor.call(vehicle);
```

```
package java.util.function;
public interface Function<T, R> {
  public R apply(T t);
  public default <V> Function<V,R> compose(Function<V,T> f) {}
  public default <V> Function<T,V> andThen(Function<R,V> f) {}
}
```

Destructured Visitor

```
public class Visitor<R> {
  public <T> Visitor<R> when(
           Class<T> type, Function<T, R> fun) { … }
  public R call(Object receiver) { … }
}

Visitor<String> visitor = new Visitor<>();
visitor.when(Car.class, car -> "car")
         .when(Moto.class, moto -> "moto");

Vehicle vehicle = ...
String text = visitor.call(vehicle);
```

Destructured Visitor

Java has no existential type :(

```
public class Visitor<R> {
  private final HashMap<Class<?>, Function<Object, R>> map =
      new HashMap<>();

  public <T> Visitor<R> when(Class<T> type, Function<T, R> f) {
    map.put(type, f);
    return this;
  }

  public R call(Object receiver) {
    return map.getOrDefault(receiver.getClass(),
                            r -> { throw new ISE(...); })
              .apply(receiver);
  }
}
```

Doesn't compile !

Destructured Visitor

Java has no existential type :(

```
public class Visitor<R> {
  private final HashMap<Class<?>, Function<?, R>> map =
      new HashMap<>();

  public <T> Visitor<R> when(Class<T> type, Function<T, R> f) {
    map.put(type, f);
    return this;
  }

  public R call(Object receiver) {
    return map.getOrDefault(receiver.getClass(), r -> { throw … })
              .apply(receiver);
  }
}
```

Doesn't compile !

Destructured Visitor

All problems can be solved by another level of indirection :)

```
public class Visitor<R> {
  private final HashMap<Class<?>, Function<Object, R>> map =
      new HashMap<>();

  public <T> Visitor<R> when(Class<T> type, Function<T, R> f) {
    map.put(type, object -> f.apply(type.cast(object)));
    return this;
  }

  public R call(Object receiver) {
    return map.getOrDefault(receiver.getClass(), r -> { throw … })
                     .apply(receiver);
  }
}
```

Destructured Visitor + function composition

And using function composition

```
public class Visitor<R> {
 private final HashMap<Class<?>, Function<Object, R>> map =
   new HashMap<>();

 public <T> Visitor<R> when(Class<T> type, Function<T, R> f) {
  map.put(type, f.compose(type::cast));
  return this;
 }

 public R call(Object receiver) {
  return map.getOrDefault(receiver.getClass(), r -> { throw … })
          .apply(receiver);
 }
}
```

Creational Patterns

Static Factory

~~Factory method~~

Singleton

Abstract Factory

Builder

Monad ?

Same problem as template method

Who want a global ?

Instance Creation

```
public interface Vehicle { … }
public class Car implements Vehicle {
  public Car(Color color) { … }
}
public class Moto implements Vehicle {
  public Moto(Color color) { … }
}
```

I want to create only either 5 red cars or 5 blue motos ?

Instance Factory ?

```
public interface VehicleFactory {
  public Vehicle create();
}

public List<Vehicle> create5(VehicleFactory factory) {
  return range(0,5)
              .mapToObj(i -> factory.create())
              .collect(toList());
}

VehicleFactory redCarFactory = ...
VehicleFactory blueMotoFactory =  ...

List<Vehicle> redCars = create5(redCarFactory);
List<Vehicle> blueMotos = create5(blueMotoFactory);
```

Instance Factory

```
public interface VehicleFactory {
  public Vehicle create();
}
public List<Vehicle> create5(VehicleFactory factory) {
  return range(0,5)
             .mapToObj(i -> factory.create())
             .collect(toList());
}

VehicleFactory redCarFactory = () -> new Car(RED);
VehicleFactory blueMotoFactory =  () -> new Moto(BLUE);

List<Vehicle> redCars = create5(redCarFactory);
List<Vehicle> blueMotos = create5(blueMotoFactory);
```

With Java 8 predefined interfaces

```
public List<Vehicle> create5(Supplier<Vehicle> factory) {
  return range(0,5)
              .mapToObj(i -> factory.get())
              .collect(toList());
}

Supplier<Vehicle> redCarFactory = () -> new Car(RED);
Supplier<Vehicle> blueMotoFactory = () -> new Moto(BLUE);

List<Vehicle> redCars =
      create5(redCarFactory);
List<Vehicle> blueMotos =
      create5(blueMotoFactory);
```

```
package java.util.function;

@FunctionalInterface
public interface Supplier<T> {
  public T get();
}
```

Instance Factory == Partial application on constructors

```
public List<Vehicle> create5(Supplier<Vehicle> factory) {
  return range(0,5)
            .mapToObj(i -> factory.get())
            .collect(toList());
}

public static <T, R> Supplier<R> partial(
                              Function<T, R> function, T value) {
  return () -> function.apply(value);
}
```

Method reference on new + constructor

```
List<Vehicle> redCars =
     create5(partial(Car::new, RED)));
List<Vehicle> blueMotos =
     create5(partial(Moto::new, BLUE)));
```

Static Factory

```
public interface Vehicle {
  public static Vehicle create(String name) {
    switch(name) {
      case "car":
        return new Car();
      case "moto":
        return new Moto();
      default:
        throw ...
    }
  }
}
```

Quite ugly isn't it ?

Abstract Factory

```
public class VehicleFactory {
  public void register(String name,
                                   Supplier<Vehicle> supplier) {
    ...
  }
  public Vehicle create(String name) {
    ...
  }
}

VehicleFactory factory = new VehicleFactory();
factory.register("car", Car::new);
factory.register("moto", Moto::new);
```

Abstract Factory impl

```
public class VehicleFactory {
  private final HashMap<String, Supplier<Vehicle>> map =
      new HashMap<>();

  public void register(String name, Supplier<Vehicle> supplier) {
    map.put(name, fun);
  }

  public Vehicle create(String name) {
    return map.getOrDefault(name, () - > { throw new ...; })
                     .get();
  }
}

VehicleFactory factory = new VehicleFactory();
factory.register("car", Car::new);
factory.register("moto", Moto::new);
```

With a singleton-like ?

```
public class VehicleFactory {
  public void register(String name, Supplier<Vehicle> supplier) {
    ...
  }
  public Vehicle create(String name) {
    ...
  }
}
```

What if I want only one instance of Moto ?

With a singleton-like

```
public class VehicleFactory {
  public void register(String name, Supplier<Vehicle> supplier) {
    ...
  }
  public Vehicle create(String name) {
    ...
  }
}

VehicleFactory factory = new VehicleFactory();
factory.register("car", Car::new);

Moto singleton = new Moto();
factory.register("moto", () -> singleton);
```

From Factory to Builder

```
public class VehicleFactory {
  public void register(String name, Supplier<Vehicle> supplier) {
    ...
  }
  public Vehicle create(String name) {
    ...
  }
}
```

How to separate the registering step from the creation step ?

Classical Builder

```
public class Builder {
  public void register(String name, Supplier<Vehicle> supplier) { … }
  public VehicleFactory create() { … }
}
public interface VehicleFactory {
  public Vehicle create(String name);
}
```

```
Builder builder = new Builder();
builder.register("car", Car::new);
builder.register("moto", Moto::new);
VehicleFactory factory = builder.create();
Vehicle vehicle = factory.create("car");
```

Lambda Builder !

```
public interface Builder {
  public void register(String name, Supplier<Vehicle> supplier);
}
public interface VehicleFactory {
  public Vehicle create(String name);

  public static VehicleFactory create(Consumer<Builder> consumer) {
    …
  }
}

VehicleFactory factory = VehicleFactory.create(builder - > {
  builder.register("car", Car::new);
  builder.register("moto", Moto::new);
});
Vehicle vehicle = factory.create("car");
```

Lambda Builder impl

```
public interface Builder {
  public void register(String name, Supplier<Vehicle> supplier);
}

public interface VehicleFactory {
  public Vehicle create(String name);

  public static VehicleFactory create(Consumer<Builder> consumer) {
    HashMap<String, Supplier<Vehicle>> map = new HashMap<>();
    consumer.accept(map::put);
    return name -> {
        return map.getOrDefault(name, () -> { throw new ...; })
                         .get();
    };
  }
}
```

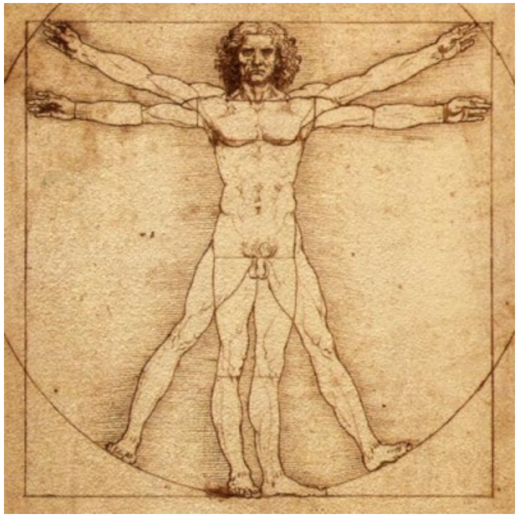
Validation ?

How to validate a user ?

```
public class User {
  private final String name;
  private final int age;
  ...
}
```

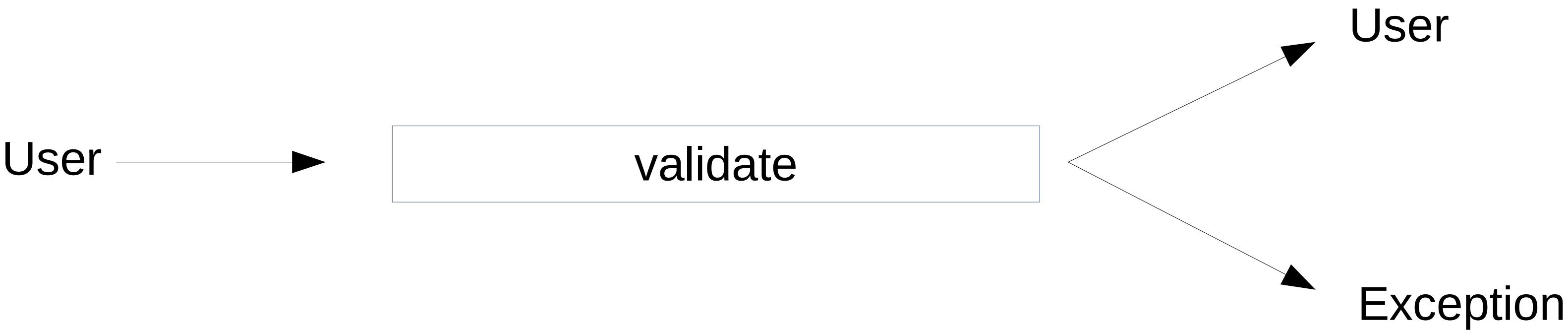
```
User user = new User("bob", 12);
if (user.getName() == null) {
  throw new IllegalStateException("name is null");
}
if (user.getName().isEmpty()) {
  throw new IllegalStateException("name is empty");
}
if (!(user.getAge() > 0 && user.getAge() < 50)) {
  throw new IllegalStateException("age isn't between 0 and 50");
}
```

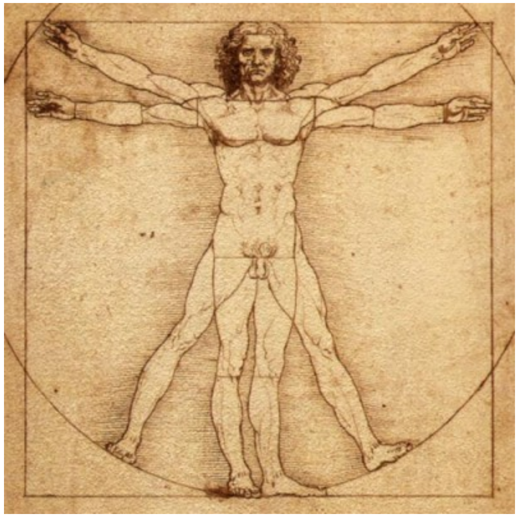
Monad



Represent 2 (or more) states has a unified value in order to compose transformations

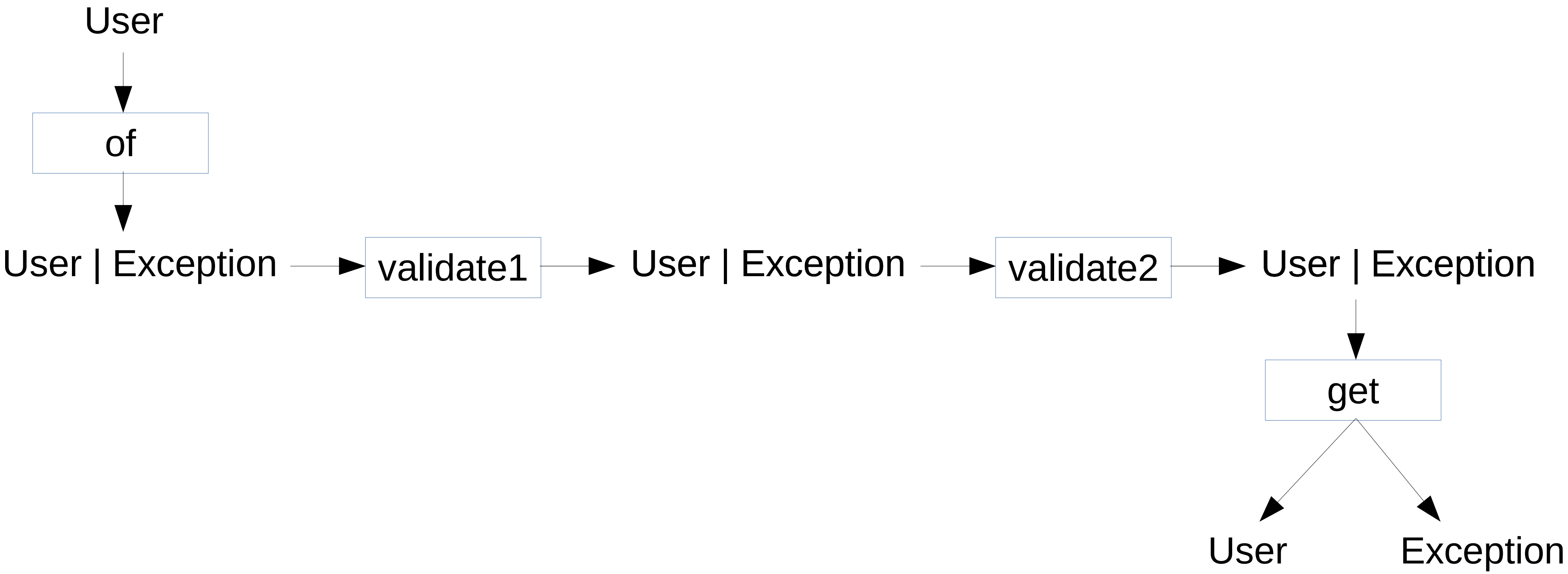
```
public static User validateName(User user) {
  if (user.getName() == null) {
    throw new IllegalStateException("...");
  }
  return user;
}
```





Monad

Represent 2 (or more) states has a unified value in order to compose transformations



User

of

User | Exception

validate1

User | Exception

validate2

User | Exception

get

User

Exception

Monad

```
public class Validator<T> {
  public static <T> Validator<T> of(T t) {
    ...
  }
  public Validator<T> validate(Predicate<T> validation, String message) {
    ...
  }
  public T get() throws IllegalStateException {
     ...
  }
}

User validatedUser = Validator.of(user)
    .validate(u -> u.getName() != null, "name is null")
    .validate(u -> !u.getName().isEmpty(), "name is empty")
    .validate(u -> u.getAge() > 0 && u.getAge() < 150, "age isn't between ...")
    .get();
```

Monad impl

```
public class Validator<T> {
  private final T t;
  private final IllegalStateException error;

  private Validator(T t, IllegalStateException error) {
    this.t = t;
    this.error = error;
  }
  public static <T> Validator<T> of(T t) {
    return new Validator<>(Objects.requireNonNull(t), null);
  }

  public Validator<T> validate(Predicate<T> validation, String message) {
    if (error == null && !validation.test(t)) {
      return new Validator<>(t, new IllegalStateException(message));
    }
    return this;
  }

  public T get() throws IllegalStateException {
     if (error == null) { return t; }
     throw error;
  }
}
```

Separate attribute from validation

```
public class Validator<T> {
  public Validator<T> validate(Predicate<T> validation,
                                                   String message) { … }

  public <U> Validator<T> validate(Function<T, U> projection,
                                   Predicate<U> validation, String message) {
   ...
  }
}

User validatedUser = Validator.of(user)
    .validate(User::getName, Objects::nonNull, "name is null")
    .validate(User::getName, name -> !name.isEmpty(), "name is ...")
    .validate(User::getAge, age -> age > 0 && age < 150, "age is ...")
    .get();
```

Higher order function

```
public static Predicate<Integer> inBetween(int start, int end) {
  return value -> value > start && value < end;
}


User validatedUser = Validator.of(user)
    .validate(User::getName, Objects::nonNull, "name is null")
    .validate(User::getName, name -> !name.isEmpty(), "...")
    .validate(User::getAge, inBetween(0, 50), "...")
    .get();
```

Monad impl

```
public class Validator<T> {
  public Validator<T> validate(Predicate<T> validation,
                              String message) {
    ...
  }

  public <U> Validator<T> validate(Function<T, U> projection,
                        Predicate<U> validation, String message) {
   return validate(t -> validation.test(projection.apply(t)), message);
  }
}
```

Monad

```
public class Validator<T> {
  public Validator<T> validate(Predicate<T> validation,
                                              String message) {
    ...
  }

  public <U> Validator<T> validate(Function<T, U> projection,
                                   Predicate<U> validation, String message) {
   return validate(
       projection.andThen(validation::test)::apply, message);
  }
}
```

Function

Predicate

Gather validation errors

```
public class Validator<T> {
  private final T t;
  private final ArrayList<Throwable> throwables = new ArrayList<>();

  public Validator<T> validate(Predicate<T> validation, String message) {
    if (!validation.test(t)) {
      throwables.add(new IllegalStateException(message));
    }
    return this;
  }

  public T get() throws IllegalStateException {
     if (throwables.isEmpty()) { return t; }
     IllegalStateException e = new IllegalStateException();
     throwables.forEach(e::addSuppressed);
     throw e;
  }
}
```

TLDR;

Functional interface

- bridge between OOP and FP

Enable several FP techniques

- Higher order function, function composition, partial application

UML is dead !

- public abstract classes are dead too !