```java
class Person {
  private String name;
  2 usages
  private int age;
  private List<String> knowledge;

  1 usage
  public boolean isAdult() {
    return age >= 18;
  }
```

data

interface

problem

```java
public boolean isAdult(int ref) {
  return age >= ref;
}
```

we can't scale
we can't run app
in another country

```java
Person jim = new Person( name: "Jim", age: 15, Arrays.asList());
boolean isAdult = jim.isAdult();
```

operation

implementation.

in Europe isAdult ≥18
US ≥21
Africa ≥15

how we can do that

we mixed everything
we made our code non-scalable
non-maintainable
:

```java
class Person3 {
  private String name;
  private int age;
  private List<String> knowledge;
}
```

data is clean

```java
interface IsAdult {
  3 usages  3 implementations
  boolean check(int age);
}
```

interface

```java
IsAdult isAdultEurope = new IsAdult() {
  3 usages
  @Override
  public boolean check(int age) {
    return age >= 18;
  }
};
IsAdult isAdultUSA = new IsAdult() {
  3 usages
  @Override
  public boolean check(int age) {
    return age >= 21;
  }
};
IsAdult isAdultAfrica = new IsAdult() {
  3 usages
  @Override
  public boolean check(int age) {
    return age >= 15;
  }
};
```

```java
IsAdult isAdultEurope = age -> age >= 18;
IsAdult isAdultUSA = age -> age >= 21;
IsAdult isAdultAfrica = age -> age >= 15;
```

} ← my implementations

```java
boolean isAdultEU = isAdultEurope.check(jim.getAge());
boolean isAdultUS = isAdultUSA.check(jim.getAge());
boolean isAdultAF = isAdultAfrica.check(jim.getAge());
```

```java
interface IsAdult4 {
  1 usage   1 implementation
  boolean check(int age);
}

2 usages
@AllArgsConstructor
class GlobalIsAdult implements IsAdult4 {
  1 usage
  private String country;

  1 usage
  private final IsAdult isAdultEurope = age -> age >= 18;
  1 usage
  private final IsAdult isAdultUSA = age -> age >= 21;
  1 usage
  private final IsAdult isAdultAfrica = age -> age >= 15;

  1 usage
  @Override
  public boolean check(int age) {
    switch (country) {
      case "Europe": return isAdultEurope.check(age);
      case "US"    : return isAdultUSA.check(age);
      case "Africa": return isAdultAfrica.check(age);
      default: return false;
    }
  }
}
```

```java
Person3 jim = new Person3( name: "Jim",  age: 15, Arrays.asList());

GlobalIsAdult isAdult = new GlobalIsAdult( country: "US");

isAdult.check(jim.getAge());
```

1) we can have as many impl as we want

2) easy to test. new impl. never affect existing one

3) we can combine them

```
private final Map<String, IsAdult5> implementations =
    new HashMap<String, IsAdult5>(){{
      put("Europe", age -> age >= 18);
      put("US", age -> age >= 21);
      put("Africa", age -> age >= 15);
    }};

1 usage
@Override
public boolean check(int age) {
  if (!implementations.containsKey(country)) return false;
  IsAdult5 isAdult = implementations.get(country);
  return isAdult.check(age);
}
```

not a value, but a function

O(1)

List.get(N) ⟶ O(N)

Map.get( ) ⟶ O(1)

true - if adult

false - if not adult

false - if country not found.

```
  public Optional<Boolean> check(int age) {
    if (!implementations.containsKey(country)) return Optional.empty();
    IsAdult5 isAdult = implementations.get(country);
    return Optional.of(isAdult.check(age));
  }
}

public class PersonTest5 {
  public static void main(String[] args) {
    Person3 jim = new Person3( name: "Jim",  age: 15, Arrays.asList());

    GlobalIsAdult5 isAdult = new GlobalIsAdult5( country: "US");
    Optional<Boolean> checked = isAdult.check(jim.getAge());
  }
}
```

- None ⟶ country not found
- Optional.of<true> isAdult
- Optional.of<false> not adult

Keep Data Description and Business logic Separate

gives you EXTRA COUPLING : IsAdult (Person)

```java
interface IsAdult5 {
  1 usage
  boolean check(Person5 p);
}
```

```java
interface IsAdult4 {
  1 usage   1 implementation
  boolean check(int age);
}
```

```java
private final Map<String, IsAdult5> implementations =
    new HashMap<String, IsAdult5>(){{
      put("Europe", p -> p.getAge() >= 18);
      put("US", p -> p.getAge() >= 21);
      put("Africa", p -> p.getAge() >= 15);
    }};

1 usage
public Optional<Boolean> check(Person5 p) {
  if (!implementations.containsKey(country)) return Optional.empty();
  IsAdult5 isAdult = implementations.get(country);
  return Optional.of(isAdult.check(p));
}
```

se1-java
se2-scala

git@github.com:alexr007/se2-java.git

se2-scala

se1
se2
] screenshots

has much nicer syntax than

```
isAdult.check(jim);
```

you can't pass a wrong value

Any Int

```
isAdult.check(jim.getAge());
```

potentially can be broken

up to you!

# Patterns

- most common composition of code regardless domain

GOF ~ 30 - 40 patterns

```
static class TaxiOrder {
  String from;
  String to;
  LocalDateTime when;
  Double price;
  Boolean smokingAllowed;
  Boolean petsAllowed;
}
```

I'm going from address 1 to address 2

I'm going to address 2 from address 1

PROBLEM

when you have a lot of fields it's not clear the order

```
new TaxiOrder(
        "Kyiv",
      "NewYork",
  LocalDateTime.parse( text: "2020-10-10 13:15"),
      500d,
              false,
          false
);
```

what do they mean?

```
TaxiOrder
  .withFrom("  ")
  .withTo("  ")
  .withSmoking(true)
  .withPet(false)
  .build()
```

→ will call constructor for me and pass variables.

very general approach to have dedicated methods to each field

```
static class TaxiOrder {
  String from;
  String to;
  LocalDateTime when;
  Double price;
  Boolean smokingAllowed;
  Boolean petsAllowed;
}

public static void main(String[] args) {
  TaxiOrder o = new TaxiOrder();
  o.setFrom("NewYork");
  o.setTo("Kyiv");
}
```

when → NULL

smokingAllowed, petsAllowed → NULL

never ever (99,99%) do that.

You can construct incomplete object

nothing WORSE than WRONG data in app

```java
@Builder
static class TaxiOrder {
  String from;
  String to;
  LocalDateTime when;
  Double price;
  Boolean smokingAllowed;
  Boolean petsAllowed;
}

public static void main(String[] args) {
  TaxiOrder order = TaxiOrder.builder()
      .from("Kyiv")
      .to("NewYork")
      .price(500d)
      .petsAllowed(false)
      .smokingAllowed(true)
      .smokingAllowed(false)
      .build();
}
```

we still can miss the date

it's times better, readable...

we can overwrite it.

```
case class TaxiOrder(
    from: String,
    to: String,
    when: LocalDateTime,
    price: Double,
    smokingAllowed: Boolean,
    petsAllowed: Boolean,
)

val order: TaxiOrder = TaxiOrder(
    from = "NewYork",
    to = "Kyiv",
    LocalDateTime.parse( text = "2020-10-10 13:15"),
    500d,
    smokingAllowed = false,
    petsAllowed = true
)
```

1. you can't create incomplete object constructor contains ALL fields
2. named parameters
   times less chances to mix parameters

even less chances to introduce a mistake

```
case class AddressFrom(value: String)
case class AddressTo(value: String)
case class SmokingAllowed(value: Boolean)
case class PetsAllowed(value: Boolean)

case class TaxiOrder(
    from: AddressFrom,
    to: AddressTo,
    when: LocalDateTime,
    price: Double,
    smokingAllowed: SmokingAllowed,
    petsAllowed: PetsAllowed)

val order: TaxiOrder = TaxiOrder(
    AddressFrom("NewYork"),
    AddressTo("Kyiv"),
    LocalDateTime.parse( text = "2020-10-10 13:15"),
    500d,
    SmokingAllowed(false),
    PetsAllowed(true),
)
```

```java
@AllArgsConstructor
static class Person {
  String name;
  String typ;
}

public static void main(String[] args) {
  Person student = new Person( name: "Jim",  typ: "student");
  Person teacher = new Person( name: "Ben",  typ: "teacher");
}
```

business value

extra discriminator

it's easily to write smth WRONG

Smart constructors

Factory

you can't pass the wrong value to the 'typ'

you can guarantee the value of the 'typ'

we built the same objects

```java
class Person {
  String name;
  String typ;

  private Person(String name, String typ) {
    this.name = name;
    this.typ = typ;
  }

  static Person student(String name) {
    return new Person(name,  typ: "student");
  }

  static Person teacher(String name) {
    return new Person(name,  typ: "teacher");
  }
}

public class CreatingObject3b {

  public static void main(String[] args) {
    Person student = Person.student( name: "Jim");
    Person teacher = Person.teacher( name: "Ben");
  }

Person wrong = new Person( name: "Jim",  typ: "friend");
```

Database

```java
public class User {
  String name;
  Email email;
  Iban iban;
  Integer id;
}
```

```java
public interface PaypalPayment {
  void doPay(Email from, Email to, int amount);
}
```

```java
public interface StripeTransaction {
  void make(Integer from, Integer to, int amount);
}
```

```java
public interface WireTransfer {
  void transfer(Iban from, Iban to, int amount);
}
```

WireTransfer (iban from, iban to, int amount)

Paypal Transfer (str email from, str email to, int amount)

Stripe Payment (int from, int to, int amt)

#12

Problems

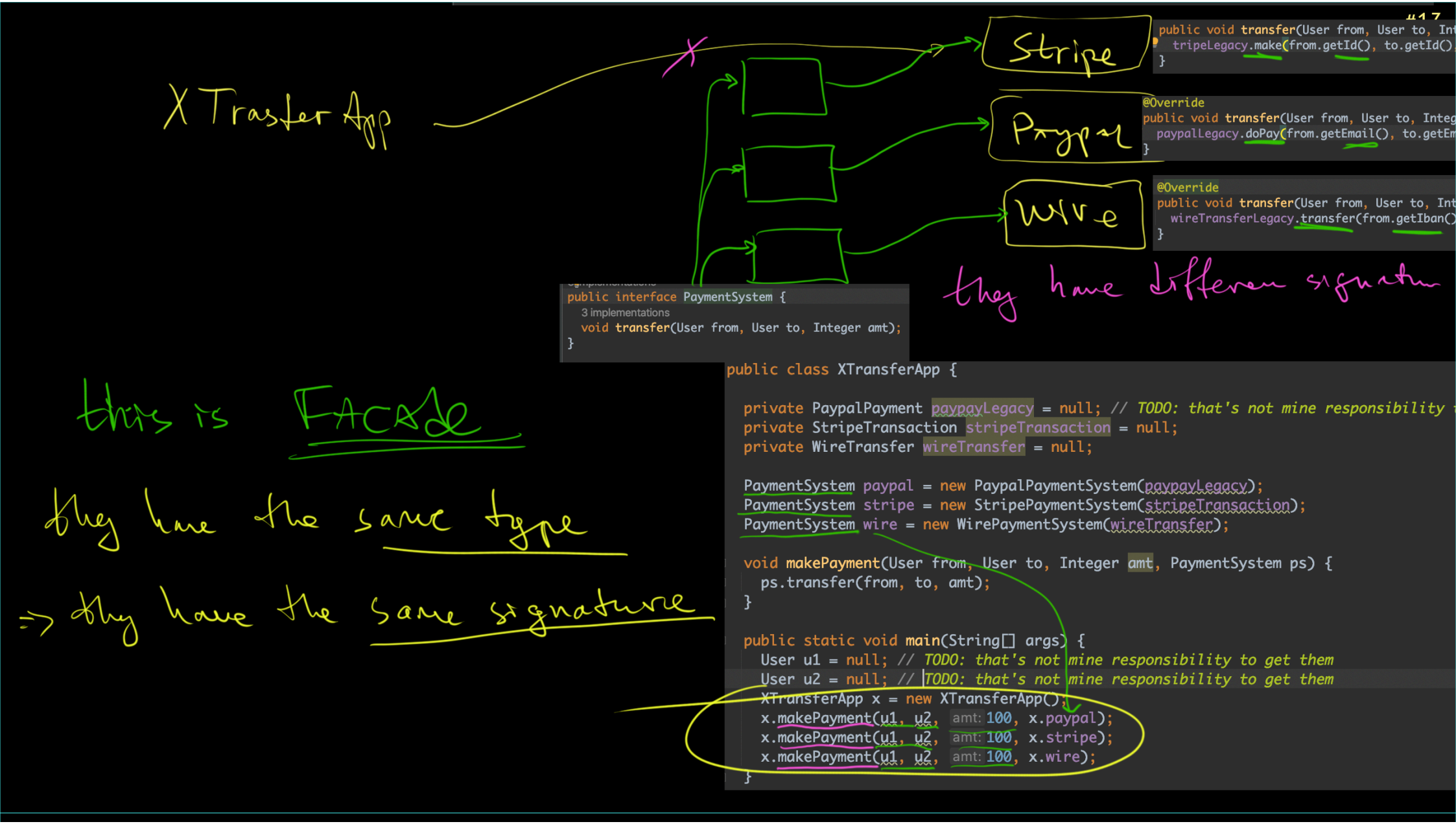
1. Different type / number/order of parameters
2. Different method names
3. —"— return types
4. Inconsistent Class Naming

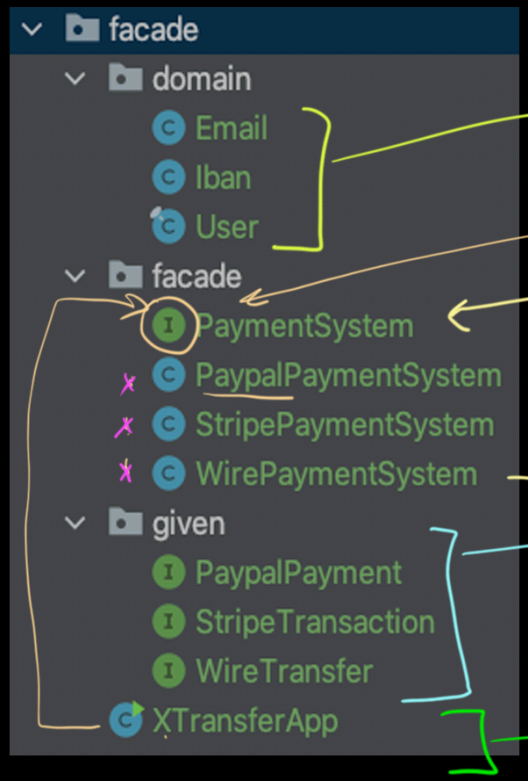
GOAL →

```java
public class XTransferApp {

  void makePayment(User from, User to, PaymentSystem ps) {
    ps.transfer(from, to);
  }

}
```


XTrasferApp
Stripe
Paypal
Wire
public void transfer(User from, User to, Int
tripeLegacy.make(from.getId(), to.getId()
}
@Override
public void transfer(User from, User to, Integ
paypalLegacy.doPay(from.getEmail(), to.getEm
}
@Override
public void transfer(User from, User to, Int
wireTransferLegacy.transfer(from.getIban()
}
they have different signatu
public interface PaymentSystem {
3 implementations
void transfer(User from, User to, Integer amt);
}
this is FACADE
they have the same type
=> they have the same signature
public class XTransferApp {
private PaypalPayment paypayLegacy = null; // TODO: that's not mine responsibility
private StripeTransaction stripeTransaction = null;
private WireTransfer wireTransfer = null;
PaymentSystem paypal = new PaypalPaymentSystem(paypayLegacy);
PaymentSystem stripe = new StripePaymentSystem(stripeTransaction);
PaymentSystem wire = new WirePaymentSystem(wireTransfer);
void makePayment(User from, User to, Integer amt, PaymentSystem ps) {
ps.transfer(from, to, amt);
}
public static void main(String[] args) {
User u1 = null; // TODO: that's not mine responsibility to get them
User u2 = null; // TODO: that's not mine responsibility to get them
XTransferApp x = new XTransferApp();
x.makePayment(u1, u2, amt: 100, x.paypal);
x.makePayment(u1, u2, amt: 100, x.stripe);
x.makePayment(u1, u2, amt: 100, x.wire);
}




→ data description

common interface ←

facades to given systems
now, conforming

given, we can't modify them

→ useCase.

SOLID

SRP
=

OCP
=
we can create
as many impl.
as we want.