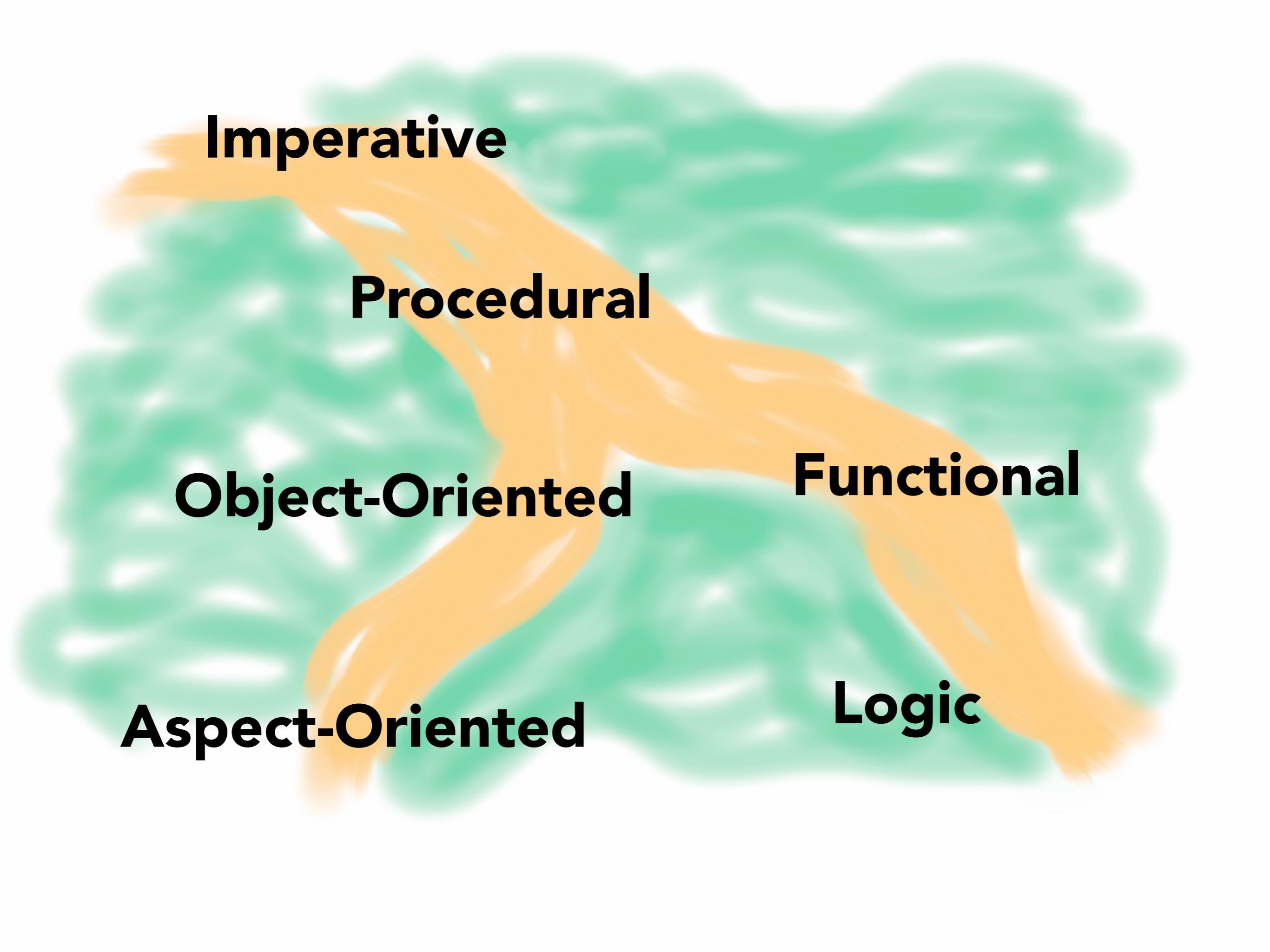


Functional principles for object-oriented development

@jessitron



The background of the image features a dynamic, abstract design composed of swirling, translucent orange and green layers. These layers overlap and flow across the frame, creating a sense of motion and depth. The colors are bright and vibrant, with the orange tones appearing in the upper right and lower left areas, while the green tones are more prominent in the center and upper left.

Imperative

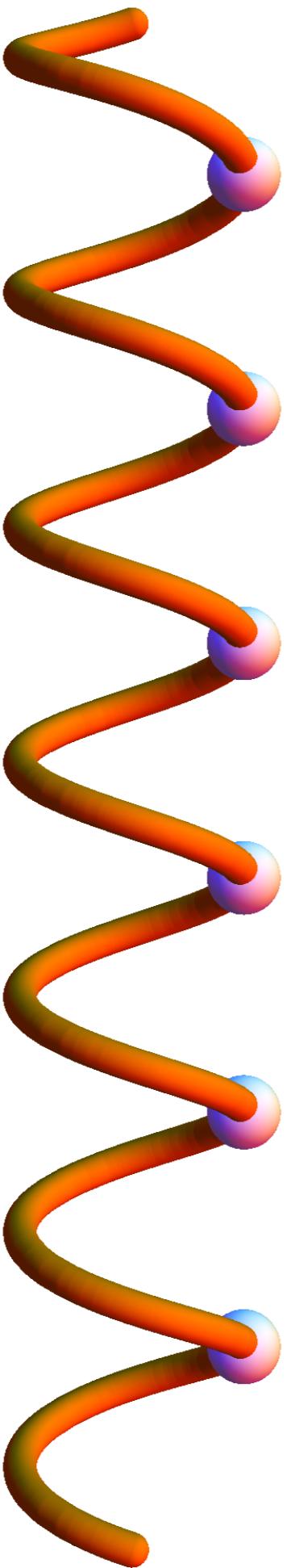
Procedural

Object-Oriented

Functional

Aspect-Oriented

Logic



Data In, Data Out

Specific Typing

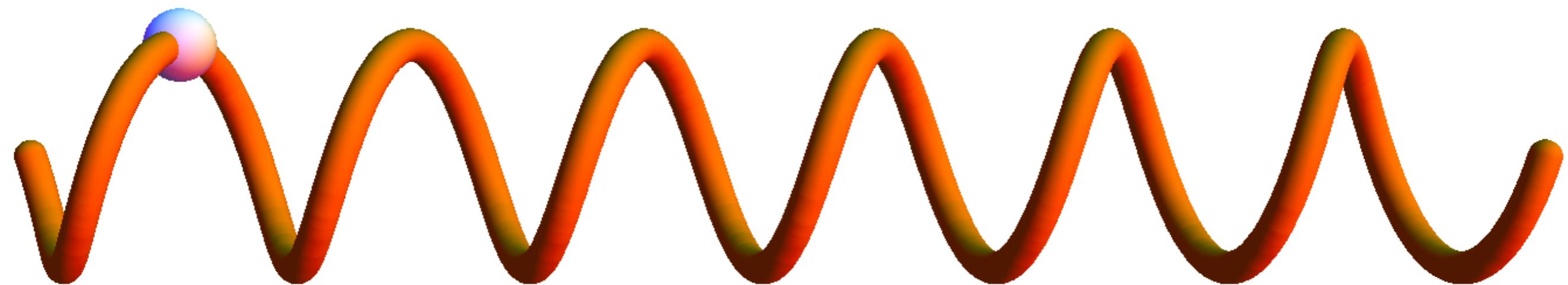
Verbs Are People Too

Immutability

Declarative Style

Lazy Evaluation

Data In, Data Out





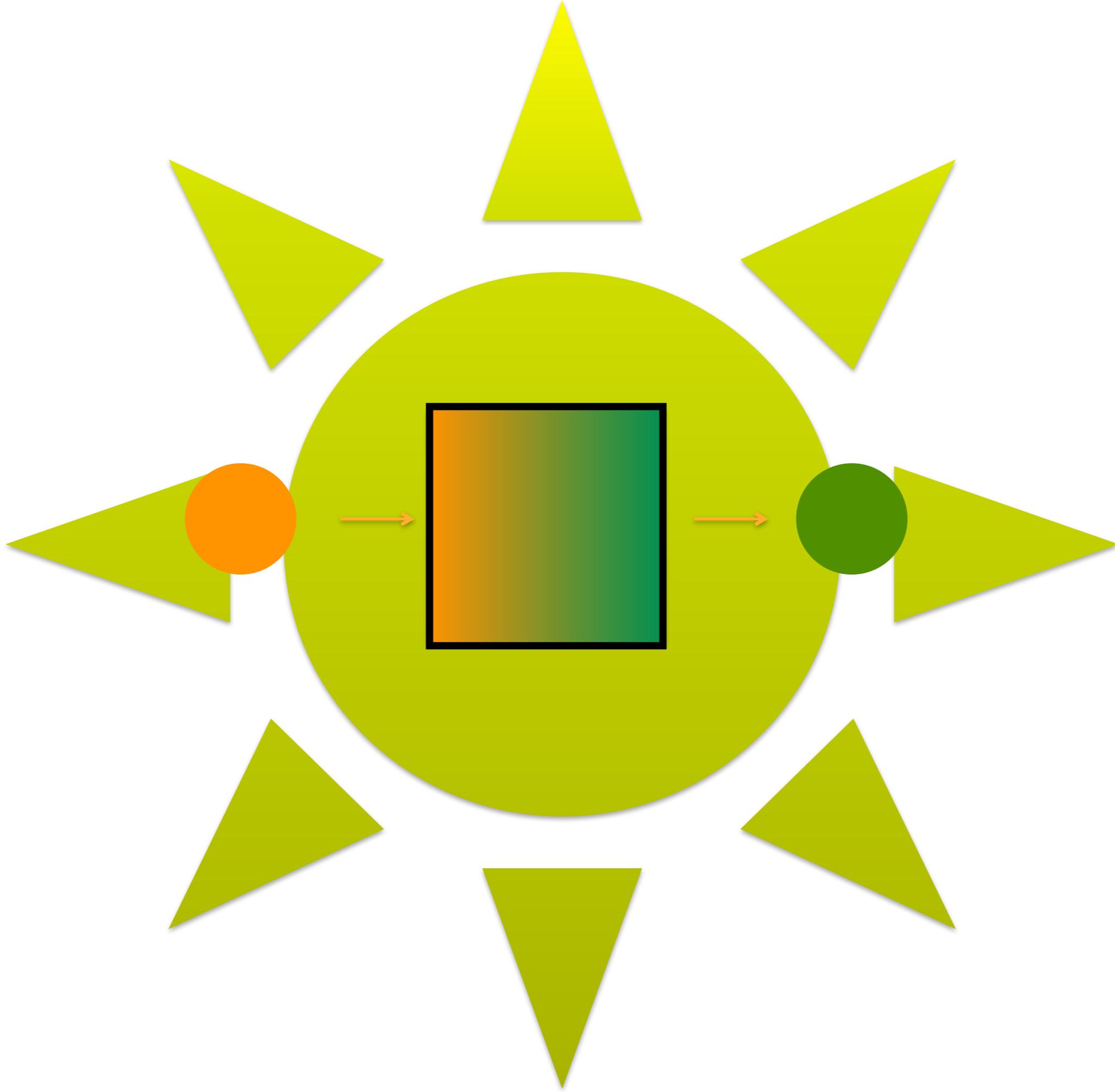
access global state



modify input



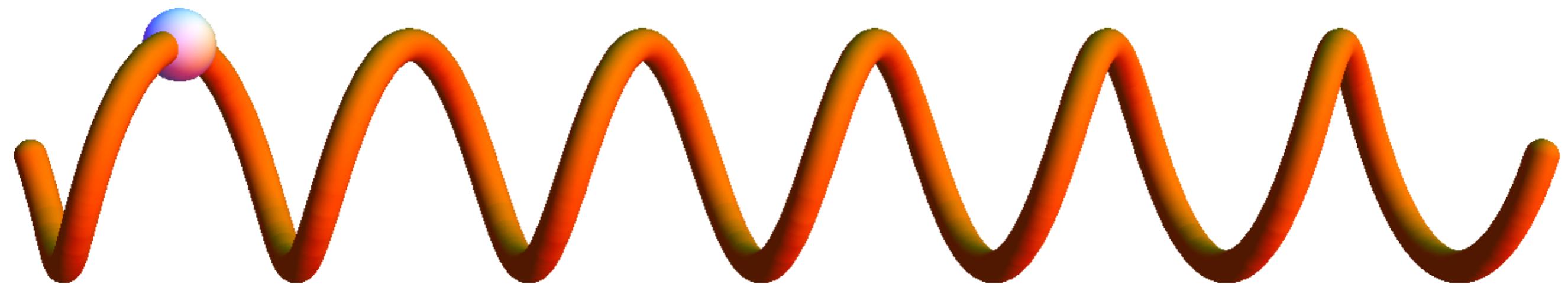
change the world



Testable



Easier to understand

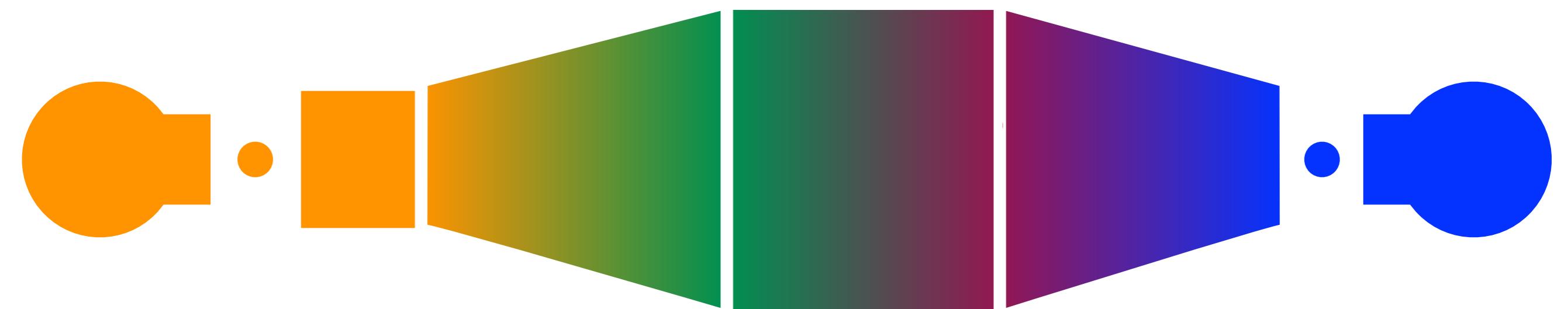


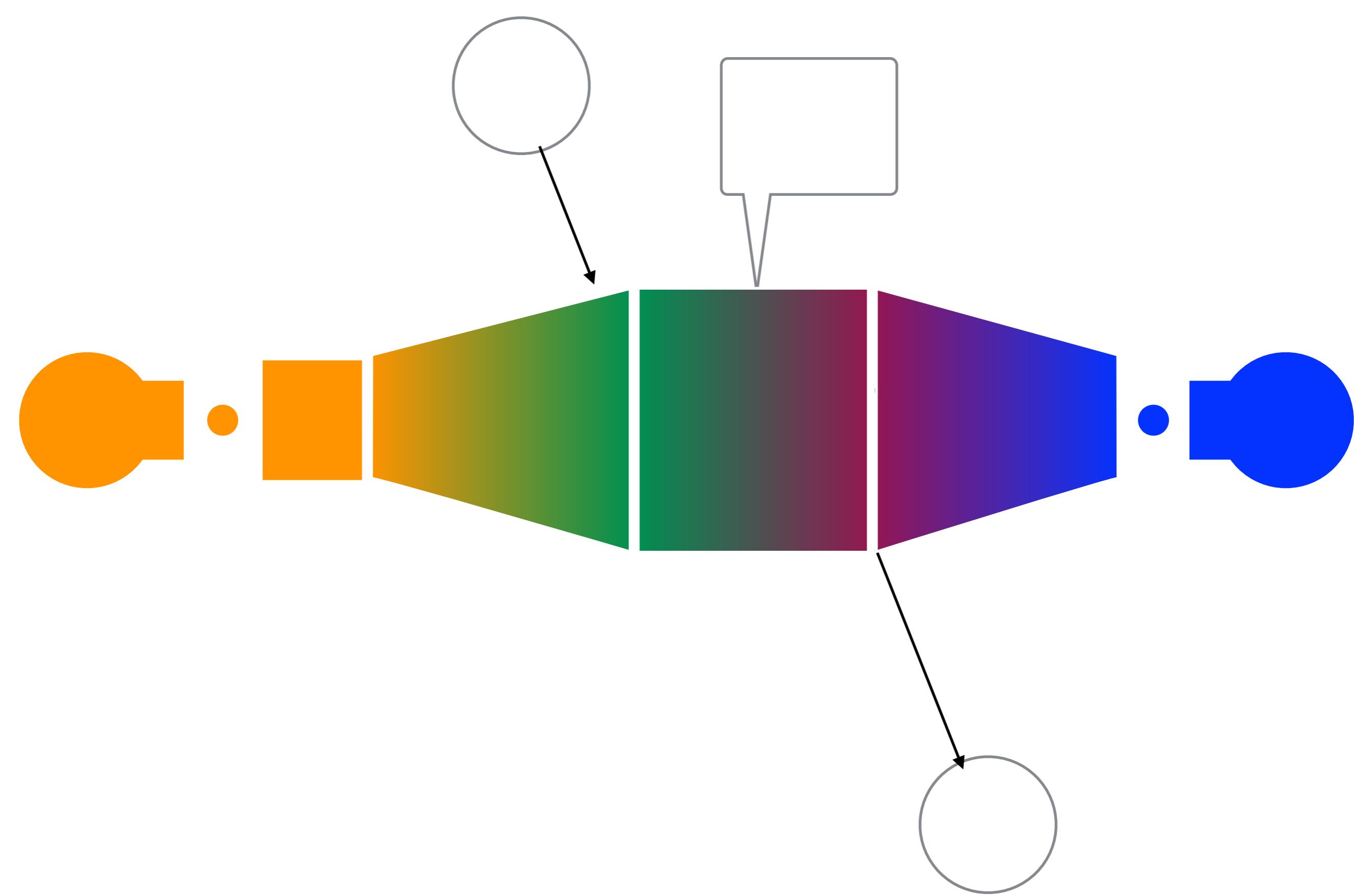
```
Password password;  
Email      email;
```

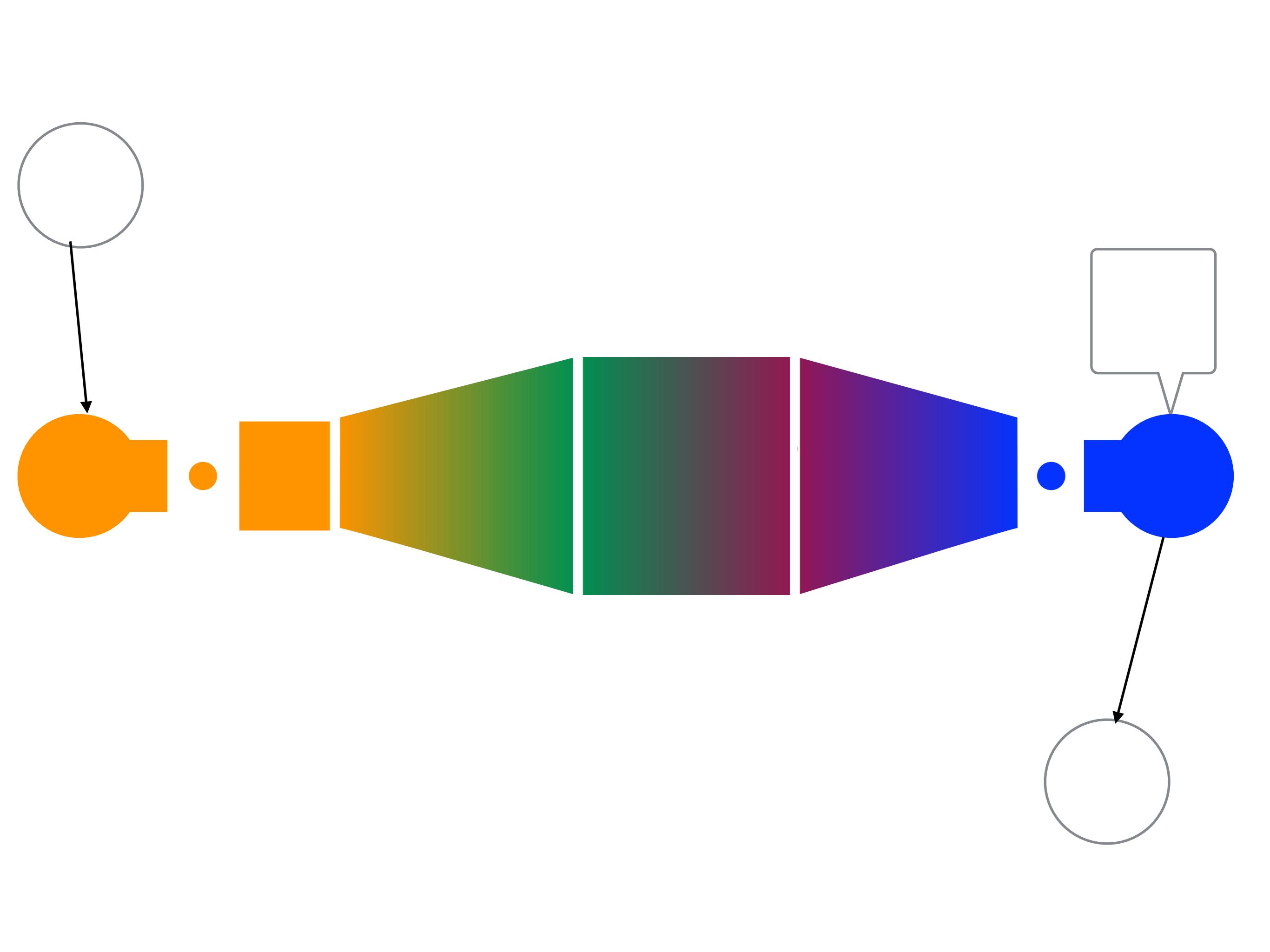
```
private boolean invalidPassword() {  
    return !password.contains(email);  
}
```

```
private boolean invalidPassword(  
    Password password,  
    Email email) {  
    return !password.contains(email);  
}
```

(Email, Password) -> boolean

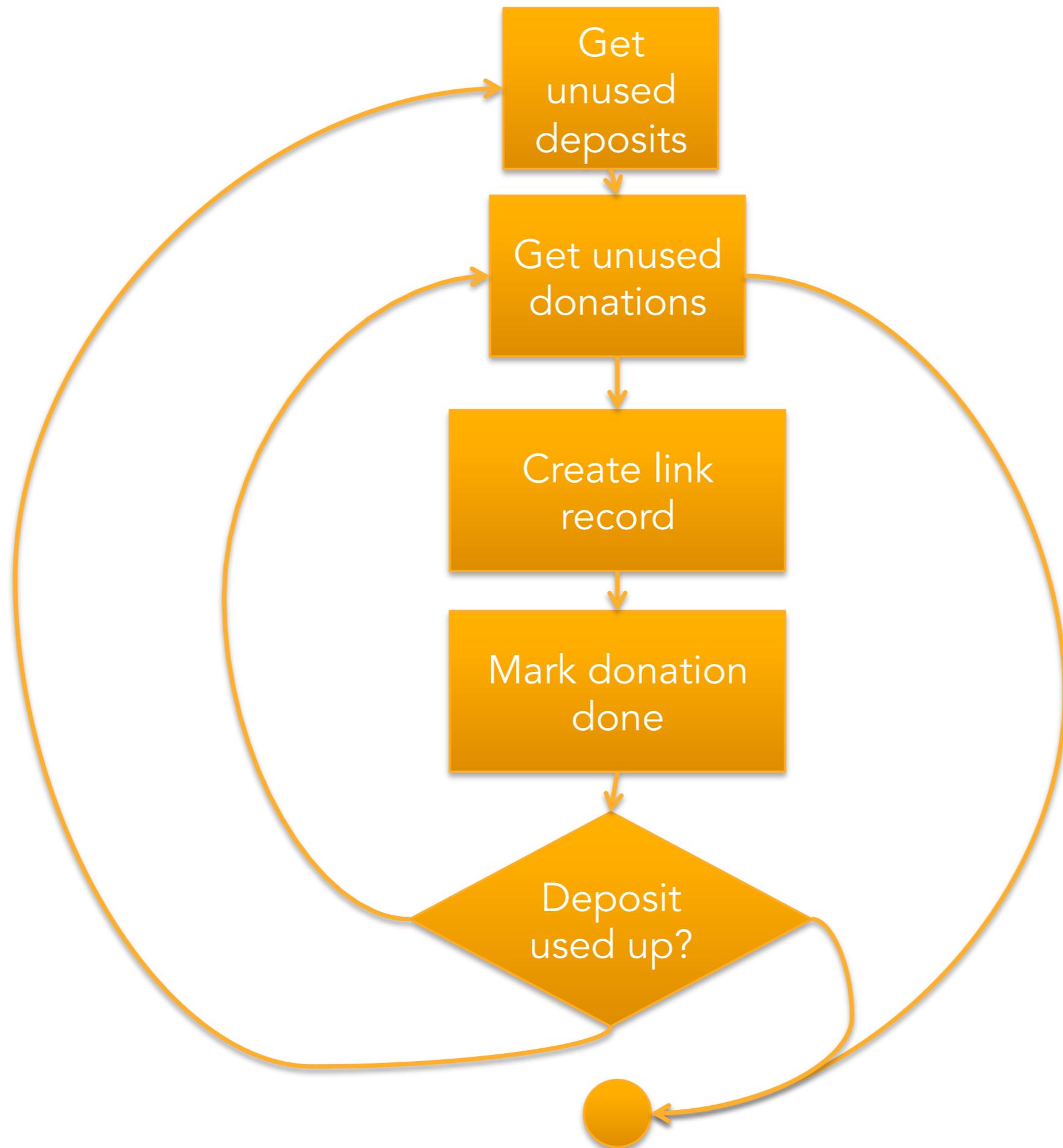


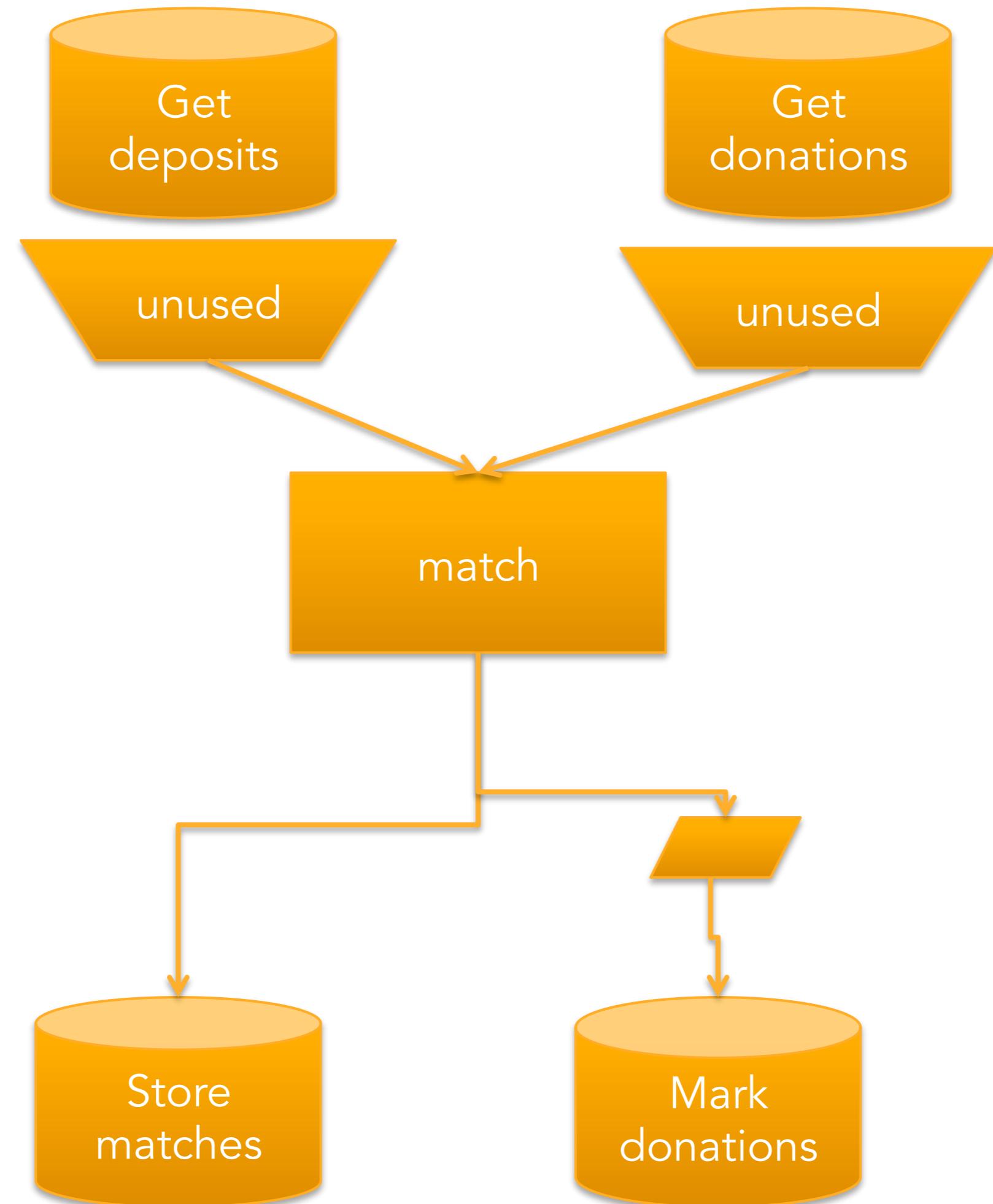


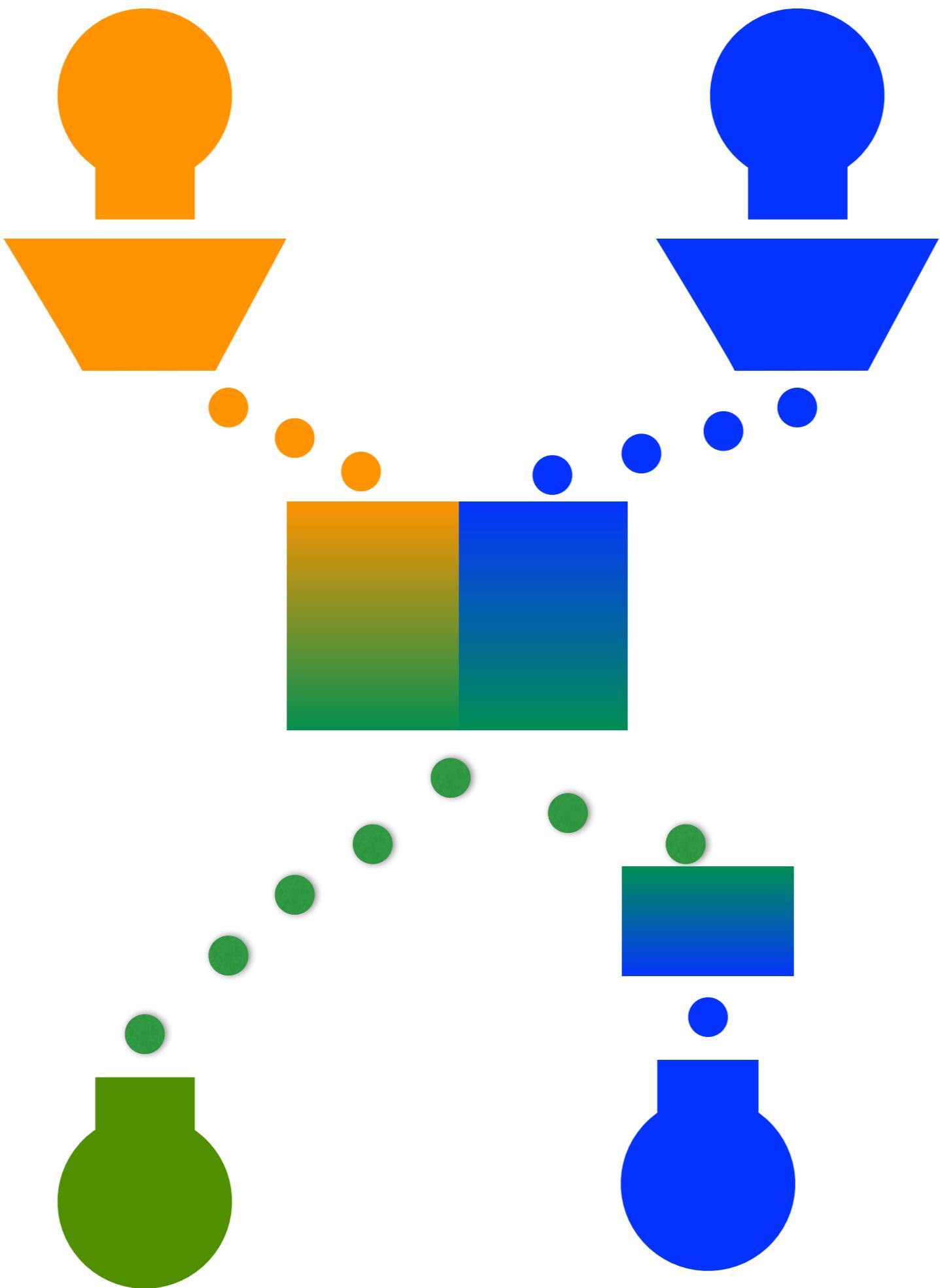




the flow of data







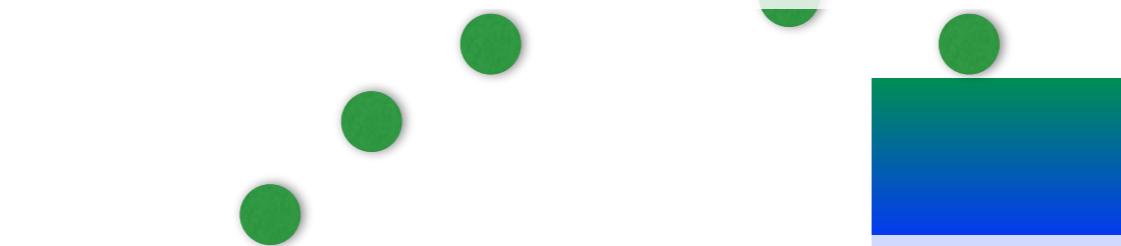


List<Deposit>



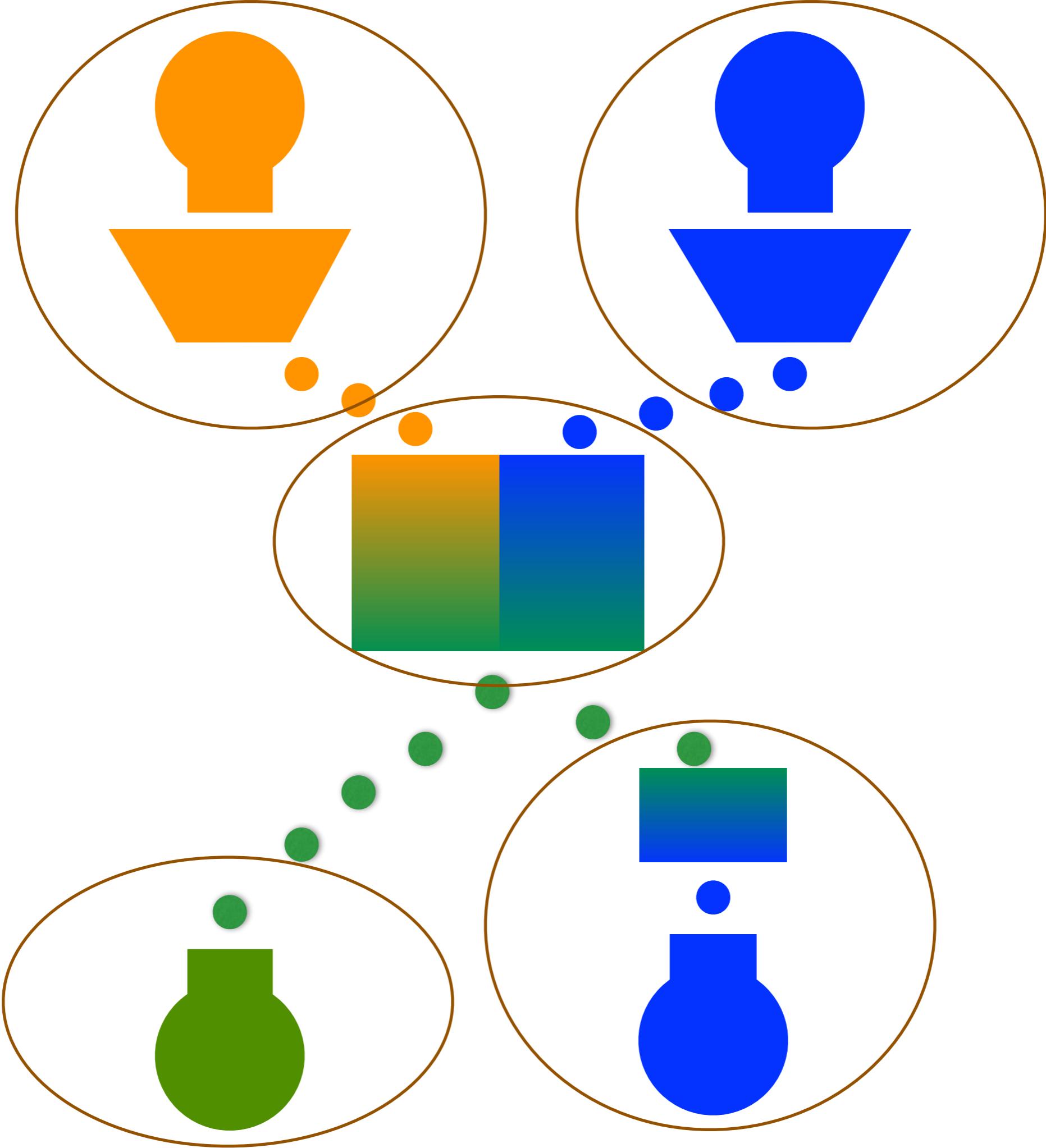
List<Donation>

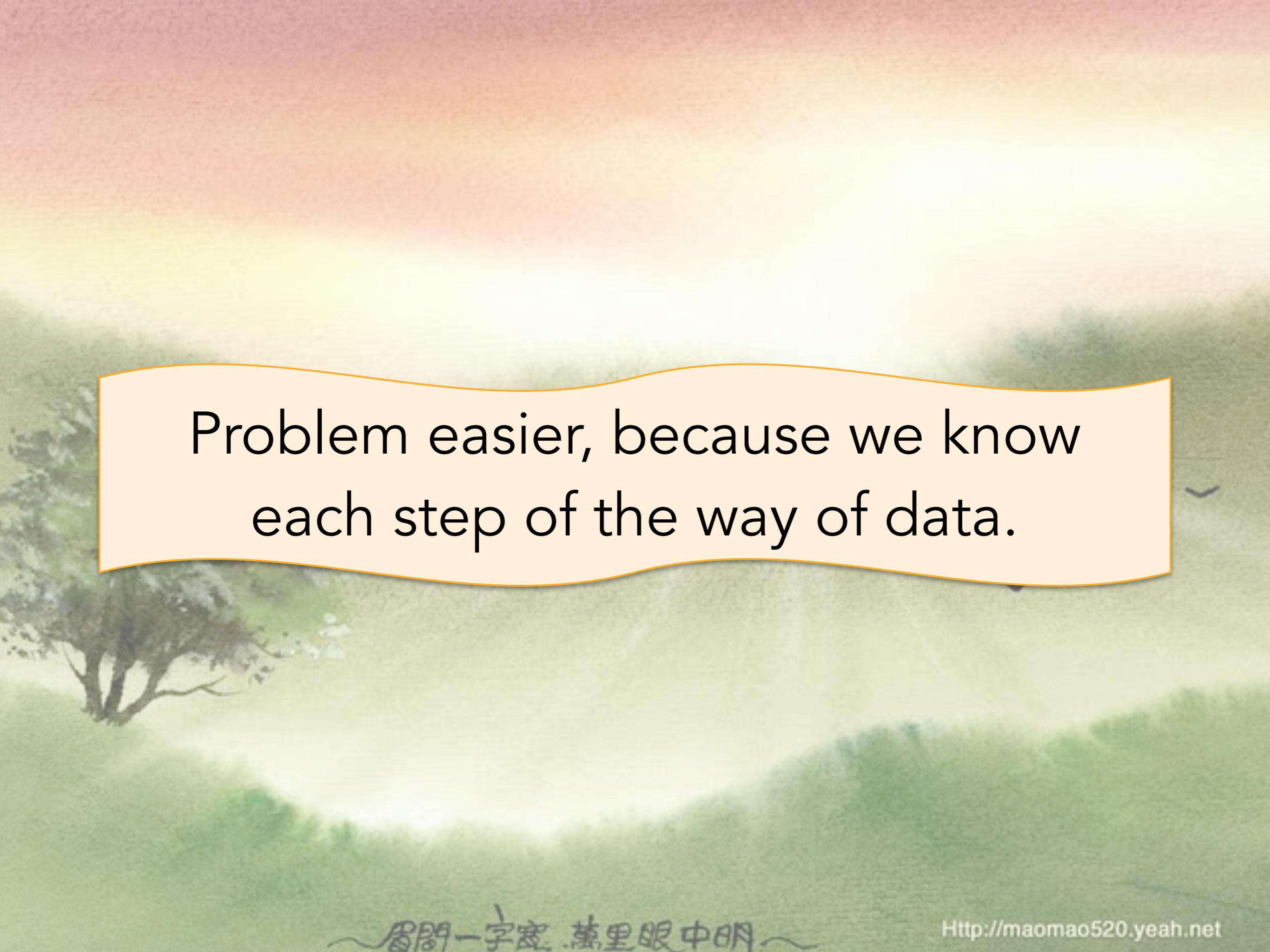
List<Match<Deposit, Donation>>



List<Donation>

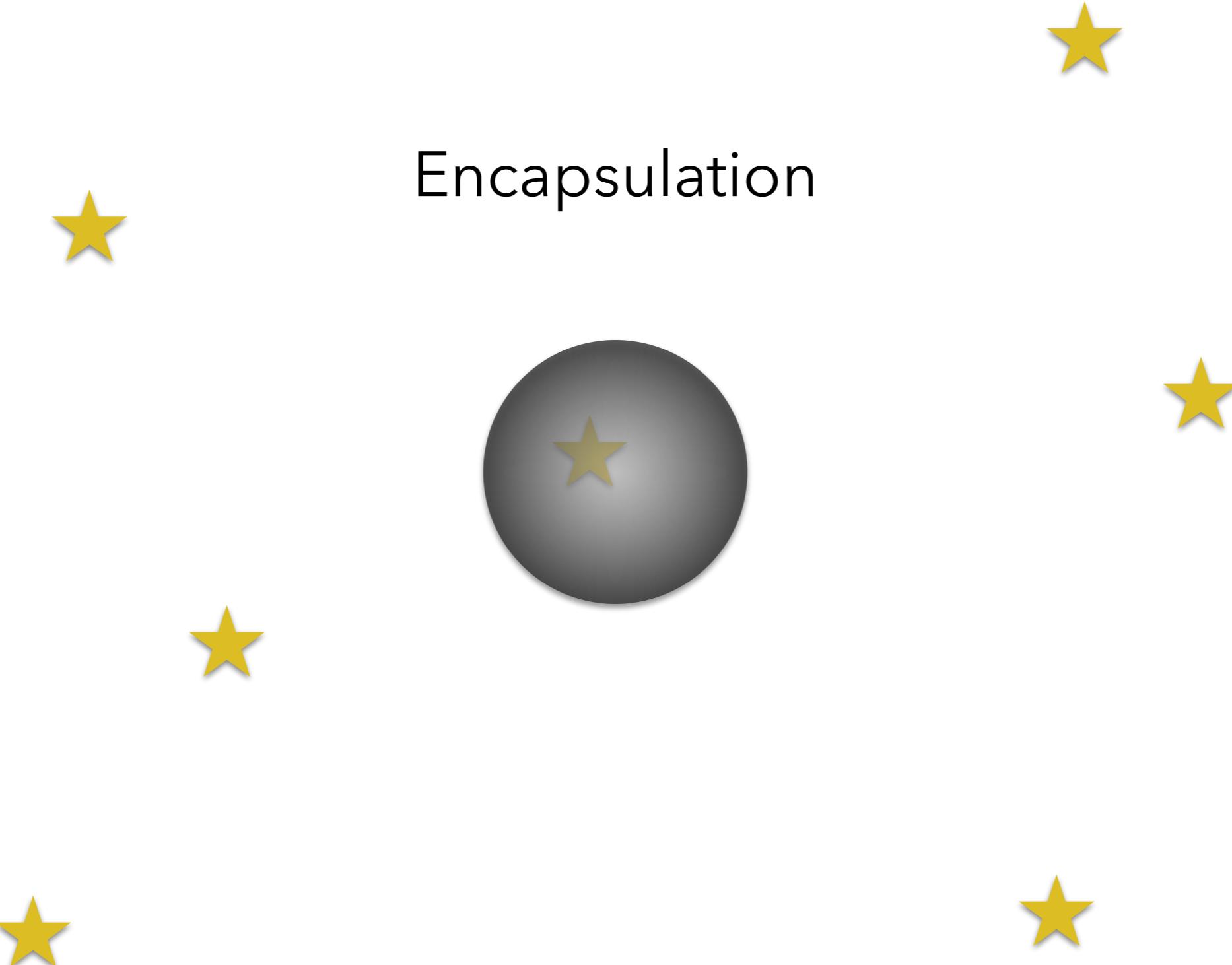




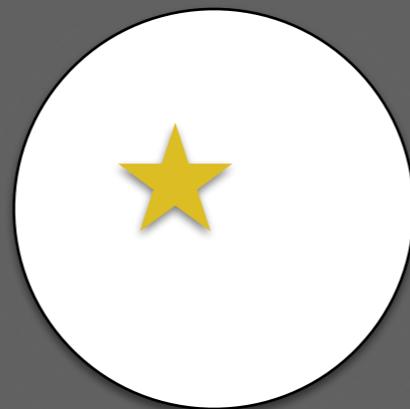


Problem easier, because we know
each step of the way of data.

Encapsulation



Isolation





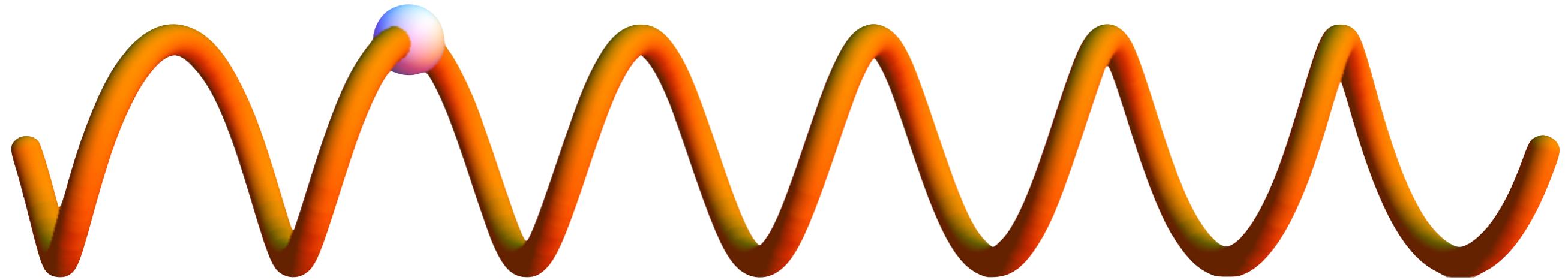
```
Response createAccount(UserDbService svc,  
                      Request req,  
                      Config cfg) {  
  
    ...  
  
    Account acc = constructAccount(...)  
    AccountInsertResult r = svc.insert(acc)  
    return respond(r);  
}
```

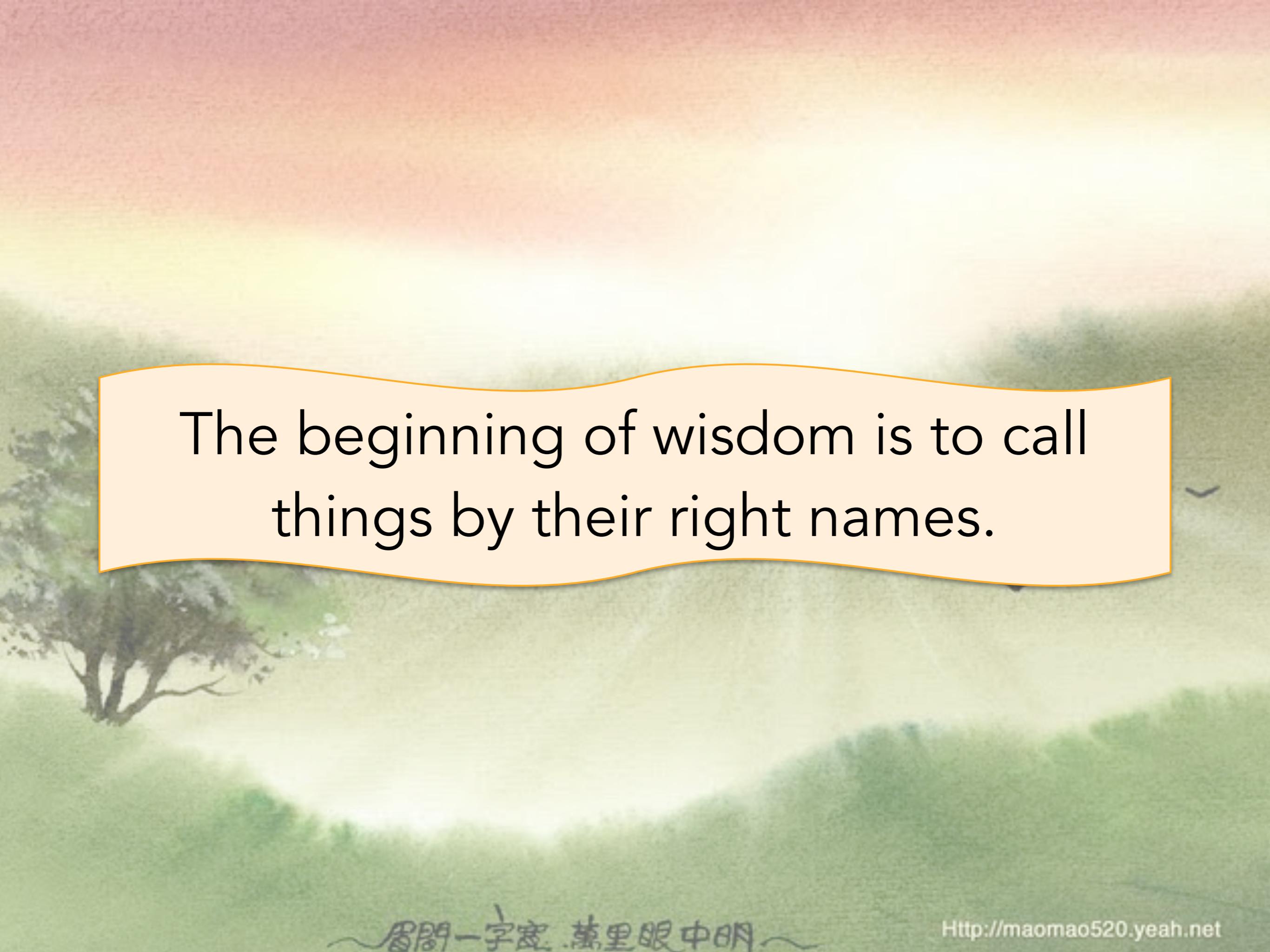


```
Response createAccount(  
    Function<Account, AccountInsertResult> svc,  
    Request req,  
    Config cfg) {  
  
    ...  
  
    Account acc = constructAccount(...)  
    AccountInsertResult r = svc.apply(acc)  
    return respond(r);  
}
```



Specific Typing





The beginning of wisdom is to call
things by their right names.

Java

```
public Customer(FirstName name, EmailAddress em)
```

```
public class FirstName {  
    public final String stringValue;  
  
    public FirstName(final String value) {  
        this.stringValue = value;  
    }  
  
    public String toString() {...}  
    public boolean equals() {...}  
    public int hashCode() {...}  
}
```

Scala

```
case class FirstName(value : String)
```

```
val friend = FirstName("Simon")
```

Scala

```
type FirstName = String
```

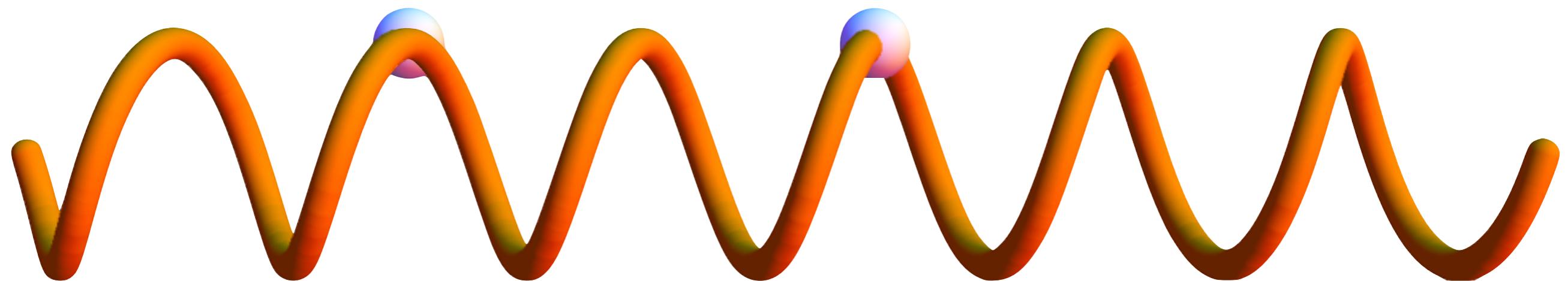
```
val friend: FirstName = "Simon"
```





layers of
thinking

expressiveness



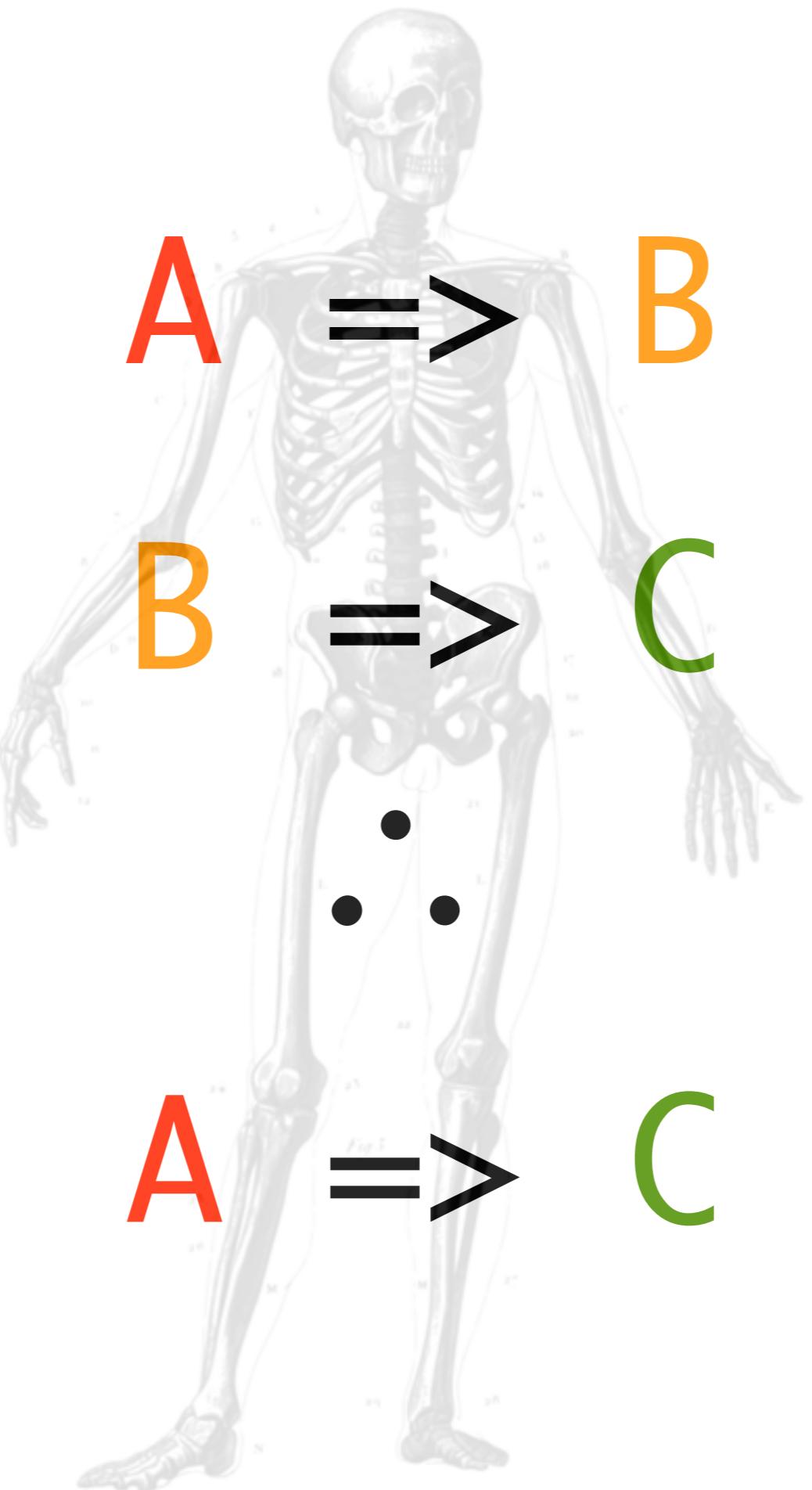


time => Cost

time => Volume

(Volume, Cost) => Expenditure

time => Expenditure



A → B

B => C

A => C

100

70

40

10

Perl

Ruby

Scala

Java

F#

Haskell

Account => AccountInsertResult

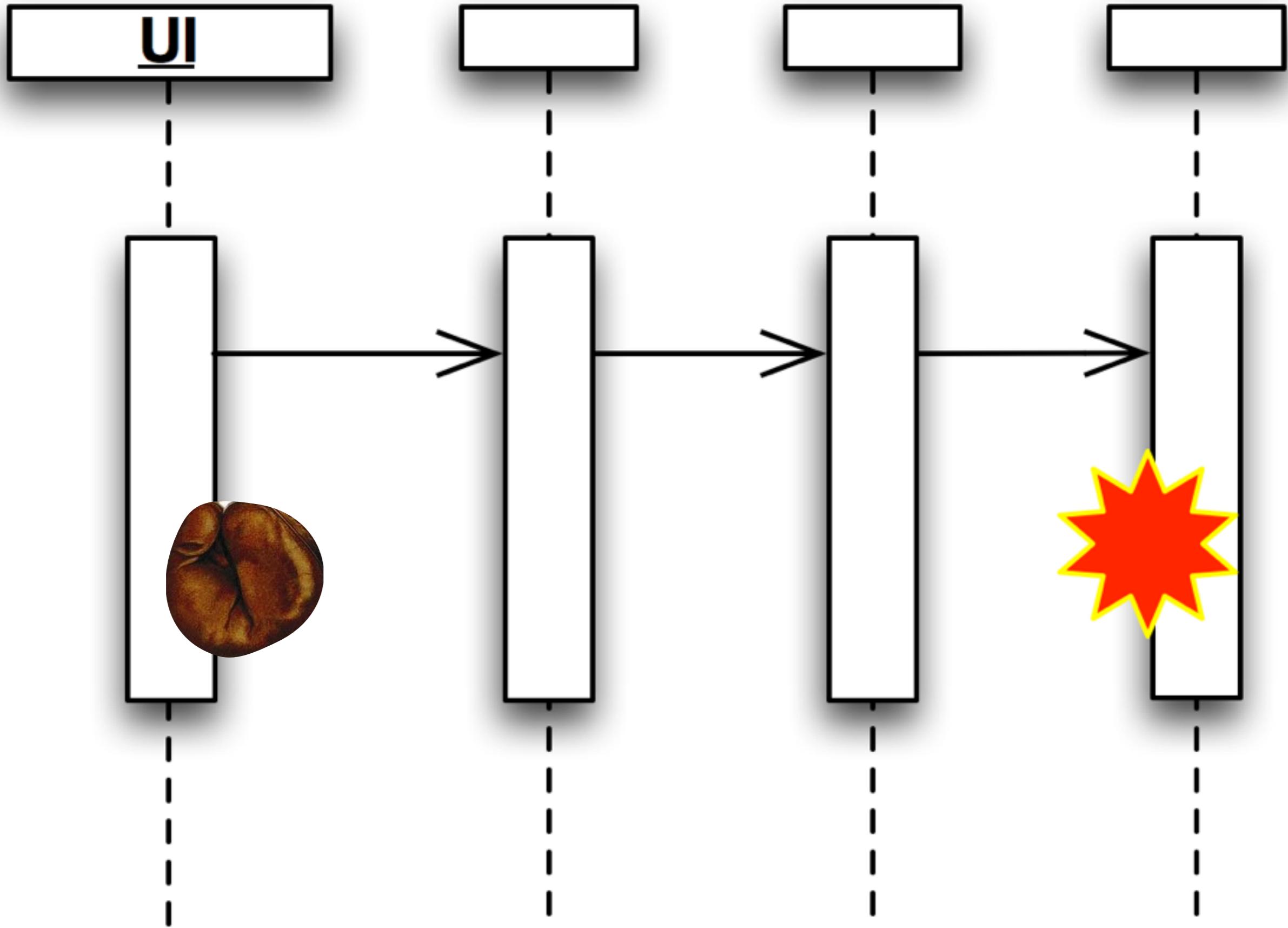
Account =>

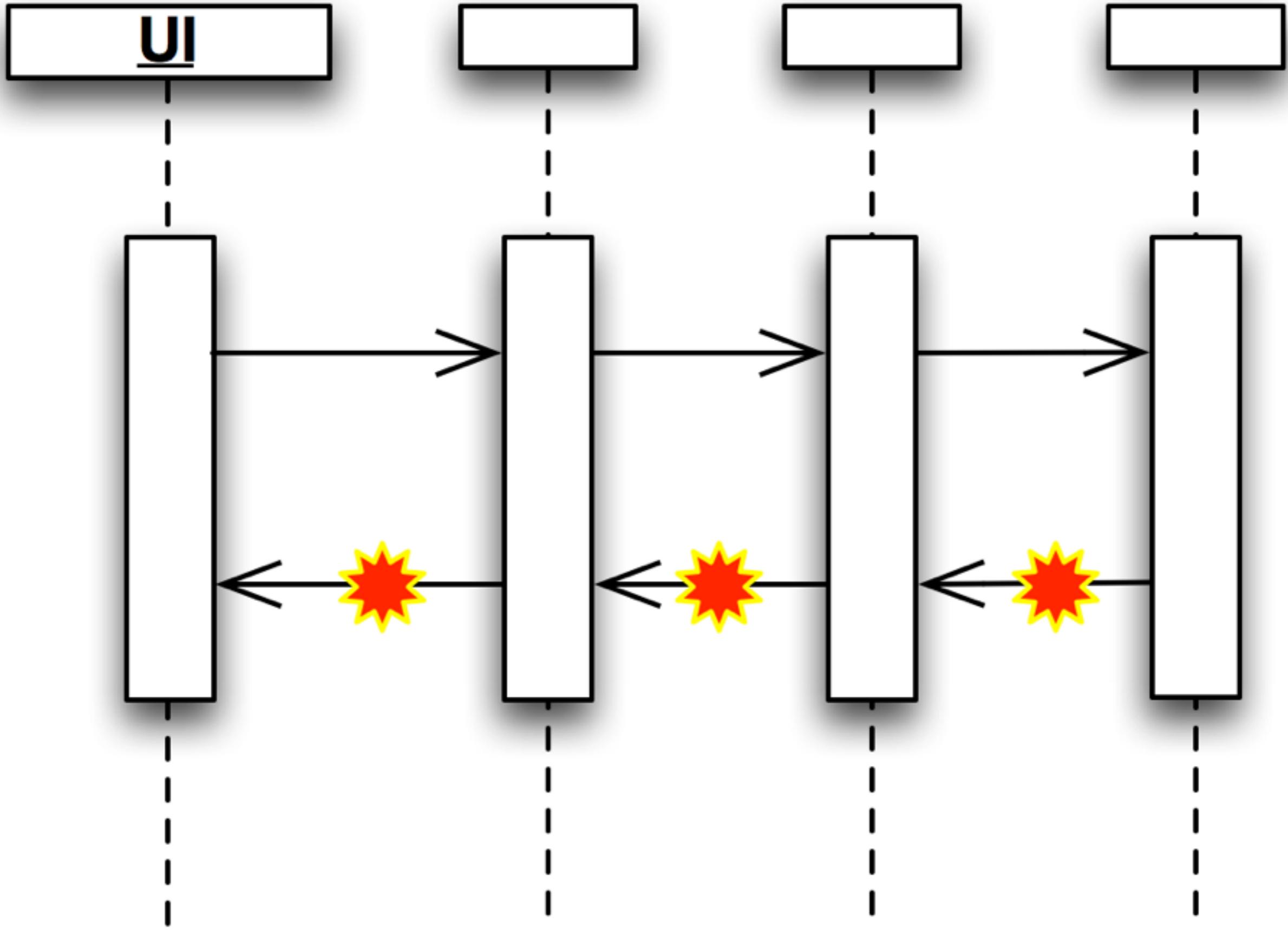
Either[**AccountCreationFailure**,
AccountInserted]

Account =>

Either[**AccountCreationFailure**,
AccountInserted]

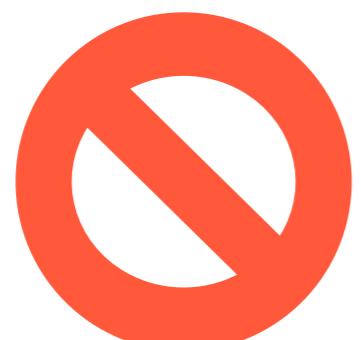
Errors are data too







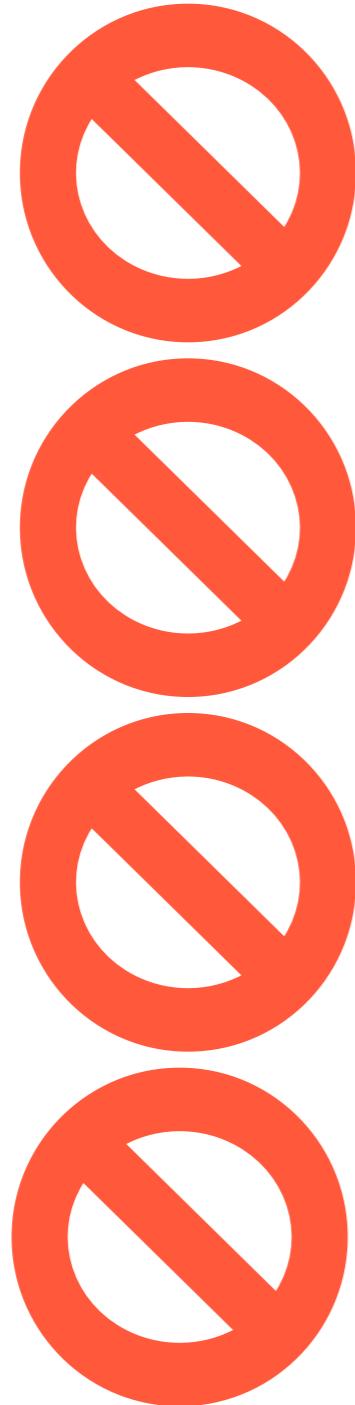
access global state



modify input



change the world



access global state

modify input

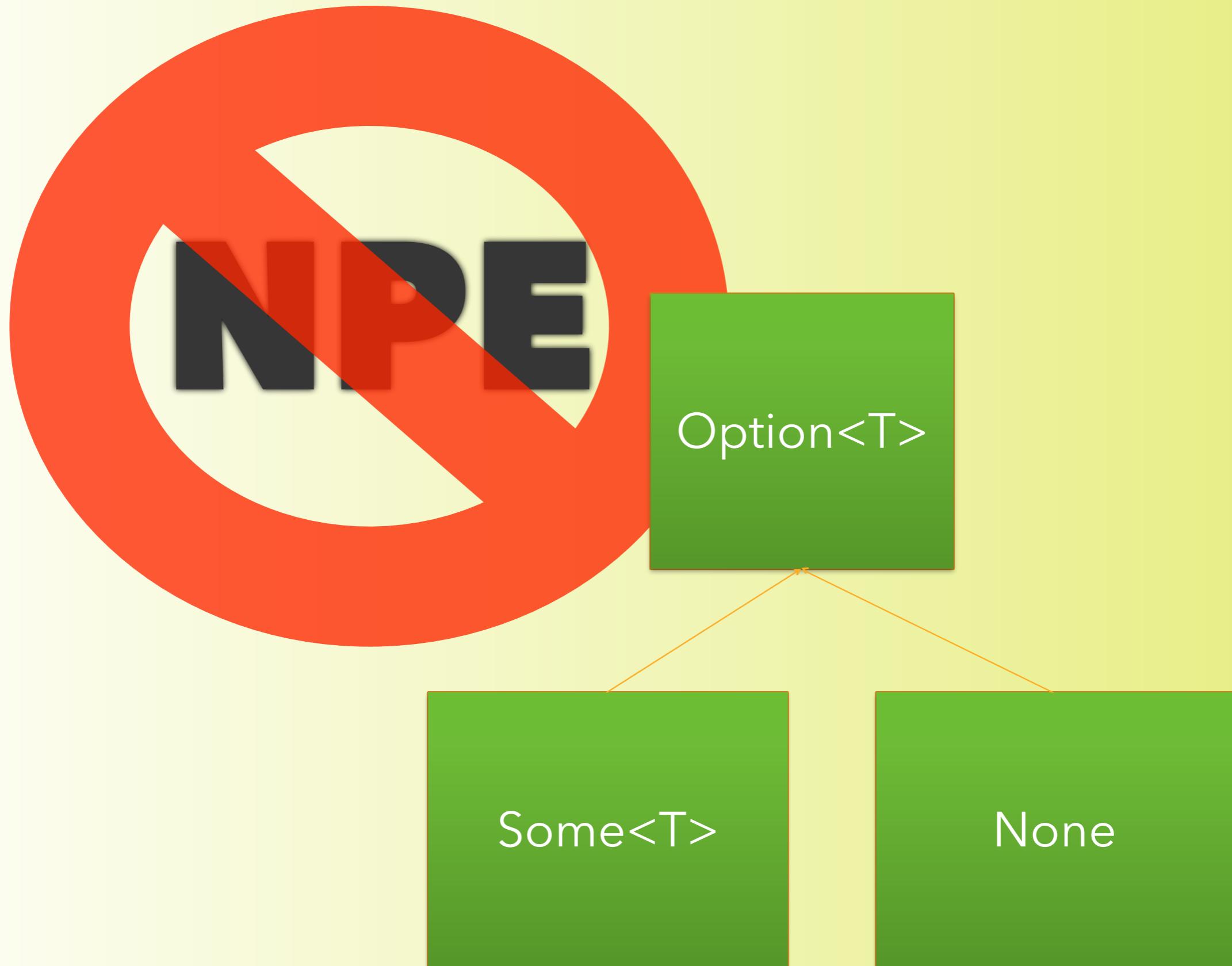
change the world

interrupt execution flow

Account =>

Either[**AccountCreationFailure**,
AccountInserted]

this talk is brought to you by... the Option type!

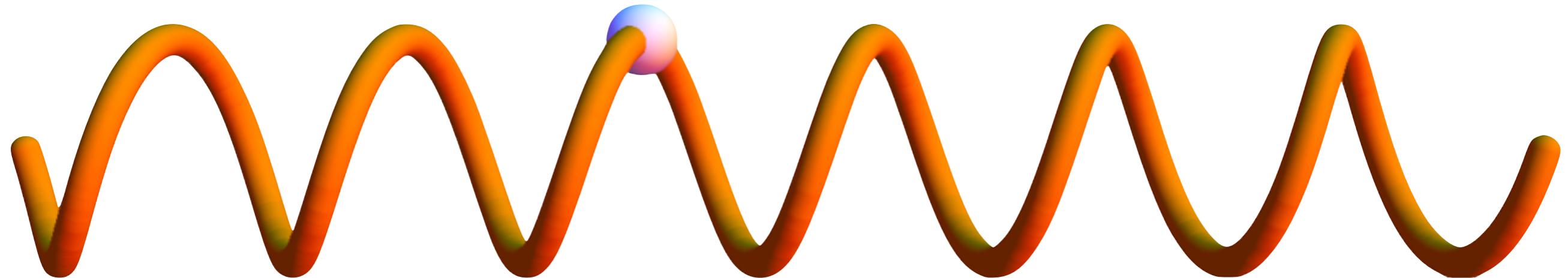


FSharpOption<T>

Optional<T>

... because null is not a valid object reference.

Verbs Are People Too





Java

```
class Inflation implements FunctionOverTime
{
    public float valueAt(int t) {
        return // some calculated value;
    }
}
```

Java

```
class Variable implements FunctionOverTime
{
    private final Function<Int, Double> vals;
    ...
    public float valueAt(int t) {
        return vals.apply(t);
    }
}
```

Java

```
Variable inflation = new Variable(  
    “Cost Inflation”,  
    t -> Math.Pow(1.15,t));  
  
inflation.valueAt(3);
```

Java 6

```
Variable inflation = new Variable("Inflation",
new Function<Integer, Double>() {
@Override
public Double apply(Integer input)
{
    return Math.pow(1.15, input) ;
}
});
```

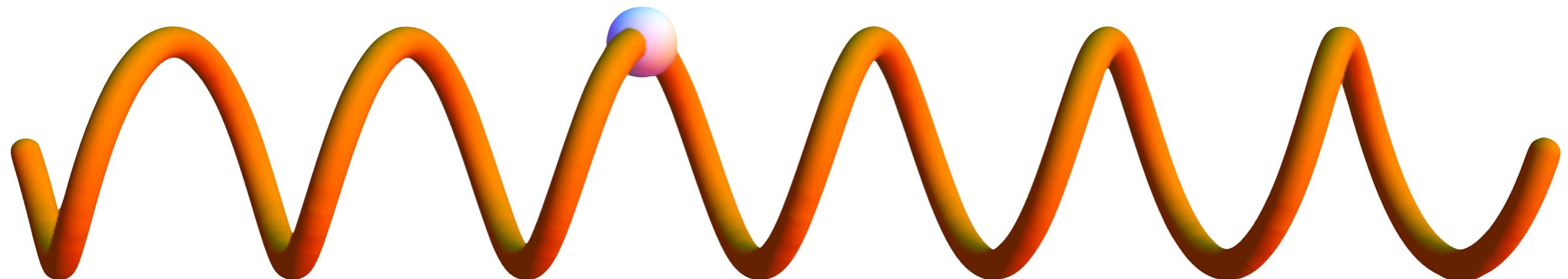


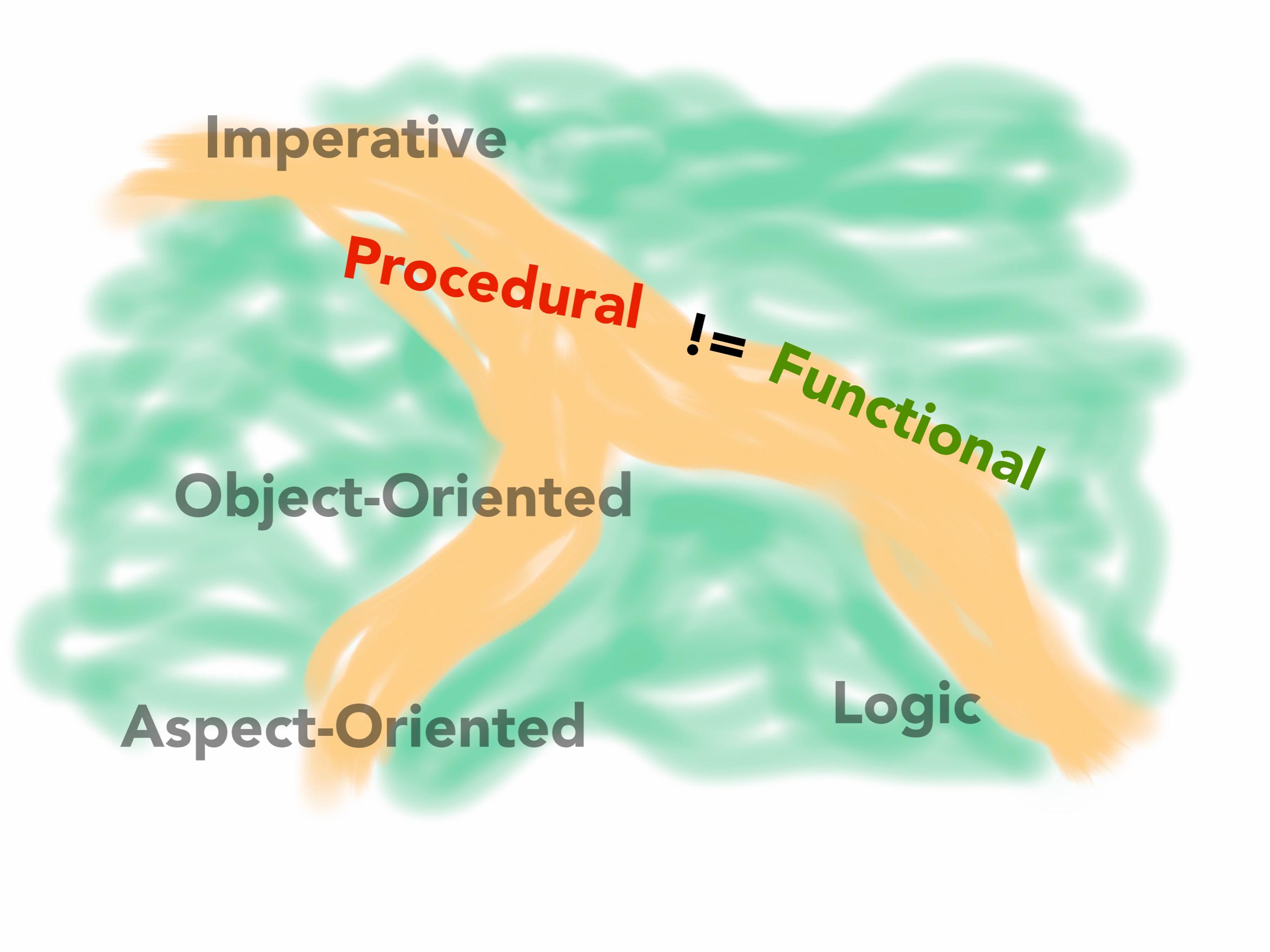
OnClick()



release ()

Command Strategy





Imperative

Procedural

!= Functional

Object-Oriented

Aspect-Oriented

Logic

```
withTransaction( )
```

```
// start
```

```
// end
```

```
Response r = createAccount(  
    (acc) -> userDB.insertAccount(acc),  
    req,  
    cfg);
```

```
Response r = createAccount(  
    acc ->  
    withTransaction(userDB.insertAccount(acc)),  
    req,  
    cfg);
```

```
Response r = createAccount(  
    acc ->  
    retrying(  
        withTransaction(userDB.insertAccount(acc))),  
    req,  
    cfg);
```

```
retrying( )
```

The diagram illustrates the nesting of three try blocks. It features a large outer rectangle with a solid black border. Inside this is a smaller rectangle with a dotted black border. Within that is another small rectangle with a dotted black border. The word "Code" is written in orange text inside each of the three inner rectangles. The top line of the outer rectangle contains the word "retrying" followed by an opening parenthesis, and the closing parenthesis is positioned at the top edge of the innermost dotted rectangle.

Code

Code



Idempotence



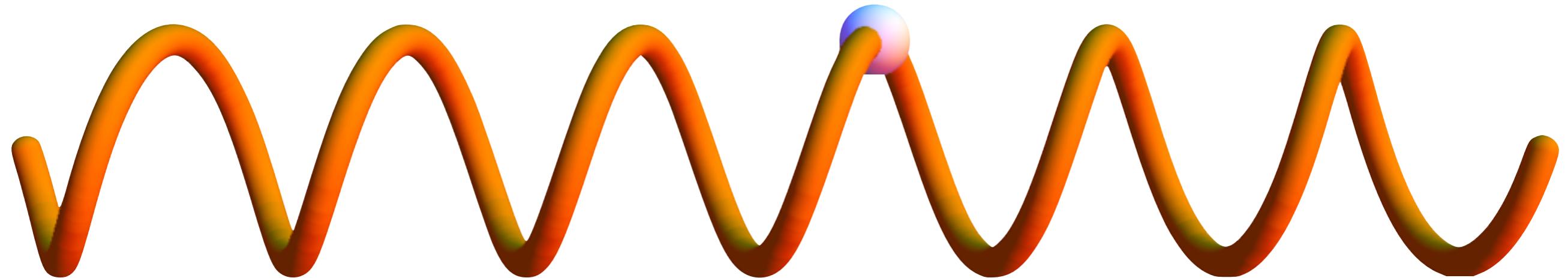
OnClick()



release ()



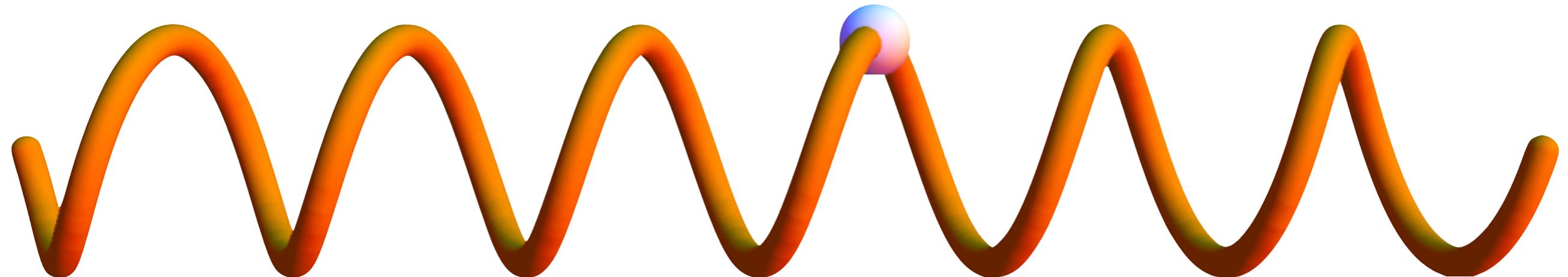
Immutability



String

Effective Java

Effective C#



Scala

```
val qty = 4 // immutable
```

```
var n = 2 // mutable
```

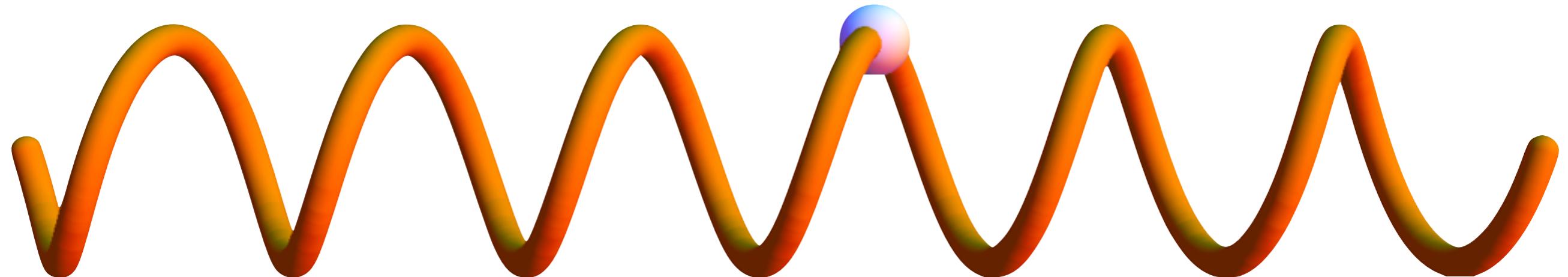
F#

```
let qty = 4          // immutable  
let mutable n = 2 // mutable  
n = 4          // false  
n <- 4        // destructive update
```

Concurrency



fewer possibilities



Java: easy

```
public class Address {  
    public final String city;  
  
    public Address(String city) {  
        this.city = city  
    }  
...}
```

Java: defensive copy

```
private final ImmutableList<Phone> phones;  
  
public Customer (Iterable<Phone> phones) {  
    this.phones = ImmutableList.copyOf(phones);  
}
```



Java: copy on mod

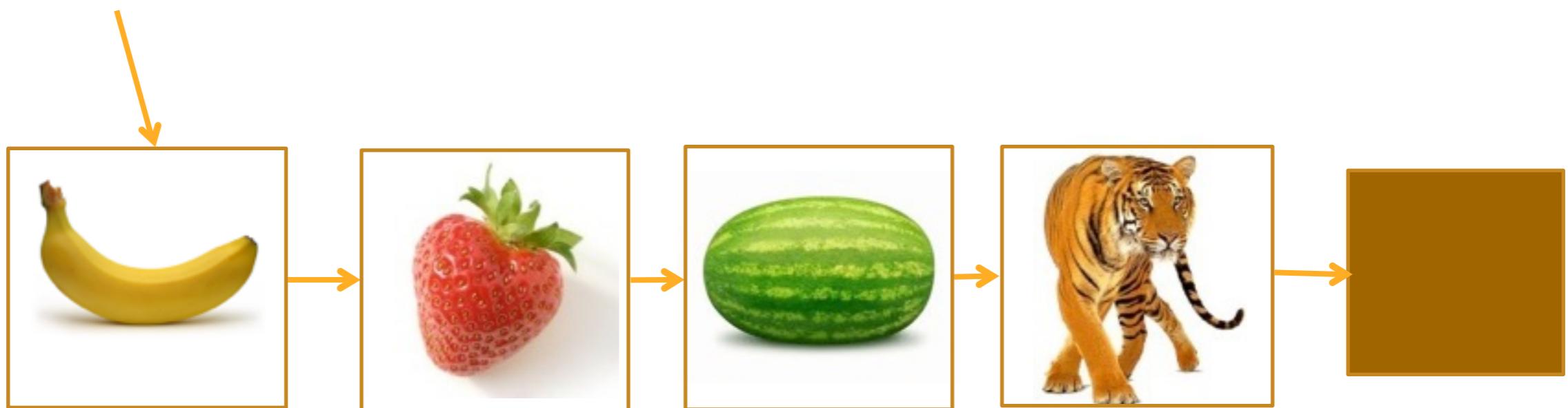
```
public Customer addPhone(Phone newPhone) {  
    Iterable<Phone> morePhones =  
        ImmutableList.builder()  
            .addAll(phones)  
            .add(newPhone).build();  
    return new Customer(morePhones);  
}
```



F#: copy on mod

```
member this.AddPhone (newPhone : Phone) {  
    new Customer(newPhone :: phones)  
}
```

fruit



fruit.add(tomato)



persistent data structure

persistent data structure

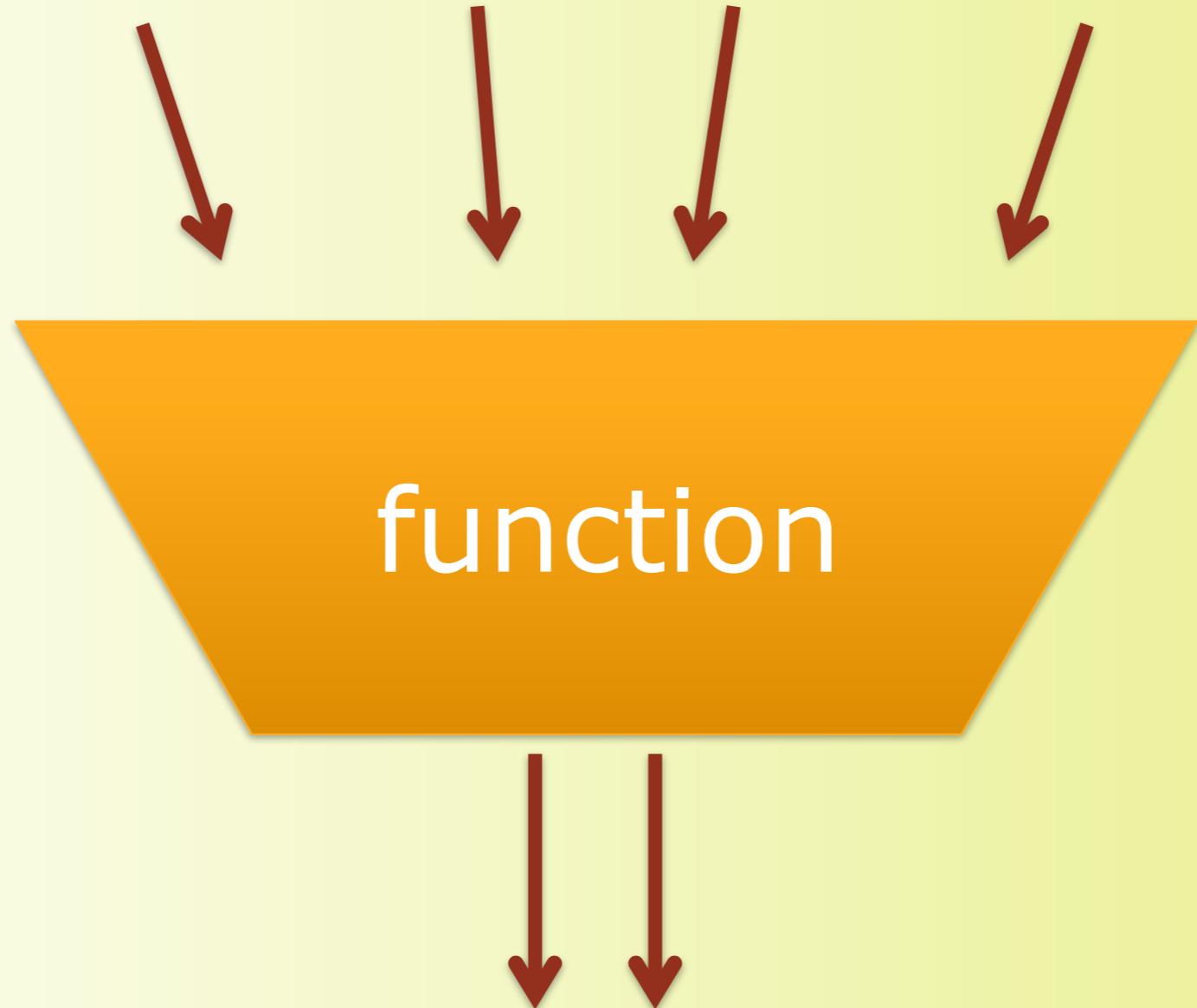


C#: shallow copy

```
public Customer AddPhone(Phone newPhone) {  
    IEnumerable<Phone> morePhones =  
        // create list  
    return new Customer(morePhones,  
                        name,  
                        address,  
                        birthday ,  
                        cousins);  
}
```



this talk is brought to you by... the Tuple type!

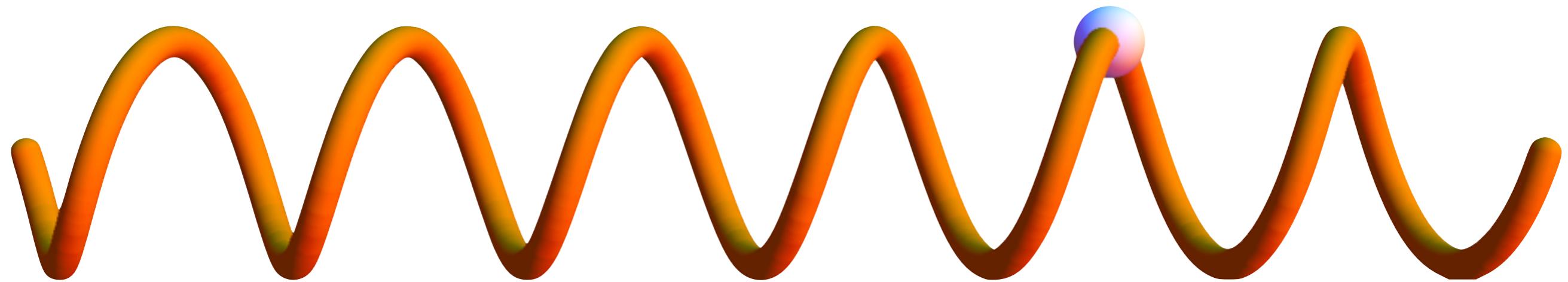


```
new Tuple<string,int>("You win!", 1000000)
```

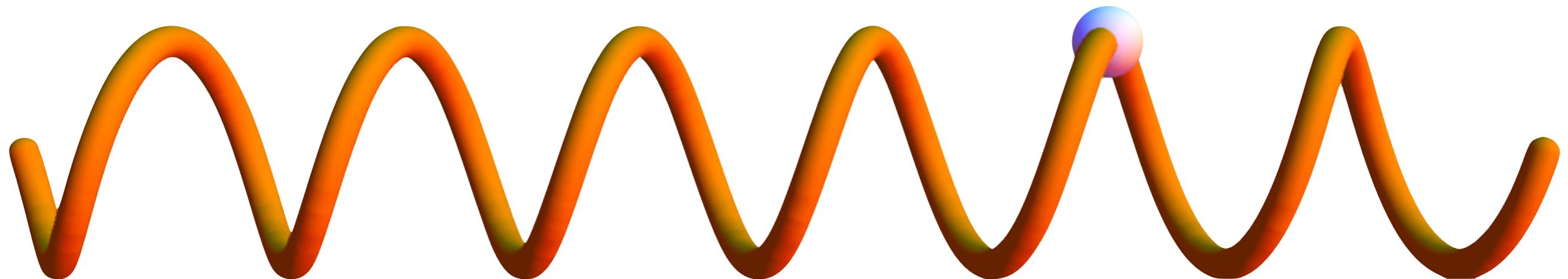
Tuple<T1,T2,...>

When one return value is not enough!

Declarative Style



say **what** you're doing,
not **how** you're doing it



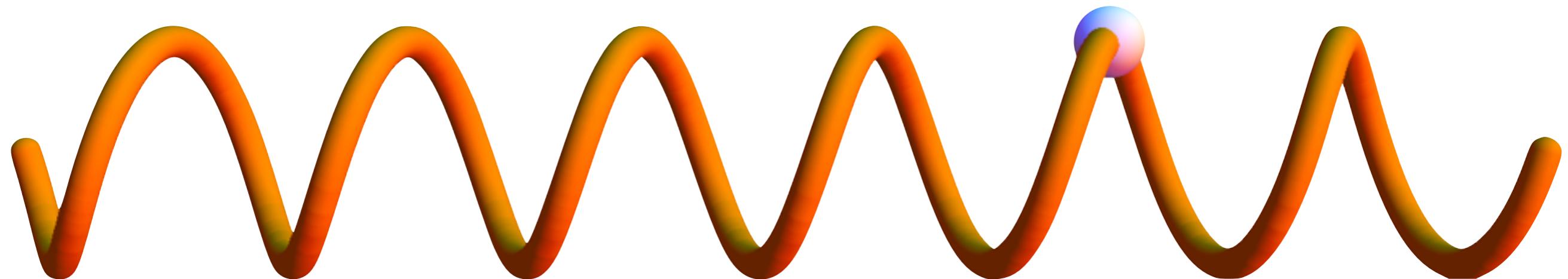
Never tell people how to do things. Tell them what to do and they will surprise you with their ingenuity.

```
select ROLE_NAME,  
UPDATE_DATE  
from USER_ROLES  
where USER_ID = :userId
```

readable code



only the essentials



Java

```
public List<String> findBugReports(List<String> lines)
{
    List<String> output = new LinkedList();
    for(String s : lines) {
        if(s.startsWith(“BUG”)) {
            output.add(s);
        }
    }
    return output;
}
```

familiar != readable

Java

```
lines.stream()  
    .filter(s -> s.startsWith("BUG"))  
    .collect(Collectors.toList)
```

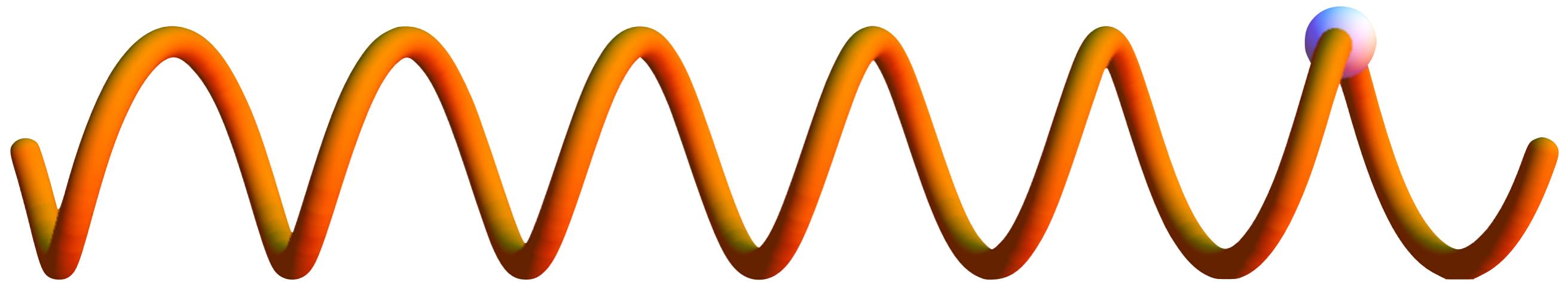
Java

```
lines.stream().parallel()  
    .filter(s -> s.startsWith("BUG"))  
    .collect(Collectors.toList)
```

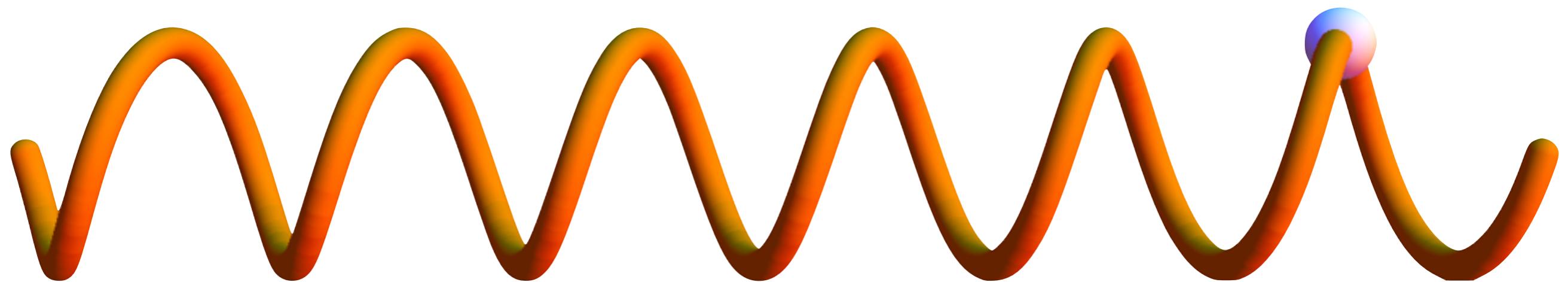
C#

```
lines.Where(s => s.StartsWith("BUG")).ToList
```

Lazy Evaluation



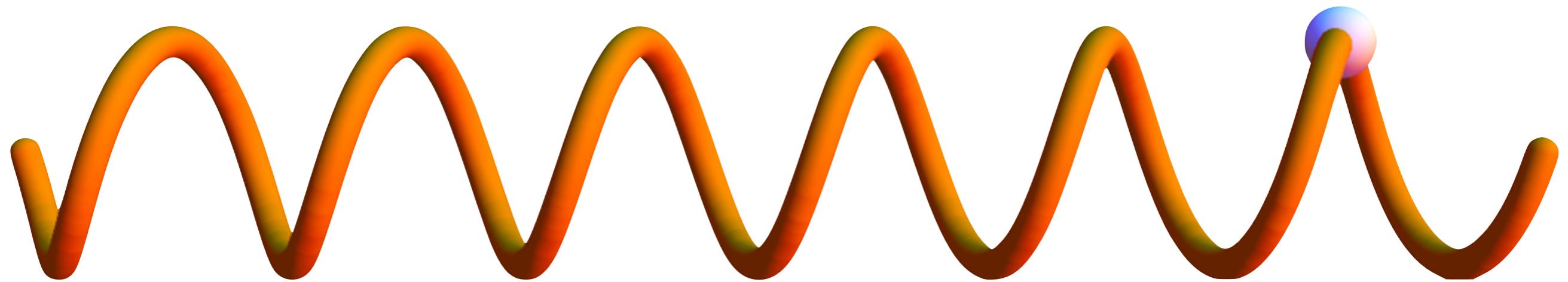
delay evaluation until the last
responsible moment



save work



let go of when



separate **what to do**
from **when to stop**

Java

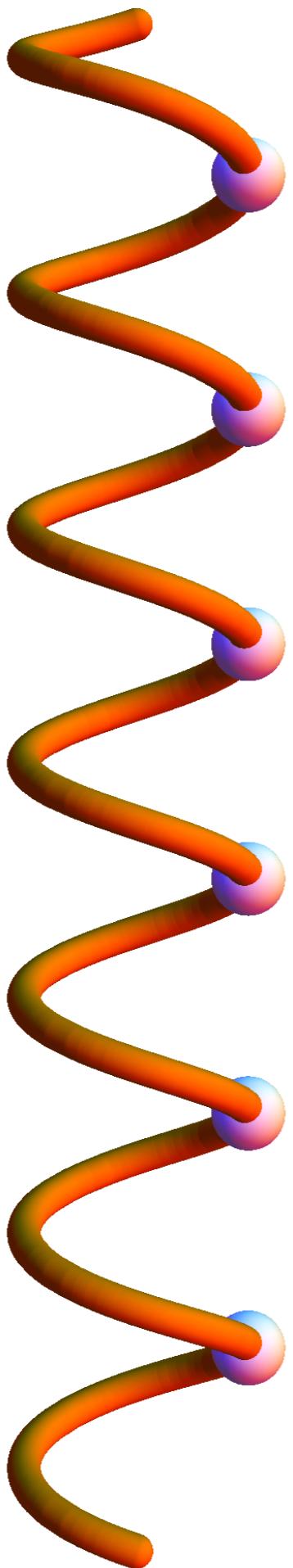
```
int bugCount = 0;
String nextLine = file.readLine();
while (bugCount < 40) {
    if (nextLine.startsWith("BUG")) {
        String[] words = nextLine.split(" ");
        report("Saw "+words[0]+" on "+words[1]);
        bugCount++;
    }
}
waitForFileHasMoreData(file);
nextLine = file.readLine();
}
```

Java 6

```
for (String s :  
    FluentIterable.of(new RandomFileIterable(br))  
    .filter(STARTS_WITH_BUG_PREDICATE)  
    .transform(TRANSFORM_BUG_FUNCTION)  
    .limit(40)  
    .asImmutableList()) {  
    report(s);  
}
```







Data In, Data Out

Specific Typing

Verbs Are People Too

Immutability

Declarative Style

Lazy Evaluation

Alistair Cockburn's Oath of Non-Allegiance

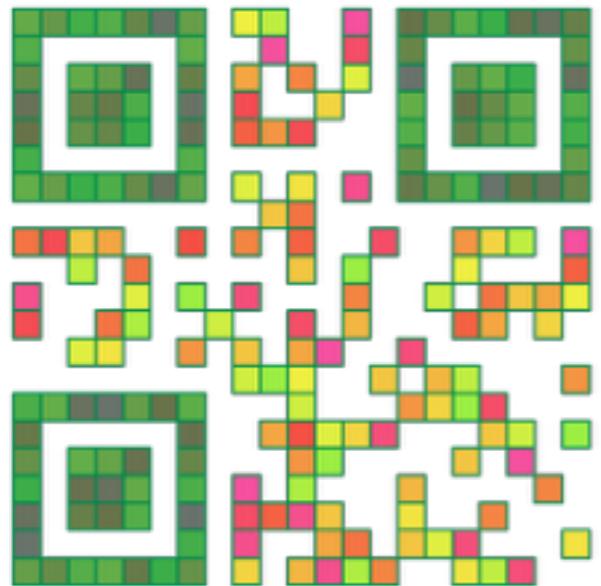
I promise not to exclude
from consideration any idea
based on its source, but to consider ideas
across schools and heritages
in order to find the ones that best suit the
current situation.

optional<T>
Tuple<T>



by Faqqotic

<http://www.sketchport.com/drawing/6606411756732416/aeiou>



Jessica Kerr
blog.jessitron.com

@jessitron

github.com/jessitron/fp4ood

