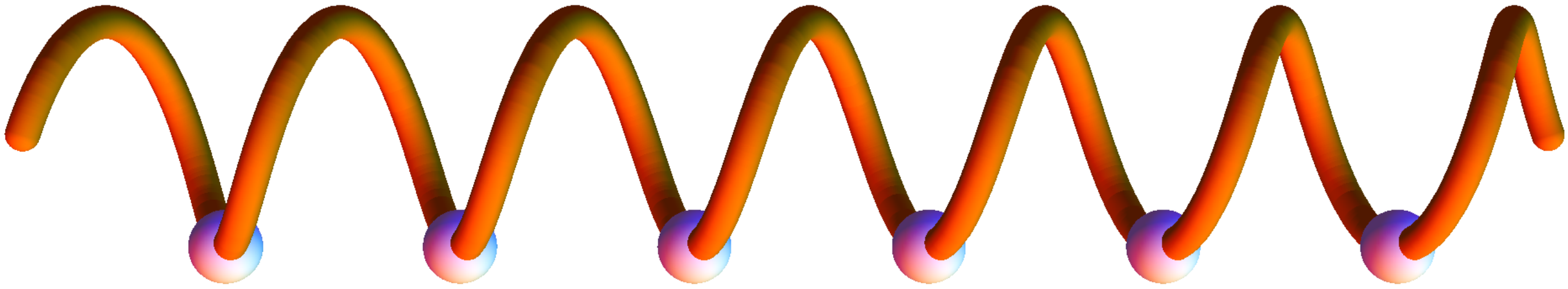
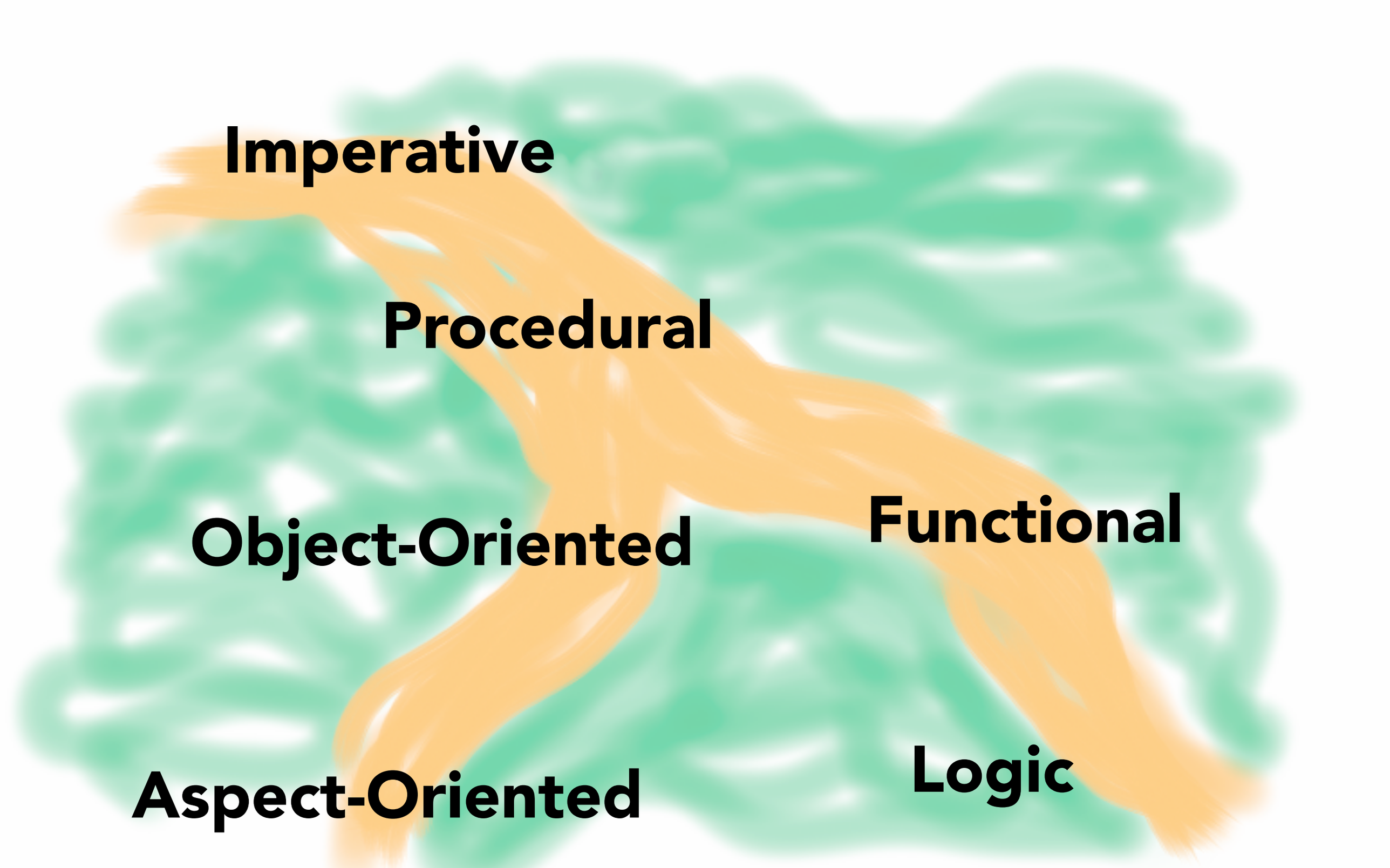
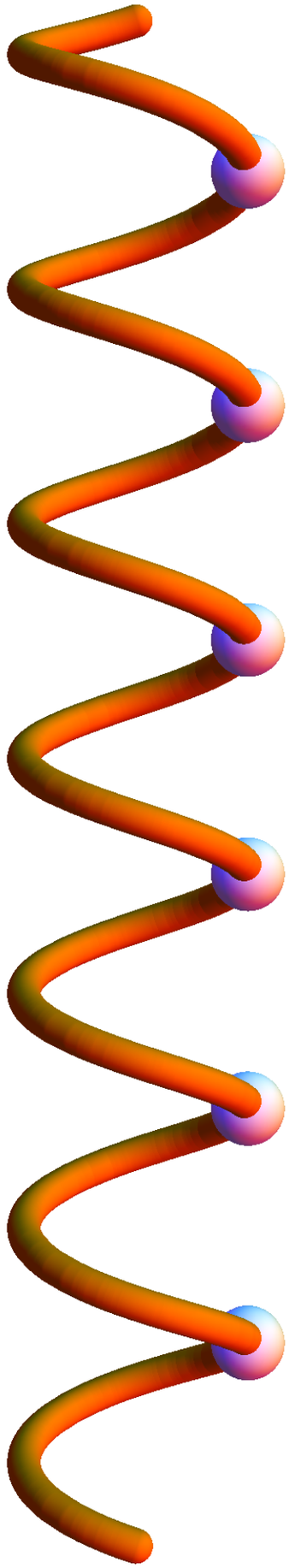


Functional principles for object-oriented development

@jessitron


Imperative
Procedural
Object-Oriented
Functional
Aspect-Oriented
Logic



Data In, Data Out

Specific Typing

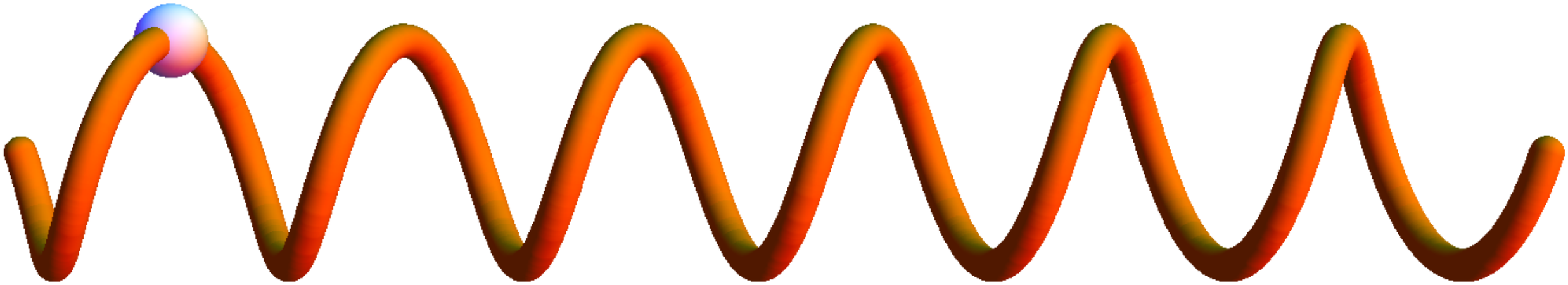
Verbs Are People Too

Immutability

Declarative Style

Lazy Evaluation

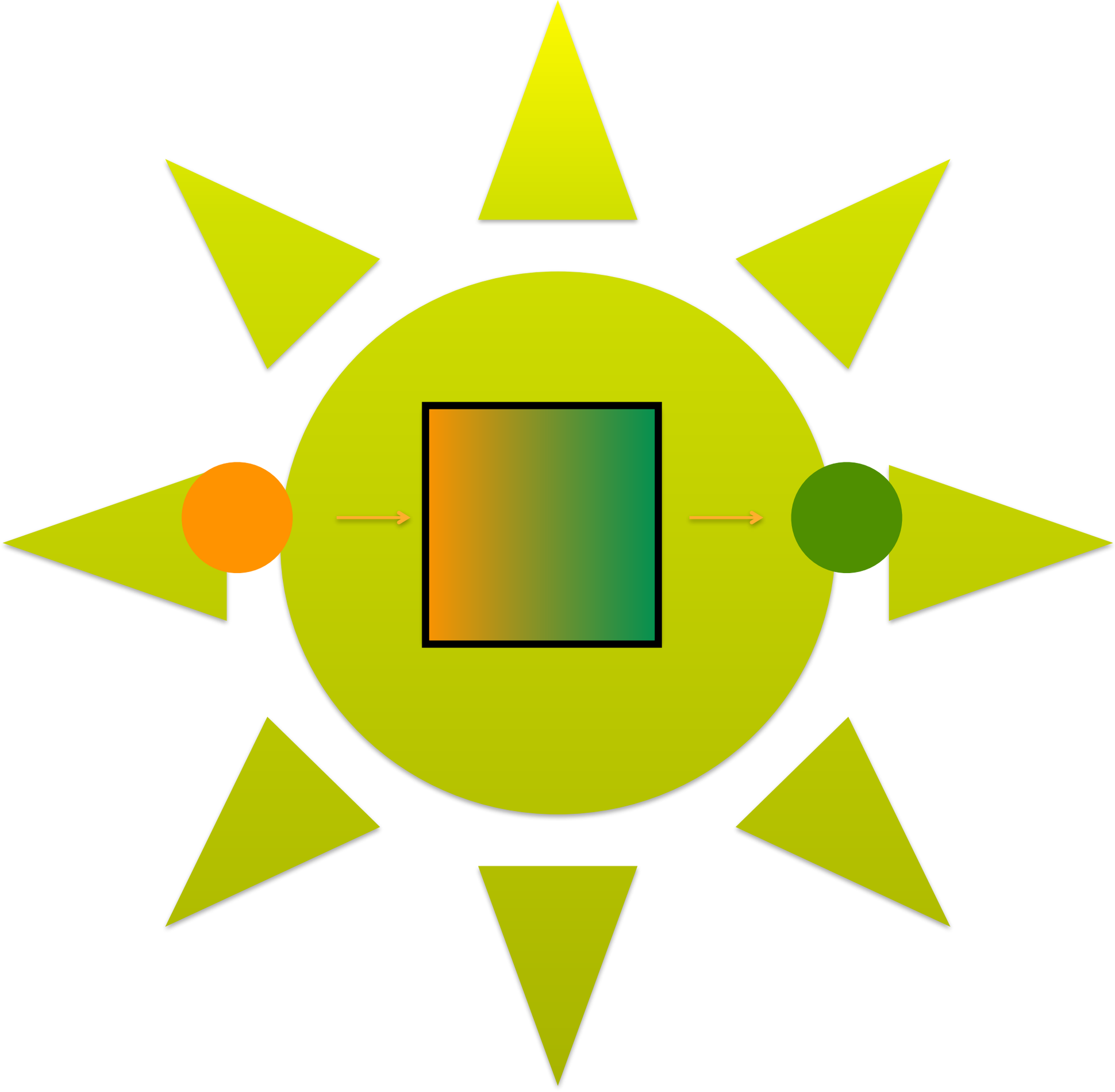
Data In, Data Out



access global state

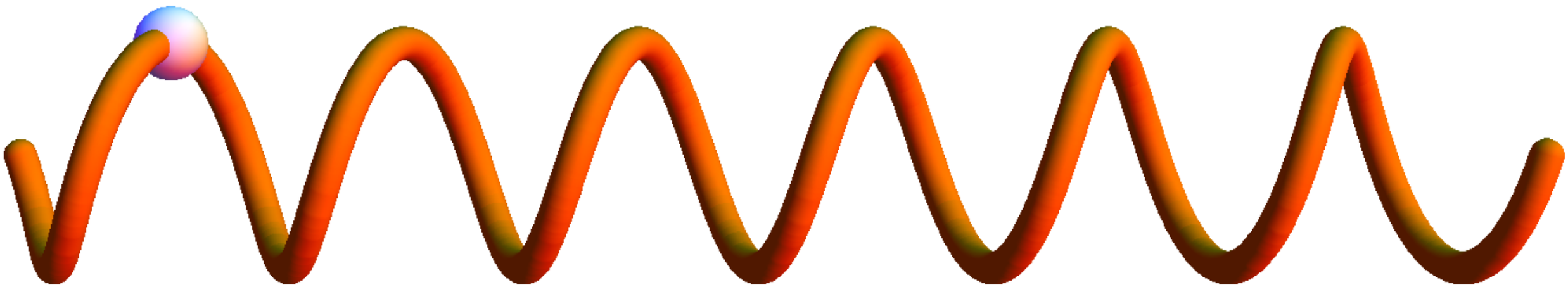
modify input

change the world



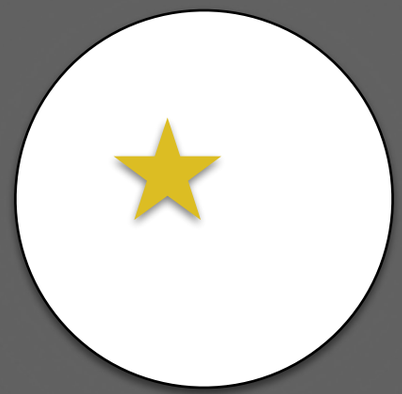
Testable

Easier to understand



Encapsulation

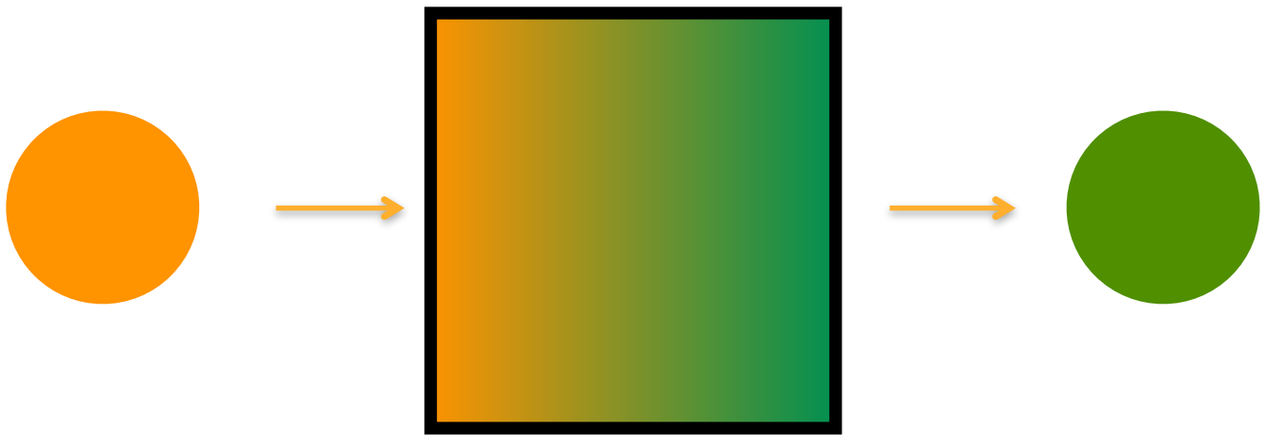
Isolation



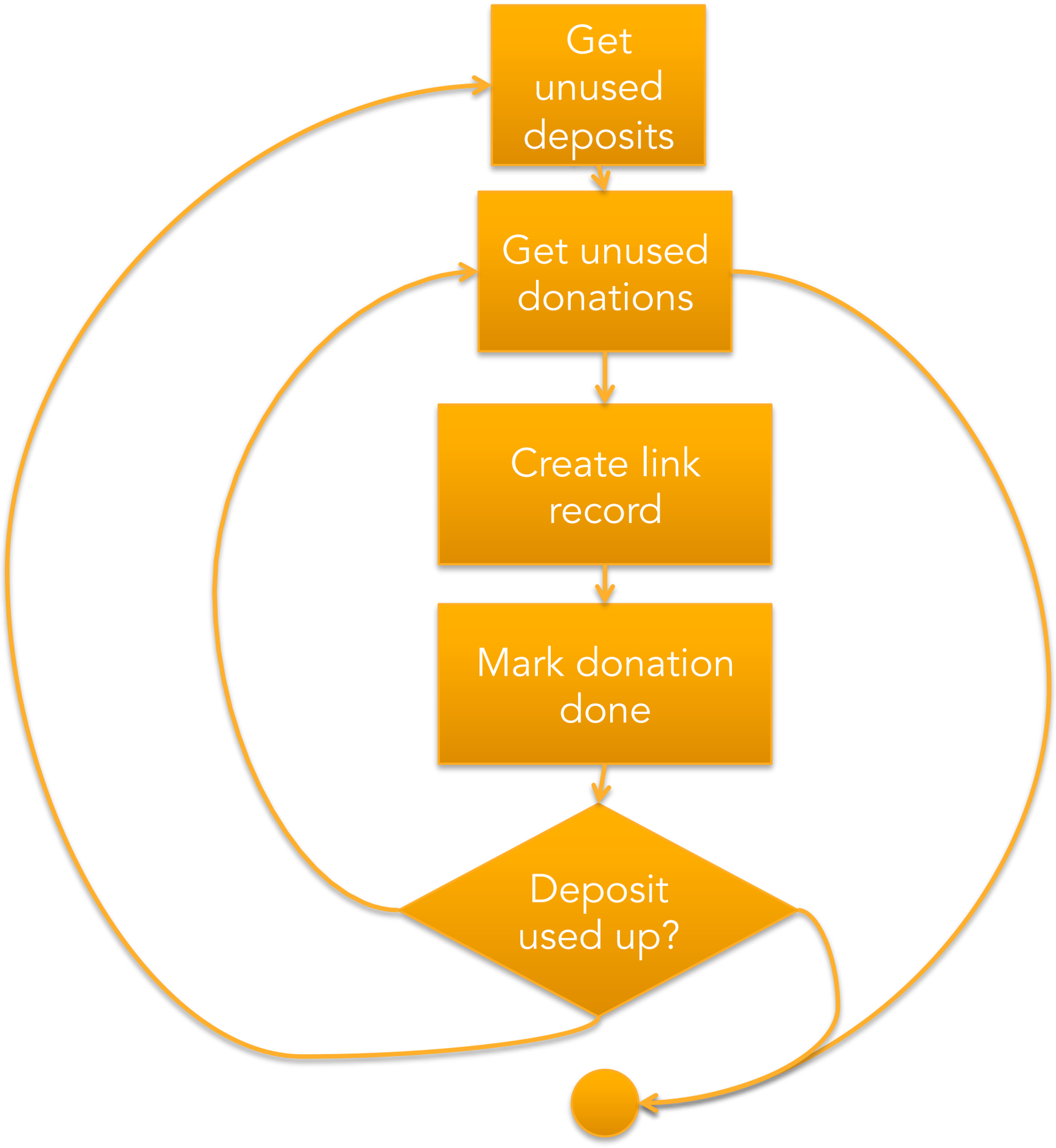
```
Password password;
Email    email;

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

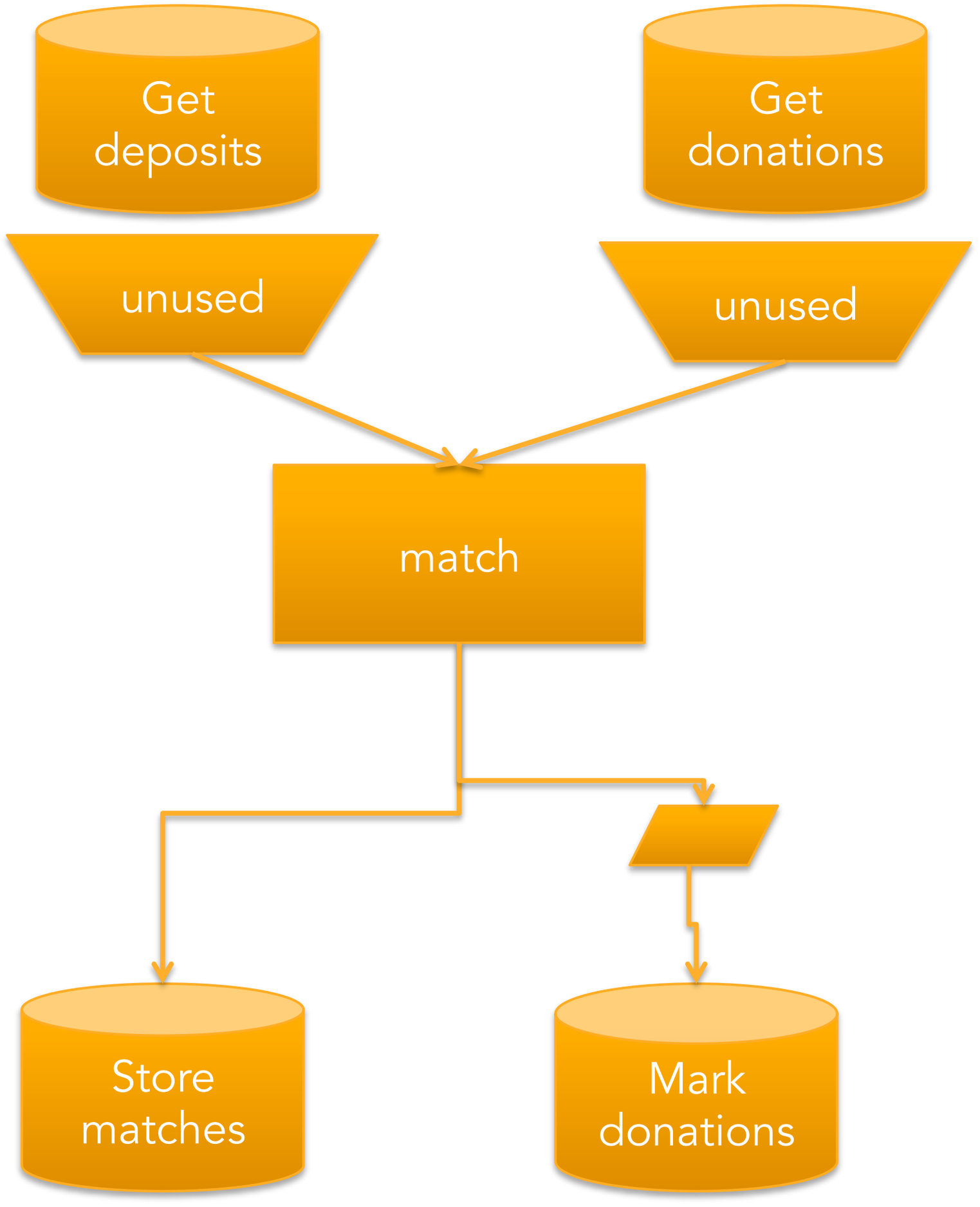
```
private boolean invalidPassword(
                                Password password,
                                Email email) {
    return !password.contains(email);
}
```

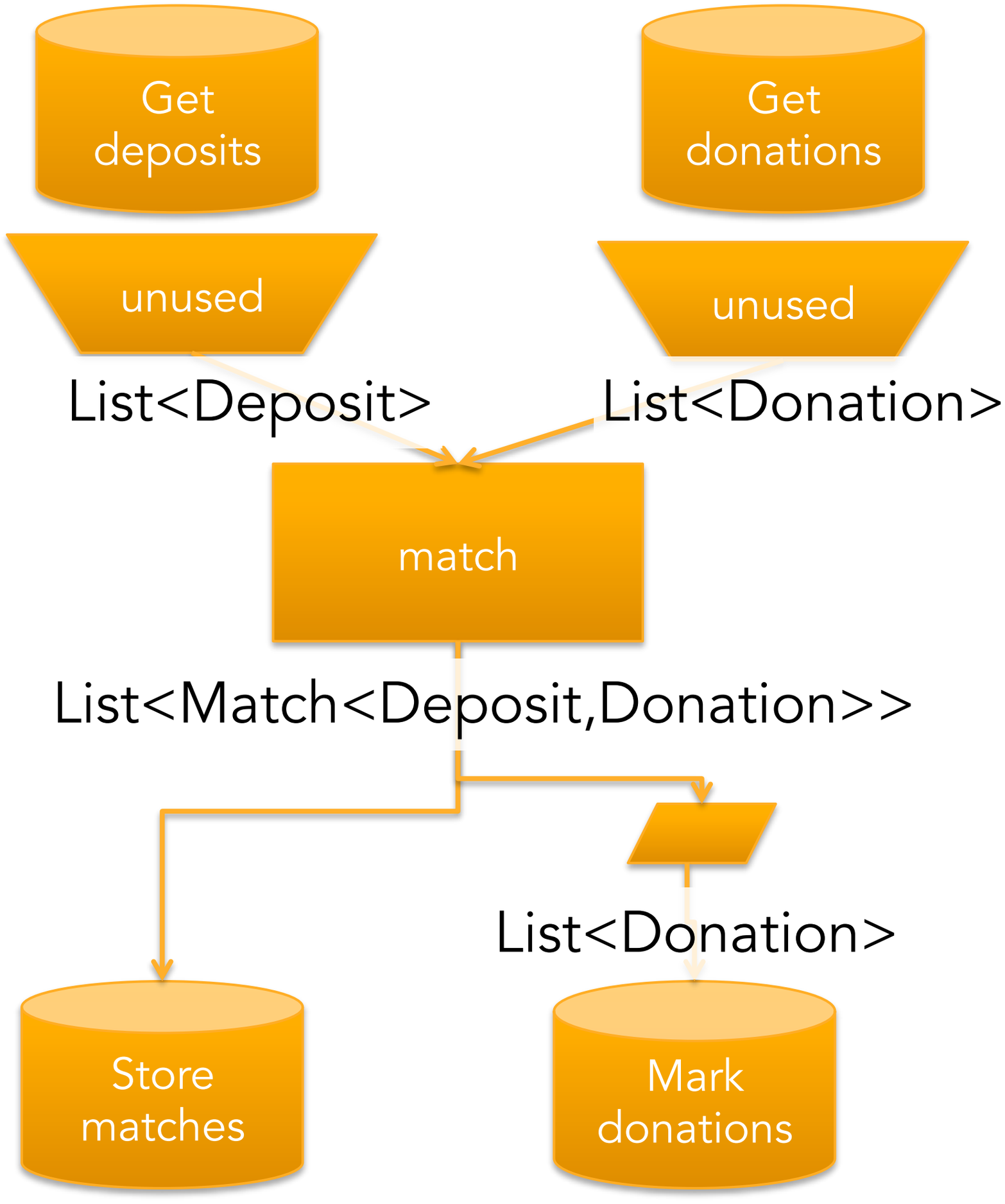
(Email, Password) => boolean

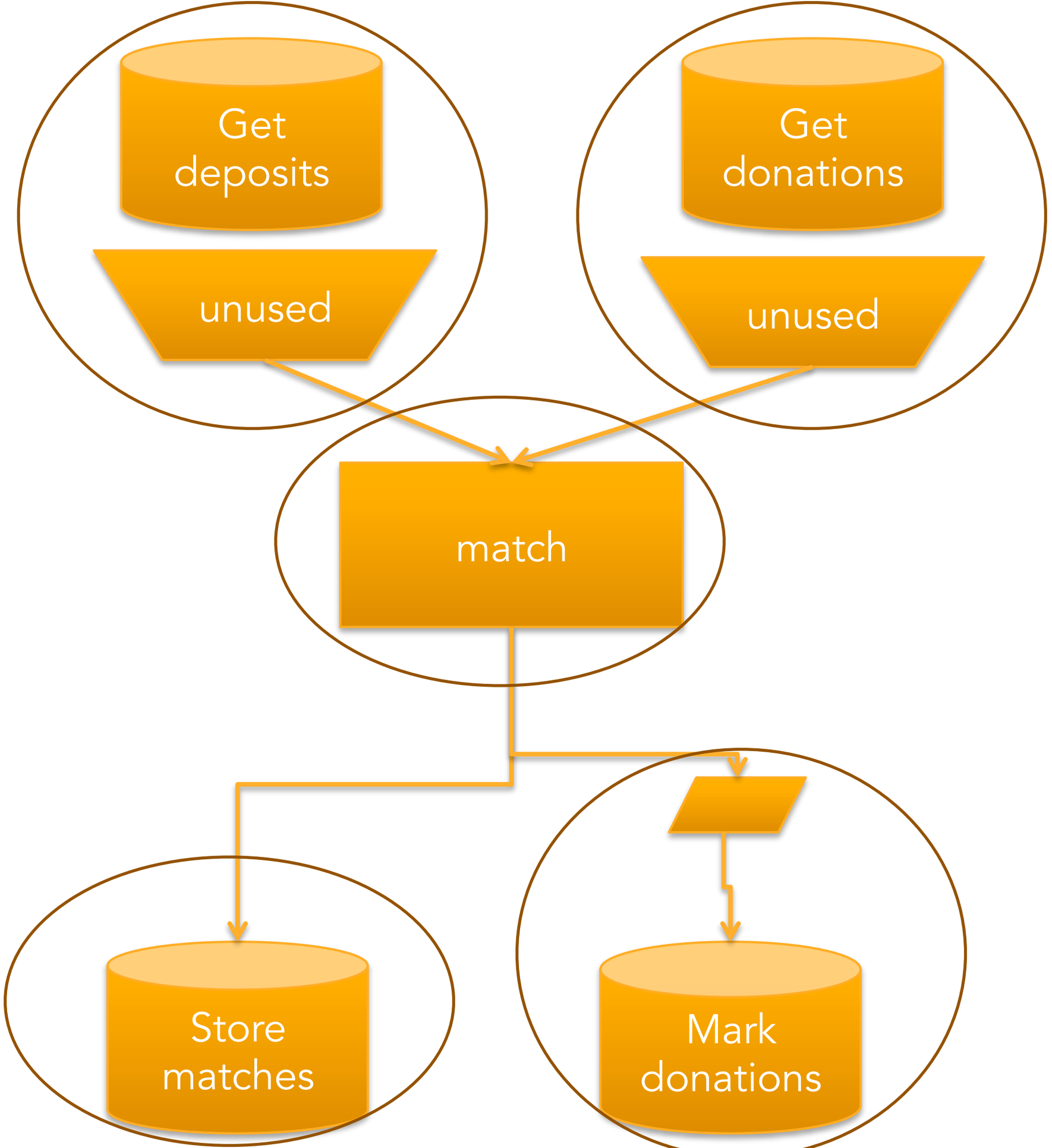


the flow of data

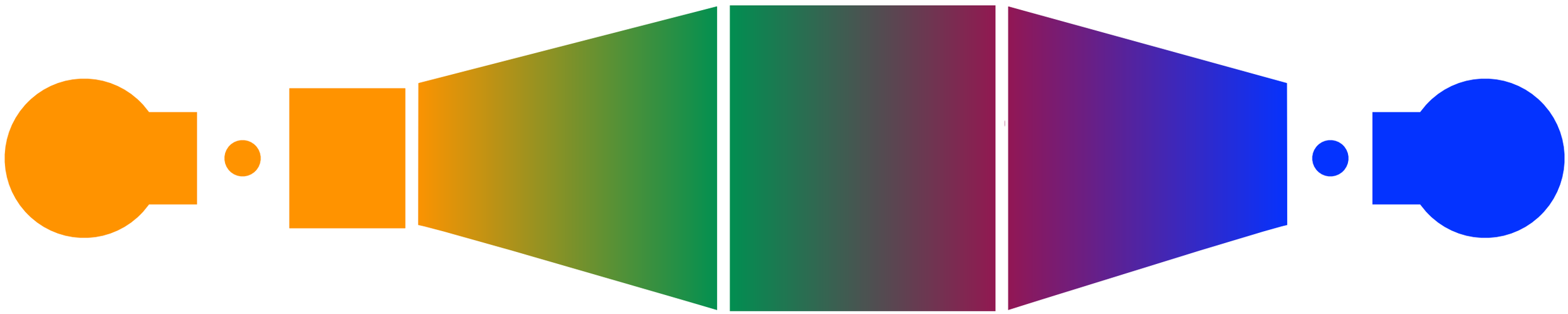

Get unused deposits
Get unused donations
Create link record
Mark donation done
Deposit used up?

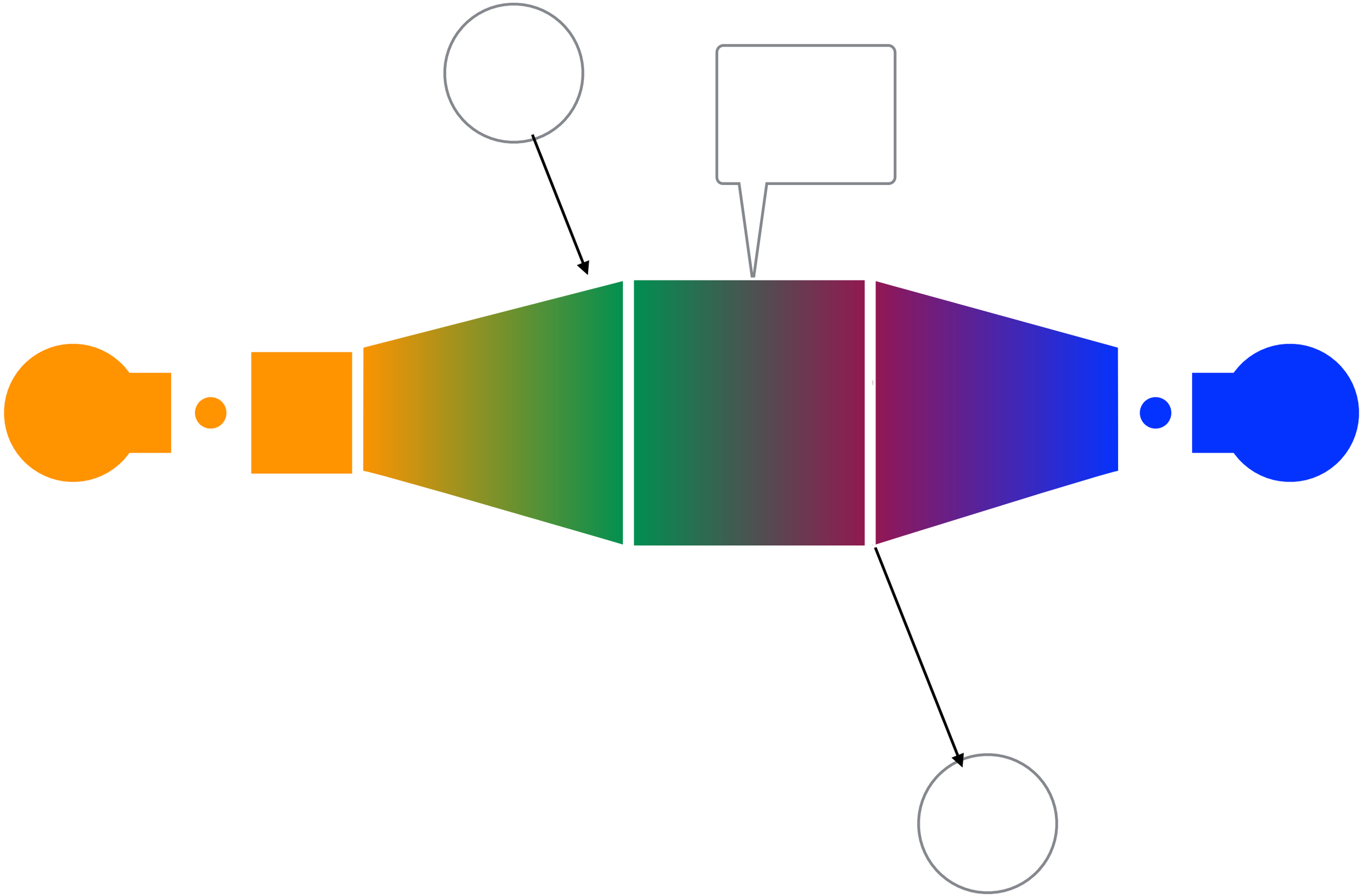

Get deposits
Get donations
unused
unused
match
Store matches
Mark donations

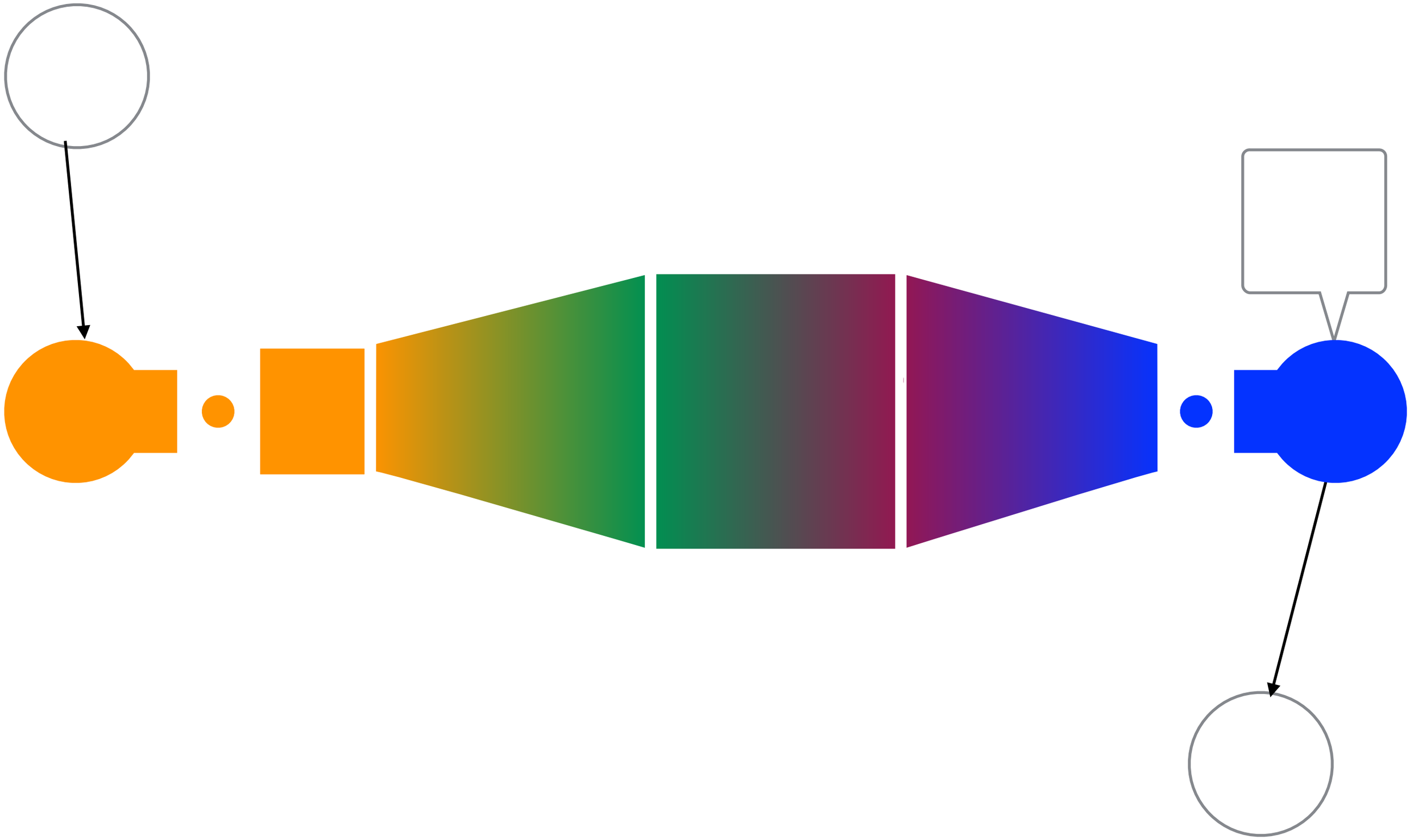

Get deposits
Get donations
unused
unused
List<Deposit>
List<Donation>
match
List<Match<Deposit,Donation>>
List<Donation>
Store matches
Mark donations


Get deposits
unused
Get donations
unused
match
Store matches
Mark donations

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



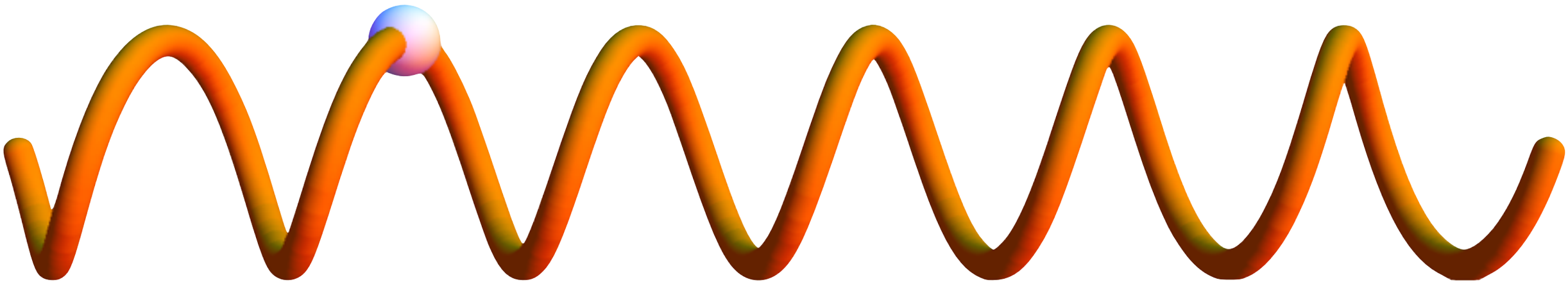




```
Response createAccount(UserDbService svc,
                       Request req,
                       Config  cfg) {
  …
  Account acc = constructAccount(…)
  AccountInsertResult r = svc.insert(acc)
  return respond(r);
}
```

```
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.

眉間一字寬，萬里眼中明

Scala

```
case class FirstName(value : String)

  val friend = FirstName(“Simon”)
```

Scala

```
type FirstName = String

val friend: FirstName = “Ivo”
```

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() {...}
}
```

layers of thinking

expressiveness

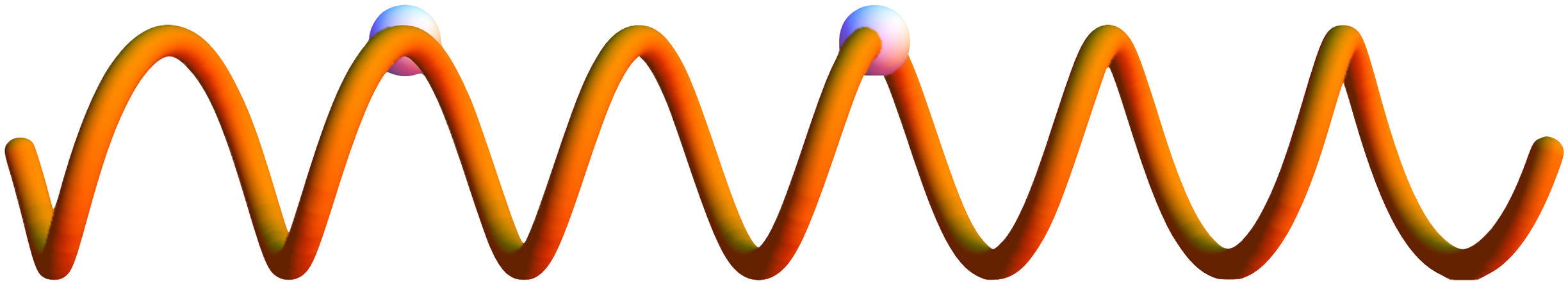
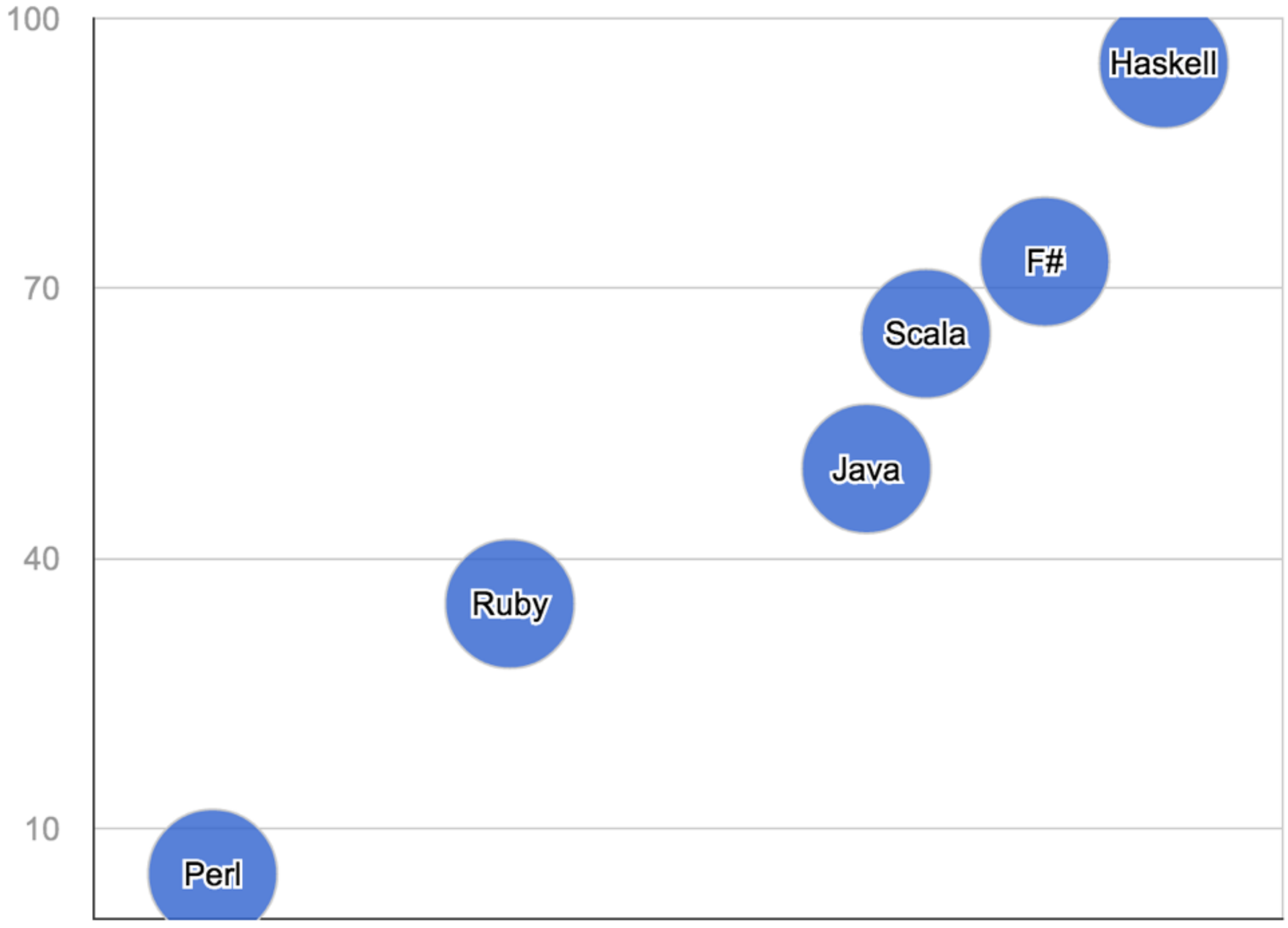
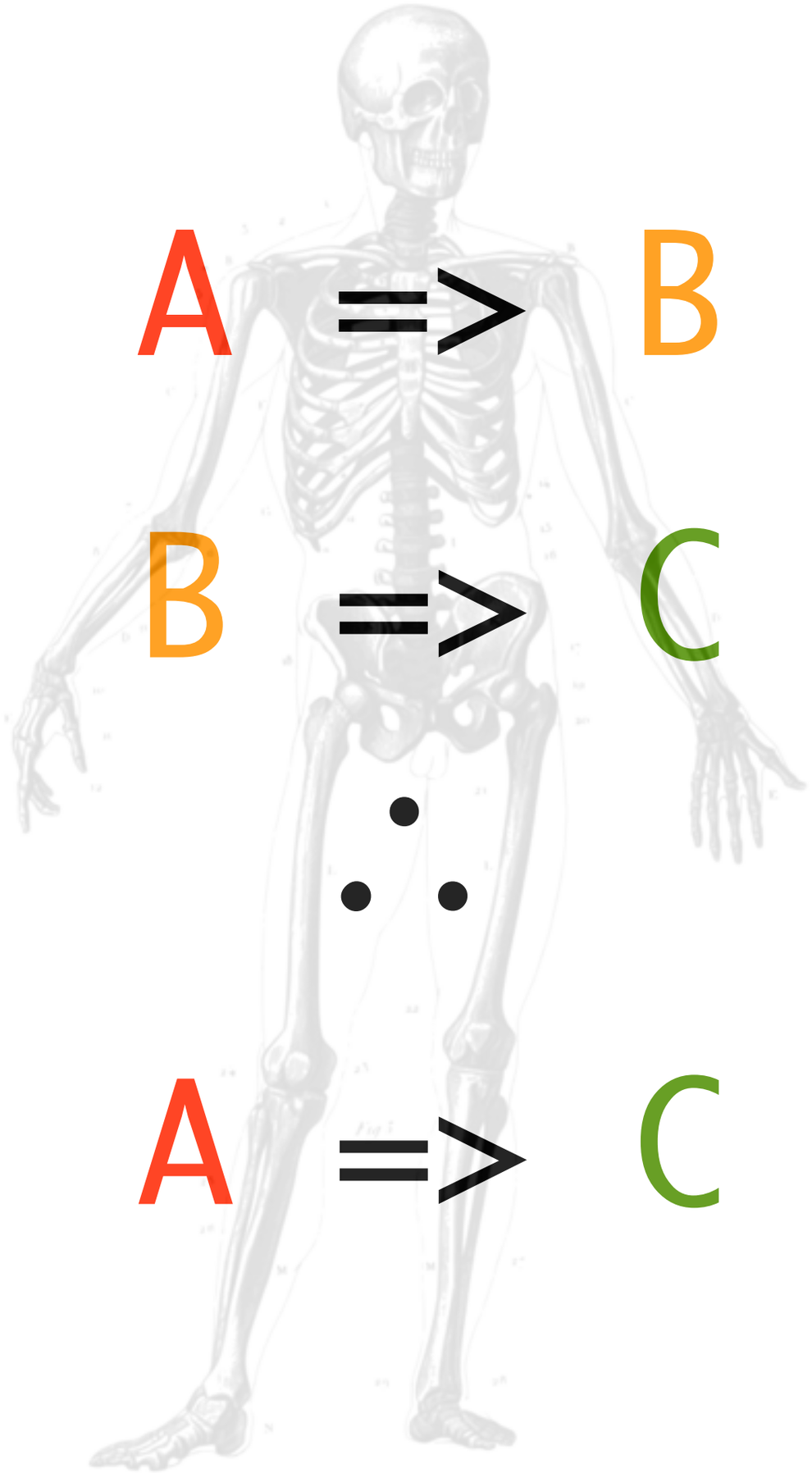


Fig 3

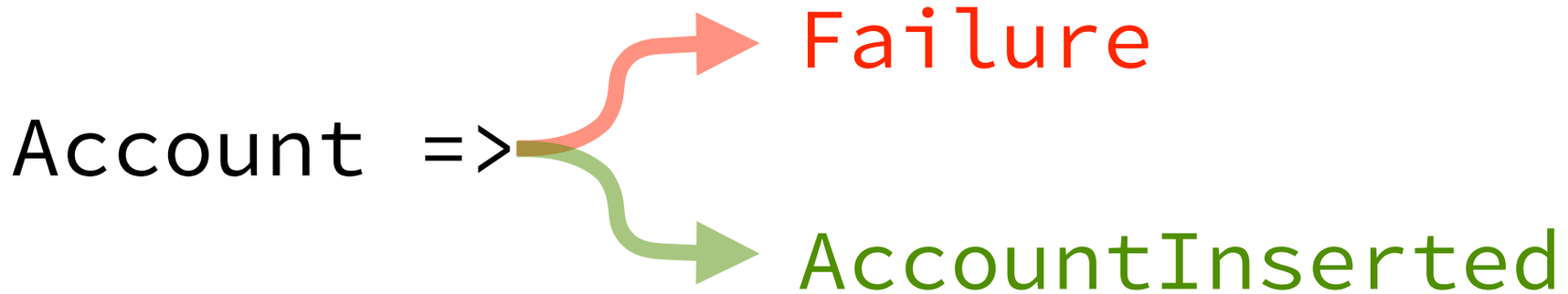

100
70
40
10
Haskell
F#
Scala
Java
Ruby
Perl

```
time => Cost
time => Volume
(Volume, Cost) => Expenditure
  .
 . .
time => Expenditure
```

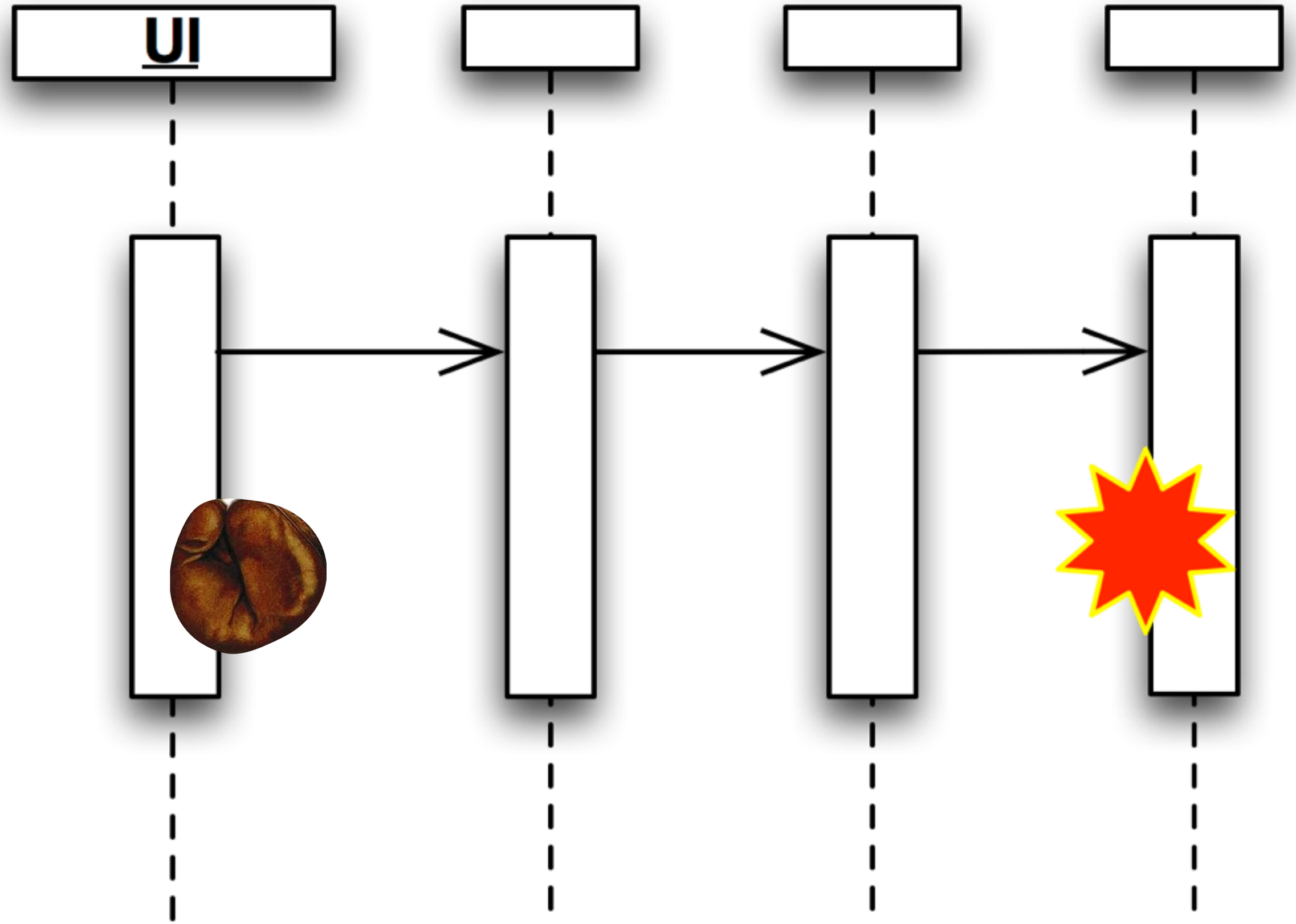

A => B
B => C
A => C

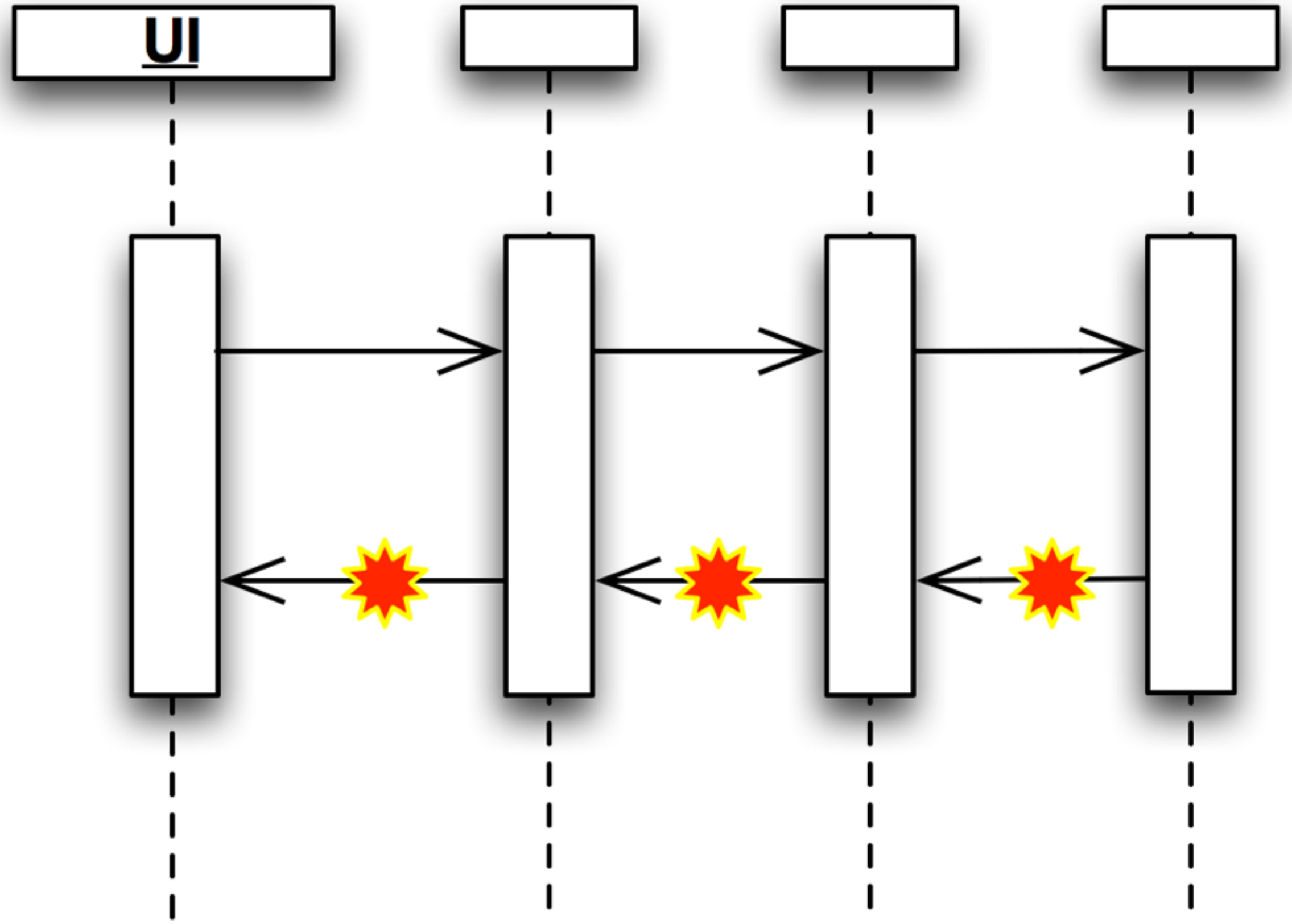
Account => AccountInsertResult

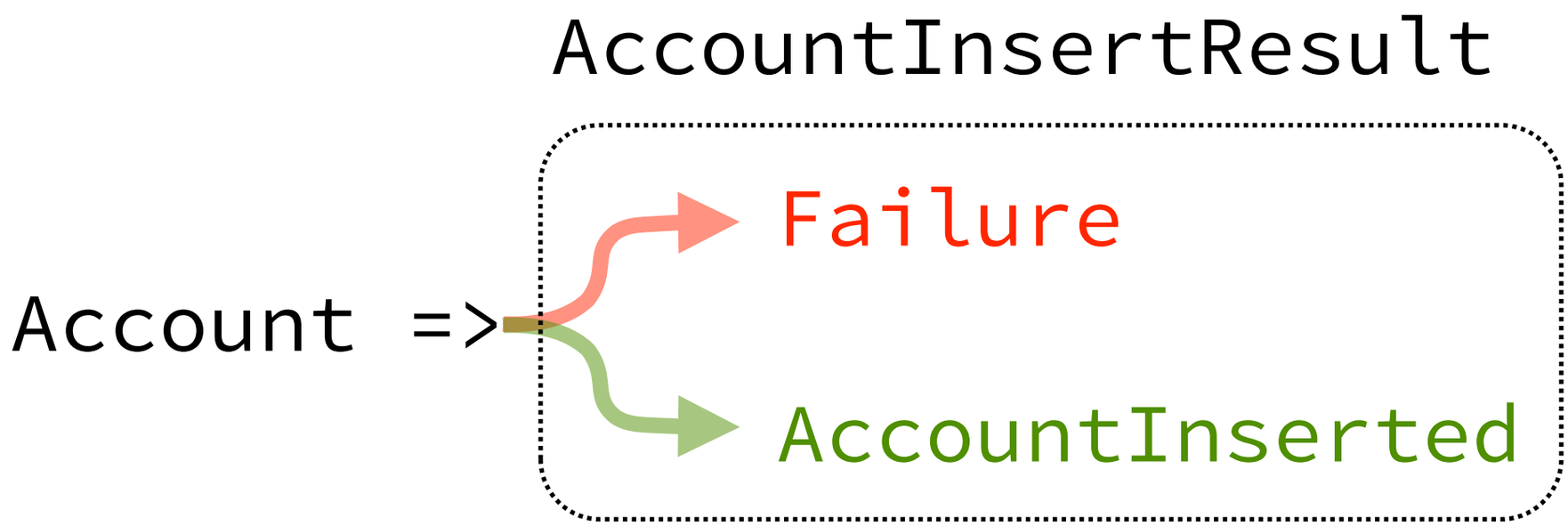
```
Account =>    AccountInserted
```


Account =>
Failure
AccountInserted

Errors are data too

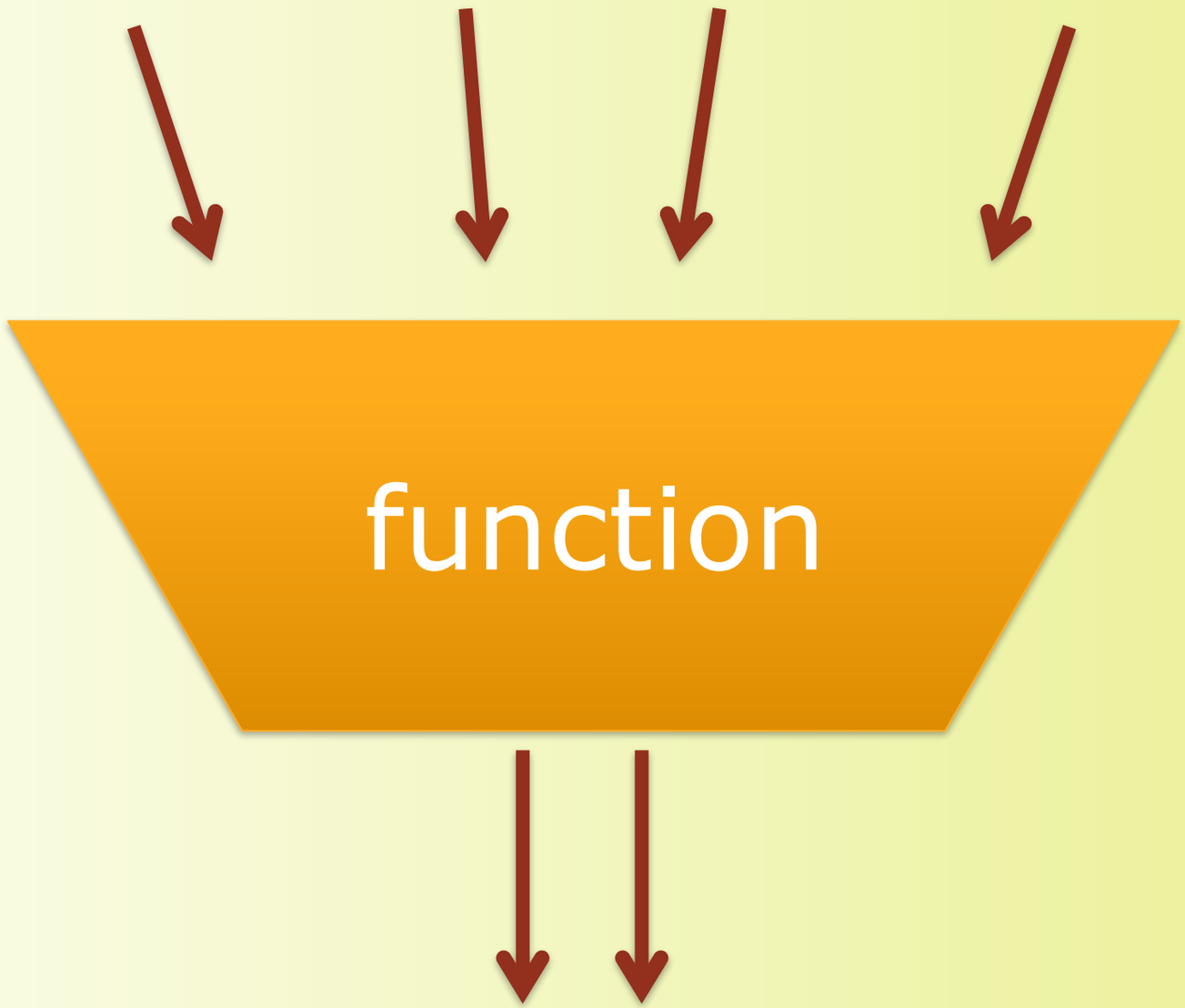

UI


UI


AccountInsertResult
Account =>
Failure
AccountInserted

```
Account => Either[Failure,
                  AccountInserted]
```

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

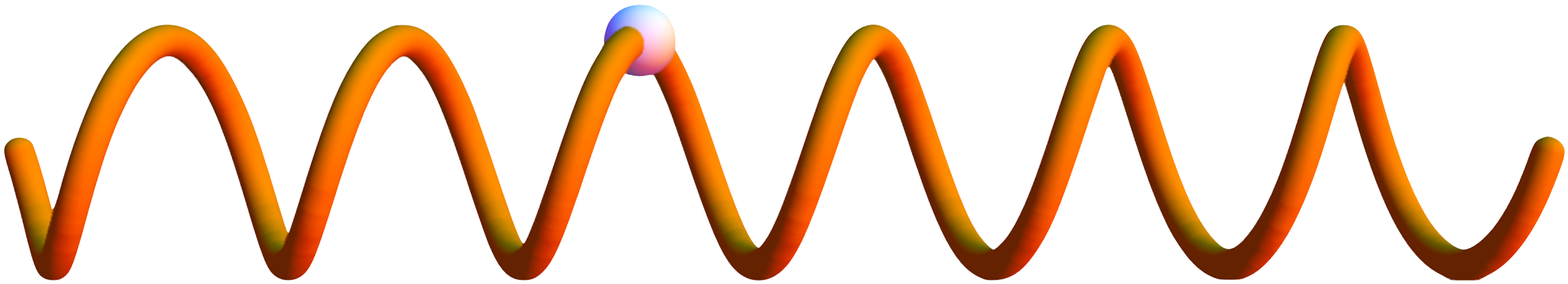


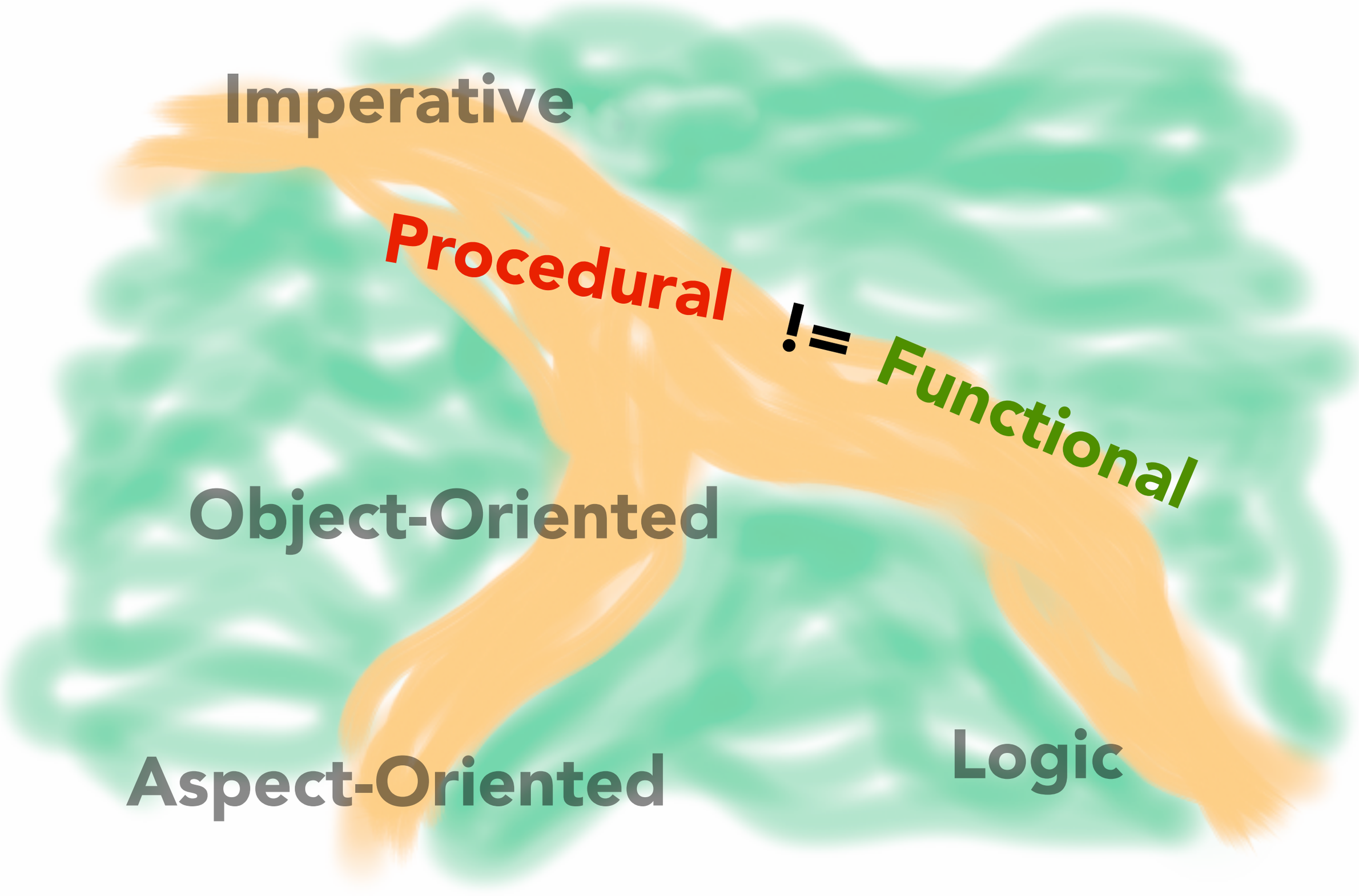

```
new Tuple<string,int>(“You win!”, 1000000)
```

Tuple<T1,T2,...>

When one return value is not enough!

Verbs Are People Too




Imperative
Procedural
!=
Functional
Object-Oriented
Aspect-Oriented
Logic

Java

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

Java 8

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

Java 8

```
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);
       }
    });
```

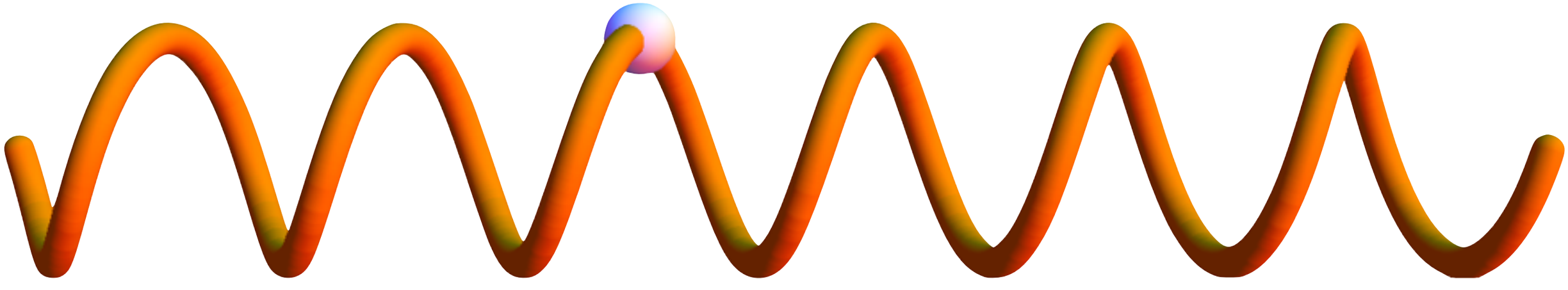
Strategy

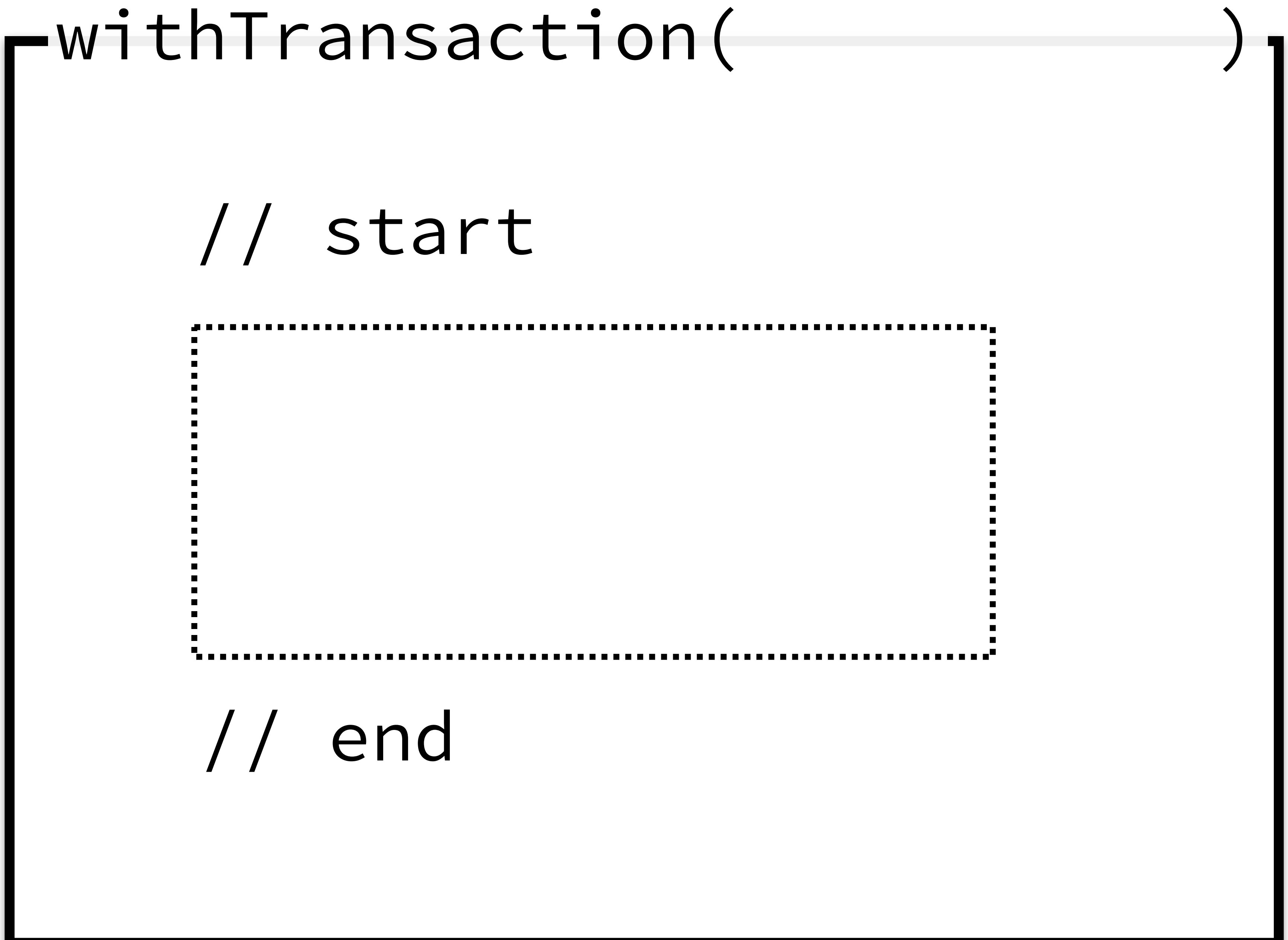
Command

context

around code

- convenience




withTransaction(   )
// start
// end

```
Response r = createAccount(
      (acc) -> userDB.insertAccount(acc),
      request,
      config);
```

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

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

retrying( )

Code

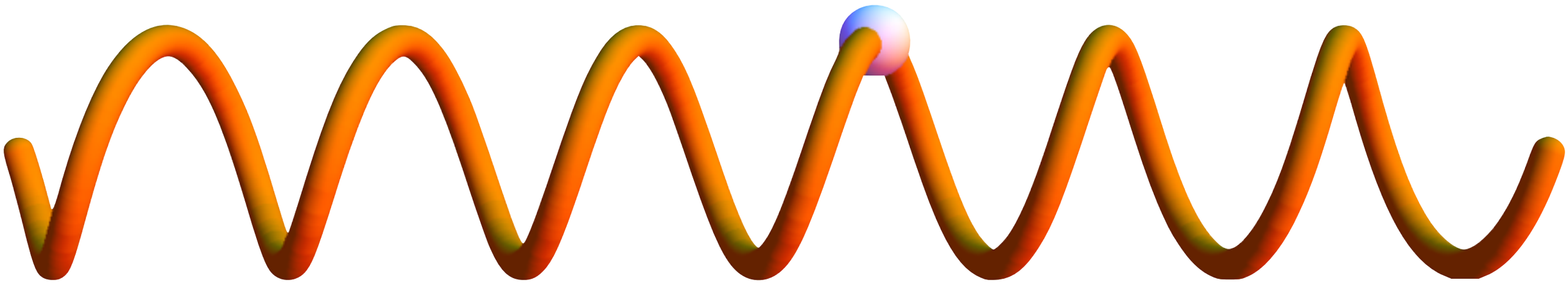
Code

Idempotence

OnClick()

release ( )

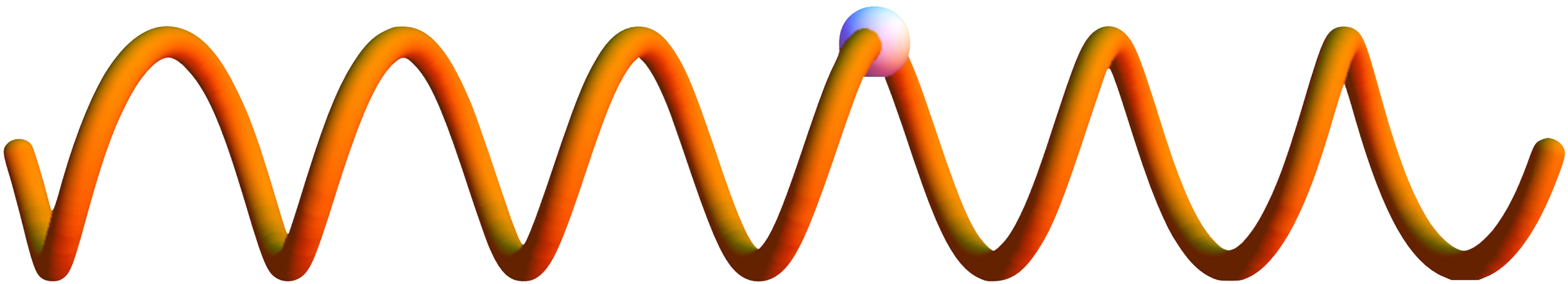
Immutability



String

Effective Java

Effective C#



Scala

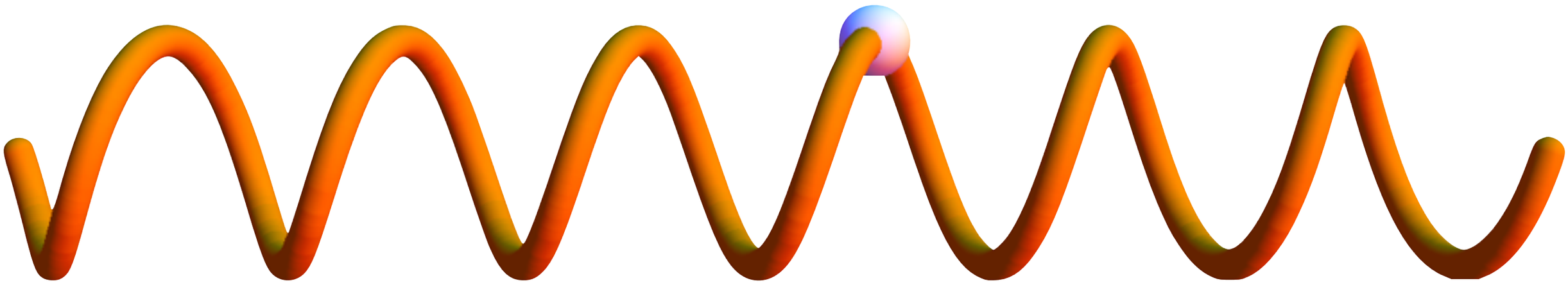
```
val qty = 4   // immutable

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);
}
```

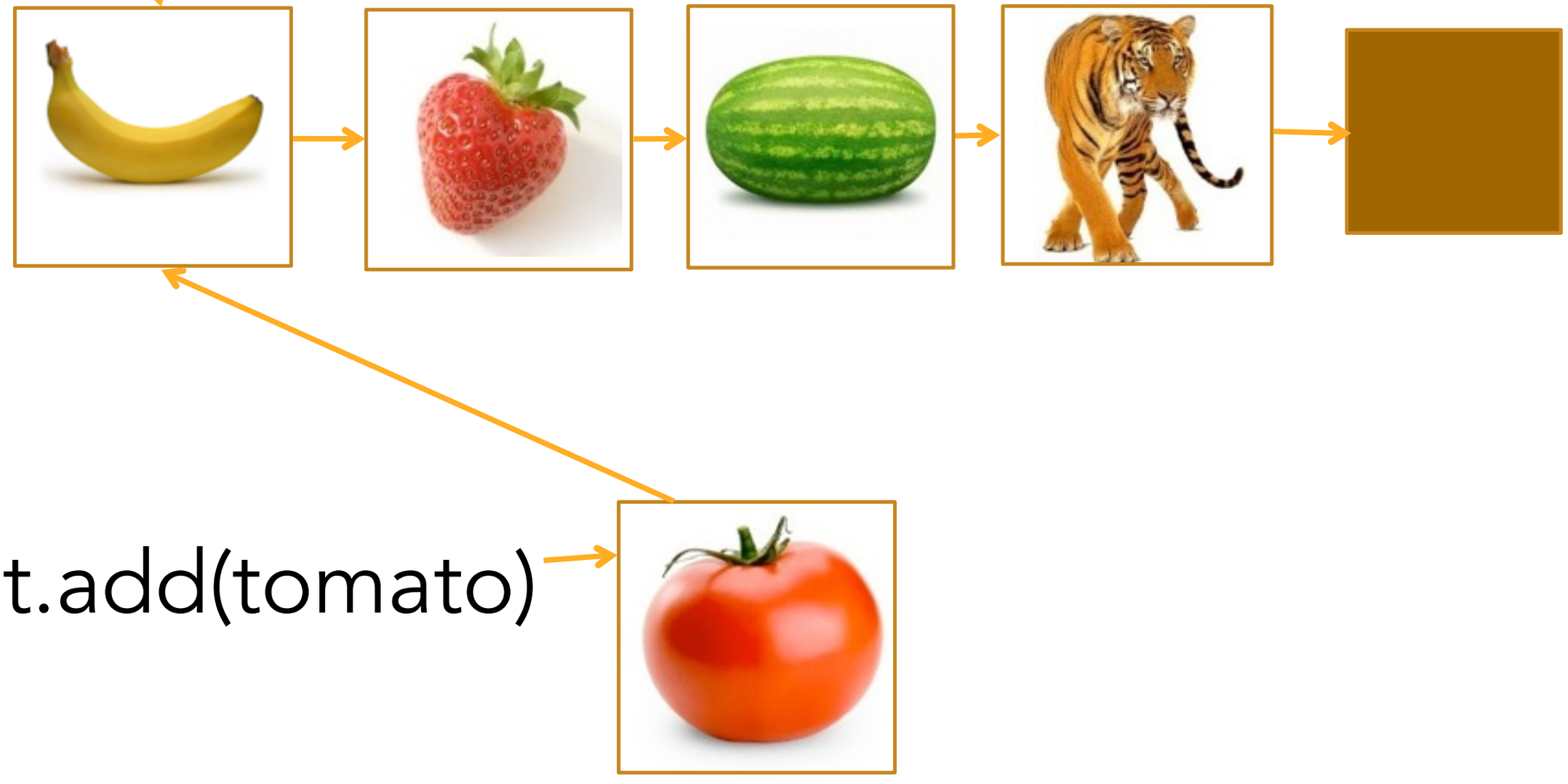
Java: copy on mod

```
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)

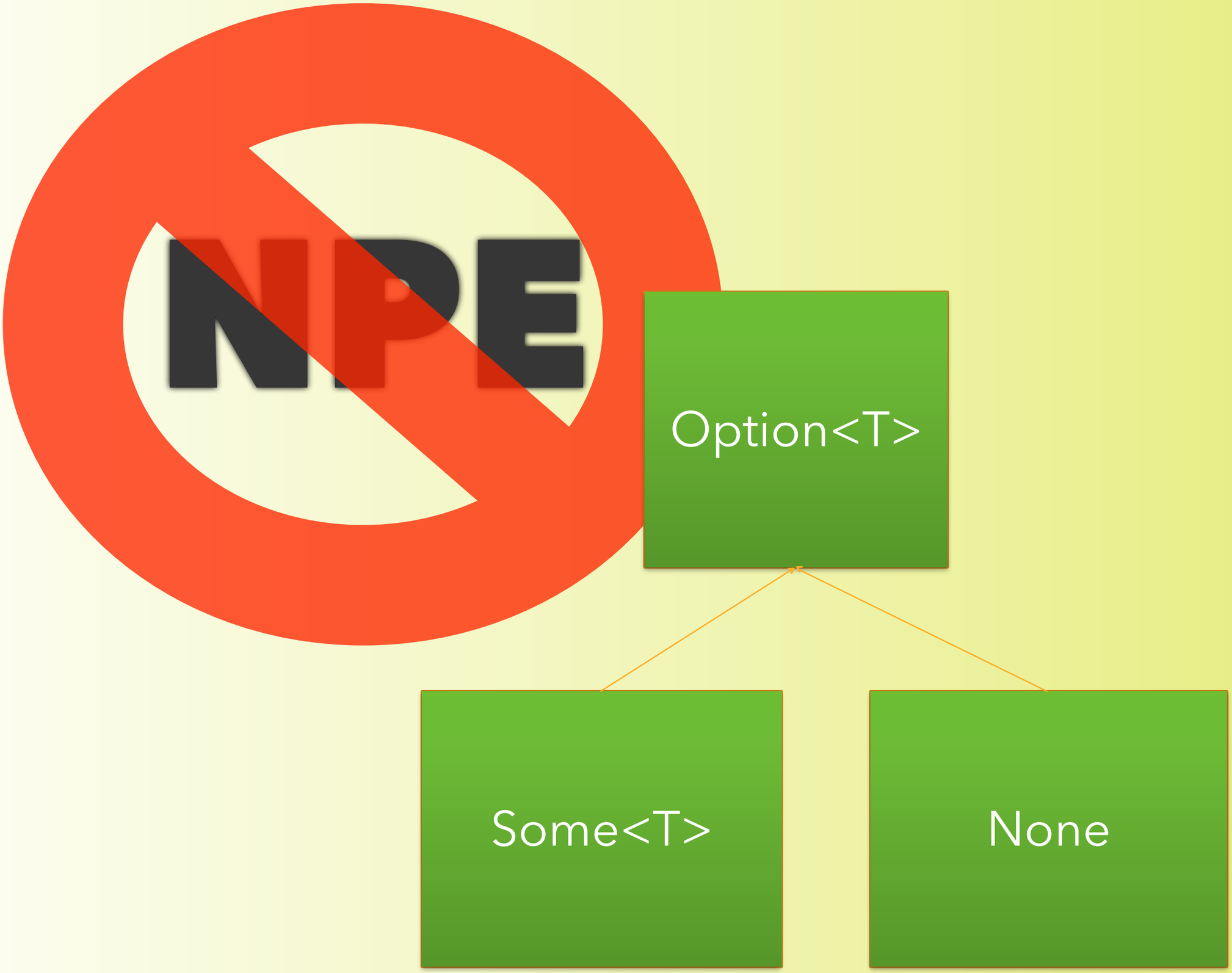
persistent data structure

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 Option type!

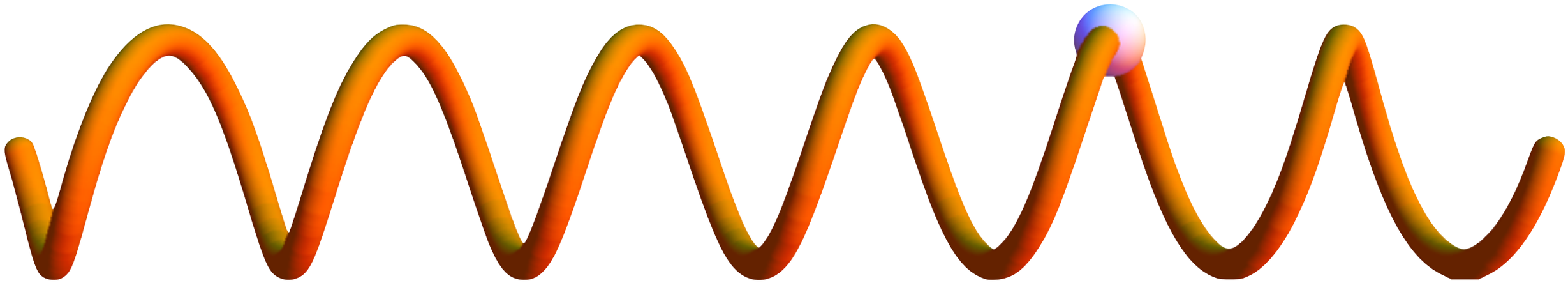


FSharpOption<T>

Optional<T>

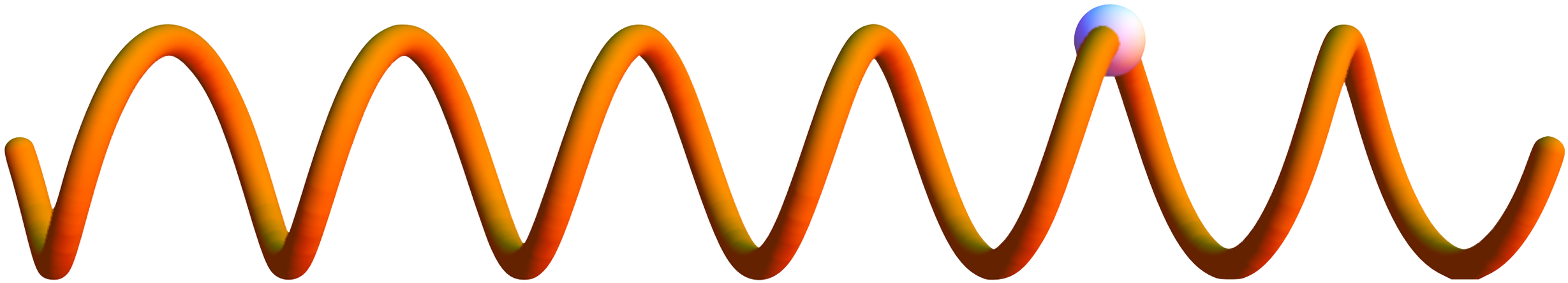
… because null is not a valid object reference.

Declarative Style



say what you're doing,

not how you're doing it



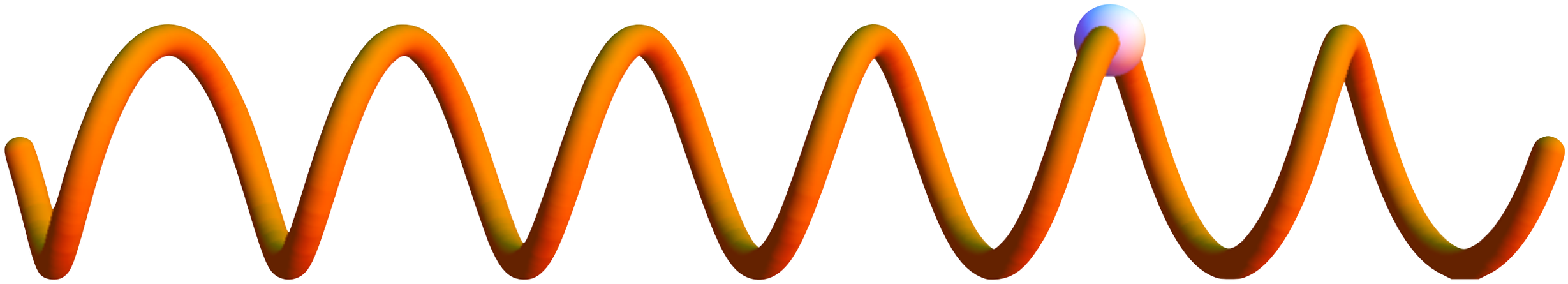
Never tell people how to do things. Tell them what to do and they will surprise you with their ingenuity.

眉間一字寬，萬里眼中明

```
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 6

```
filter(list, startsWithBug);
```

```
final Predicate<String> startsWithBug =
    new Predicate<String>() {
      public boolean apply(String s) {
        return s.startsWith("BUG");
      }
    };
```

C#

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

Java

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

Java

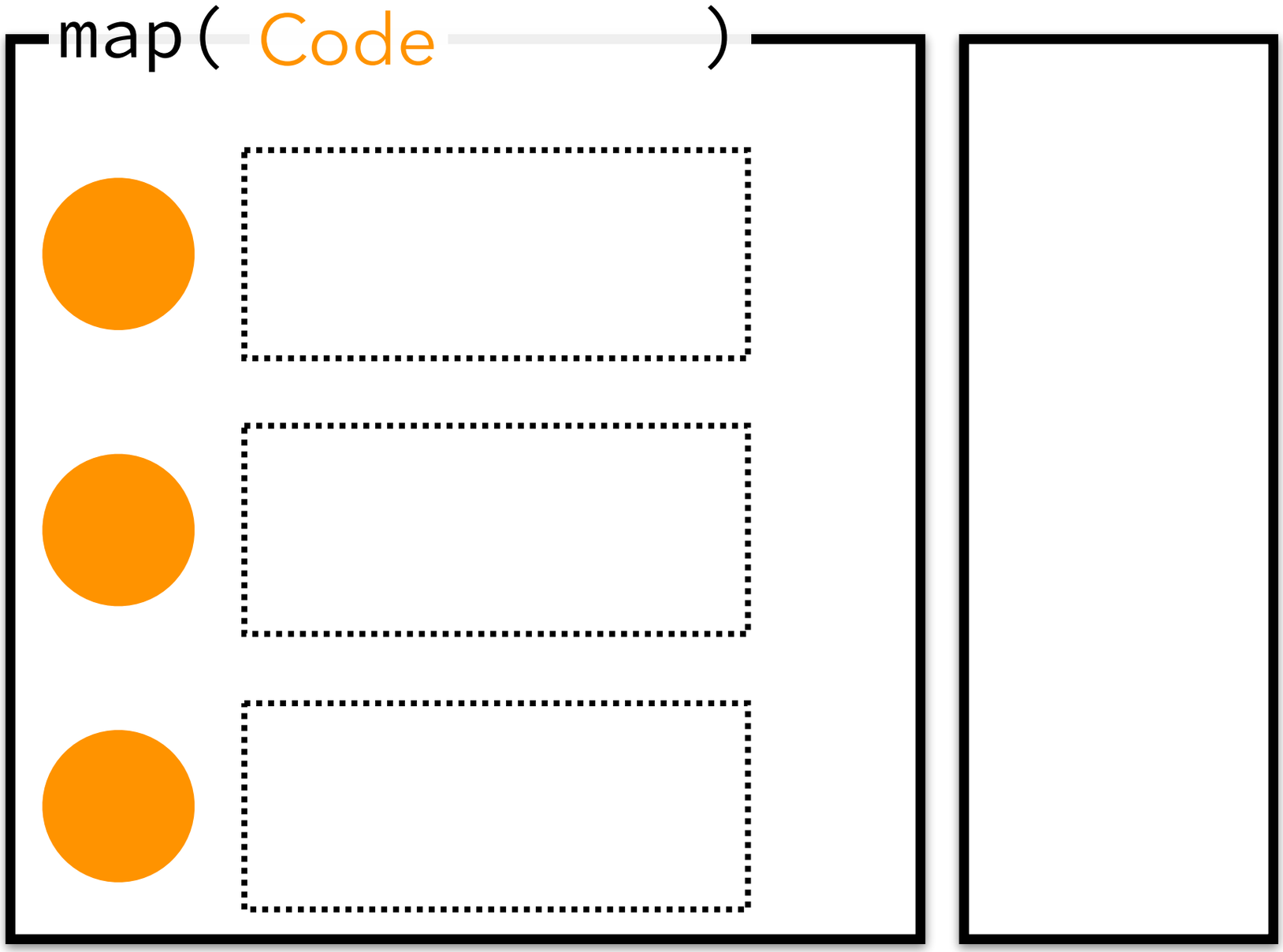
```
lines.stream().parallel()
    .filter(s -> s.startsWith(“BUG”))
    .collect(Collectors.toList)
```

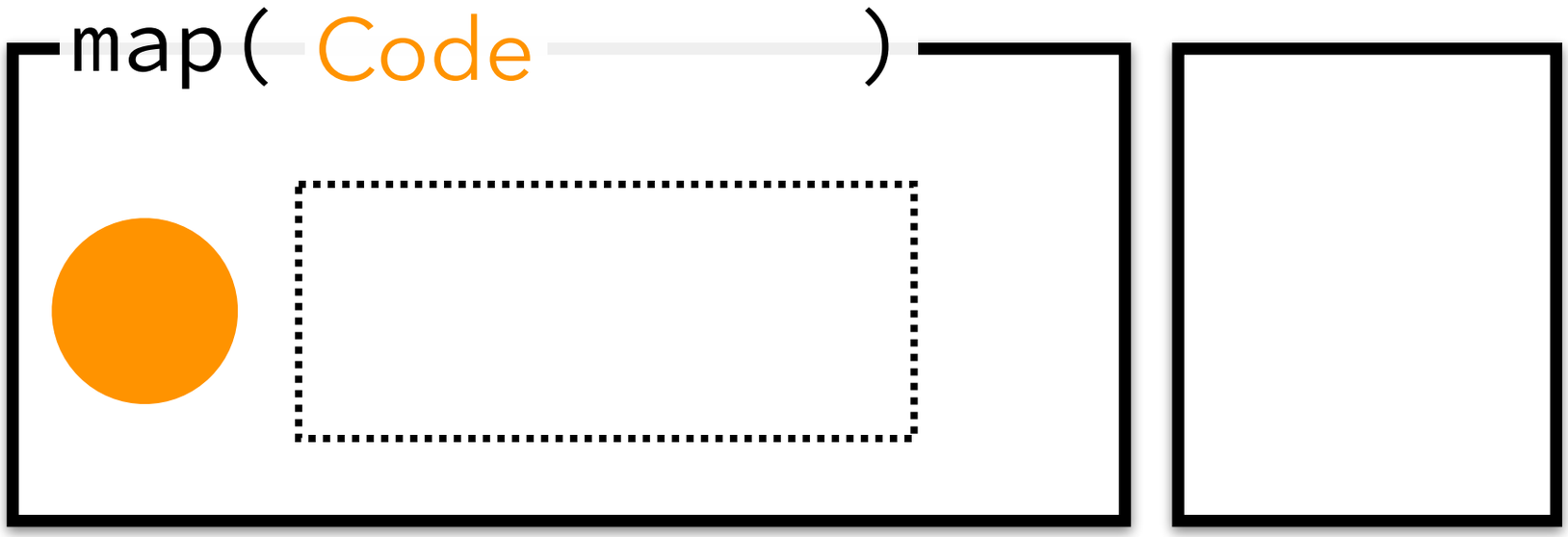
Java

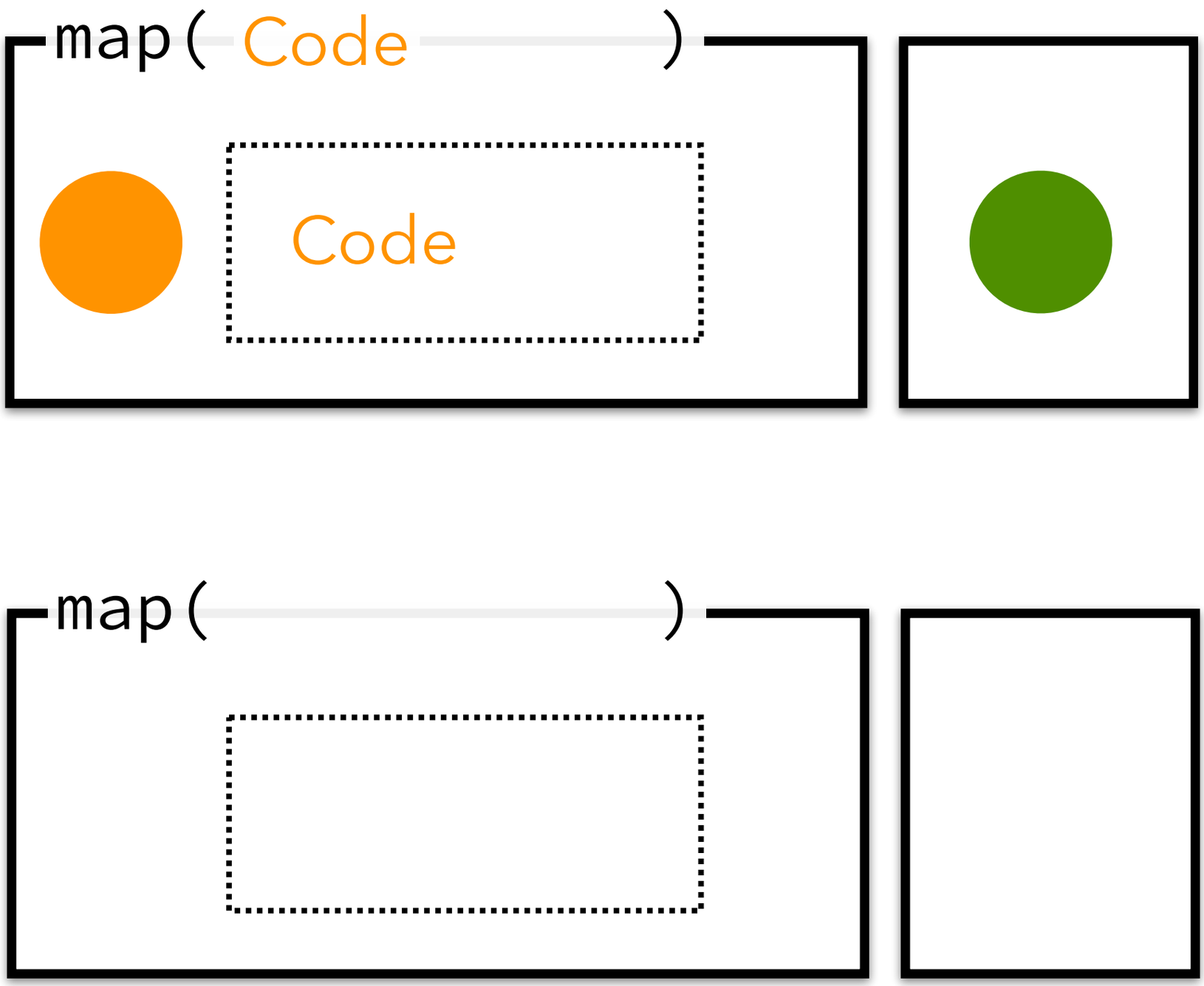
```
public List<String> formatReports(List<String> lines)
{
    List<String> output = new LinkedList();
      for(String s : lines) {
        output.add(formatBugReport(s));
      }
    return output;
}
```

Java

```
lines.stream()
    .map(s -> formatBugReport(s))
    .collect(Collectors.toList)
```

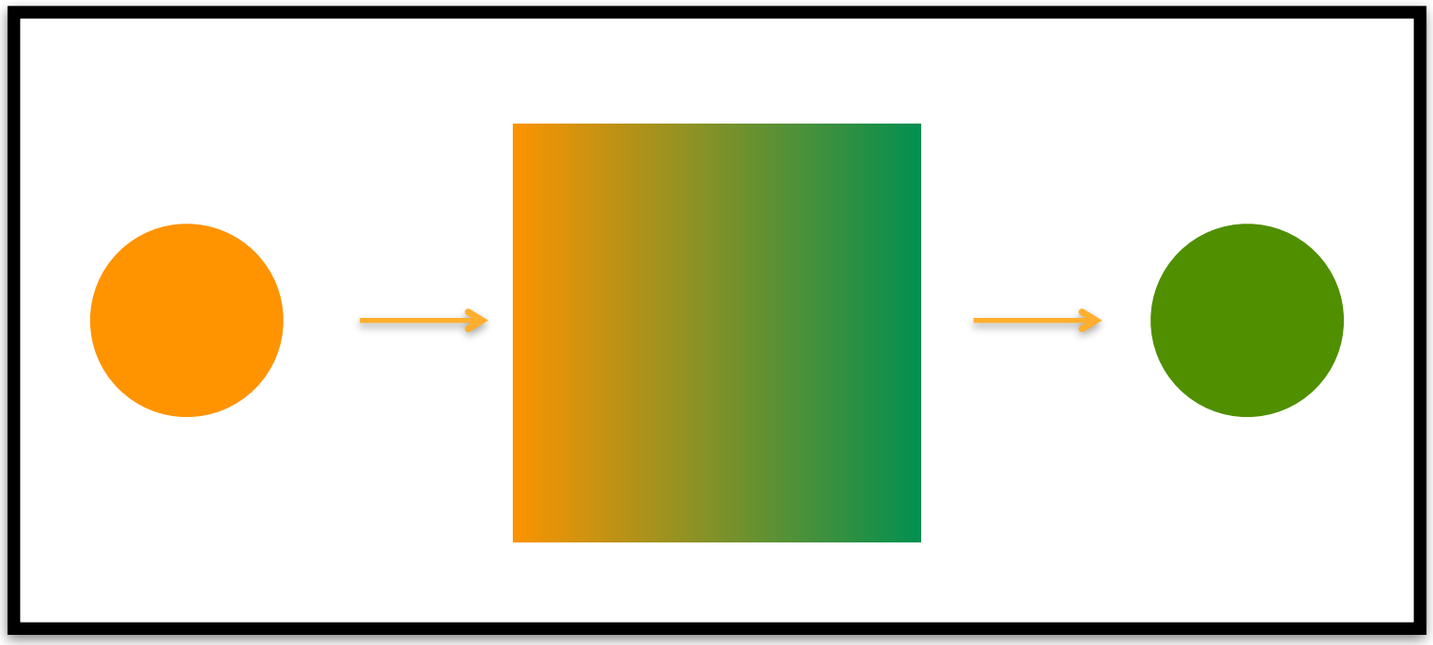

map(Code)


map(Code)

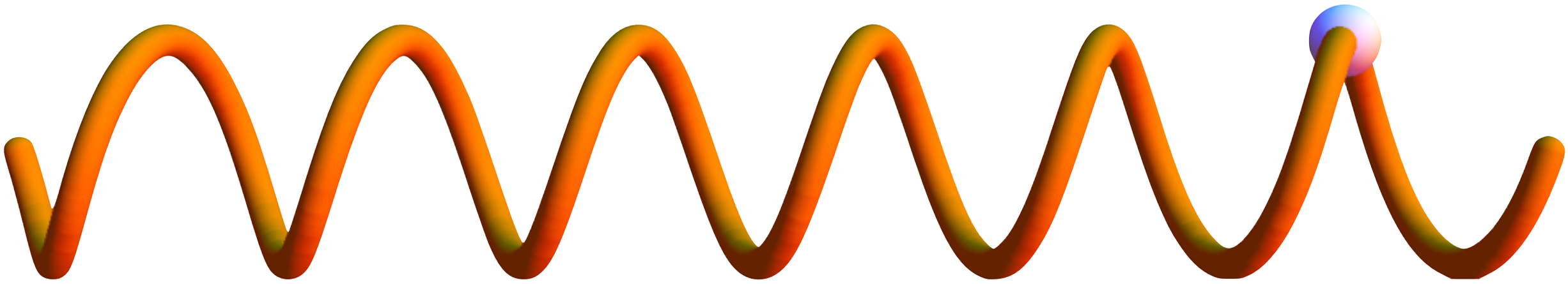

map(Code)
Code
map( )

Java 8

```
Optional<String>
        sampleBugReport(
                Stream<String> bugLines) {
    Optional<String> bugReport =
                        bugLines.first();

    return bugReport.map(formatBugReport);
}
```



Lazy Evaluation

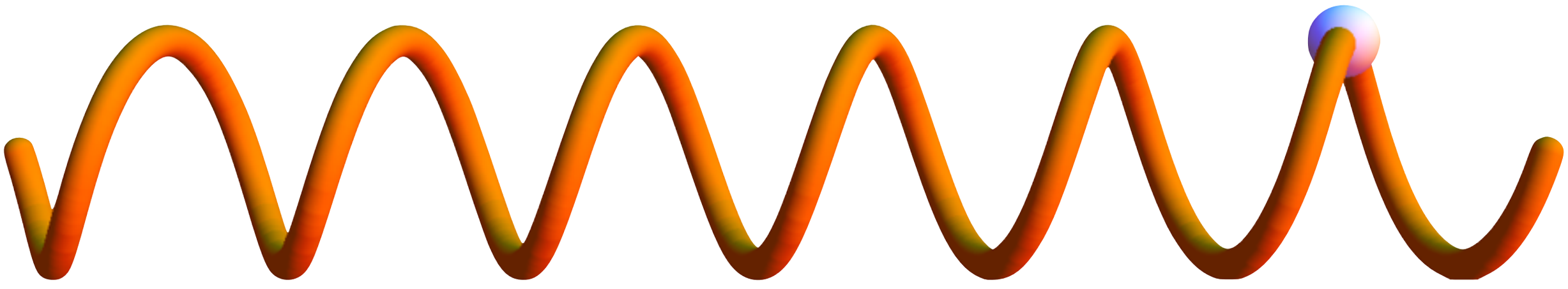


Why do today what you can put off unti

眉間一字寬 .萬里眼中明

save work

ignore ‘when’



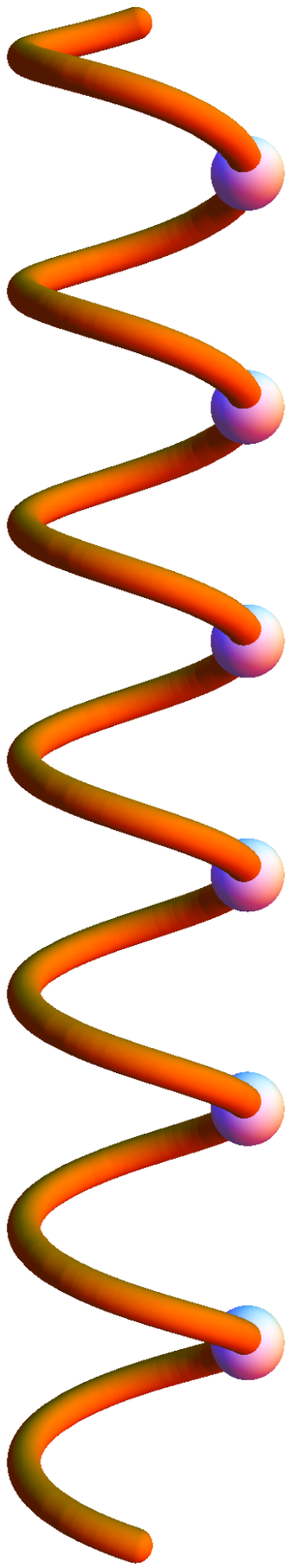
separate **what to do**
from **when to stop**

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++;
    }
    waitUntilFileHasMoreData(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

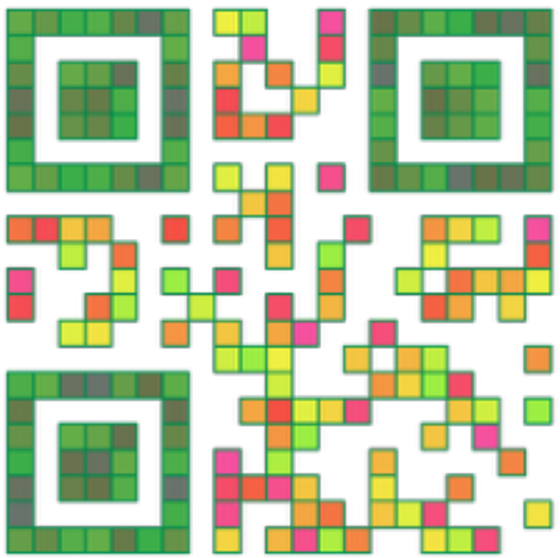
I promise not to exclude
from consideration any idea
based on its source, but to consider ideas
across schools and heritages
in order to find the ones that best suit the
current situation.

Optional<T>

Tuple<T>

by Faqqotic

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



Jessica Kerr

blog.jessitron.com

@jessitron

github.com/jessitron/fp4ood