On the way to functional programming

@PatrickGIRY

Who am I ?

Patrick GIRY

Software gardener

arolla

patrick.giry@arolla.fr

@PatrickGIRY

https://github.com/PatrickGIRY

20 ans d'expérience dans le développement logiciel

Test Driven Development
Behavior Driven Development
Domain Driven Design
Functional programming

CO-ORGANIZER

Meetup

Coding Dojo

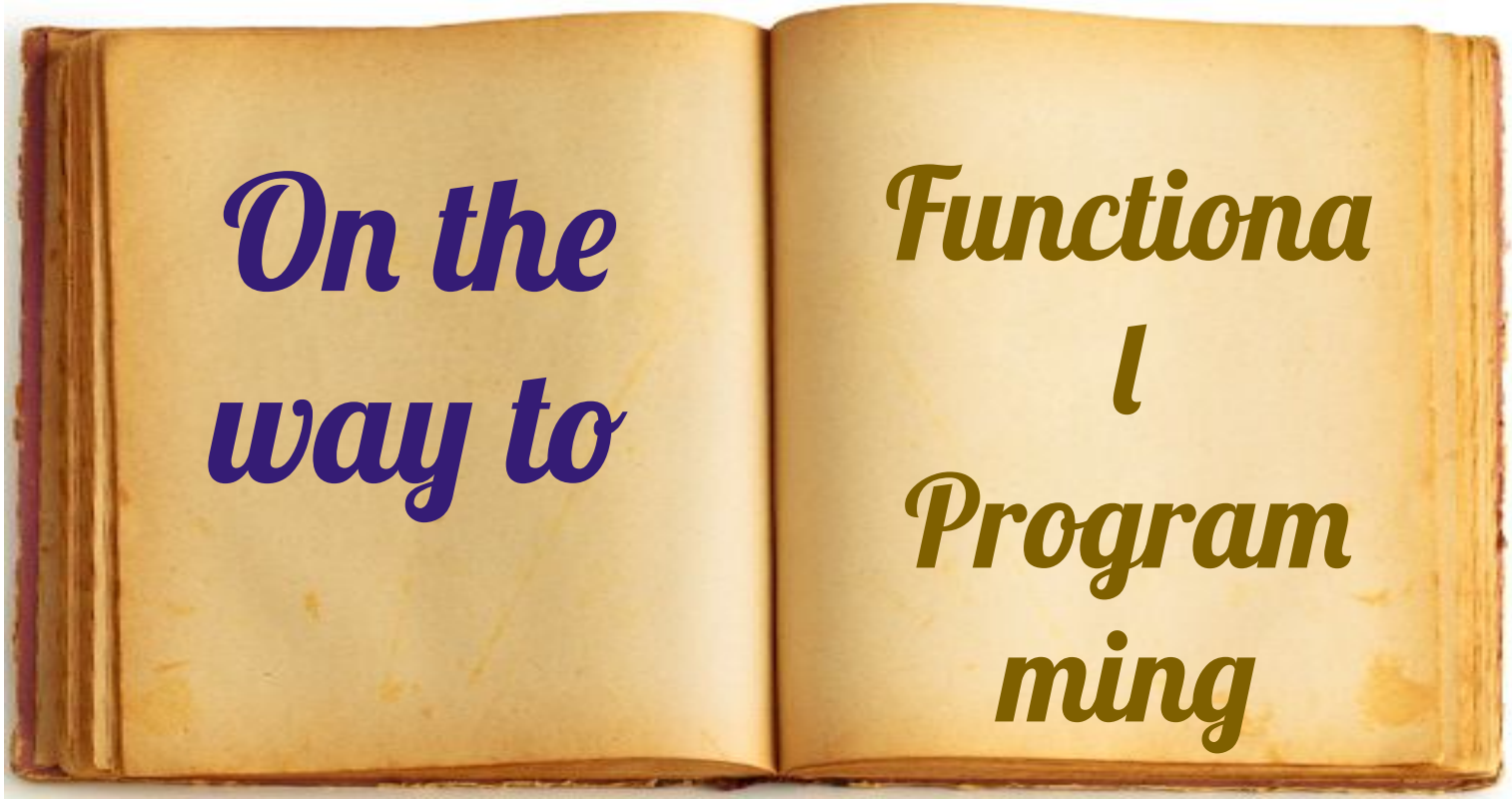
Dojo développement Paris

KataCatalogue

- KataBankOCR
- KataFizzBuzz
- FooBarQix
- KataPotter
- KataRomanNumerals
- KataRomanCalculator
- KataNumbersInWords
- KataArgs
- KataAnagram
- KataDepthFirstSearch
- KataNumberToLCD

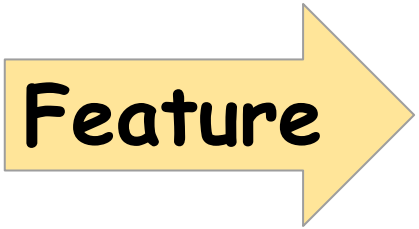
https://github.com/dojo-developpement-paris/

Co-organizer



Patrick Giry - @PatrickGiry

Red thread project



As an event organizer, **I want to** search for registered attendees by first name, **in order to** be able to create the list of attendees actually present at the event.

List test case

0, 1, many…

Acceptance criteria

- The finder should return
  - an empty result when no attendee first name matches the query string.
  - one result when only one attendee first name matches the query string.
  - all the results when many attendees first names match the query string.

Demo

Unit test setup

```
@DisplayName("The attendees finder should return")
class AttendeesFinderTest {

    private static final Attendee MARC = Attendee.withFirstName("Marc");
    private static final Attendee CHRISTELLE =
                                            Attendee.withFirstName("Christelle");
    private static final Attendee CHRISTOPHE =
                                            Attendee.withFirstName("Christophe");

    private Attendees attendees;

    @BeforeEach
    void setUp() {
        attendees = new InMemoryAttendeesRepository();
        attendees.append(MARC);
        attendees.append(CHRISTOPHE);
        attendees.append(CHRISTELLE);
    }
```

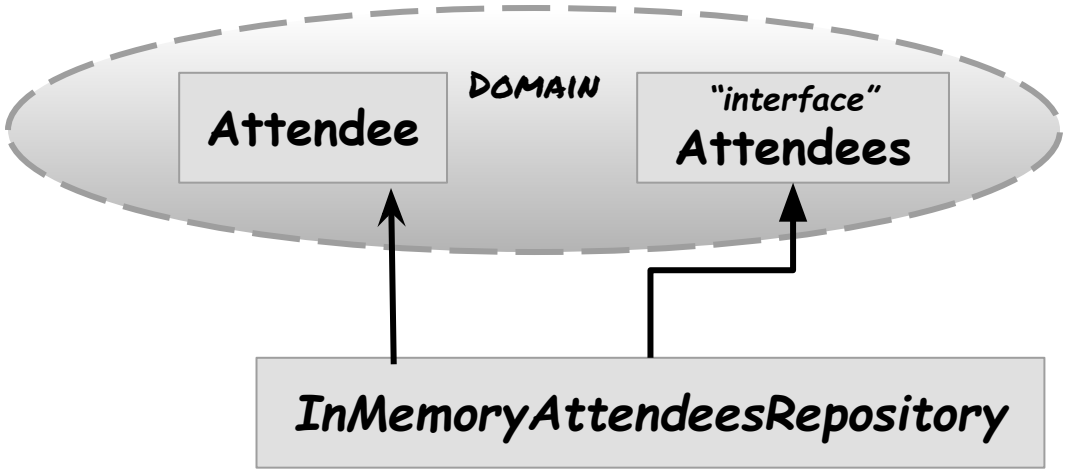
Unit test case

```
@Test
@DisplayName("an empty result when no attendee first name matches the query string")
void no_attendee_matches() {
   List<Attendee> result = attendees.findByInfixOfFirstName("Paul");
   assertThat(result).isEmpty();
}

@Test
@@DisplayName("one result when only one attendee first name matches the query string")
void one_attendee_matches() {
   List<Attendee> result = attendees.findByInfixOfFirstName("Marc");
   assertThat(result).containsOnly(MARC);
}

@Test
@DisplayName("all the results when many attendees first names match the query string")
void many_attendees_matche() {
   List<Attendee> result = attendees.findByInfixOfFirstName("hri");
   assertThat(result).containsOnly(CHRISTELLE, CHRISTOPHE);
}
```

Architecture



Domain - Attendee

```
public class Attendee {
   private final String firstName;

   public static Attendee withFirstName(String firstName) { return new Attendee(firstName); }

   public static Attendee copyOf(Attendee attendee) {
     return withFirstName(attendee.firstName);
   }

   private Attendee(String firstName) { this.firstName = firstName; }

   public boolean isFirstNameInfixOf(String query) { return firstName.contains(query); }

   @Override
   public boolean equals(Object o) { }

   @Override
   public int hashCode() { }

   @Override
   public String toString() { }
}
```

Domain - Attendees

```
public interface Attendees {
   void append(Attendee attendee);

   List<Attendee> findByInfixOfFirstName(String query);
}
```

First imperative implementation

```
public class InMemoryAttendeesRepository implements Attendees {
    private final List<Attendee> attendees = new ArrayList<>();

    @Override
    public void append(Attendee attendee) {
        this.attendees.add(Attendee.copyOf(attendee));
    }

    @Override
    public List<Attendee> findByInfixOfFirstName(String query) {
        List<Attendee> result = new ArrayList<>();
        while (!attendees.isEmpty()) {
            Attendee attendee = attendees.get(0);
            if (attendee.isFirstNameInfixOf(query)) {
                result.add(attendee);
            }
            attendees.remove(0);
        }
        return result;
    }
}
```

Unit test setup - Repository as singleton

```
@DisplayName("The attendees finder should return")
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class AttendeesFinderTest {

    private static final Attendee MARC = Attendee.withFirstName("Marc");
    private static final Attendee CHRISTELLE =
                                    Attendee.withFirstName("Christelle");
    private static final Attendee CHRISTOPHE =
                                    Attendee.withFirstName("Christophe");

    private Attendees attendees;

    @BeforeAll
    void setUp() {

        attendees = new InMemoryAttendeesRepository();
        attendees.append(MARC);
        attendees.append(CHRISTOPHE);
        attendees.append(CHRISTELLE);
    }
```

First imperative implementation with side effect

```
public class InMemoryAttendeesRepository implements Attendees {
    private final List<Attendee> attendees = new ArrayList<>();

    @Override
    public void append(Attendee attendee) {
        this.attendees.add(Attendee.copyOf(attendee));
    }

    @Override
    public List<Attendee> findByInfixOfFirstName(String query) {
        List<Attendee> result = new ArrayList<>();
        while (!attendees.isEmpty()) {
            Attendee attendee = attendees.get(0);
            if (attendee.isFirstNameInfixOf(query)) {
                result.add(attendee);
            }
            attendees.remove(0);
        }
        return result;
    }
}
```

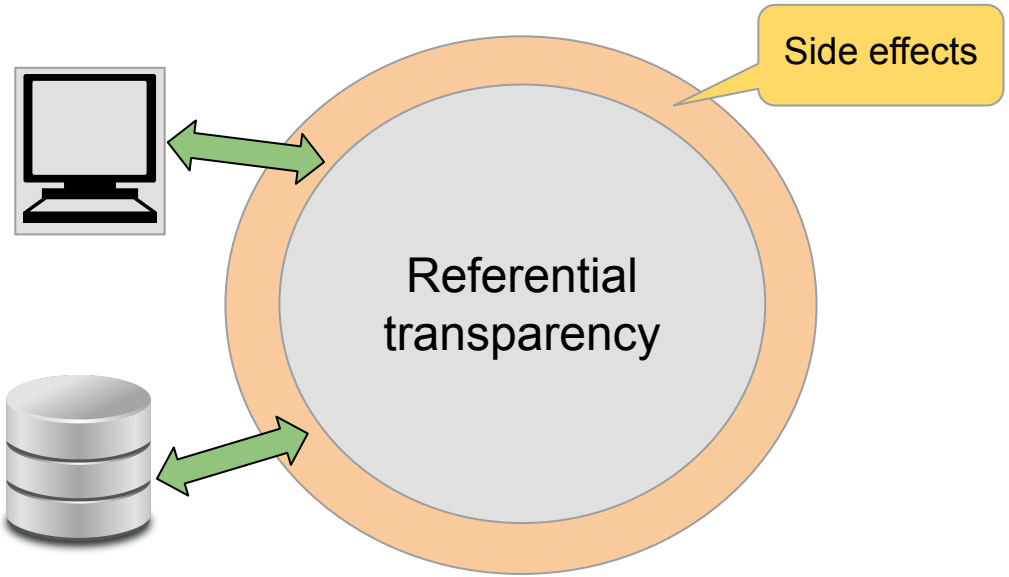
Fix side effects

```
public class InMemoryAttendeesRepository implements Attendees {
    private final List<Attendee> attendees = new ArrayList<>();

    @Override
    public void append(Attendee attendee) {
        this.attendees.add(Attendee.copyOf(attendee));
    }

    @Override
    public List<Attendee> findByInfixOfFirstName(String query) {
        List<Attendee> result = new ArrayList<>();
        for (Attendee attendee : attendees) {
            if (attendee.isFirstNameInfixOf(query)) {
                result.add(attendee);
            }
        }
        return result;
    }
}
```

Put side effects at the periphery



From imperative to procedural paradigm

```
public class InMemoryAttendeesRepository implements Attendees {
    ...
    @Override
    public List<Attendee> findByInfixOfFirstName(String query) {
        List<Attendee> result = new ArrayList<>();
        for (Attendee attendee : attendees) {
            if (matches(query, attendee)) {
                result.add(attendee);
            }
        }
        return result;
    }

    private boolean matches(String query, Attendee attendee) {
        return attendee.isFirstNameInfixOf(query);
    }
}
```

Suppress second level of indentation

```
public class InMemoryAttendeesRepository implements Attendees {
  ...
   @Override
   public List<Attendee> findByInfixOfFirstName(String query) {
       List<Attendee> result = new ArrayList<>();
       for (Attendee attendee : attendees) {
           addIfMatches(query, result, attendee);
       }
       return result;
   }

   private void addIfMatches(String query, List<Attendee> result,
                                                        Attendee attendee) {
       if (matches(query, attendee)) {
           result.add(attendee);
       }
   }
     ...
}
```

Looking for side effects

If a method signature indicates a return void and the method is used the it does a side effect.

```
void addIfMatches(String query, List<Attendee> result, Attendee attendee) {
    if (matches(query, attendee)) {
        result.add(attendee);
    }
}
```

Function as value - First class citizen

```
public class InMemoryAttendeesRepository implements Attendees {
 ...
  @Override
  public List<Attendee> findByInfixOfFirstName(String query) {
     List<Attendee> result = new ArrayList<>();
     BiPredicate<String, Attendee> predicate = (q, attendee) ->
                                                    matches(q, attendee);
     for (Attendee attendee : attendees) {
        addIf(predicate, query, result, attendee);
     }
     return result;
  }

  private void addIf(BiPredicate<String, Attendee> predicate, String query, List<Attendee> result,
                                                                  Attendee attendee) {
     if (predicate.test(query, attendee)) {  result.add(attendee); }
  }
```

Closure - Avoid to pass result to addIf

```
public class InMemoryAttendeesRepository implements Attendees {
 ...
  @Override
  public List<Attendee> findByInfixOfFirstName(String query) {
    List<Attendee> result = new ArrayList<>();
    BiPredicate<String, Attendee> predicate = this::matches;
    Consumer<Attendee> append = attendee -> result.add(attendee);
     for (Attendee attendee : attendees) {
        addIf(predicate, query, attendee, append);
     }
     return result;
  }

  private void addIf(BiPredicate<String, Attendee> predicate, String query,
                                                          Attendee attendee, Consumer<Attendee> append) {
     if (predicate.test(query, attendee)) {
        append.accept(attendee);
     }
  }
```

Higher order function - To not be silly, let others do

```
public List<Attendee> findByInfixOfFirstName(String query) {
   List<Attendee> result = new ArrayList<>();
   BiPredicate<String, Attendee> predicate = this::matches;
   Consumer<Attendee> append = result::add;
   for (Attendee attendee : attendees) {
       var consumer = addIf(predicate, query, attendee, append);
       consumer.accept(attendee);
   }
   return result;
}

private Consumer<Attendee> addIf(BiPredicate<String, Attendee> predicate, String query,
                    Attendee attendee, Consumer<Attendee> append) {
   if (predicate.test(query, attendee)) {
        return append;
    } else {
        return attendee1 -> {};
    }
}
```

HOF - Change matches and remove query parameter of addIf

```
public List<Attendee> findByInfixOfFirstName(String query) {
    List<Attendee> result = new ArrayList<>();
    Predicate<Attendee> predicate = matches(query);
    Consumer<Attendee> append = result::add;
    for (Attendee attendee : attendees) {
        final var consumer = addIf(predicate, attendee, append);
        consumer.accept(attendee);
    }
    return result;
}

private Predicate<Attendee> matches(String query) {
    return attendee -> attendee.isFirstNameInfixOf(query);
}
```

Replace append by concat - Be honest

```
public List<Attendee> findByInfixOfFirstName(String query) {
  Predicate<Attendee> predicate = matches(query);
  BiFunction<Attendee, List<Attendee>, List<Attendee>> concat = this::concat;
  List<Attendee> result = new ArrayList<>();
  for (Attendee attendee : attendees) {
     var fn = addIf(predicate, attendee, concat);
     result = fn.apply(attendee, result);
  }
  return result;
}

private List<Attendee> concat(Attendee attendee, List<Attendee> attendees) {
   var newAttendees = new ArrayList<>(attendees);
   newAttendees.add(attendee);
   return newAttendees;
}

private BiFunction<Attendee, List<Attendee>, List<Attendee>> addIf(Predicate<Attendee> predicate, Attendee attendee,
                                                                   BiFunction<Attendee, List<Attendee>, List<Attendee>> concat) {
  if (predicate.test(attendee)) {
     return concat;
  } else {
     return (attendee1, attendees) -> attendees;
  }
}
```

Currying concat

```
public List<Attendee> findByInfixOfFirstName(String query) {
  Predicate<Attendee> predicate = matches(query);
  Function<Attendee, Function<List<Attendee>, List<Attendee>>> concat =
                                        attendee -> attendees -> concat(attendee, attendees);
  List<Attendee> result = new ArrayList<>();
  for (Attendee attendee : attendees) {
    var fn = addIf(predicate, attendee, concat);
    result = fn.apply(attendee).apply(result);
  }
  return result;
}

private List<Attendee> concat(Attendee attendee, List<Attendee> attendees) {
    var newAttendees = new ArrayList<>(attendees);
    newAttendees.add(attendee);
    return newAttendees;
}

private Function<Attendee, Function<List<Attendee>, List<Attendee>>>addIf(Predicate<Attendee> predicate,
                Attendee attendee, Function<Attendee, Function<List<Attendee>, List<Attendee>>> concat) {
  if (predicate.test(attendee)) {
    return concat;
  } else {
    return (attendee1, attendees) -> attendees;
  }
```

Partial application

```
public List<Attendee> findByInfixOfFirstName(String query) {
   final Predicate<Attendee> predicate = matches(query);
   final Function<Attendee, Function<List<Attendee>, List<Attendee>>> concat =
                                    attendee -> attendees -> concat(attendee, attendees);
   List<Attendee> result = new ArrayList<>();
   for (Attendee attendee : attendees) {
       final var fn = addIf(predicate, attendee, concat);
       result = fn.apply(result);
   }
   return result;
}
private Function<List<Attendee>, List<Attendee>> addIf(Predicate<Attendee> predicate,
 Attendee attendee, Function<Attendee, Function<List<Attendee>, List<Attendee>>> concat) {
   if (predicate.test(attendee)) {
       return concat.apply(attendee);
   } else {
       return attendees -> attendees;
   }
}
```

Extract filter method

```
public List<Attendee> findByInfixOfFirstName(String query) {
    final Predicate<Attendee> predicate = matches(query);
    return filter(predicate, attendees);
}

private static List<Attendee> filter(Predicate<Attendee> predicate,
                                     List<Attendee> attendees) {
    final Function<Attendee, Function<List<Attendee>, List<Attendee>>> concat =
                        attendee -> attendees -> concat(attendee, attendees);
    List<Attendee> result = new ArrayList<>();
    for (Attendee attendee : attendees) {
        result = addIf(predicate, attendee, concat).apply(result);
    }
    return result;
}
```

Simplify filter - transform in stream collect

```
private static List<Attendee> filter(Predicate<Attendee> predicate,
                                     List<Attendee> attendees) {
    List<Attendee> result = new ArrayList<>();
    for (Attendee attendee : attendees) {
        if (predicate.test(attendee)) {
            result.add(attendee);
        }
    }
    return result;
}
```

```
private static List<Attendee> filter(Predicate<Attendee> predicate,
                                     List<Attendee> attendees) {
    return attendees.stream()
                    .filter(predicate).collect(Collectors.toList());
}
```

InMemoryAttendeesRepository - final version

```
public class InMemoryAttendeesRepository implements Attendees {
    private final List<Attendee> attendees = new ArrayList<>();

    @Override
    public void append(Attendee attendee) {
        this.attendees.add(Attendee.copyOf(attendee));
    }

    @Override
    public List<Attendee> findByInfixOfFirstName(String query) {
        return filter(matches(query), attendees);
    }

    private static List<Attendee> filter(Predicate<Attendee> predicate,
                                         List<Attendee> attendees) {
        return attendees.stream().filter(predicate).collect(Collectors.toList());
    }

    private Predicate<Attendee> matches(String query) {
        return attendee -> attendee.isFirstNameInfixOf(query);
    }
}
```

Haskell - data

```haskell
data Attendee = Attendee {
    firstName :: String
} deriving (Show, Eq)
```

```java
public class Attendee {
   private final String firstName;

   private Attendee(String firstName) {
     this.firstName = firstName;
   }

   public String firstName() {
     return firstName;
   }

   @Override
   public boolean equals(Object o) { }

   @Override
   public int hashCode() { }

   @Override
   public String toString() { }
}
```

Haskell - unit tests

```
main = hspec $ do
    let attendees = [Attendee { firstName = "Marc" },
                     Attendee { firstName = "Christelle" },
                     Attendee { firstName = "Christophe" }]

    describe "The attendees finder should return" $ do
        it "an empty result when no attendee first name matches the query string" $ do
            findByInfixOfFirstName "Paul" attendees `shouldBe` []

        it "one result when only one attendee first name matches the query string" $ do
            findByInfixOfFirstName "Marc" attendees `shouldBe`
                                                    [Attendee {firstName = "Marc"}]

        it "all the results when many attendees first names match the query string" $ do
            findByInfixOfFirstName "hri" attendees `shouldBe`
                [Attendee {firstName = "Christelle"}, Attendee {firstName = "Christophe"}]
```

Haskell - Implementation

```
findByInfixOfFirstName :: String -> [Attendee] -> [Attendee]
findByInfixOfFirstName query attendees = filter (matches query) attendees
   where matches :: String -> Attendee -> Bool
         matches query attendee = query `isInfixOf` (firstName attendee)
```

```
  public List<Attendee> findByInfixOfFirstName(String query) {
        return filter(matches(query), attendees);
    }

    private static List<Attendee> filter(Predicate<Attendee> predicate,
                                         List<Attendee>
                                    attendees) {
        return attendees.stream().filter(predicate).collect(Collectors.toList());
    }

    private Predicate<Attendee> matches(String query) {
        return attendee -> attendee.isFirstNameInfixOf(query);
    }
```

Summary

- Imperative paradigm
- Side effects - Put at the periphery
- Referential transparency
- Procedural paradigm
- Only one level of indentation
- Function as value - First class citizen
- Lambda - method reference
- Closure
- Higher order function
- Be honest
- Currying
- Partial application
- Filter
- Stream
- Haskell

Conclusion

- FP code is more concise
- Minimizes side effects
- Easier to test and debug
- Facilitates parallelization (mutli-cores)
- Haskell can help think functional to develop


The End

https://github.com/PatrickGIRY/attendeesFinder