Refactoring to Functional Architecture Patterns

George Fairbanks
SATURN 2018
9 May 2018

This talk

Talk last year at SATURN:

- Functional Programming Invades Architecture
- Ideas from the functional programming (FP) community
- Relevant to software architecture

Today's talk:

- Experience report based on applying those ideas
- Joins FP and OO design

Boring slides (repeated from last year) about an interesting topic

Big ideas in Functional Programming

Function composition

- Build programs by combining small functions

  `g(f(x))` or `f(x) |> g(x)`

- Seen in pipelines, event-based systems, machine learning systems, reactive

  `ls | grep “foo” | uniq | sort`

Note: We’re just covering the FP ideas that seem relevant to architecture

Pure functions, no side effects

- Calling function with same params always yields same answer
- So: Reasoning about the outcome is easier

```
curl http://localhost/numberAfter/5 → [always 6]
curl http://localhost/customer/5/v1 → [always v1 of customer]
```

vs

```
curl http://localhost/customer/5 → [may be different]
```

Statelessness and Immutability

Statelessness

- If there's no state:

- Easy to reason about
- All instances are equivalent

Immutability

- If you have state, but it never changes:

- Easy to reason about
- Concurrent access is safe

Idempotence

- Idempotence: get the same answer regardless of how many times you do it

  `resizeTo100px(image)` vs `shrinkByHalf(image)`

- Often hard to guarantee something is done exactly once

- Eg: Is the task stalled or failed?
  - Go ahead and schedule it again without fear that you'll get a duplicate result

Declarative, not imperative

- Define what something ***is***
  … or how it relates to other things
- Versus
  - Series of operations that yield desired state

- Definition, not procedure

Declarative, not imperative

Example: how much paint do I need?

```
while(!done) { fence.paint(); }
```

Versus:

```
float paintNeeded(float len, float wid) {
  float coverage = 0.52;
  return len * wid * coverage;
}
```

We need to go deeper

Immutable models

```
@Autovalue
public abstract class Customer {
  String getName();

  static Customer of(String name) {
    return new Autovalue_Customer(name);
  }
}
...
Customer ann = Customer.of(“Ann Smith”);
Product car = Car.of(“Mustang”, 500);
Sale sale1 = Sale.of(car, ann);
```

Also:
Make illegal states unrepresentable

Yaron Minsky, Make illegal states unrepresentable. 2011.

Expressions vs statements

```
List<Integer> primes = new ArrayList<>();
primes.add(2);
primes.add(3);
primes.add(5);
```

Statements

```
ImmutableList<Integer> primes = ImmutableList.of(2, 3, 5);

private static final ImmutableList<Integer> PRIMES =
  ImmutableList.builder()
    .add(2)
    .add(3)
    .add(5)
    .build();
```

Expressions

Fluent streams / collection pipeline

```
ImmutableList<Customer> premiumCustomers =
  customers.stream()
    .filter(customer -> customer.isPremium())
    .collect(toImmutableList());
```

Martin Fowler, Collection Pipeline https://martinfowler.com/articles/collection-pipeline, 2014

Fluent streams / collection pipeline

```
ImmutableList<Customer> premiumCustomers =
  customers.stream()
    .filter(Customer::isPremium)
    .collect(toImmutableList());
```

Martin Fowler, Collection Pipeline https://martinfowler.com/articles/collection-pipeline, 2014

Build up a DSL

```
/** Returns true iff customer buys a lot. */
boolean isPremium(Customer c) {
  return 5 < allSales.stream()
    .filter(sale -> sale.customer().equals(c))
    .count();
}
```

Build up a DSL (2)

```
/** Returns true iff customer buys a lot. */
boolean isPremium(Customer c) {
  return PREMIUM_THRESHOLD < allSales.stream()
    .filter(sale -> sale.customer().equals(c))
    .count();
}
```

Build up a DSL (3)

```
/** Returns true iff customer buys a lot. */
boolean isPremium(Customer c) {
  return PREMIUM_THRESHOLD < salesCount(c);
}

/** Returns the number of sales made by customer. */
boolean salesCount(Customer c) {
  return allSales.stream()
    .filter(sale -> sale.customer().equals(c))
    .count();
}
```

See ‘definition’, ‘designation’, ‘narrow bridge’ in Michael Jackson’s Software Requirements and Specifications. Buy it now.

Domain model and DSL

- Advice: Grow a DSL organically so the code reads naturally
  - Verbose or awkward code → refactor
  - `PREMIUM_THRESHOLD < salesCount(customer)`

- Some domains are well sorted out already
  - Eg everyone knows banking has accounts, debits, credits, transactions, etc.
  - Other domains are invented with the application, in part by the developers

Problem and solution domains

- Code expresses a combination of the **problem domain** and the **solution domain**
  - Eg customers (problem) and relational tables (solution)

- For IT code, expressions in the code should lean towards the domain
  - It's possible that you need a spin lock in your IT application, but it shouldn't be the first thing I see in the code

Problem and solution domains

Problem domain

- Customer
- Product
- Sale

Solution domain

- ListCustomersRequest, ListCustomersResponse
- CustomerProto
- ProductProto

Parsing at system boundary

Problem domain

- Customer
- Product
- Sale

No protos in here

Solution domain

- ListCustomersRequest, ListCustomersResponse
- CustomerProto
- ProductProto

Converters between domains

- Converter<Customer, CustomerProto>
- Converter<Product, ProductProto>

Guava Converter interface

Domain model and DSL

The **domain model** is a
domain-specific language (**DSL**) that has:

- Predicates on state
  (eg `isPremium`)
- Query methods
  (eg `salesCount`)
- Transformations (pure functions)

It does NOT have:

- Mutation
- Business logic
  (that is in the Rich Service Layer)

Probably an Anemic Domain Model

Rich service layer pattern

- Old tension:
  - Transaction Script pattern
  - Domain Model pattern

- We followed a pattern we call the **Rich Service Layer**
  - Pure functions on top of an Anemic Domain Model

- Our domain model
  - Immutable; minimal optional data (ie not just data bags)
  - Strictly defined types
  - Integrity checks at system boundary
  - Less behavior than traditional OO

Lookup, validate, operate

As we refactored to use Functional Programming, this pattern emerged:

1. Lookup / load the needed data
2. Validate it
3. Operate on it

In earlier days, I would have done all of these in the domain model objects.
Now, the last step is rarely in the objects -- it's instead in a Rich Service.

Monads

Optional, not NULL

```
Optional<Customer> customer = customerDao.getCustomer(5);
String zipCode = customer.address().zipCode().orElse("missing");

String zipCode =
  customerDao.getCustomer(5)
    .address()
    .zipCode()
    .orElse("missing");
```

Result: Essentially zero null pointer exceptions in our code.

Try, really try

```
/** Returns an active customer with an address, or null. */
Customer getValidCustomer(CustomerId id) {
  Customer c = getCustomer(id);
  if (c == null) {return null;}
  if (c.address() == null) {return null;}
  if (!c.isActive()) {return null;}
  return c;
}
```

Try, really try

```
/** Returns an active customer with an address, or null. */
Customer getValidCustomer(CustomerId id) {
  Customer c = getCustomer(id);
  if (c == null) {return null;}
  if (c.address() == null) {return null;}
  if (!c.isActive()) {return null;}
  return c;
}
```

I hate writing this code
I hate reviewing this code

- Null checks everywhere
- Mechanics obscure the simple logic

Try, really try

```
/** Returns an active customer with an address, or empty. */
Optional<Customer> getValidCustomer(CustomerId id) {
  Optional<Customer> c = getCustomer(id);
  if (c.isEmpty()) {return c;}
  if (!c.get().address().isPresent()) {return Optional.empty();}
  if (!c.get().isActive()) {return Optional.empty();}
  return c;
}
```

Try, really try

```
/** Returns an active customer with an address, or failure. */
Try<Customer> getValidCustomer(CustomerId id) {
  Try<Customer> c = Try.fromOptional(getCustomer(id));
  if (c.isFailure()) { return c; }
  if (!c.get().address().isPresent()) {return Try.fail(); }
  if (!c.get().isActive()) {return Try.fail(); }
  return c;
}
```

Try, really try

```
/** Returns an active customer with an address, or failure. */
Try<Customer> getValidCustomer(CustomerId id) {
  return Try.fromOptional(getCustomer(id))
    .checkState(c -> c.address().isPresent(), "no address")
    .checkState(c -> c.isActive(), "inactive");
}
```

Try, really try

```
/** Returns an active customer with an address, or failure. */
Try<Customer> getValidCustomer(CustomerId id) {
  return Try.fromOptional(getCustomer(id))
    .checkState(Customer::hasAddress, "no address")
    .checkState(Customer::isActive, "inactive");
}
```

Try, really try

```
/** Returns an active customer with an address, or failure. */
Try<Customer> getValidCustomer(CustomerId id) {
  return Try.fromOptional(getCustomer(id))
    .checkState(Customer::hasAddress, "no address")
    .checkState(Customer::isActive, "inactive");
}
```

I like writing this code
I like reviewing this code

- No null checks
- Easy to see the simple logic

The end of the boring slides

Conclusion

FP helps express intent

- Express intent better:
  - Use fluent streaming operations
  - Use DSL
  - Optional / Try

- Grow the DSL:
  - Keep tweaking the expressiveness of your operations
  - Tests read easily when the DSL is great

- If you can’t express intent like: `salesCount(c) > PREMIUM_THRESHOLD`
  - Then your DSL / domain model is weak
  - Don’t: compensate with increasingly complex FP magic

Functional code smells

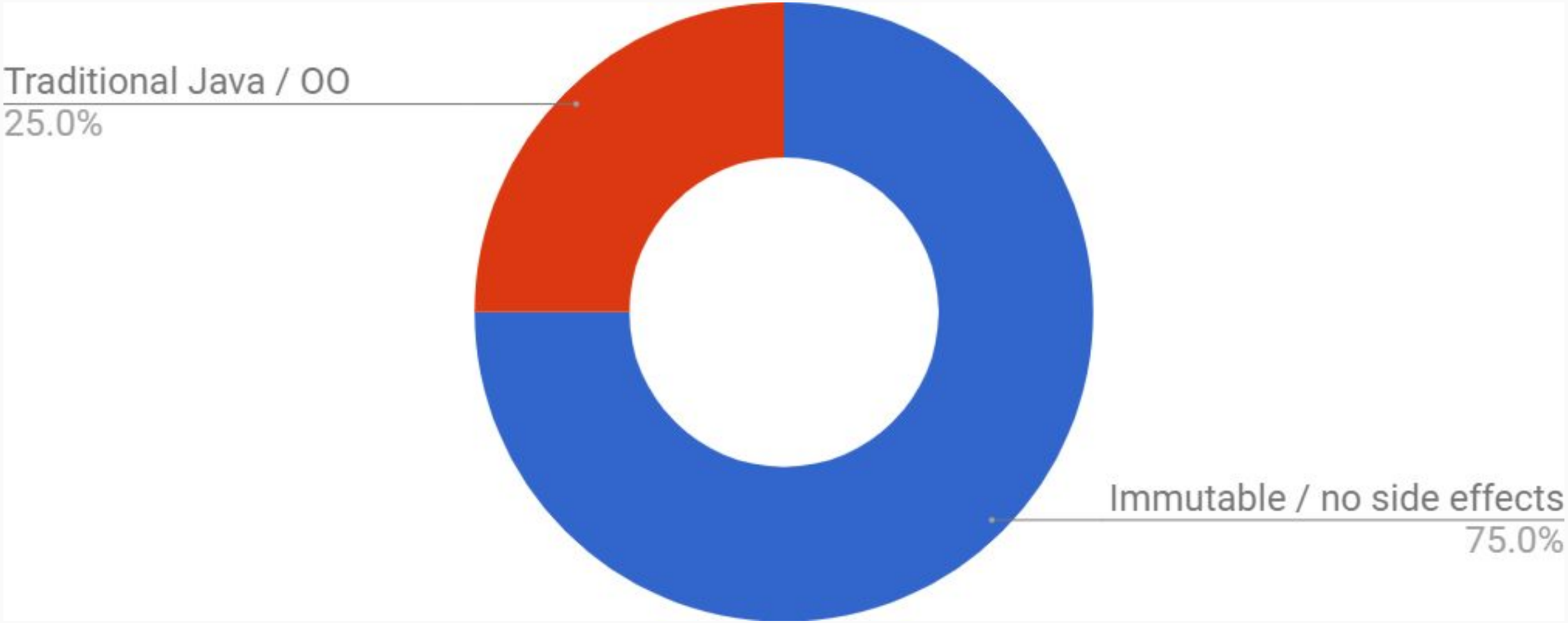
1. Lots of built-in types like `Map<Integer, String>`
   - Perhaps those are CustomerId and Address in disguise

Is it legal to add to this Integer? If not, it's not really an Integer.

2. Doubly or more nested types like `List<Map<Customer, Money>>`
   - Your model needs more types. Is that map an Account or a Ledger?

3. Complex or tricky functional transformations
   - If you wonder "why does this work?", your model is too simple

So, is any of this architecture?



Summary

We used these ideas / patterns

- Java8 streaming features (collection pipeline)
- No NULLs (Optional / Try instead)
- Domain model in immutable datatypes (Autovalue/Lombok)
- Make illegal states unrepresentable + Design by contract
- Domain Specific Language (DSL)
- Separate Lookup, Validation, Operation
- Favoring declarations over procedures
- Rich service layer (see below)
- Parsing at system boundary (no protos in the model)

The durable valuable parts

- Domain model
- Converters to/from external representations
- DAOs to external systems / datastores (gateway pattern)

Further reading

George Fairbanks, Functional Programming Invades Architecture. SATURN 2017. http://www.georgefairbanks.com/blog/saturn-2017-functional-programming-invades-architecture

Martin Fowler, Collection Pipeline. https://martinfowler.com/articles/collection-pipeline. 2014.

Michael Jackson, Software Requirements and Specifications. 1995. https://www.goodreads.com/book/show/582861.Software_Requirements_and_Specifications. Buy it now.

Guava Java library. https://github.com/google/guava.

Yaron Minsky, Make illegal states unrepresentable. https://blog.janestreet.com/effective-ml-revisited. 2011.