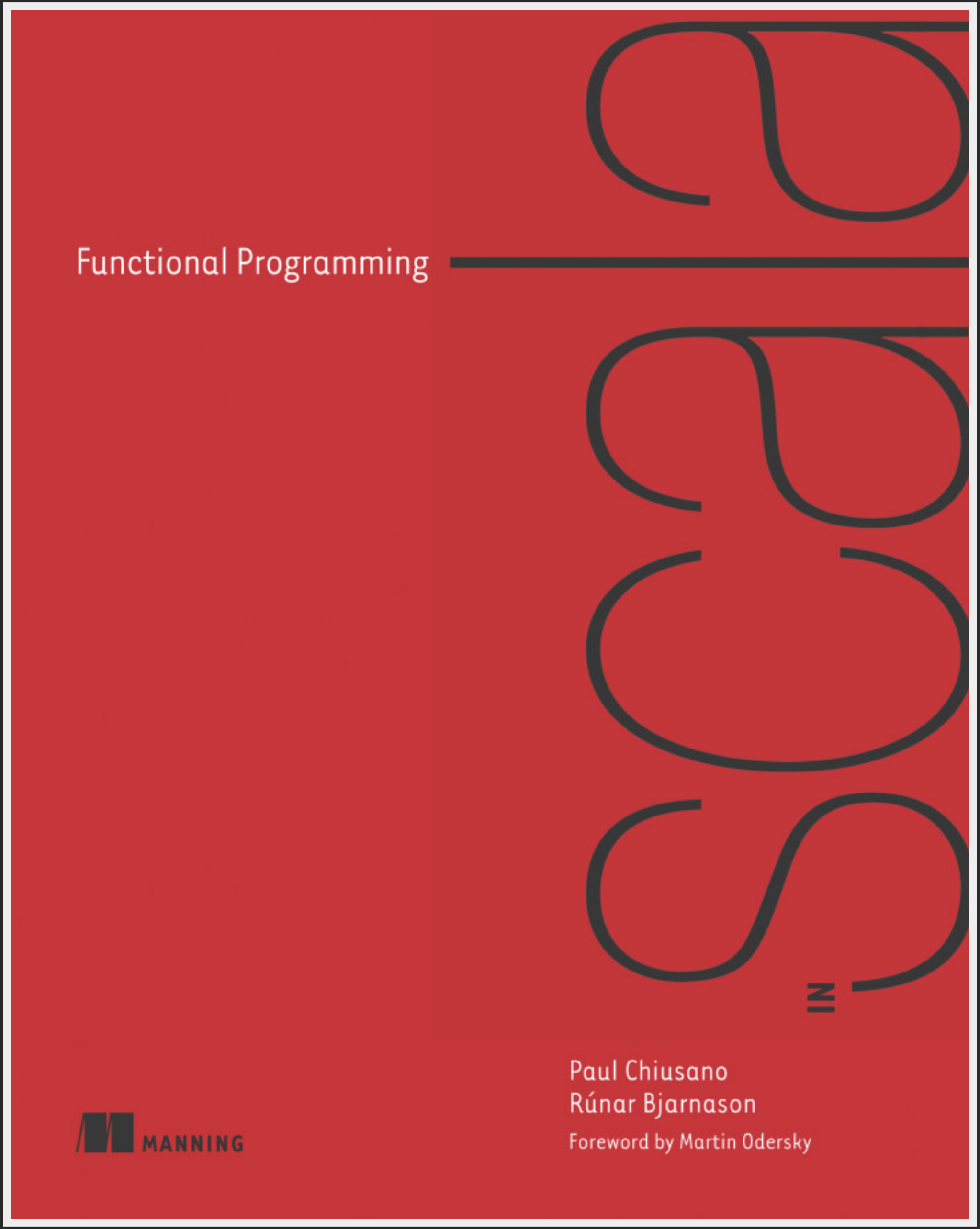
SIDE EFFECTS AND COMPOSITION

STOJAN ANASTASOV



Functional Programming IN scala

Paul Chiusano
Rúnar Bjarnason

Foreword by Martin Odersky

MANNING

MODELING A COFFEE SHOP

- Buying coffee
- Card payment

BUYING COFFEE V1

```
import coffee.*

fun buyCoffee(cc: CreditCard): Coffee {
    val cup = Coffee()
    cc.charge(cup.price)
    return cup
}
```

TESTABILITY

```
cc.charge(cup.price) // <-- Side effect
```

TESTABILITY

```
cc.charge(cup.price) // <-- Side effect
```

Talks to the CC company via API/SDK

TESTABILITY

```
cc.charge(cup.price) // <-- Side effect
```

Talks to the CC company via API/SDK

Should a CC know how it's charged

BUYING COFFEE V2

```
fun buyCoffee(cc: CreditCard, p: Payments): Coffee {
    val cup = Coffee()
    p.charge(cc, cup.price)
    return cup
}
```

DI to the rescue - mock `Payments` in tests

NEW REQUIREMENTS

NEW REQUIREMENTS

- Buy coffee
- Card payment

NEW REQUIREMENTS

- Buy coffee
- Card payment
- Buy X coffees

NEW REQUIREMENTS

- Buy coffee
- Card payment
- Buy X coffees
- One charge per credit card

COMPOSITION

Reuse the code for 1 coffee for X coffees

COMPOSITION

Reuse the code for 1 coffee for X coffees

- It's complex (could be)

COMPOSITION

Reuse the code for 1 coffee for X coffees

- It's complex (could be)
- It's tested

COMPOSITION

Reuse the code for 1 coffee for X coffees

- It's complex (could be)
- It's tested
- It's debugged

COMPOSITION

Reuse the code for 1 coffee for X coffees

- It's complex (could be)
- It's tested
- It's debugged
- Saves time

BUY MULTIPLE COFFEES

```
fun buyCoffees(
    cc: CreditCard,
    p: Payments,
    n: Int
): List<Coffee> = List(n) { buyCoffee(cc, p) }
```

BUY MULTIPLE COFFEES

```
fun buyCoffees(
    cc: CreditCard,
    p: Payments,
    n: Int
): List<Coffee> = List(n) { buyCoffee(cc, p) }
```

Charges the same CC multiple times :(

SIDE EFFECT

```
p.charge(cc, cup.price) // <-- Side effect
```

Charging the Credit Card prevents composition

WHAT IS A SIDE EFFECT

WHAT IS A SIDE EFFECT

Something that breaks referential transparency

REFERENTIAL TRANSPARENCY

An expression is called referentially transparent if it can be replaced with its corresponding value without changing the program's behavior. - Wikipedia

RT EXAMPLE

```
fun buyCoffee(cc: CreditCard, p: Payments): Coffee {
    val cup = Coffee()
    p.charge(cc, cup.price)
    return cup
}
```

```
val coffeeA = buyCoffee(cc, p)

val coffeeB = Coffee()
```

RT EXAMPLE

```
fun buyCoffee(cc: CreditCard, p: Payments): Coffee {
    val cup = Coffee()
    p.charge(cc, cup.price)
    return cup
}
```

```
val coffeeA = buyCoffee(cc, p)

val coffeeB = Coffee()
```

coffeeA is not the same as coffeeB -> means buyCoffee() is not referentially transparent

BATCHPAYMENTPROCESSOR

A Payment processor that can batch requests for the same Card

BATCHPAYMENTPROCESSOR

A Payment processor that can batch requests for the same Card

- How long do we wait
- How many charges do we batch
- Does buyCoffee() indicate start/end of a batch

BATCHPAYMENTPROCESSOR

A Payment processor that can batch requests for the same Card

- How long do we wait
- How many charges do we batch
- Does buyCoffee() indicate start/end of a batch

Also code doesn't fit in a slide

CAN WE DO BETTER

BUYING COFFEE V3

```
fun buyCoffee(cc: CreditCard): Pair<Coffee, Charge> {
    val cup = Coffee()
    val charge = Charge(cc, cup.price)
    return Pair(cup, charge)
}
```

Return a value (indicating the effect) instead of performing a side effect. We perform the side effect later.

COMBINE CHARGES

```kotlin
fun combine(c1: Charge, c2: Charge): Charge =
    if (c1.cc == c2.cc) Charge(c1.cc, c1.price + c2.price)
    else throw IllegalArgumentException(
        "Can't combine charges with different cc"
    )
```

BUYING X COFFEES

```kotlin
fun buyCoffees(
    cc: CreditCard,
    n: Int
): Pair<List<Coffee>, Charge> {
    val purchases: List<Pair<Coffee, Charge>> =
        List(n) { buyCoffee(cc) }
    val (coffees, charges) = purchases.unzip()
    val charge = charges.reduce { c1, c2 -> combine(c1, c2) }
    return Pair(coffees, charge)
}
```

BENEFITS

BENEFITS

- Better testability (no need for mocks)

BENEFITS

- Better testability (no need for mocks)
- Composition

BENEFITS

- Better testability (no need for mocks)
- Composition
- Separation of concerns

COMPOSITION AND SIDE EFFECTS

To achieve composition don't mix side effect with business logic.

Questions?