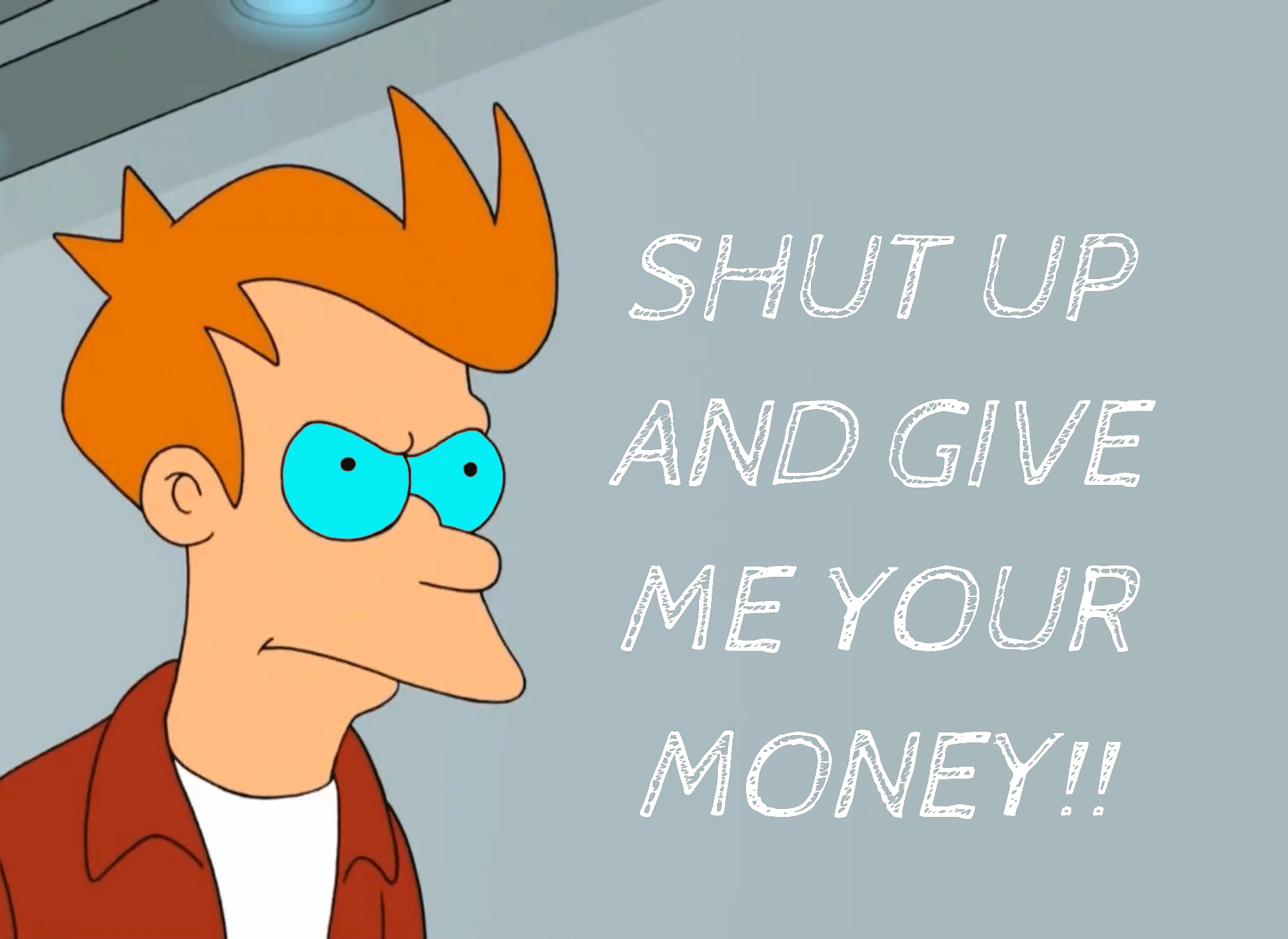

SHUT UP
AND GIVE
ME YOUR
MONEY!!

ticketbis

Álvaro Salazar

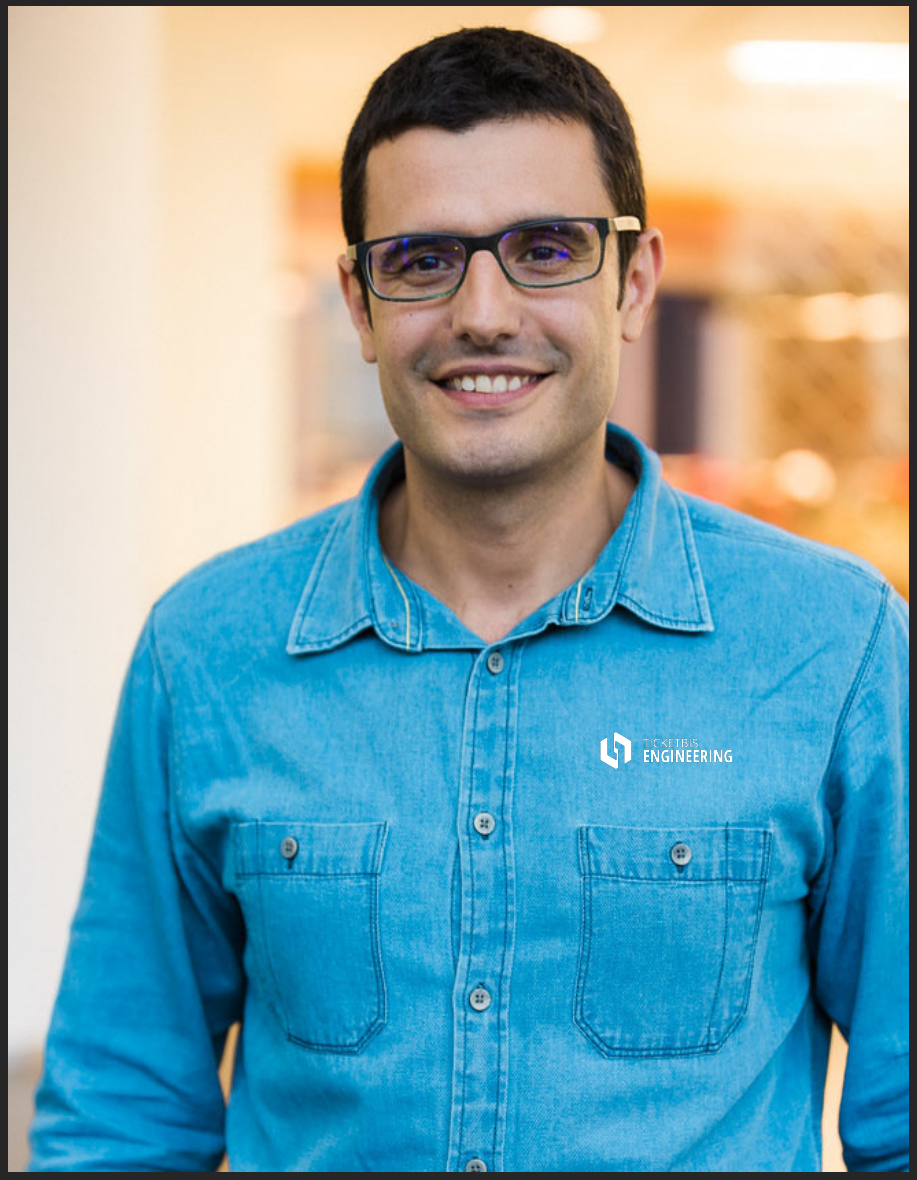
xala3pa

@xala3pa

@xala3pa

Greach

the Groovy spanish conf



Problems to Avoid

- Currency exchange and number format logic spread
- Using differents round modes
- Operations between diffent currencies
- Accuracy errors

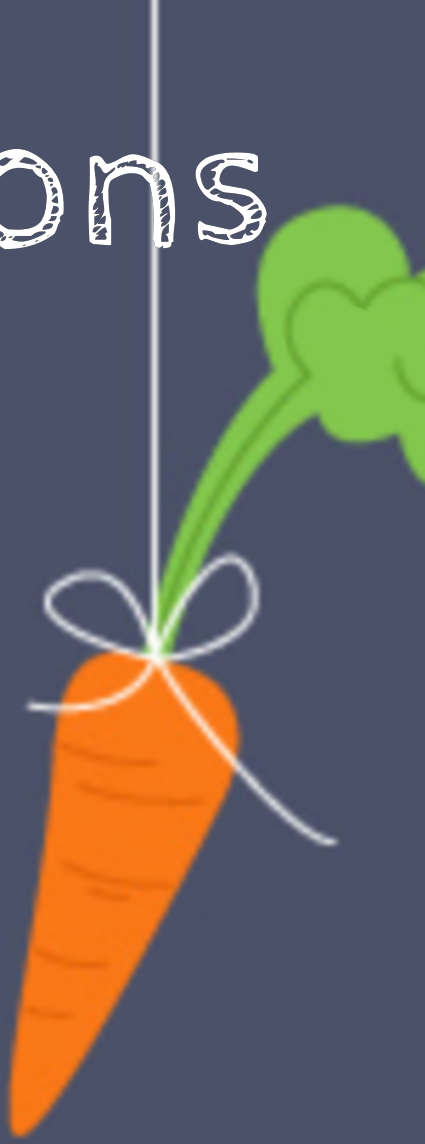
Design Motivations

↑ COHESION

↑ ENCAPSULATION

↓ PRIMITIVE OBSESSION

↓ FEATURE ENVY



TICKETBIS
ENGINEERING

COHESION

" *Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module*

https://en.wikipedia.org/wiki/Cohesion_(computer_science)

COHESION

 Increased system maintainability

 Increased module reusability

TICKETBIS
ENGINEERING

COHESION

Before Money:

```
[ 'totalPrice': CurrencyUtils.format( purchase.totalPrice, purchase.country ) ]


CurrencyService currencyService

BigDecimal totalPrice = currencyService.convertCurrency( purchase.totalPrice,
                                purchase.currency, gateway.currency, round )
```

After Money:

```
['totalPrice': purchase.totalPrice.format( Locale.US )]


Money totalPrice = purchase.totalPrice.exchangeTo( gateway.currency )
```

ENCAPSULATION

" Encapsulation binds together the data and functions that manipulate the data, and keeps both safe from outside interference and misuse

http://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm

ENCAPSULATION

Control the way data is accessed or modified

Makes the class easy to use for clients

Increase reusability

Encapsulation promotes maintenance

ENCAPSULATION

```groovy
@groovy.transform.CompileStatic
final class Money implements Serializable, Comparable<Money>, MoneyExchange, MoneyFormat {

    final BigDecimal amount
    final Currency currency

    // ...

}
```

ENCAPSULATION

```groovy
@groovy.transform.PackageScope
trait MoneyExchange {

        //...

    Money exchangeTo(Currency to, Exchange exchange = getCurrentExchange()) {

        //...
    }
}
```

```groovy
@groovy.transform.PackageScope
trait MoneyFormat {

    //...

    String format(Locale locale = Locale.default) {

        //...
    }
}
```

PRIMITIVE OBSESSION

" Primitive Obsession is using primitive data types to represent domain ideas. For example, use a String to represent a message or a Big Decimal to represent an amount of money,

TICKETBIS
ENGINEERING

http://c2.com/cgi/wiki?PrimitiveObsession

PRIMITIVE OBSESSION

Treatment:

Group primitive fields into their own class

Refactor -> Replace Data Value with Object.

TICKETBIS
ENGINEERING

https://sourcemaking.com/refactoring/smells/primitive-obsession

PRIMITIVE OBSESSION

Benefits:

- *Code becomes more flexible thanks to use of objects instead of primitives.*
- Better understandability and organization of code.
- *Easier finding of duplicate code.*

TICKETBIS
ENGINEERING

https://sourcemaking.com/refactoring/smells/primitive-obsession

PRIMITIVE OBSESSION

Before Money:

```
class Ticket implements Serializable {

    BigDecimal totalPrice
    Currency currency

    //...
}
```

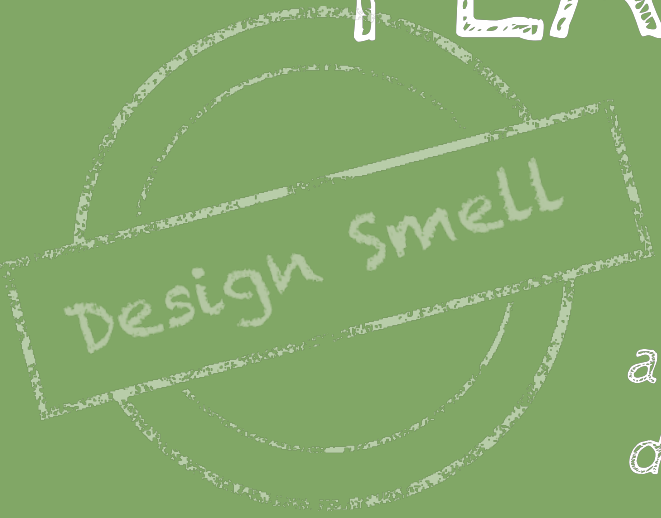
After Money:

```
class Ticket implements Serializable {

    Money totalPrice

    //...
}
```

FEATURE ENVY



" *A method accesses the data of another object more than its own data.*

A rule of thumb: *If things change at the same time, you should keep them in the same place.*

TICKETBIS
ENGINEERING

https://sourcemaking.com/refactoring/smells/feature-envy

FEATURE ENVY

Treatment:

Refactor -> Extract Method

Refactor -> Move Method

TICKETBIS
ENGINEERING

https://sourcemaking.com/refactoring/smells/feature-envy

FEATURE ENVY

Benefits

- Less code duplication
- Better code organization

TICKETBIS
ENGINEERING

https://sourcemaking.com/refactoring/smells/feature-envy

FEATURE ENVY

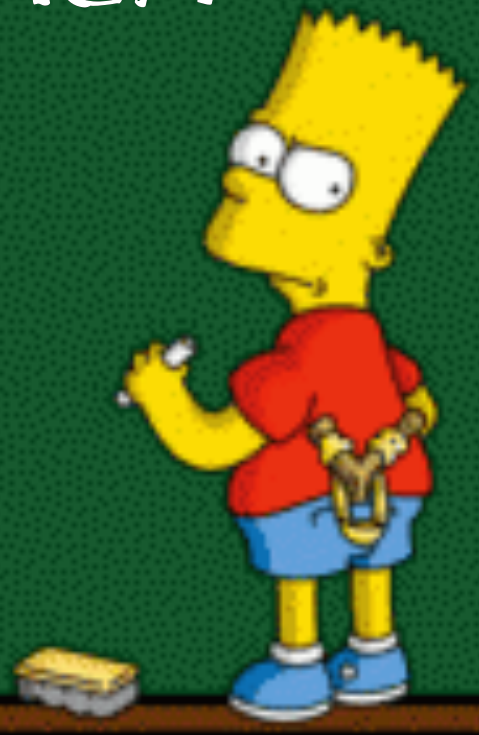
Before Money:

```
BigDecimal totalPrice = currencyService.convertCurrency( purchase.totalPrice,
                                purchase.currency, gateway.currency, round )
```

After Money:

```
Money totalPrice = purchase.totalPrice.exchangeTo( gateway.currency )
```

TECHNICAL APPROACH
TECHNICAL APPROACH
TECHNICAL APPROACH



TICKETBIS
ENGINEERING

CREATE A CONTAINER TO OUR MONEY OBJECT

```
@groovy.transform.CompileStatic
final class Money implements Serializable, Comparable<Money>, MoneyExchange, MoneyFormat {
    final BigDecimal amount
    final Currency currency

    private final static MathContext MONETARY_CONTEXT = MathContext.DECIMAL128

    final static class CurrencyMismatchException extends RuntimeException {
        CurrencyMismatchException(String aMessage) {
            super(aMessage)
        }
    }

        //...

    Money(Number amount, Currency currency) {
        this.amount = (BigDecimal) amount
        this.currency = currency
    }

        //...
}
```

INTRODUCE MONEY TO GORM

TICKETBIS
ENGINEERING

```
class MoneyUserType implements UserType, ParameterizedType {

    private final static String DEFAULT_CURRENCY_COLUMN = 'currency'
    private final static int[] SQL_TYPES = [Types.DECIMAL] as int[]

    Properties parameterValues

    //...

    Object nullSafeGet(ResultSet rs, String[] names, Object owner) {
        //...
    }

    void nullSafeSet(PreparedStatement st, Object value, int index) {
        //...
    }

    Class returnedClass() {
        Money.class
    }

    int[] sqlTypes() {
        SQL_TYPES
    }
}
```

USE MONEY TYPE IN DOMAIN OBJECTS

```
class Ticket implements Serializable {
    Money totalPrice

    //...

    static mapping = {
        totalPrice  type: MoneyUserType, params: [currencyColumn: 'divisa']
    }
}
```

ADD BEHAVIOUR TO OUR MONEY

```
interface Exchange {
    BigDecimal getRate(Currency from, Currency to)
}
```

```
trait MoneyExchange {

    //...

    Money exchangeTo(Currency to, Exchange exchange = getCurrentExchange()) {

        //...
    }
}
```

ADD BEHAVIOUR TO OUR MONEY

```
trait MoneyFormat {

    //...

    String format(Locale locale = Locale.default) {

        //...
    }

    //...
}
```

INTRODUCE MONEY TO GRAILS

TICKETBIS
ENGINEERING

```groovy
class StructuredMoneyEditor extends AbstractStructuredBindingEditor<Money> {

  private static final String currencyPlaceholder = '¤'

  Money getPropertyValue(Map values) {

      DecimalFormat formatter = getCustomDecimalFormatter(values)
      BigDecimal parsedAmount = getParsedAmount(formatter, (String) values.amount)

      new Money(parsedAmount, (String) values.currency)
  }

  //...
}
```

```
def doWithSpring = {

    //Custom structured property editor data binding for Money type
    moneyEditor com.ticketbis.money.StructuredMoneyEditor

}
```

```
class GreaterThanZeroConstraint extends AbstractConstraint {

    private static final String DEFAULT_INVALID_MESSAGE_CODE = 'default.gtZero.invalid'
    static final String CONSTRAINT_NAME = 'gtZero'

    private boolean gtZero

    //...

    protected void processValidate(Object target, Object propertyValue, Errors errors)
        if (!validate(propertyValue)) {
            def args = (Object[]) [constraintPropertyName,
                                   constraintOwningClass, propertyValue]

            rejectValue(target, errors,
                  DEFAULT_INVALID_MESSAGE_CODE, "not.${CONSTRAINT_NAME}", args)
        }
    }
    //...
}
```

```
def doWithSpring = {

    //...

    ConstrainedProperty.registerNewConstraint(
        GreaterThanZeroConstraint.CONSTRAINT_NAME,
        GreaterThanZeroConstraint)

    //...
}
```

```
class Ticket implements Serializable {

    Money totalPrice

    //...

    static constraints = {
        totalPrice( nullable: false, gtZero: true )

        //...
    }
```

```
class MoneyTagLib {
    static namespace = 'money'

    def inputField = { attrs ->
        def name = attrs.remove('name')
        def value = attrs.remove('value')

        //...
    }

    def format = { attrs ->
        Money value = new Money(attrs.value)

        //...
    }
}
```

```
<money:inputField name="totalPrice" value="123.45" currency="EUR"/>

<money:inputField name="totalPrice" value="${ money }"/>

<money:format value="${ money }" pattern="¤ ##,##0.00"/>

<money:format value="${ money }" numberFormat="${ formatter }"/>
```

Summing up

- Cleaner code
- Less code duplication
- Easy to maintain
- Increase reusability
- Avoid Operations between different currencies
- Better Accuracy

TICKETBIS
ENGINEERING

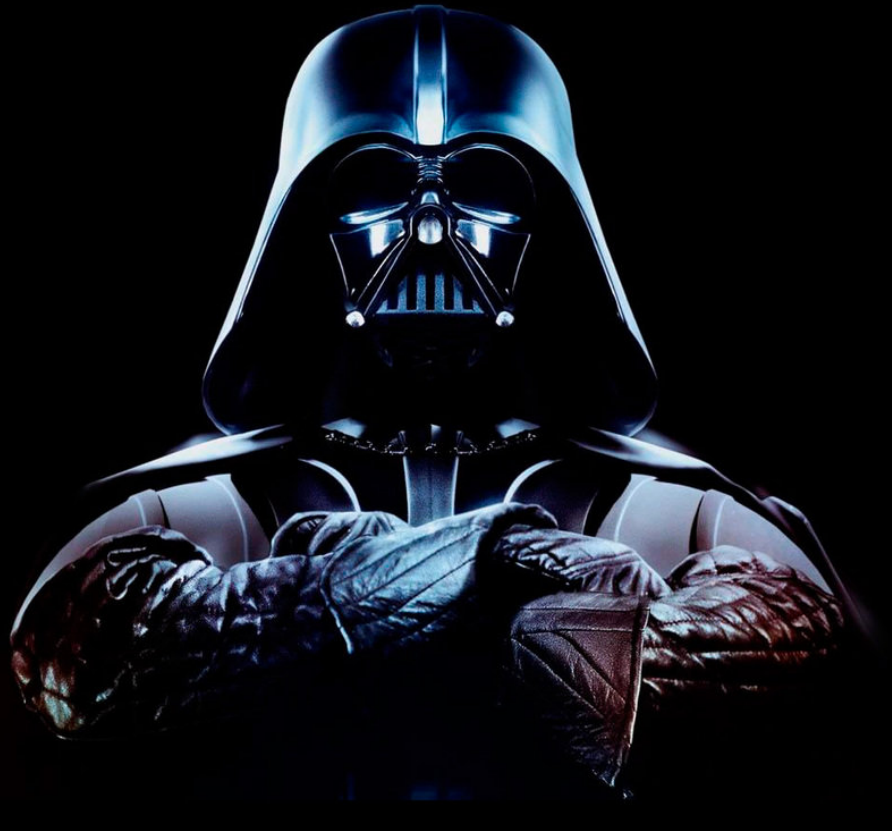
Source Code : https://github.com/ticketbis/grails-money

Slides : https://slides.com/xala3pa

itjobs@ticketbis.com

http://stackoverflow.com/jobs/companies/ticketbis

KEEP
CALM
AND
JOIN THE
DARK SIDE



Thanks!!