SOLID através de BDD

um guia prático para rubistas

Lucas Húngaro
software developer

SOLID

Conjunto de princípios desenvolvidos por Bob Martin que devem ser aplicados para melhorar a qualidade do código orientado a objetos.

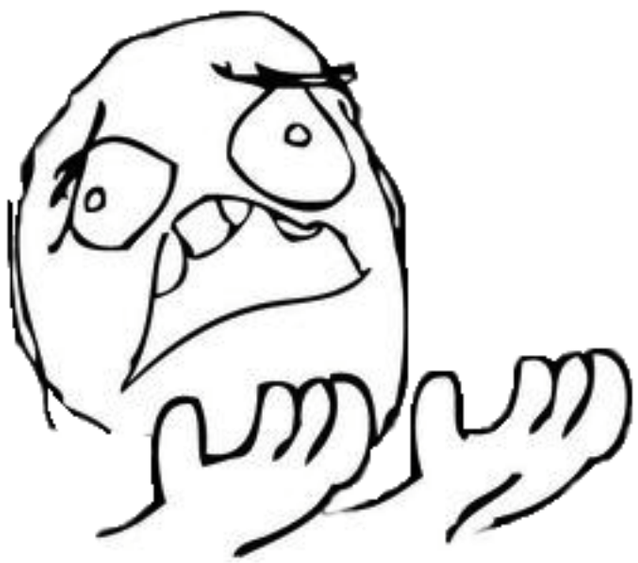
Rails, MVC

Rails, MVC e facilitam muito a vida do desenvolvedor, principalmente em aplicações simples. Mas também deixamos de prestar atenção à aspectos importantes enquanto produzimos código rapidamente.

Fat Models aka programação procedural

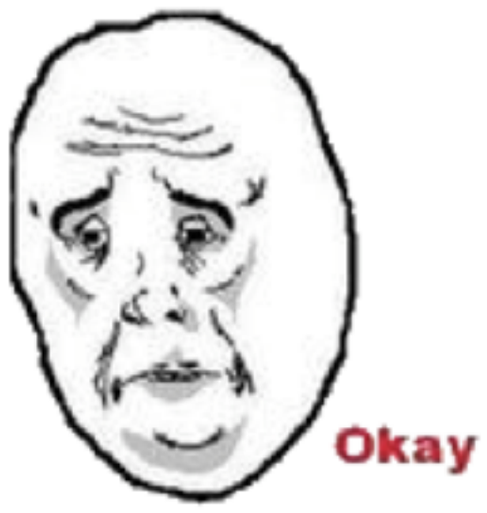
Assim acabamos desenvolvendo alguns anti-patterns. Pior do que isso é quando alguns desses anti-patterns são vistos como boas práticas mas, na verdade, são apenas atalhos enganosos (parecem vantajosos a princípio, mas acabam aumentando o custo de manutenção)

OOP, BDD, SOLID



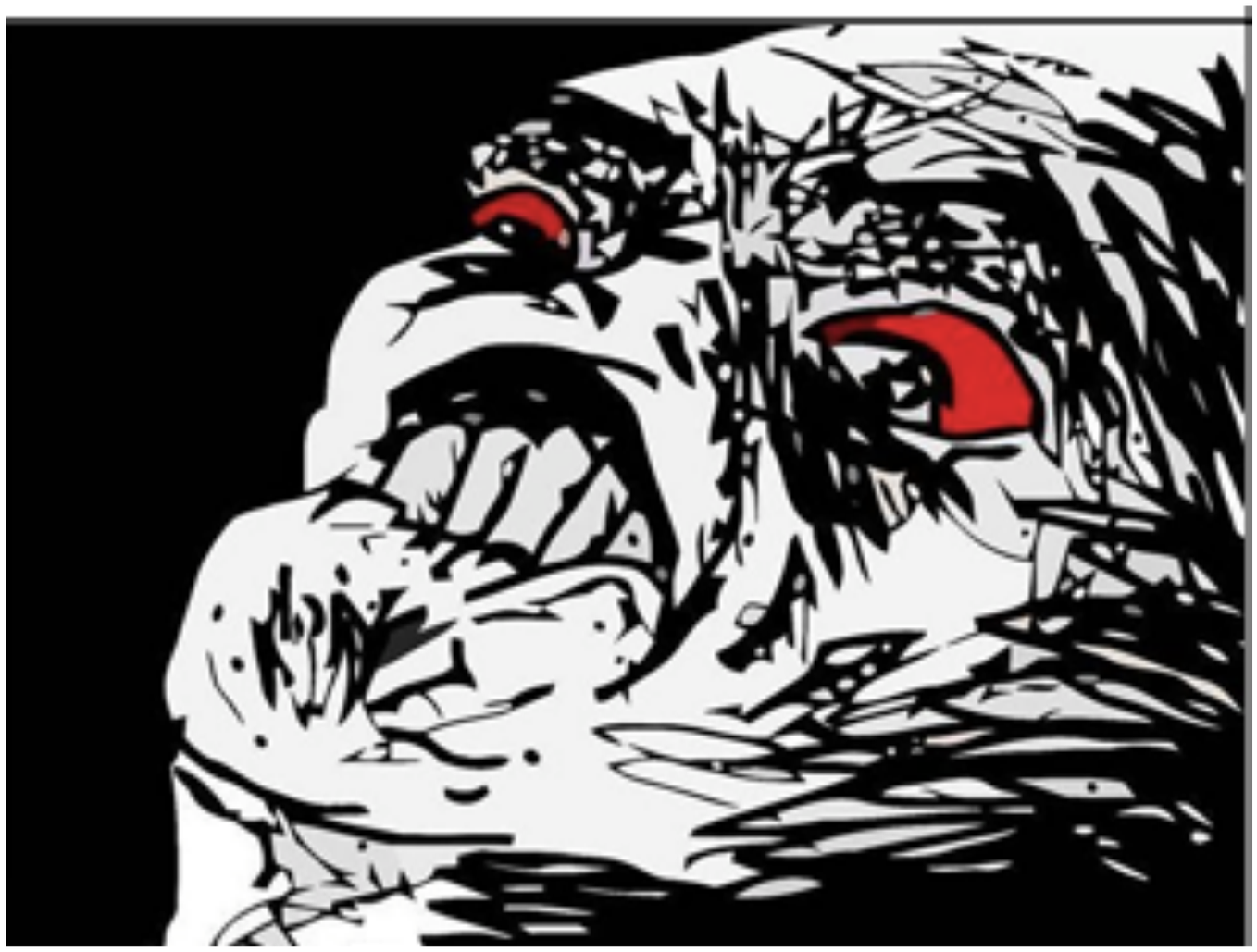
As soluções são simples mas, muitas vezes, acabam ficando de lado porque a maioria dos desenvolvedores pensam que são assuntos chatos.

Acadêmico...
(chato)



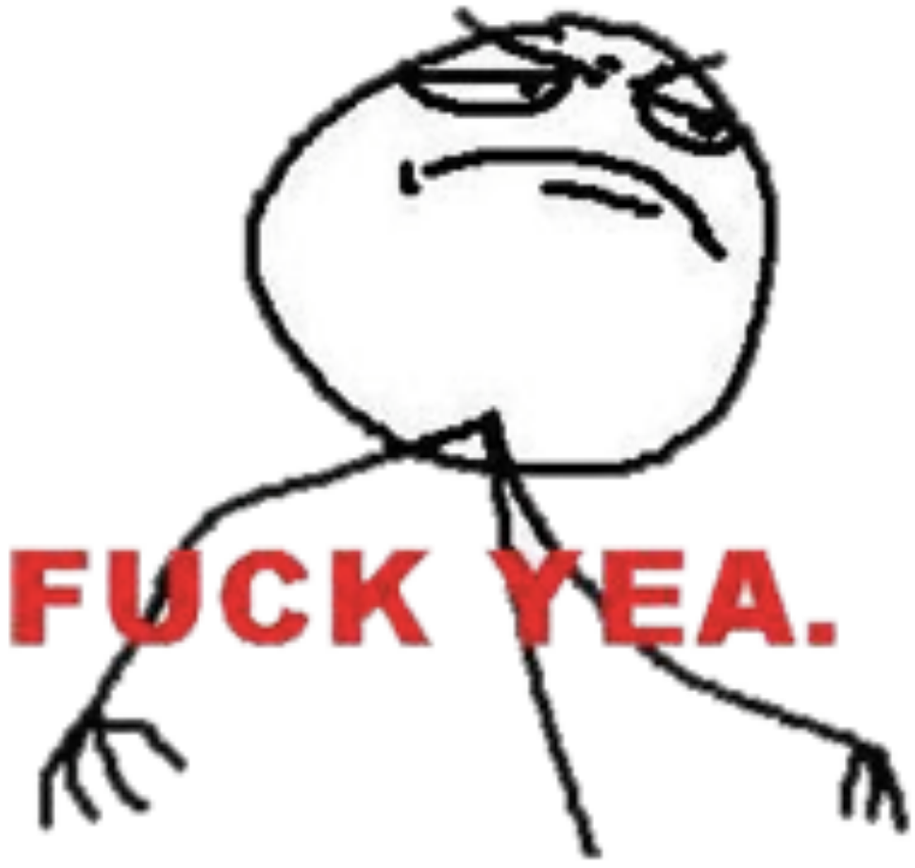
Um dos motivos para isso é a origem acadêmica, onde muitas vezes o assunto é discutido apenas na esfera teórica, com poucos exemplos de aplicação.

Coisa de Javeiro...



Outro problema é que muitos associam essas práticas à plataformas como Java e .NET que, infelizmente, remetem a sentimentos de excesso de burocracia e baixa produtividade.

Linguagem dinâmica



Mas, para nossa sorte, utilizar esses princípios em linguagens dinâmicas é muito mais fácil e menos burocrático.

SRP e DIP

Pra facilitar as coisas, vamos falar apenas de dois princípios: Single Responsibility e Dependency Inversion - os que acredito causarem o maior impacto

Coesão e Acoplamento

“Traduzindo” esses princípios, identificamos que eles falam sobre duas características do código: coesão e acoplamento.

Baixa coesão:

executa parte de uma responsabilidade ou mais de uma

Problemas que queremos eliminar

Alto acoplamento:

código preso a um caso de uso e suas dependências

Problemas que queremos eliminar

Code!

Exemplos

Processo usual:

rails new myapp

migrations

finders

data-centric apps

fat models

Começamos pelos dados (schema) e depois tentamos derivar algum comportamento disso, resultando em objetos “gordos”, pouco coesos e muito acoplados.

Rails apps

==

model User gigante

mais algumas

classes sem

importância

Domínios anêmicos

```ruby
class User < ActiveRecord::Base
  attr_accessible :email, :password,
                  :password_confirmation
  has_secure_password
  validates_presence_of :password, :on => :create

  def send_password_reset
    generate_token(:password_reset_token)
    self.password_reset_sent_at = Time.zone.now
    save!
    UserMailer.password_reset(self).deliver
  end

  def generate_token(column)
    begin
      self[column] = SecureRandom.urlsafe_base64
    end while User.exists?(column => self[column])
  end
end
```

Exemplo clássico (railscasts 275). Falta coesão, sobra acoplamento.

```ruby
class User < ActiveRecord::Base
  attr_accessible :email, :password,
                  :password_confirmation

  has_secure_password
  validates_presence_of :password, :on => :create

  def send_password_reset
    generate_token(:password_reset_token)
    self.password_reset_sent_at = Time.zone.now
    save!
    UserMailer.password_reset(self).deliver
  end

  def generate_token(column)
    begin
      self[column] = SecureRandom.urlsafe_base64
    end while User.exists?(column => self[column])
  end
end
```

Dependências

```ruby
class User < ActiveRecord::Base
  attr_accessible :email, :password,
                  :password_confirmation
  has_secure_password
  validates_presence_of :password, :on => :create

  def send_password_reset
    generate_token(:password_reset_token)
    self.password_reset_sent_at = Time.zone.now
    save!
    UserMailer.password_reset(self).deliver
  end

  def generate_token(column)
    begin
      self[column] = SecureRandom.urlsafe_base64
    end while User.exists?(column => self[column])
  end
end
```

Dependências

```ruby
class User < ActiveRecord::Base
  attr_accessible :email, :password,
                  :password_confirmation
  has_secure_password
  validates_presence_of :password, :on => :create

  def send_password_reset
    generate_token(:password_reset_token)
    self.password_reset_sent_at = Time.zone.now
    save!
    UserMailer.password_reset(self).deliver
  end

  def generate_token(column)
    begin
      self[column] = SecureRandom.urlsafe_base64
    end while User.exists?(column => self[column])
  end
end
```

Dependências

```ruby
describe User do
  describe "#send_password_reset" do
    let(:user) { Factory(:user) }

    it "generates a unique password_reset_token each time" do
      user.send_password_reset
      last_token = user.password_reset_token
      user.send_password_reset
      user.password_reset_token.should_not eq(last_token)
    end

    it "saves the time the password reset was sent" do
      user.send_password_reset
      user.reload.password_reset_sent_at.should be_present
    end

    it "delivers email to user" do
      user.send_password_reset
      last_email.to.should include(user.email)
    end
  end
end
```

Estrutural

```ruby
describe User do
  describe "#send_password_reset" do
    let(:user) { Factory(:user) }

    it "generates a unique password_reset_token each time" do
      user.send_password_reset
      last_token = user.password_reset_token
      user.send_password_reset
      user.password_reset_token.should_not eq(last_token)
    end

    it "saves the time the password reset was sent" do
      user.send_password_reset
      user.reload.password_reset_sent_at.should be_present
    end

    it "delivers email to user" do
      user.send_password_reset
      last_email.to.should include(user.email)
    end
  end
end
```

Estrutural

```ruby
describe User do
  describe "#send_password_reset" do
    let(:user) { Factory(:user) }

    it "generates a unique password_reset_token each time" do
      user.send_password_reset
      last_token = user.password_reset_token
      user.send_password_reset
      user.password_reset_token.should_not eq(last_token)
    end

    it "saves the time the password reset was sent" do
      user.send_password_reset
      user.reload.password_reset_sent_at.should be_present
    end

    it "delivers email to user" do
      user.send_password_reset
      last_email.to.should include(user.email)
    end
  end
end
```

Dependente de banco de dados == lento

```ruby
describe User do
  describe "#send_password_reset" do
    let(:user) { Factory(:user) }

    it "generates a unique password_reset_token each time" do
      user.send_password_reset
      last_token = user.password_reset_token
      user.send_password_reset
      user.password_reset_token.should_not eq(last_token)
    end

    it "saves the time the password reset was sent" do
      user.send_password_reset
      user.reload.password_reset_sent_at.should be_present
    end

    it "delivers email to user" do
      user.send_password_reset
      last_email.to.should include(user.email)
    end
  end
end
```

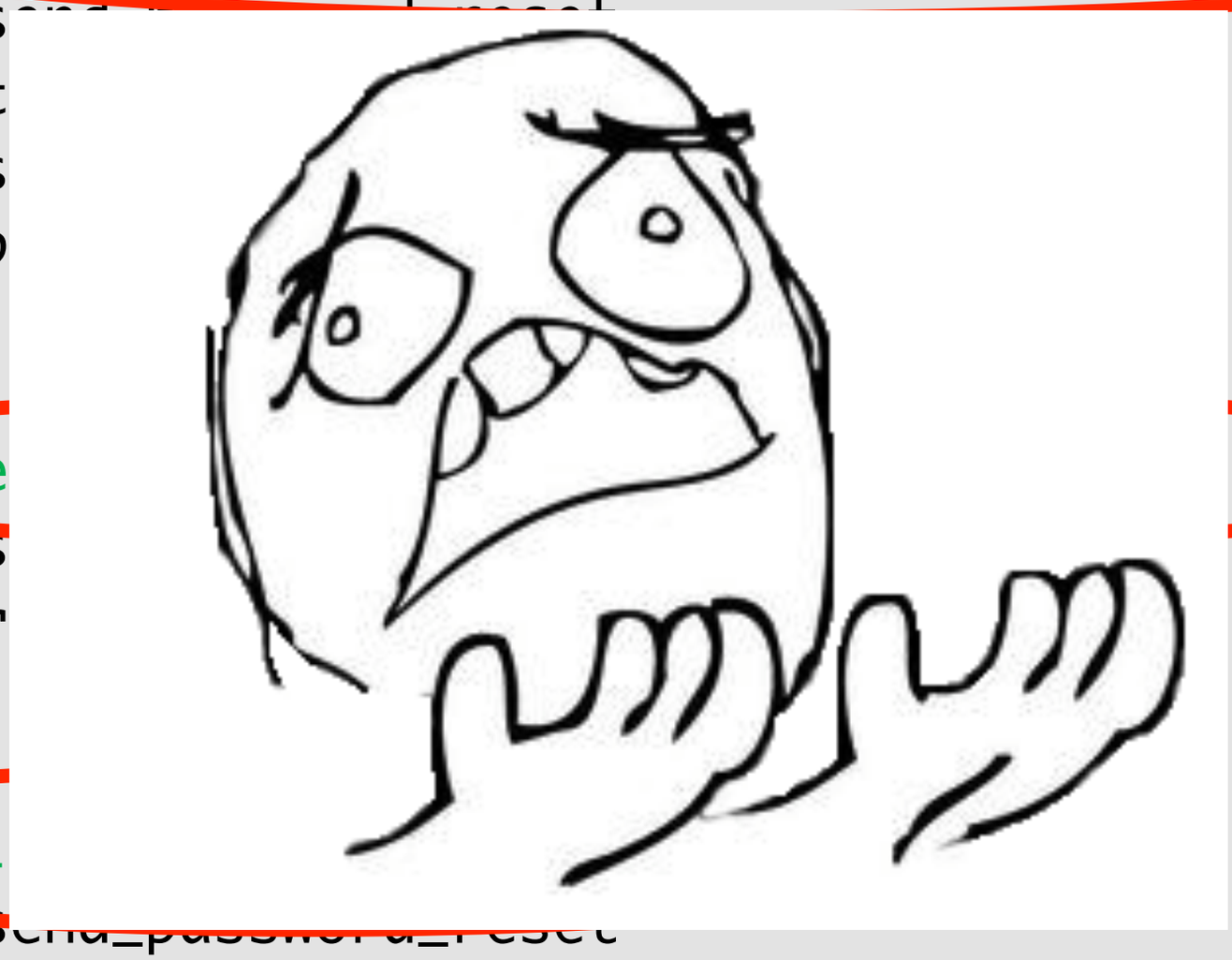
Muitas responsabilidades (isso porque o “has_secure_password” já esconde outras)

```ruby
describe User do
  describe "#send_password_reset" do
    let(:user) { Factory(:user) }

    it "generates a unique password_reset_token each time" do
      user.s                                    reset
      last_t
      user.s
      user.p                                                st_token)
    end

    it "save                                                nt" do
      user.s
      user.r                                                be_present
    end

    it "deli
      user.send_password_reset
      last_email.to.should include(user.email)
    end
  end
end
```



Specs lentas, código “rígido” (difícil de modificar), custo de manutenção alto.

Escrever specs isoladas para AR é difícil

ActiveRecord

==

Repositório (classe)
Model (objeto)

BD (reflection)

AR faz coisas demais e quebra o SRP por design

Uncle Bob: "ActiveRecord is a Data Structure"

http://goo.gl/OEBvX

AR deve ser usado como estrutura de dados para não se tornar um "buraco negro" de comportamento. Minha guideline: AR contém apenas validação, associações e finders, nada mais.

validações
associações
finders
nada mais

AR deve ser usado como estrutura de dados para não se tornar um “buraco negro” de comportamento. Minha guideline: AR contém apenas validação, associações e finders, nada mais.

Comece pelo **comportamento**

Vamos fazer diferente. Primeiro, identificamos o comportamento desejado.

```ruby
describe UserAuthentication do
  let(:user) { Factory(:user) }
  let(:password) { "123456" }

  context "with valid credentials" do
    subject { UserAuthentication.new(user.username, password) }

    it "allows access" do
      subject.authenticate.should be_true
    end
  end

  context "with invalid credentials" do
    subject { UserAuthentication.new(user.username, "invalid") }

    it "denies access" do
      subject.authenticate.should be_false
    end
  end
end
```

Isso é o que eu quero que faça

```ruby
class UserAuthentication
  def initialize(username, password)
    @username = username
    @password = password
  end

  def authenticate
    if user = User.find_by_username(@username)
      user.password_hash ==
BCrypt::Engine.hash_secret(@password, user.password_salt)
    else
      false
    end
  end
end
```

Uma primeira implementação

```ruby
class UserAuthentication
  def initialize(username, password)
    @username = username
    @password = password
  end

  def authenticate
    if user = User.find_by_username(@username)
      user.password_hash ==
BCrypt::Engine.hash_secret(@password, user.password_salt)
    else
      false
    end
  end
end
```

Melhoramos quanto à coesão... mas ainda há acoplamento

"The negative effects on testability in the Active Record pattern can be minimized by using mocking or dependency injection"

Wikipedia

object doubles são essenciais

YAY for mocks, stubs, spies, proxies and friends!

BDD == design

Eu costumava ser da turma do “mocks e stubs são apenas para isolamento de sistemas externos (como gateways) pq geram testes quebradiços quando usados internamente”

Testes quebradiços

==

Design ruim

Até que tomei vergonha na cara e fui estudar essa bagaça! ;)

object doubles
+
dependency injection

Através disso conseguimos: testes rápidos, objetos desacoplados e coesos, custo de manutenção reduzido.

```ruby
class UserAuthentication
  def initialize(username, password)
    @username = username
    @password = password
  end

  def authenticate(user_repo = User,
                   encryption_engine = BCrypt::Engine)
    if user = user_repo.find_by_username(@username)
      user.password_hash == encryption_engine.
                              hash_secret(@password,
                                          user.password_salt)
    else
      false
    end
  end
end
```

Como estamos aprendendo, vou inverter as coisas e mostrar a implementação primeiro: uma forma de utilizar o DIP é passar as dependências como parâmetros.

```ruby
class UserAuthentication
  def initialize(username, password)
    @username = username
    @password = password
  end

  def authenticate(user_repo = User,
                   encryption_engine = BCrypt::Engine)
    if user = user_repo.find_by_username(@username)
      user.password_hash == encryption_engine.
                              hash_secret(@password,
                                          user.password_salt)
    else
      false
    end
  end
end
```

```
describe UserAuthentication do
  let(:user) { double("an user").as_null_object }
  let(:user_repo) { double("an user repository") }
  let(:encryption_engine) { double("an encryption engine") }

  before(:each) do
    user.stub(:password_hash).and_return "the hash"
    user_repo.stub(:find_by_username).and_return user
  end
```

```
describe UserAuthentication do
  let(:user) { double("an user").as_null_object }
  let(:user_repo) { double("an user repository") }
  let(:encryption_engine) { double("an encryption engine") }

  before(:each) do
    user.stub(:password_hash).and_return "the hash"
    user_repo.stub(:find_by_username).and_return user
  end
```

São objetos dublês, não me importo se stub ou mock (ou outro tipo). Essas decisões tomo sobre mensagens (métodos), não sobre o objeto todo. Isso é uma feature muito legal do framework de mocks do RSpec

```ruby
describe UserAuthentication do
  let(:user) { double("an user").as_null_object }
  let(:user_repo) { double("an user repository") }
  let(:encryption_engine) { double("an encryption engine") }

  before(:each) do
    user.stub(:password_hash).and_return "the hash"
    user_repo.stub(:find_by_username).and_return user
  end
```

```
  context "with valid credentials" do
    before(:each) { encryption_engine.
                              stub(:hash_secret).
                               and_return user.password_hash }

    subject { UserAuthentication.new("username", "123456") }

    it "allows access" do
      subject.authenticate(user_repo,
                           encryption_engine).should be_true
    end
  end
```

E agora as specs: sem banco de dados, dependências injetáveis, código simples. No fim das contas, tudo o que preciso é de três objetos que respondam a uma interface bem definida.

```
context "with valid credentials" do
  before(:each) { encryption_engine.
                          stub(:hash_secret).
                           and_return user.password_hash }

  subject { UserAuthentication.new("username", "123456") }

  it "allows access" do
    subject.authenticate(user_repo,
                         encryption_engine).should be_true
  end
end
```

E agora as specs: sem banco de dados, dependências injetáveis, código simples. No fim das contas, tudo o que preciso é de três objetos que respondam a uma interface bem definida.

```
  context "with invalid credentials" do
    before(:each) { encryption_engine.
                                stub(:hash_secret).
                                and_return "another hash" }

    subject { UserAuthentication.new("username", "invalid") }

    it "denies access" do
      subject.authenticate(user_repo,
                           encryption_engine).should be_false
    end
  end
```

```
context "with invalid credentials" do
  before(:each) { encryption_engine.
                                  stub(:hash_secret).
                                  and_return "another hash" }

  subject { UserAuthentication.new("username", "invalid") }

  it "denies access" do
    subject.authenticate(user_repo,
                         encryption_engine).should be_false
  end
end
```

```ruby
describe MyBCryptAdapter do
  context "encrypting text with a salt" do
    it "respects the lib protocol" do
      BCrypt::Engine.expects(:hash_secret)
      MyBCryptAdapter.encrypt("text", "salt")
    end
  end
end
```

```ruby
class MyBCryptAdapter
  def self.encrypt(plain_text, salt)
    BCrypt::Engine.hash_secret(plain_text, salt)
  end
end
```

Podemos abstrair ainda mais...

```ruby
describe MyBCryptAdapter do
  context "encrypting text with a salt" do
    it "respects the lib protocol" do
      BCrypt::Engine.expects(:hash_secret)
      MyBCryptAdapter.encrypt("text", "salt")
    end
  end
end
```

```ruby
class MyBCryptAdapter
  def self.encrypt(plain_text, salt)
    BCrypt::Engine.hash_secret(plain_text, salt)
  end
end
```

... e usar um mock para garantir a interface.

```
def authenticate(user_repo = User,
                 encryption_engine = MyBCryptAdapter)

  if user = user_repo.find_by_username(@username)
    user.password_hash == encryption_engine.
                            encrypt(@password,
                                    user.password_salt)
  else
    false
  end
end
```

Até onde?

Quanto de abstração vale a pena? Indireção em excesso é tão ruim quanto alto acoplamento. A dica é abstrair quando é um componente que mudará muito ou quando você não tem a mínima ideia disso, pois isso preserva seu “direito” de mudar caso seja necessário.

```ruby
def authenticate(encryption_engine = MyBCryptAdapter)
 if user = User.find_by_username(@username)
 ...
end
```

Por ex: caso eu tenha certeza de que meu repositório de usuários será sempre um modelo AR (e nunca precisarei buscá-los em um LDAP, por exemplo), podemos eliminar essa injeção e deixar acoplado.

Mudanças...

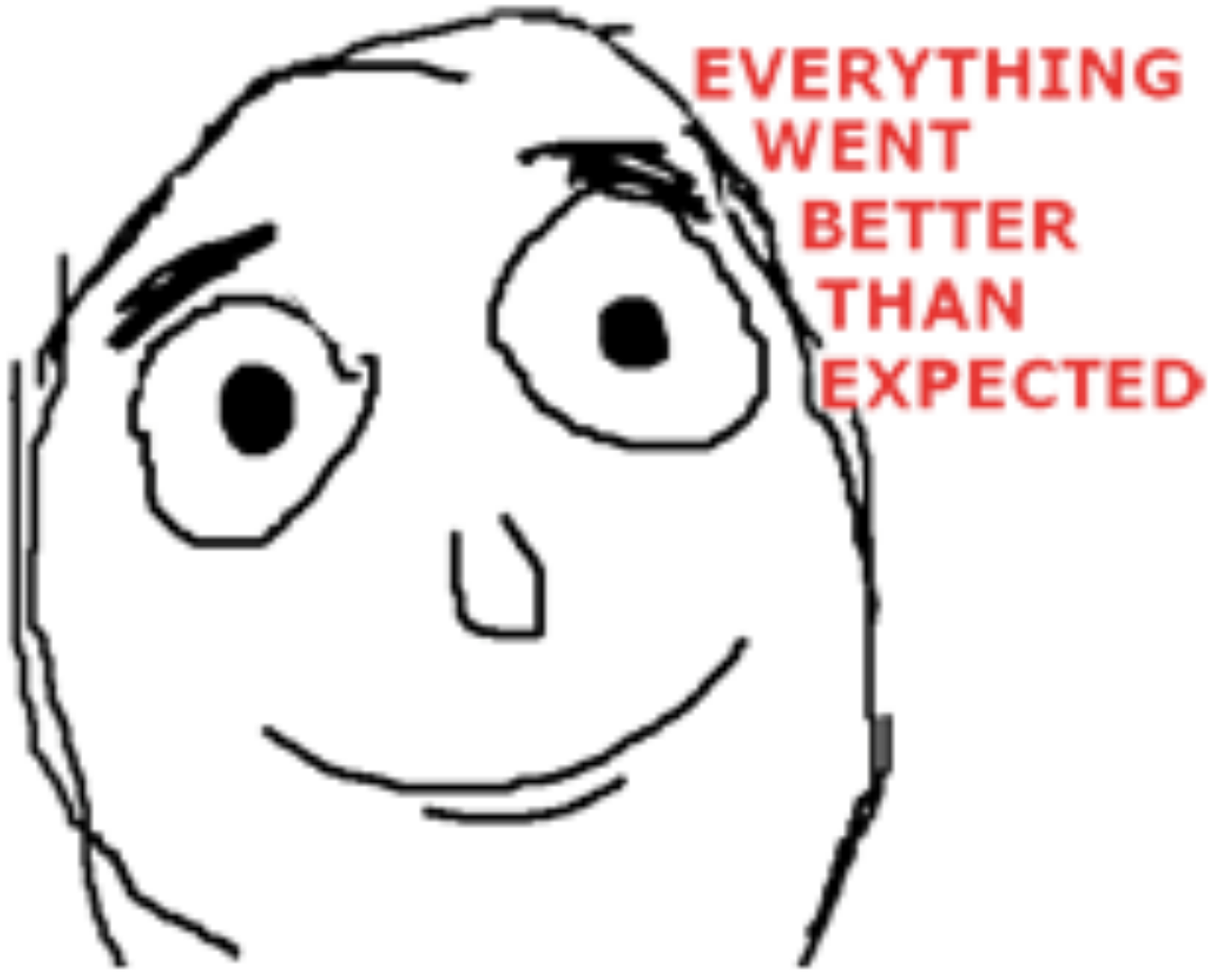
E se precisarmos mudar o algoritmo de criptografia?

```
describe MyZYCryptAdapter do
  context "encrypting text with a salt" do
    it "respects the lib protocol" do
      ZYCrypt::Engine::Passwords.
          should_receive(:hash_password_with_salt)
      MyZYCryptAdapter.encrypt("text", "salt")
    end
  end
end

class MyZYCryptAdapter
  def self.encrypt(plain_text, salt)
    ZYCrypt::Engine::Passwords.
        hash_password_with_salt(plain_text, salt)
  end
end
```

```
def authenticate(user_repo = User,
                 encryption_engine = MyZYCryptAdapter)

  if user = user_repo.find_by_username(@username)
    user.password_hash == encryption_engine.
                            encrypt(@password,
                                    user.password_salt)
  else
    false
  end
end
```



WIN! \o/

Mais uma alteração

Logging!

```
def authenticate(user_repo = User,
                 encryption_engine = MyBCryptAdapter
                 logger = MyFancyLogger)
  ...
end
```

Passando como parâmetro

```
context "with invalid credentials" do
  ...

  it "logs the authentication attempt" do
    my_logger = double("a logger")
    my_logger.should_receive(:log).with("something")

    subject.authenticate(user_repo, encryption_engine, logger)
  end
end
```

Um caso de uso meio forçado, apenas como exemplo. Vamos verificar as interações entre os componentes. O “como” fica à cargo do teste unitário do logger, não do autenticador.

```
context "with invalid credentials" do
  ...

  it "logs the authentication attempt" do
    my_logger = double("a logger")
    my_logger.should_receive(:log).with("something")

    subject.authenticate(user_repo, encryption_engine, logger)
  end
end
```

Um caso de uso meio forçado, apenas como exemplo. Vamos verificar as interações entre os componentes. O “como” fica à cargo do teste unitário do logger, não do autenticador.

```
context "with valid credentials" do
  ...

  it "doesn't log the authentication attempt" do
    my_logger = double("a logger")
    my_logger.should_receive(:log).never

    subject.authenticate(user_repo, encryption_engine, logger)
  end
end
```

Um caso de uso meio forçado, apenas como exemplo. Vamos verificar as interações entre os componentes. O "como" fica à cargo do teste unitário do logger, não do autenticador.

```
context "with valid credentials" do
  ...

  it "doesn't log the authentication attempt" do
    my_logger = double("a logger")
    my_logger.should_receive(:log).never

    subject.authenticate(user_repo, encryption_engine, logger)
  end
end
```

Um caso de uso meio forçado, apenas como exemplo. Vamos verificar as interações entre os componentes. O "como" fica à cargo do teste unitário do logger, não do autenticador.

Sintomas

BDD é uma ótima forma de apontar problemas com o design do código - os sintomas costumam ser claros

Muitos mocks/stubs numa mesma dependência ==

superfície de contato muito ampla

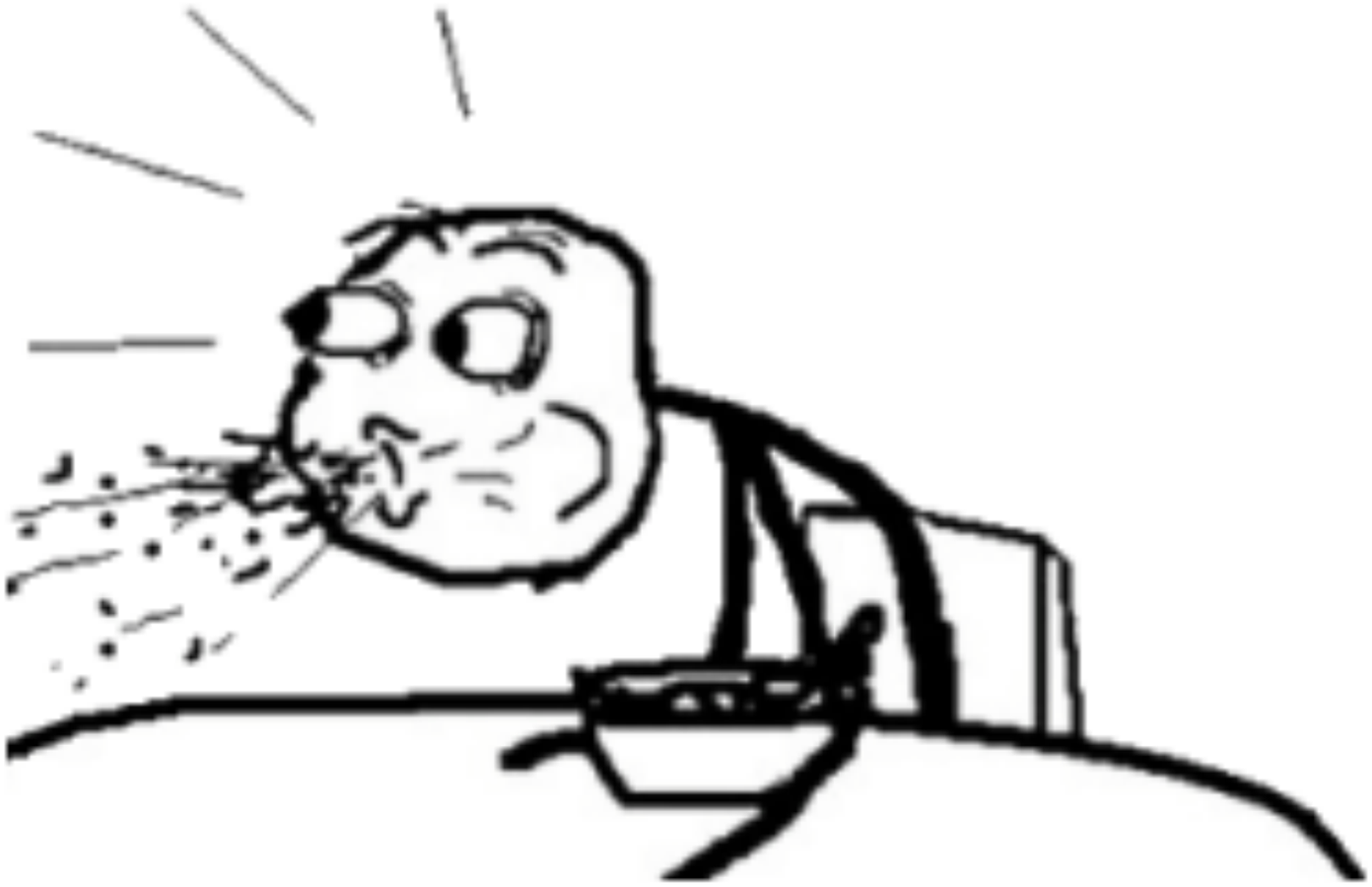
Essa é uma das principais reclamações dos desenvolvedores que são contra o uso pesado de dublês - criar muitos mocks/stubs nos setups dos cenários de testes. A questão é que isso, na verdade, está revelando problemas com o design.
Exemplo: teste de controller fazendo stub de vários métods de um model => falta de encapsulamento e model quebrando SRP

Superfície de contato

```ruby
class BlogController < ApplicationController
  def index
    @posts = Post.published.page params[:page]
    @posts_grouped_by_year = Post.
                                all.
                                group_by
                                 {|post| post.
                                          created_at.
                                          beginning_of_year }

    @posts_highlights = Post.published.featured.limit(3)
    @categories = Category.includes(:posts)

    @faqs = Faq.tagged_with(@posts.map {|post|
                                        post.tag_list}.
                                        join(",").
                                        split,
                                        :any => true)
  end
end
```



tha fuuuuck??!?!?!

```ruby
class BlogController < ApplicationController
  def index
    @posts = Post.published.page params[:page]
    @posts_grouped_by_year = Post.
                               all.
                               group_by
                                {|post| post.
                                         created_at.
                                         beginning_of_year }

    @posts_highlights = Post.published.featured.limit(3)
    @categories = Category.includes(:posts)

    @faqs = Faq.tagged_with(@posts.map {|post|
                                        post.tag_list}.
                                        join(",").
                                        split,
                                        :any => true)
  end
end
```

Controller: quanto mais “burro”, melhor

Minha sugestão para esse caso seria extrair toda essa “filtragem” para uma classe especializada em montar a página inicial do blog (poderia ser utilizado um Presenter).

Muitos contextos
em diferentes níveis
de abstração
==
muitas
responsabilidades

```
describe User do
  context "persistence logic" do
    it "validates ..."
  end

  context "data gathering" do
    it "finds records under certain conditions ..."
  end

  context "making payments" do
    it "register an error in case the key is invalid"

    it "writes some info to the log in case of success"

    # ...
  end
end
```

```
describe User do
  context "persistence logic" do
    it "validates ..."
  end

  context "data gathering" do
    it "finds records under certain conditions ..."
  end

  context "making payments" do
    it "register an error in case the key is invalid"

    it "writes some info to the log in case of success"

    # ...
  end
end
```

Isso nem sempre é “exato”, mas pode ser um ótimo indicativo de problemas.

Dicas

Algumas formas rápidas de verificar se seu código está “sólido” :D

Pequenas peças de comportamento facilmente acessíveis

Isso garante que não precisemos de setups muito elaborados e nos “força” a abstrair conceitos de negócio em forma de código.

app console

.buy(user, product, cart)

Dica: abra o console da sua aplicação e veja se os processos de negócio podem ser executados facilmente na linha de comando

@unclebobmartin
Uncle Bob Martin

e.g. OO is _NOT_ about "modeling the real world". I'd like to find the guy who started that rumor and have a serious public debate.

Modele processos, não se prenda apenas a entidades.

```ruby
class Customer
  def initialize(user, supplier)
    @user = user
    @supplier = supplier
  end

  def pay(amount, gateway = SomeGatewayAdapter)
    ...
  end
end
```

Exemplo: User assume o papel Customer para o processo de pagamento.

Don't mock types you don't own

Write wrappers

Evite ao máximo colocar dublês em tipos externos. Caso necessário, crie um wrapper/adapter

"Always check code in better than you checked it out."

— Uncle Bob

Seja um bom menino! :P

Crie o hábito de passar dependências como parâmetros

"Não escreva 'fat models'"

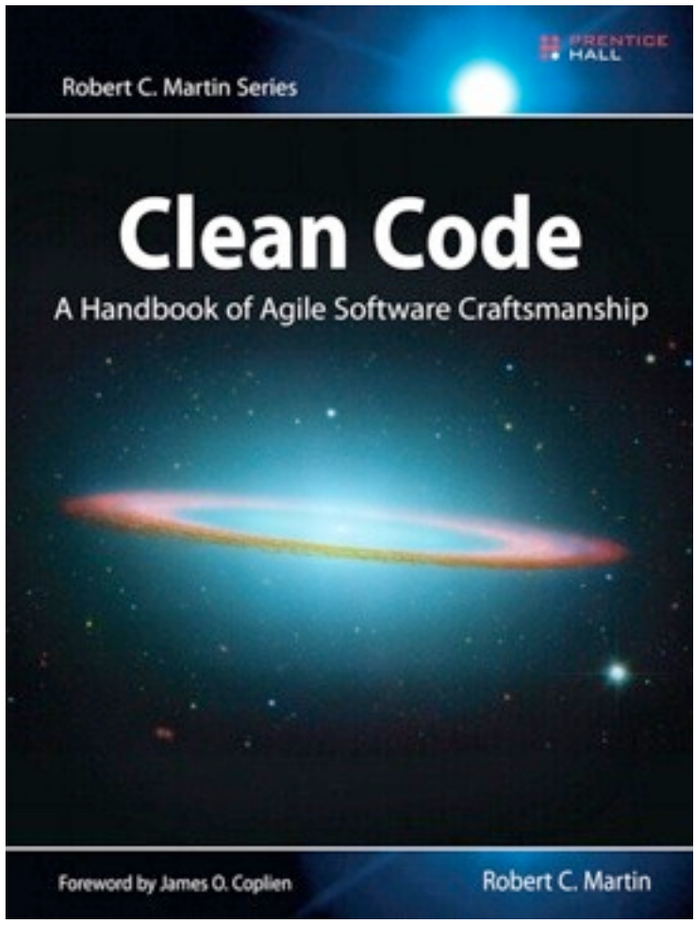
— OOP

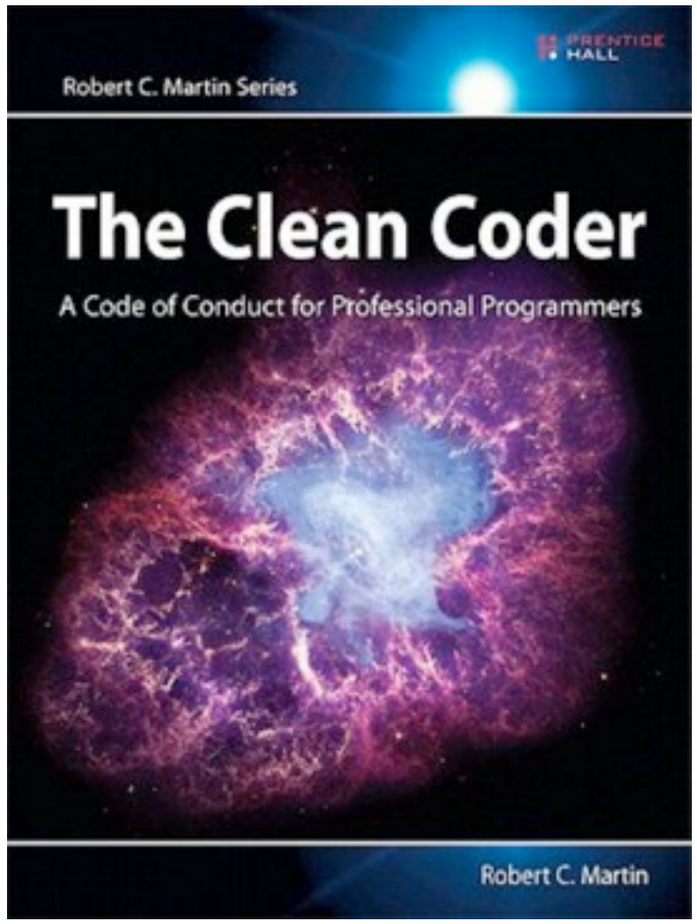
Próximos passos

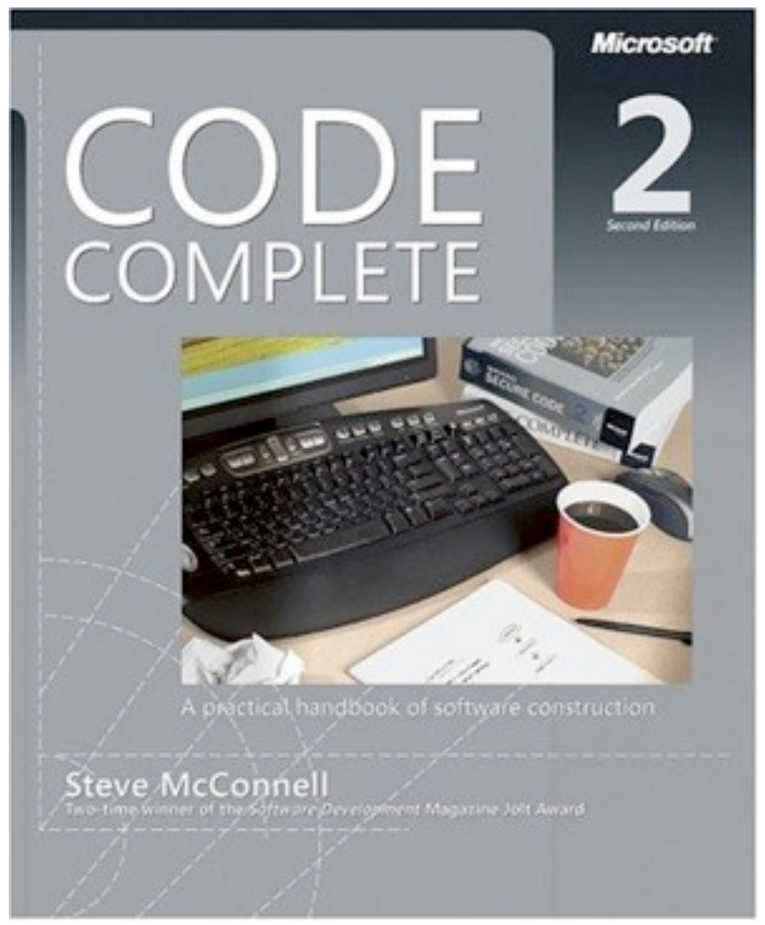
Não basta apenas alguns slides pra mostrar como fazer isso. É preciso ler, estudar e praticar. Por isso, seguem algumas referências.

Uncle Bob
Michael Feathers
Corey Haines
Gary Bernhardt
Pat Maddox
Avdi Grimm

Esses são apenas alguns nomes, que levarão a outros. Há muita gente altamente capacitada falando sobre e praticando essas técnicas.







Ótimas guidelines (!= regras)

Obrigado

@lucashungaro

github.com/lucashungaro

lucashungaro@gmail.com