

FIBONACCI SOLUÇÕES ÁGEIS

# Desenvolvimento de Aplicações Corporativas com Grails

---

*Treinamento avançado em desenvolvimento de sistemas*

Frederico Policarpo Martins

08/06/2011



Essa apostila é referente a um curso completo de desenvolvimento de software, com duração de 120hs. O objetivo do curso é tornar o aluno pronto para desenvolver aplicações corporativas completas, usando o framework de desenvolvimento rápido Grails.

## Conteúdo

Capítulo 01: Conceitos Básicos de Arquitetura de Software.....	5
1.1 - O que é Arquitetura de Software? .....	5
1.2 - O que são aplicações corporativas? .....	6
1.3 - Quais os tipos de aplicações corporativas? .....	7
1.4 - Soluções comprovadas para problemas comuns: Padrões.....	7
1.5 - O sistema para estacionamento: Estudo de Caso.....	7
1.6 - Tecnologia base do treinamento: Grails .....	8
1.7 - A estrutura de uma aplicação Grails .....	9
Prática 01 .....	10
1.8 - A linguagem Groovy .....	11
Prática 02 .....	11
1.9 - O IDE Spring Tool Suite.....	12
Prática 03 .....	12
Capítulo 02: Camada de Domínio.....	13
2.1 - Domain-Driven Design (DDD) .....	13
2.2 - Programação Orientada a Objetos (POO).....	13
2.2.1 - Classe e Objetos.....	13
Prática 04 .....	16
2.2.2 - Herança e Polimorfismo .....	17
Prática 05 .....	21
2.2.3 - Interfaces .....	21
Prática 06 .....	23
2.2.4 - Pensando em Papeis+Comportamentos+Colaborações na hora de definir seus objetos.....	23
Prática 07 .....	24
2.3 - Testes de Unidade.....	24
Prática 08 .....	26
2.4 - Cobertura de Testes.....	26
2.4.1 - Cobertura por Linha de Código x Cobertura por <i>Branch</i> .....	27
Prática 09 .....	28

2.5 - Desenvolvimento Guiado por Testes (TDD) .....	28
Prática 10 .....	29
Capítulo 03: Camada de Persistência .....	31
3.1 - Mapeamento Objeto Relacional .....	31
Prática 11 .....	32
3.1.1 - Mapeamento de Associações .....	32
3.1.1.1 - One-to-One.....	32
3.1.1.2 - One-to-Many .....	34
3.1.1.3 - Many-to-Many .....	35
3.1.2 - Mapeamento de Composições .....	35
3.1.3 - Mapeamento de Herança .....	36
3.2 - Realizando operações de banco de dados .....	37
3.3 - Customizando o Mapeamento.....	37
3.4 - Consultas via <i>Finders</i> Dinâmicos.....	38
3.5 - Consultas via HQL.....	38
Prática 12 .....	39
3.6 - Sessão e Transação .....	39
Prática 13.....	39
3.7 - Configurações de Banco.....	39
Prática 14 .....	41
3.8 - Testes de Integração.....	41
Prática 15 .....	42
Capítulo 04: Camada de Serviços .....	43
4.1 - O que são aspectos?.....	43
4.2 - Como criar meus serviços? .....	43
Prática 16 .....	44
Capítulo 05: Camada de Apresentação .....	45
5.1 - Model View Controller (MVC).....	45
5.2 - Controladores.....	47
5.3 - Grails Server Pages (GSP) .....	50
5.4 - Customizando nossa interface gráfica .....	50

5.4.1 – Questões de Usabilidade.....	50
5.4.2 – Redirecionamento de Páginas.....	50
Capítulo 06: Configurações do Grails.....	51
6.1 – BootStrap.groovy .....	51
6.2 – BuildConfig.groovy .....	51
6.3 – DataSources.groovy.....	51
6.4 – UrlMappings.groovy .....	51
6.5 – Fazendo o deploy de uma aplicação Grails.....	51
6.6 – Configurações de Idioma .....	51
6.7 – Customizando controles GSP com as taglibs.....	51
6.8 – Scripts de customização de Builds .....	51

# Capítulo 01: Conceitos Básicos de Arquitetura de Software

## 1.1 - O que é Arquitetura de Software?

No mundo dos desenvolvedores de sistemas temos variadas definições sobre o que vem a ser a tal '*Arquitetura de Software*'. Para o nosso estudo vamos adotar a seguinte definição:

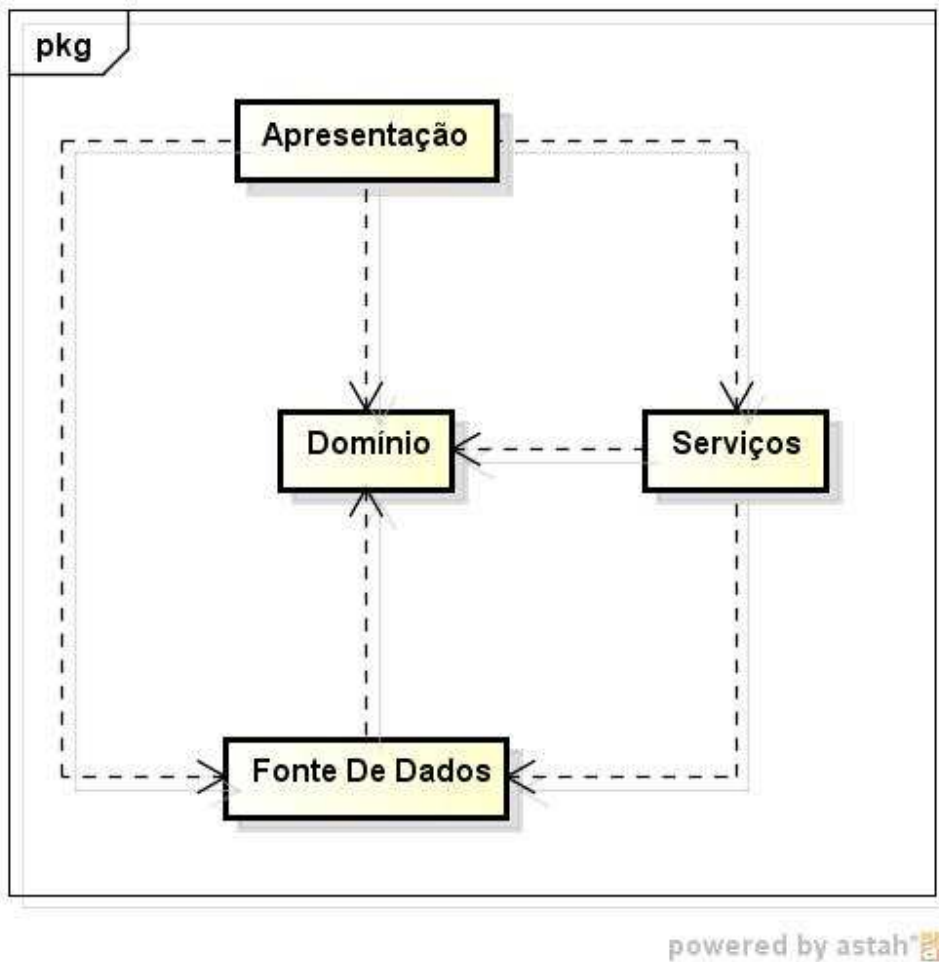
*Arquitetura de software é um conjunto de estratégias que define como um software é dividido em partes que possuem funcionamento bem definido e limitado e também as formas como essas partes se comunicam entre si.*

Pelo fato da arquitetura ser a base do sistema, suas partes maiores, alterações nela são extremamente caras de se realizar.

Neste curso iremos trabalhar com uma arquitetura com quatro camadas:

- **Apresentação** – Responsável por exibir dados ao usuário e também receber os comandos do mesmo.
- **Domínio** – É o centro desta arquitetura, onde se encontram as representações dos objetos para solução do problema, e que contém a implementação das regras de negócio extraídas das necessidades do cliente e dos requisitos da aplicação.
- **Fonte de Dados** – Essa camada é responsável pelo armazenamento não volátil dos dados.
- **Serviços** – Camada que adiciona ao sistema algumas lógicas de aplicação: Controle de *Log*, controle de transações, dentre outros.

O diagrama abaixo mostra como é a comunicação entre essas camadas. Observe que a camada de domínio está no centro, e que todas as outras camadas dependem dela, ao mesmo tempo em que ela não depende de ninguém. É fundamental compreender essa importância da camada de domínio já no começo do curso, pois todo o desenvolvimento dele será em torno dela.



## 1.2 - O que são aplicações corporativas?

Toda sequência de comandos que escrevemos em uma linguagem de programação, que passa por um compilador e que no final das contas é executada por um computador é um software (também chamado de aplicação). Todavia existem diversos tipos de software, de acordo com o uso e as características do mesmo, e para cada um temos, também, diferentes estratégias de desenvolvimento. Neste curso vamos estudar as estratégias de desenvolvimento para aplicações corporativas.

Vamos exemplificar um pouco alguns tipos de aplicações. Para construir um jogo é necessário um bom conhecimento de inteligência artificial. Um sistema crítico em tempo real precisa, acima de qualquer coisa, de ter uma resposta rápida e precisa em 100% dos casos, um software de uma aeronave, por exemplo. Cada tipo de software tem suas necessidades específicas, o jogo pode precisar ser executado em um hardware

simplificado, o sistema real deve responder de imediato, um compilador precisa ter boas mensagens de erro, e assim por diante.

Consideramos aplicações corporativas aquelas que manipulam um conjunto de dados muito grande, e também muito complexo. Além disso, essas aplicações possuem uma lógica de negócio extremamente complicada, não é raro ver um programador dizendo que o cliente está pedindo para ele fazer algo que não tem sentido algum. As aplicações corporativas são acessadas por muitos usuários ao mesmo tempo, o que implica cuidados peculiares quanto à concorrência e *performance*. Outro ponto importante é que nestas aplicações os requisitos mudam com muita frequência, se comparados aos requisitos de um jogo ou de um compilador, por exemplo.

As técnicas que serão ministradas neste curso são para resolver essas, e outras, questões sobre o desenvolvimento de aplicações corporativas.

### 1.3 - Quais os tipos de aplicações corporativas?

Mesmo após levantar algumas características comuns às aplicações corporativas, elas têm variâncias entre si. Um sistema feito para usuários comuns, pessoas que navegam pela web, por exemplo, tem uma exigência de interface gráfica maior do que um sistema que será usado por funcionários de uma empresa, onde estes poderão ter um treinamento para a utilização do mesmo. Sistemas com manipulações monetárias têm questões particulares e mais delicadas, do que um sistema de inscrição em eventos esportivos, por exemplo. Cada um dessas variações é melhor atendida por estratégias de arquitetura também variadas.

No nosso curso iremos considerar uma dessas variações, porém teremos o cuidado de exemplificar cenários alternativos e as variadas opções para cada caso.

### 1.4 - Soluções comprovadas para problemas comuns: Padrões

Os padrões nada mais são do que soluções modelo, propostas por alguém, para a resolução de problemas recorrentes no desenvolvimento de software. O foco deste curso não é a apresentação de padrões. Não teremos uma lista de padrões a serem exercitados, teremos uma sequência de passos, para o desenvolvimento do estudo de caso, e à medida que esses passos forem mostrando suas problemáticas, os padrões coerentes serão trabalhados. Para cada problema adotaremos um padrão de solução e citaremos as outras possibilidades.

### 1.5 - O sistema para estacionamento: Estudo de Caso

Um dono de estacionamento deseja utilizar um sistema para executar algumas regras de cálculo, em relação ao valor que tem que ser pago por um período dentro do estacionamento.

Ele irá criar três portões de entrada em seus estacionamentos. No menos deles apenas terão acesso motocicletas, no intermediário, terão acesso apenas carros de passeio e no portão maior poderão passar as caminhonetes. Para cada um deste tipo de veículo haverá uma forma específica de se calcular o valor do estacionamento.

- Sempre que um veículo entrar no estacionamento, sua entrada será registrada com a data e hora, placa do veículo e categoria do mesmo, de acordo com o portão de entrada;
- Esse registro será usado para o cálculo do valor, que será efetivado quando o motorista decidir pagar o estacionamento para poder retirar seu veículo;
- Um bilhete com essas informações será entregue ao motorista;
- Além destas informações o bilhete contém um código de identificação;
- O bilhete é usado para que ele possa efetuar o pagamento e então poder retirar seu veículo.
- O bilhete também pode ser usado para verificar o valor parcial da conta, com base em um portal de acompanhamento aberto aos clientes.
- O sistema será feito de forma gradual, com base nas prioridades dos requisitos, que serão dadas a cada prática presente nesta apostila;
- Deverá haver um tempo de tolerância no qual o motorista não precisa pagar para poder retirar seu veículo;
- O tempo de tolerância poderá variar de acordo com o tipo de veículo
- O sistema deverá dar suporte a períodos promocionais;
- Pode ser definido um valor de hora menor, para as primeiras horas de permanência de um veículo;
- Promoções poderão ser válidas para um ou mais tipos de veículos;
- Veículos podem permanecer no estacionamento o tempo que quiserem;
- Acima de 12 horas será cobrada uma diária;
- O tempo que ultrapassar 24 horas será contabilizado normalmente, até que o veículo inicie outra diária:
  - Ex 01: 26 horas = 1 diária + 2 horas
  - Ex 02: Das 37hs às 48hs = 2 diárias

Como foi avisado, esses requisitos serão detalhados e implementados gradativamente, à medida que o conteúdo do curso for progredindo. Essa lista de requisitos serve apenas para dar uma boa idéia geral do que se trata nosso aplicativo, não se prenda muito a cada item levantado, pense no sistema como um todo, pelo menos por enquanto.

## 1.6 - Tecnologia base do treinamento: Grails



Ao pé da letra Grails significa Groovy On Rails, isto é, um aplicativo que fornece a arquitetura Rails para a linguagem dinâmica Groovy (principal linguagem dinâmica para a plataforma Java).

O Grails (assim como outras plataformas Rails) foi construído para facilitar o desenvolvimento de sistemas de informação para a web. Através do uso de **Convenção sobre Configuração**, essa ferramenta diminui radicalmente o tempo que o programador gastaria para integrar e configurar todos os módulos do seu aplicativo web (banco de dados, regras de negócio, controladores de tela, testes, etc..). Para arrematar podemos dizer que com Grails o desenvolvedor pode se preocupar quase que exclusivamente apenas com a lógica de negócio do sistema que irá desenvolver, a lógica de aplicação fica por conta do framework.

*"Groovy é a principal linguagem de programação de código aberto para a Máquina Virtual Java, que oferece uma sintaxe flexível em Java, de modo que a maioria dos desenvolvedores Java pode aprender em questão de horas. Groovy fornece recursos vistos em outras linguagens dinâmicas como Ruby, Python e Smalltalk. Groovy realmente brilha em sua capacidade de definir facilmente novas Domain Specific Languages (DSLs) que podem ser usadas como uma camada de abstração que permite que especialistas no assunto, mesmo não sendo programadores, codifiquem regras de negócio.*

*A combinação de Groovy e Grails oferece benefícios de produtividade rivalizando com Ruby on Rails, mas na plataforma Java que já é comprovada a um tempo, escalável e integrada."* (Texto extraído do site [www.springsource.com](http://www.springsource.com), acessado em 13/05/2011, às 14h40min; Traduzido pelo Google Tradutor)

É importante dizer que, como Groovy é executado sobre a plataforma Java, podemos usar qualquer biblioteca ou classe Java que desejarmos em nossos códigos em Groovy, e vice-versa.

Além dos benefícios já citados vale a pena dizer que o Grails fornece uma interface de comandos bastante simples para realizar suas tarefas (compilar, executar, rodar testes, criar classes, etc.). Não obstante, essa plataforma fornece uma robusta estrutura de plugins, que fornece a possibilidade de adicionar à sua aplicação, funcionalidades feitas por terceiros de forma muito rápida.

## 1.7 – A estrutura de uma aplicação Grails

As aplicações feitas em Grails seguem uma arquitetura padrão, essa arquitetura é dividida nos seguintes módulos principais:

- Classes de Domínio
- Testes
- Controladores de Tela

- Telas em GSP (*Groovy Server Pages*)
- Configurações
- GORM

### Prática 01

*Abra uma janela de comandos e execute o seguinte comando para criar uma nova aplicação grails:*

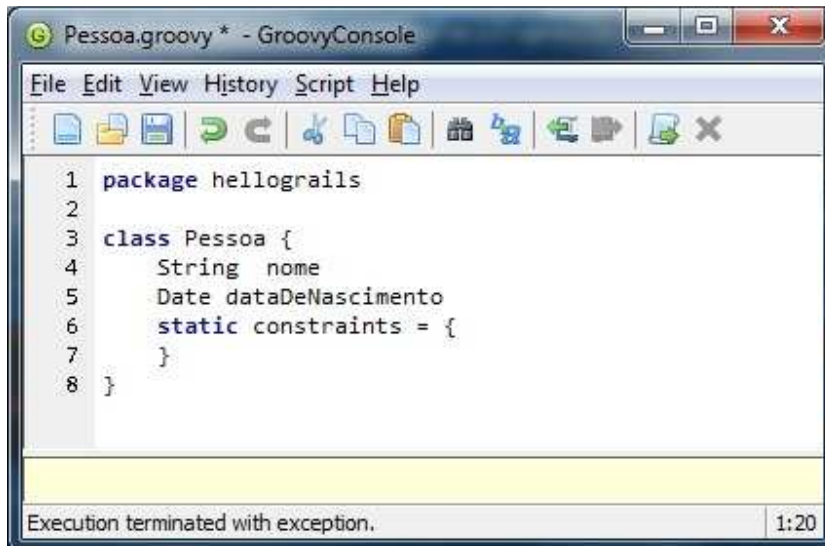
```
grails create-app HelloGrails
```

*Verifique a estrutura de diretórios e arquivos que foi gerada por esse comando. Não tenha preguiça, olhe e compreenda, ao menos razoavelmente, artefato por artefato.*

*Execute mais um comando para criar uma classe de domínio:*

```
grails create-domain-class Pessoa
```

*Abra o arquivo Pessoa.groovy, no diretório grails-app/domain/hellograils, com o aplicativo GroovyConsole, e adicione as propriedades dataDeNascimento e nome. O Arquivo ficará assim:*



*Para finalizar nosso primeiro contato execute o seguinte comando:*

```
grails generate-all hellograils.Pessoa
```

*Esse comando irá criar telas, controladores e testes para a classe de domínio Pessoa. Verifique novamente esses arquivos criados e posteriormente execute a aplicação para saborear o resultado:*

```
grails run-app
```

## 1.8 - A linguagem Groovy

A linguagem que iremos trabalhar tem uma sintaxe muito próxima à da linguagem Java, com a diferença que essa é uma linguagem dinamicamente tipada, enquanto Java é estaticamente tipada. Como contato introdutório a esta linguagem vamos dar uma rápida analisada no trecho de código abaixo:

```
1 class Teste {
2     String nome
3     List lista = [1,2,3,4,5]
4
5     void imprimaLista(){
6         print lista
7     }
8
9     int somaLista() {
10         def total = 0
11         lista.each{total += it}
12         total
13     }
14
15     static constraints = {
16         nome(blank:false)
17         lista(size:5..5)
18     }
19 }
20
21 // em outro lugar
22 def teste = new Teste(nome:'Primeiro Teste')
23 teste.save(flush:true)
24 List todosTestes = Teste.findAllByNameLike('Prim%')
25
26
```

Obs.: Os métodos `save` e `findAllByNameLike` são injetados pelo Grails, eles não irão funcionar no GroovyConsole.

### Prática 02

Abra uma janela de comando e execute o aplicativo GroovyConsole. Escreva códigos para experimentar os seguintes cenários:

- Identificar o maior elemento de uma lista (testar para lista de inteiros, floats e strings)
- Imprimir na tela todos os elementos da lista
- Imprimir na tela todos os elementos pares (para lista de tipos numéricos)

- *Converta uma string para inteiro*

## 1.9 - O IDE Spring Tool Suite

Existem diversos IDEs disponíveis no mercado para trabalharmos com essa plataforma, dentre eles o NetBeans, Spring Tool Suite e o IntelliJIDEA. Na minha opinião o IntelliJIDEA é o melhor dentre todos, mas como ele é pago nós vamos usar o Spring Tool Suite que é grátis e que também é um bom IDE. O Spring Tool Suite é uma customização do Eclipse, feito pela Spring Source, que já vem pronto para o trabalho com a plataforma Grails.

Os comandos do Grails são simples e completos o suficiente para que você possa trabalhar bem nele sem nenhum IDE, porém os ambientes de desenvolvimento acrescentam ferramentas que agilizam ainda mais nosso trabalho.

### Prática 03

*Execute os mesmos passos da PRÁTICA 01, porém ao invés de executar os comandos do Grails diretamente por linha de comando, execute-os via STS. E para editar o código fonte utilize também essa IDE, e não o GroovyConsole.*

## Capítulo 02: Camada de Domínio

### 2.1 - Domain-Driven Design (DDD)

DDD é uma estratégia de desenvolvimento de sistemas na qual toda lógica de negócio é codificada dentro da camada de domínio, a lógica de negócio também pode ser chamada de lógica de domínio. Essa lógica é dividida entre os diversos objetos do sistema, quem mantém dados, disponibilizam operações que utilizam esses dados e que se comunicam entre si (mais detalhes no próximo tópico).

Domain-Driven Design não é uma tecnologia, ou uma metodologia, é uma forma de pensar na solução do problema: "Através dos requisitos nós extraímos os objetos e seus comportamentos, e toda lógica deve estar inserido nestes objetos, que por sua vez compõe a camada de domínio".

#### DDD Vs. BO-LO-VO

Pessoas que trabalham com aplicações corporativas em Java estão acostumados com a estrutura BO-LO-VO:

BO - **B**ussines **O**bjects

LO - **L**ayer **O**bjects

VO - **V**alue **O**bjects

Nesta estrutura existe um objeto para conter os dados de uma dada entidade, os *Value Objects*, e outro que contém o processamento que aplica lógica de negócio a esses dados, os *Bussines Objects*. Desta forma temos dados separados de operações, isso não é orientado a objetos! Com o DDD nós juntamos tudo nos objetos de domínio!

O fato de tudo estar bem isolado torna o sistema mais fácil de compreender, mais simples de manter e bem mais fácil de testar também, o que é muito importante. Não se assuste se você ainda não consegue visualizar isso na prática, à medida que o curso for progredindo isso ficará mais claro.

### 2.2 - Programação Orientada a Objetos (POO)

Primeiramente vamos estudar algumas definições básicas para que possamos ter um código orientado a objetos, e posteriormente vamos analisar uma forma mais orgânica de se chegar a um resultado.

#### 2.2.1 - Classe e Objetos

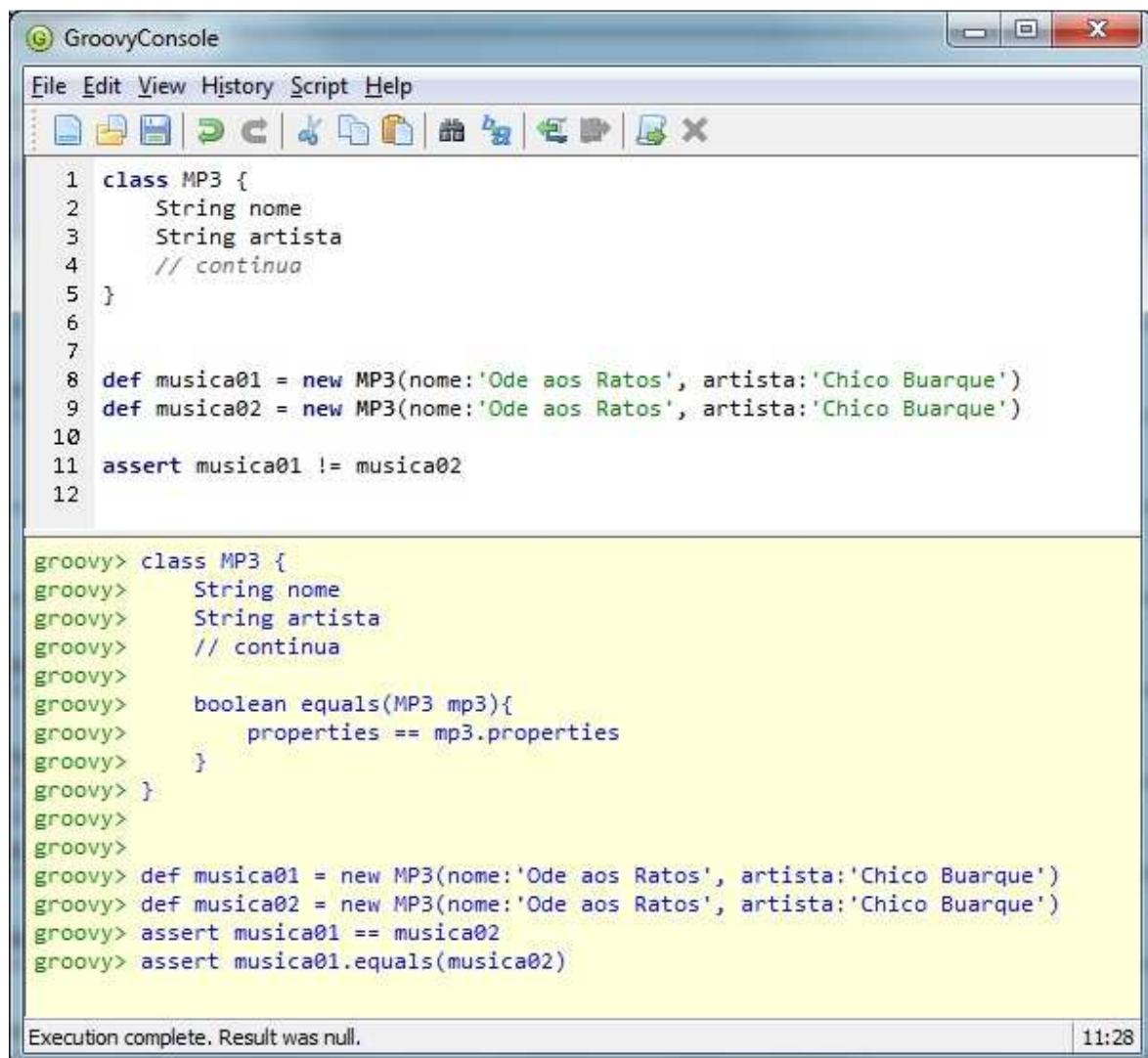
Começemos com a definição de objeto:

*Objeto é uma unidade de um software que contém dados relacionados a um conceito bem definido, e que contém operações que utilizam, alterando ou não, esses dados. Essas operações realizam as responsabilidades deste objeto.*

Exemplo: Um arquivo de texto é um objeto, seus dados são o conteúdo de seu texto, seu tamanho, sua codificação, etc. Suas responsabilidades são: adicionar um texto ao fim do arquivo, renomear-se, verificar se pode ser lido, verificar se pode ser alterado, dentre outras.

Uma observação importante é quanto à identificação dos objetos. Dois objetos, mesmo que contenham o mesmo conteúdo, são objetos distintos. Veja o trecho de código abaixo.

Primeiro criamos dois objetos idênticos e verificamos se são diferentes:

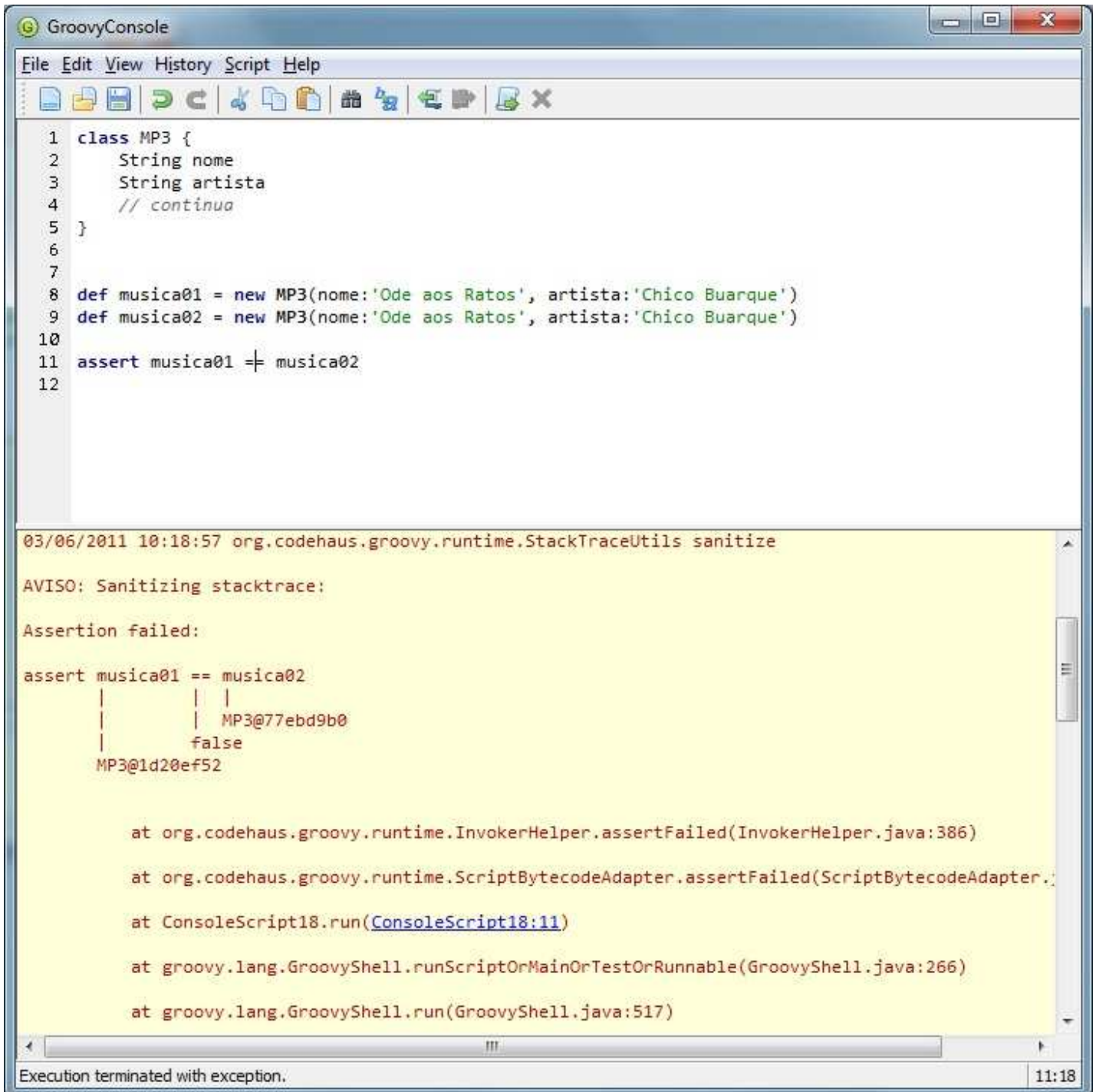


```
1 class MP3 {
2     String nome
3     String artista
4     // continua
5 }
6
7
8 def musica01 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
9 def musica02 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
10
11 assert musica01 != musica02
12
```

```
groovy> class MP3 {
groovy>     String nome
groovy>     String artista
groovy>     // continua
groovy>
groovy>     boolean equals(MP3 mp3){
groovy>         properties == mp3.properties
groovy>     }
groovy> }
groovy>
groovy>
groovy> def musica01 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
groovy> def musica02 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
groovy> assert musica01 == musica02
groovy> assert musica01.equals(musica02)
```

Execution complete. Result was null. 11:28

Agora pegamos esses mesmo objetos idênticos e verificamos se são iguais:



The screenshot shows the GroovyConsole application window. The script editor contains the following code:

```
1 class MP3 {
2     String nome
3     String artista
4     // continua
5 }
6
7
8 def musica01 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
9 def musica02 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
10
11 assert musica01 == musica02
12
```

The output pane shows the following messages:

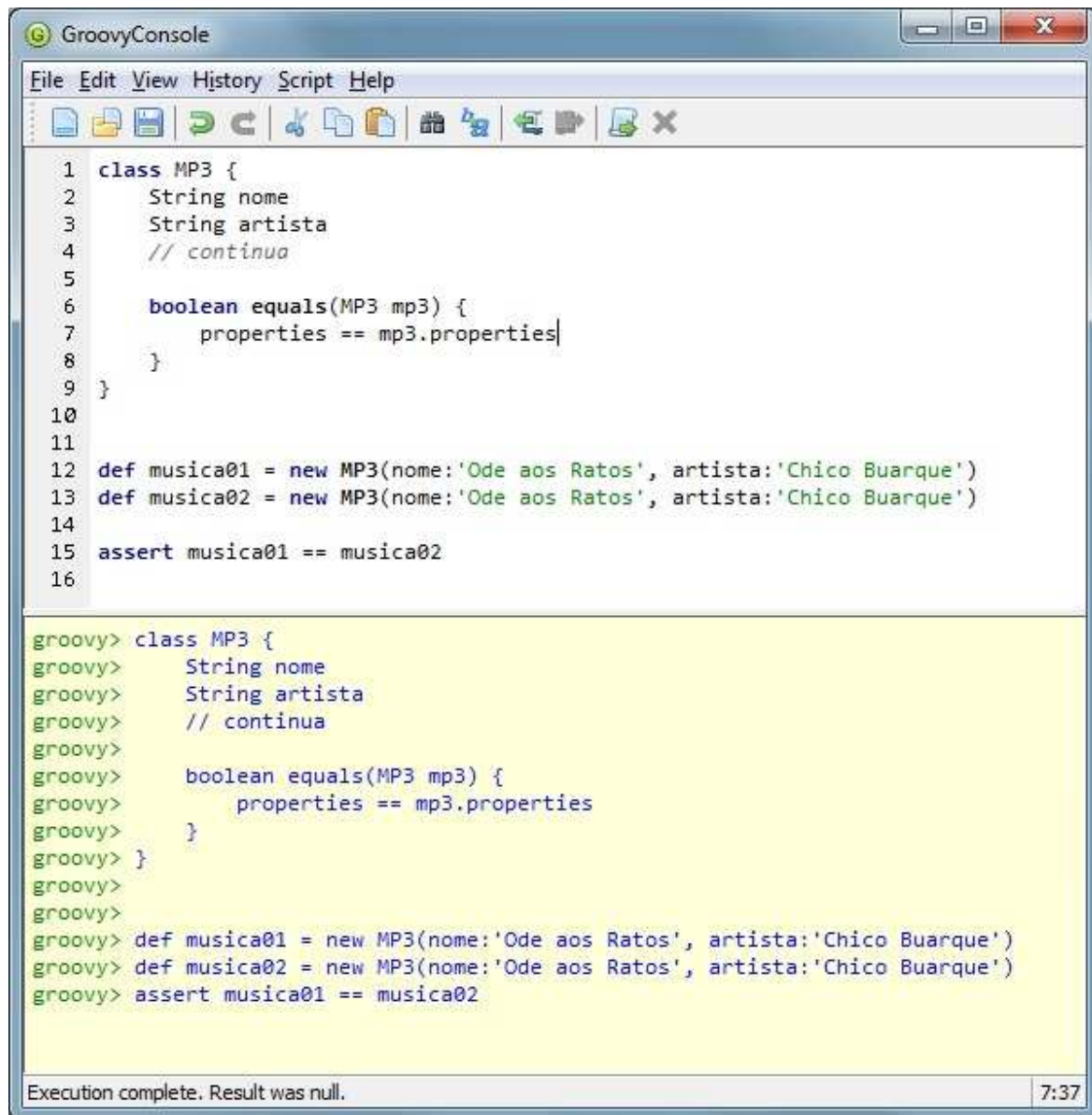
```
03/06/2011 10:18:57 org.codehaus.groovy.runtime.StackTraceUtils sanitize
AVISO: Sanitizing stacktrace:
Assertion failed:
assert musica01 == musica02
    |             |             |
    |             |             MP3@77ebd9b0
    |             false
    MP3@1d20ef52

    at org.codehaus.groovy.runtime.InvokerHelper.assertFailed(InvokerHelper.java:386)
    at org.codehaus.groovy.runtime.ScriptBytecodeAdapter.assertFailed(ScriptBytecodeAdapter.:
    at ConsoleScript18.run(ConsoleScript18:11)
    at groovy.lang.GroovyShell.runScriptOrMainOrTestOrRunnable(GroovyShell.java:266)
    at groovy.lang.GroovyShell.run(GroovyShell.java:517)
```

Execution terminated with exception. 11:18



Porém, se de acordo com suas regras de negócio dois objetos idênticos devam ser considerados iguais, você pode fazer isso alterando o método de comparação:



The screenshot shows a window titled "GroovyConsole" with a menu bar (File, Edit, View, History, Script, Help) and a toolbar. The main text area contains Groovy code for a class and object creation. Below the code is a yellow-highlighted area showing the same code being executed in a REPL. The status bar at the bottom indicates "Execution complete. Result was null." and the time "7:37".

```
1 class MP3 {
2     String nome
3     String artista
4     // continua
5
6     boolean equals(MP3 mp3) {
7         properties == mp3.properties
8     }
9 }
10
11
12 def musica01 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
13 def musica02 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
14
15 assert musica01 == musica02
16
```

```
groovy> class MP3 {
groovy>     String nome
groovy>     String artista
groovy>     // continua
groovy>
groovy>     boolean equals(MP3 mp3) {
groovy>         properties == mp3.properties
groovy>     }
groovy> }
groovy>
groovy>
groovy> def musica01 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
groovy> def musica02 = new MP3(nome:'Ode aos Ratos', artista:'Chico Buarque')
groovy> assert musica01 == musica02
```

Execution complete. Result was null. 7:37

Uma classe é a fôrma para a construção de um objeto. É na classe que definimos quais são os dados do objeto e codificamos o comportamento do mesmo. Com essa fôrma podemos criar nossos objetos, como pode ser notado nas imagens anteriores temos a definição da **classe MP3** e a criação dos **objetos musica01** e **musica02**.

#### Prática 04



*Construa um modelo de domínio, usando apenas o que foi visto até agora, para o seguinte cenário: Um computador contém um processador e um sistema operacional. O sistema operacional é responsável por gerenciar tarefas, ele mantém uma fila de tarefas. O SO delega a um processador que execute cada uma das tarefas. Após executar um trecho da tarefa, o SO verifica se a tarefa finalizou, se tiver finalizado não faz nada e continua o processamento da lista, se não finalizou ele coloca a tarefa no final da fila e pega outra para processar. Cada tarefa tem um nome, um código e uma quantidade de passos. O SO fornece a funcionalidade de criar tarefas. O SO tem um nome e uma versão.*

### 2.2.2 - Herança e Polimorfismo

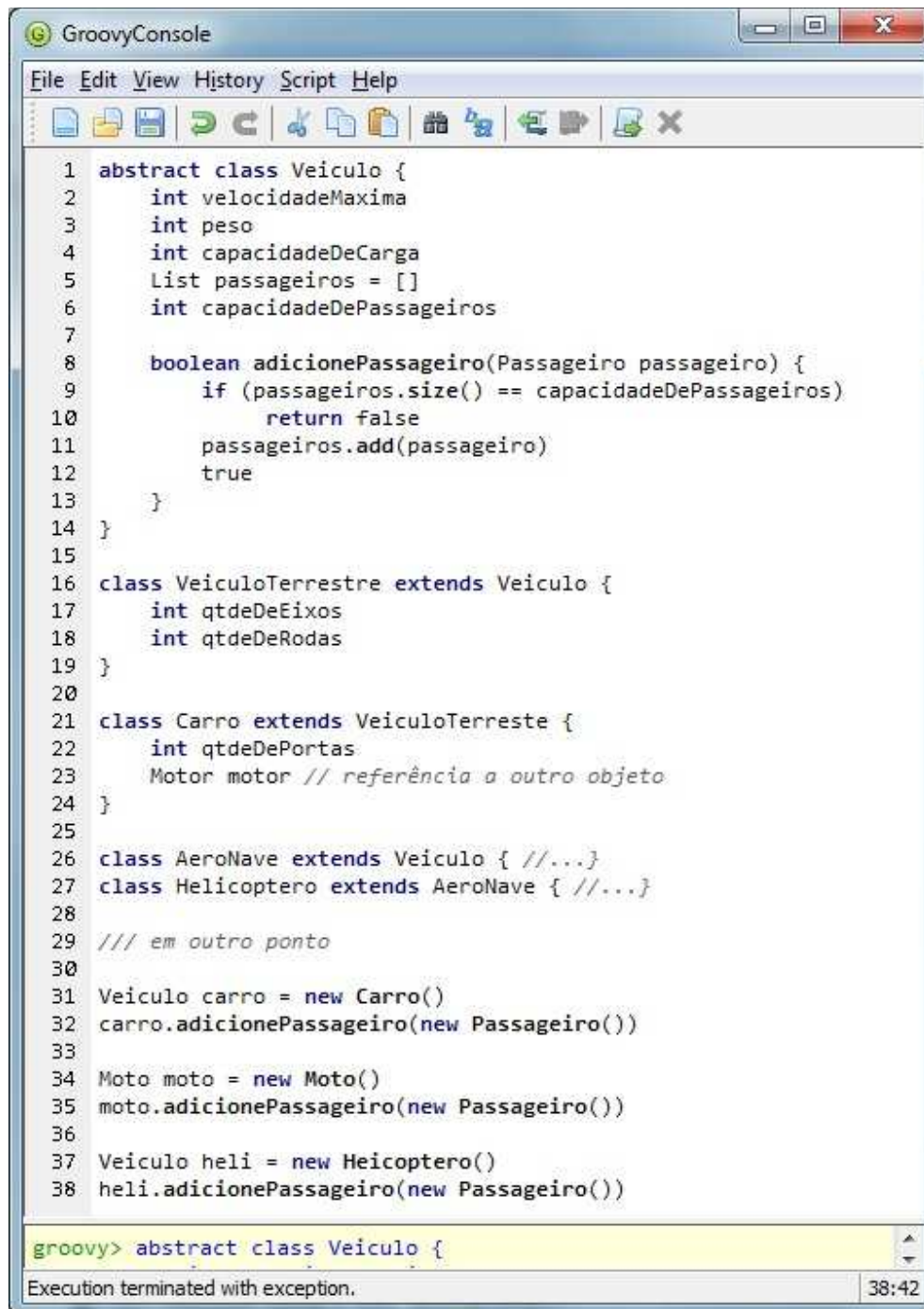
A herança é um recurso da orientação a objetos que nos permite reutilizar código e definir uma hierarquia para nossas classes, no sentido de classes mais genéricas para classes mais especializadas.

Essas especializações são simples de compreender se olharmos para o mundo real. Pense em veículos por exemplo. Para pensarmos no conceito mais genérico nos precisamos pensar em características comuns a todos os veículos, como: velocidade máxima de locomoção, peso, meio de locomoção (água, terra, ar), cor, capacidade de carga, capacidade de passageiros, e outras.

Dentre esses tipos de veículo cada um tem suas características peculiares, veículos terrestres têm quantidade de rodas, aeronaves têm quantidade de motores e veículos aquáticos podem ser divididos em tração humana, movido pelo vento ou por motor. *(note que estou pegando alguns pontos para poder explicar o que é herança, não vamos ficar aqui levantando todas possibilidades de veículos)*

Da mesma forma que no mundo real temos entidades genéricas (Veículos), e entidade especializadas (Carros, Motos, Bicicletas, Navios, etc), no mundo da programação orientada a objetos também podemos fazer isso, através da herança de classes. Podemos também ter especialização em mais de um nível: Veículo se especializa em Veículos terrestres, veículos aquáticos e veículos aéreos, os terrestres por sua vez se especializam em carros, motos, caminhões, e assim por diante.

Vejamos uma possível codificação para esse exemplo:



```
1 abstract class Veiculo {
2     int velocidadeMaxima
3     int peso
4     int capacidadeDeCarga
5     List passageiros = []
6     int capacidadeDePassageiros
7
8     boolean adicionePassageiro(Passageiro passageiro) {
9         if (passageiros.size() == capacidadeDePassageiros)
10             return false
11         passageiros.add(passageiro)
12         true
13     }
14 }
15
16 class VeiculoTerrestre extends Veiculo {
17     int qtdeDeEixos
18     int qtdeDeRodas
19 }
20
21 class Carro extends VeiculoTerrestre {
22     int qtdeDePortas
23     Motor motor // referência a outro objeto
24 }
25
26 class AeroNave extends Veiculo { //...}
27 class Helicoptero extends AeroNave { //...}
28
29 /// em outro ponto
30
31 Veiculo carro = new Carro()
32 carro.adicionePassageiro(new Passageiro())
33
34 Moto moto = new Moto()
35 moto.adicionePassageiro(new Passageiro())
36
37 Veiculo heli = new Heicoptero()
38 heli.adicionePassageiro(new Passageiro())
39
40 groovy> abstract class Veiculo {
41
42 Execution terminated with exception. 38:42
```

Note que a operação *adicionePassageiro* está disponível para todas classes que herdam de veículo. E as subclasses (classes que herdam) podem ter operações específicas que não são encontradas nas superclasses (classes que são herdadas). Poderíamos ter a operação *troquePneus* para a classe Carro, por exemplo, e que não faria muito sentido para a classe Navio. Resumidamente podemos dizer que uma subclasse é a superclasse e mais um

pouco, isto é, ela tem todos dados e operações da sua classe pai, e mais alguns dados e operações específicas para ela.

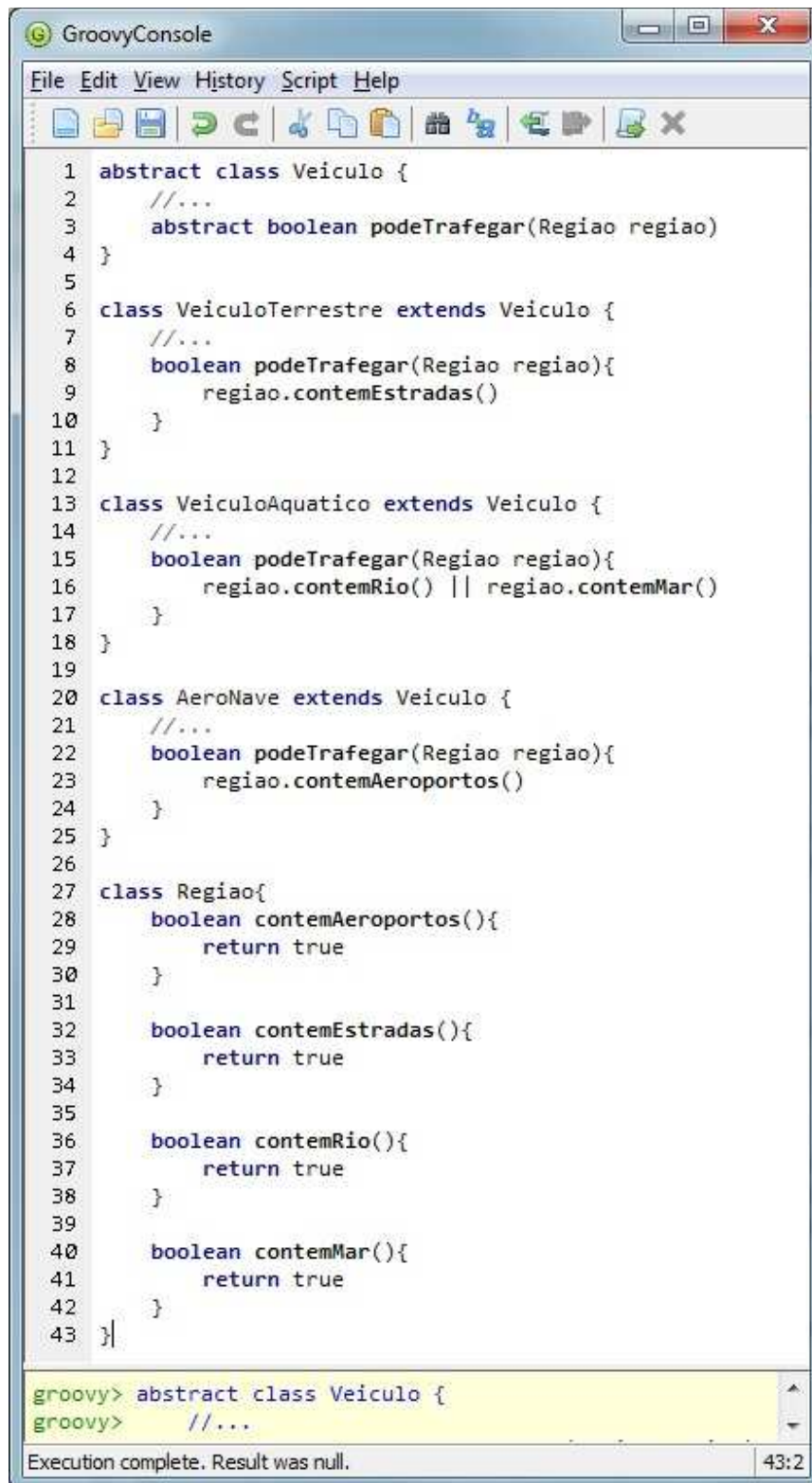
Note que a classe veículo é iniciada com a palavra *abstract*, o que indica, obviamente, que está é uma classe abstrata. Classes abstratas não podem ser instanciadas, elas apenas servem de molde para suas classes filhas, não faz sentido existir uma classe abstrata que não contenha classes filhas.

Esse é um assunto bastante extenso, porém no momento nosso objetivo é apenas introduzir esses conceitos, e à medida que formos aplicando-os na prática, que é o que nos interessa, eles serão compreendidos com mais detalhes e clareza. Vamos apenas dar uma pincelada sobre polimorfismo antes de passar adiante.

Além de poder adicionar novas operações e dados, as subclasses também podem alterar comportamentos definidos na classe pai, o que chamamos de **Polimorfismo**. Ou seja:

*Polimorfismo é a capacidade que classes filhas têm de alterar algum comportamento definido na classe pai, através da sobrescrita de métodos. Ou, por outro ponto de vista, é a capacidade que objetos têm de realizar uma mesma operação de forma distinta, de acordo com a especialização de cada um.*

Exemplo:



```
1 abstract class Veiculo {
2     //...
3     abstract boolean podeTrafegar(Regiao regiao)
4 }
5
6 class VeiculoTerrestre extends Veiculo {
7     //...
8     boolean podeTrafegar(Regiao regiao){
9         regiao.contemEstradas()
10    }
11 }
12
13 class VeiculoAquatico extends Veiculo {
14     //...
15     boolean podeTrafegar(Regiao regiao){
16         regiao.contemRio() || regiao.contemMar()
17    }
18 }
19
20 class AeroNave extends Veiculo {
21     //...
22     boolean podeTrafegar(Regiao regiao){
23         regiao.contemAeroportos()
24    }
25 }
26
27 class Regiao{
28     boolean contemAeroportos(){
29         return true
30     }
31
32     boolean contemEstradas(){
33         return true
34     }
35
36     boolean contemRio(){
37         return true
38     }
39
40     boolean contemMar(){
41         return true
42     }
43 }
```

groovy> abstract class Veiculo {  
groovy> //...

Execution complete. Result was null. 43:2

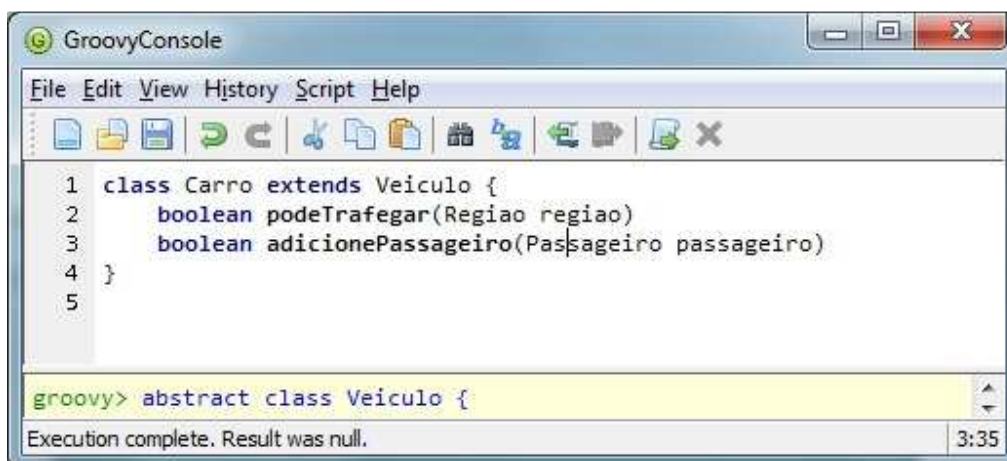
## Prática 05

*Modifique o código do exercício anterior, de modo a dar suporte às seguintes alterações: O nosso sistema operacional deve dar suporte a três tipos de processo: Processos de processamento lento, de processamento neutro e de processamento rápido. Os de processamento lento executam um passo de cada vez, os de processamento médio dois passos por vez, e os últimos três passos por vez. O SO deve dar suporte à criação de todos esses novos tipos de tarefas.*

### 2.2.3 - Interfaces

Como foi dito anteriormente, em um sistema orientado a objetos existe uma comunicação entre os objetos do sistema, para que a lógica de domínio possa ser implementada. Para que os objetos possam conversar é necessário que eles estabeleçam um contrato de comunicação. Esse contrato pode ser definido direto na classe, ou pode ficar separada, em uma interface.

Para nosso exemplo dos veículos a interface da classe Carro, por exemplo, é:



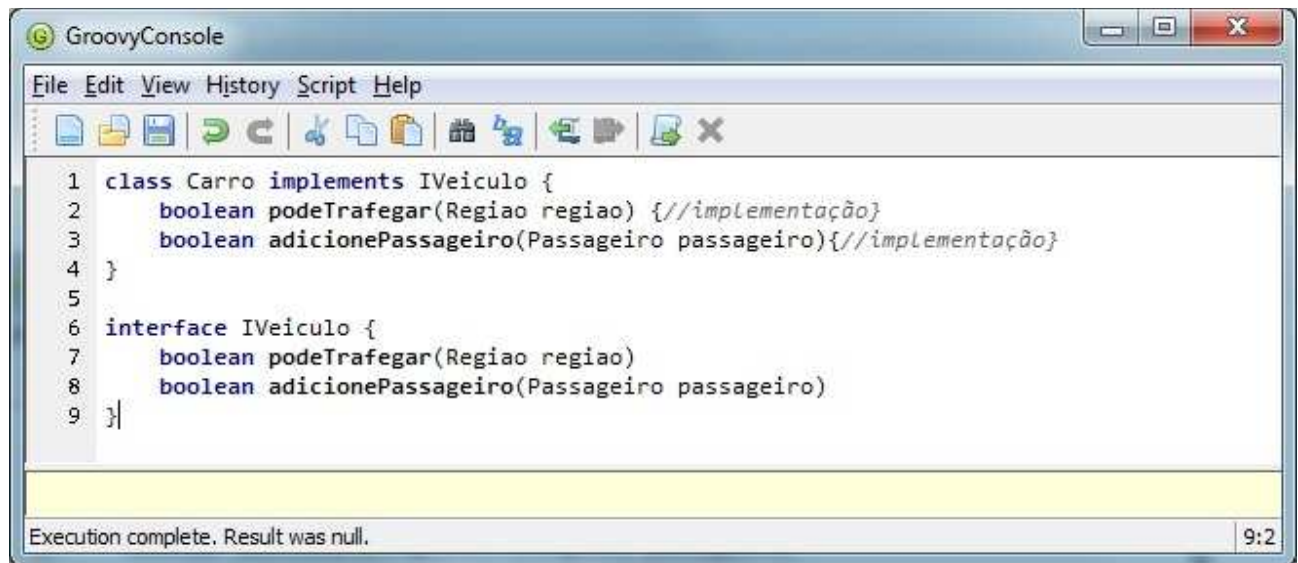
```
1 class Carro extends Veiculo {
2     boolean podeTrafegar(Regiao regiao)
3     boolean adicionePassageiro(Passageiro passageiro)
4 }
5

groovy> abstract class Veiculo {
Execution complete. Result was null. 3:35
```

Você pode estar se perguntando: Mas isso não é a própria classe? Na verdade não, isso é a definição das operações que os objetos desta classe podem realizar. A classe envolve também a implementação destas operações.

A outra forma é extrair para uma interface, que é um conceito específico das linguagens de programação orientadas a objeto.

Vamos mudar um pouco o a nossa solução de veículos, vamos remover a herança da classe Veículo e criar uma interface IVeiculo:



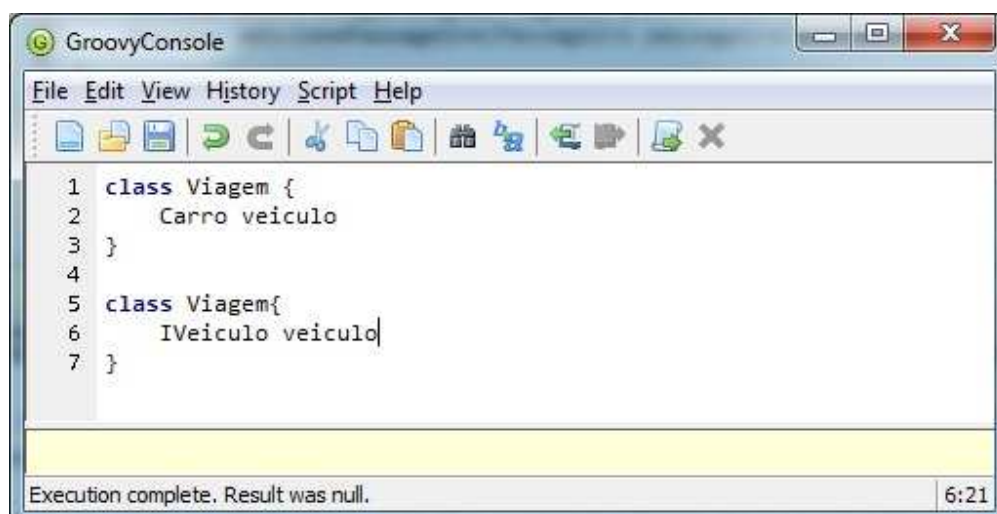
```
1 class Carro implements IVeiculo {
2     boolean podeTrafegar(Regiao regiao) { //implementação
3     boolean adicionePassageiro(Passageiro passageiro) { //implementação
4 }
5
6 interface IVeiculo {
7     boolean podeTrafegar(Regiao regiao)
8     boolean adicionePassageiro(Passageiro passageiro)
9 }
```

Execution complete. Result was null. 9:2

Da mesma forma que no caso anterior as operações estarão disponíveis para os objetos da classe Carro.

A questão se o contrato será definido apenas na classe ou se será definido em uma interface é apenas tecnológica, conceitualmente ambas servem para estabelecer a forma que os objetos destas classes irão trocar mensagens com outros.

Conceitualmente são equivalentes, porém com a separação da interface podemos deixar a solução mais flexível. Suponha que uma companhia de viagem dependa de um carro para realizar uma viagem de turismo. Porém se pensarmos um pouco a viagem precisa é de um veículo qualquer, desta forma viagem pode depender da interface IVeiculo e não da classe Carro, pois assim podemos fornecer qualquer implementação de IVeiculo(Carro, Moto, Jegue) que a viagem acontece de qualquer forma!



```
1 class Viagem {
2     Carro veiculo
3 }
4
5 class Viagem{
6     IVeiculo veiculo
7 }
```

Execution complete. Result was null. 6:21



Novamente não se assuste com essa teoria, são apenas pinceladas, vamos aprender mesmo é na prática.

## Prática 06

*Suponhamos que algumas de nossas tarefas possam ser interrompidas. Implemente uma interface que forneça uma operação para a interrupção de processos. O nosso SO também deve ser alterado de modo que forneça uma opção para interromper processos, caso o processo não seja interrompido uma exceção deve ser lançada, caso contrário ele deve ser removido da lista.*

### 2.2.4 - Pensando em Papeis+Comportamentos+Colaborações na hora de definir seus objetos

No tópico sobre DDD foi dito que os objetos de domínio contem dados e algoritmos que alteram esses dados. No produto final o que realmente temos é realmente isso, porém a forma de se chegar neste resultado pode ser diferente. Existe uma outra forma de se pensar no problema, para que chegamos ao nosso modelo de domínio, essa forma de pensar é definida por Craig Larman. Essa forma é pensarmos em Papeis, Comportamentos e Colaborações. O Objetivo desta nova forma de se pensar em objeto é aproxima ainda mais essa observação de componentes do sistema à observação que temos do mundo real.

Para chegar ao nosso modelo de objetos devemos seguir os passos abaixo, após ter compreendido bem nossos requisitos é claro:

- Identificar os *objetos* no domínio do problema
- Pensar nas *responsabilidades* destes objetos
- Pensar em como os objetos *colaboram* para cumprir as responsabilidades

Para facilitar um pouco a compreensão do que foi dito vamos pedir ajuda à tabelinha de definições abaixo:

Termo	Definição
<b>Aplicação</b>	Um conjunto de objetos que se interagem
<b>Objeto</b>	A implementação de um ou mais papéis
<b>Papel</b>	Um conjunto de responsabilidades relacionadas
<b>Responsabilidade</b>	Uma obrigação de executar uma tarefa ou conhecer uma informação
<b>Colaboração</b>	Uma interação de objetos o papéis (ou ambos)

<b>Contrato</b>	Um acordo que define os termos de uma colaboração
-----------------	---

### Tipos de responsabilidades

- Responsabilidade de *conhecer*
- Responsabilidade de *fazer*

### Como ocorre a colaboração?

A colaboração entre os objetos ocorre através da troca de *mensagens*. Um objeto envia uma *mensagem* a outro objeto, relativa a alguma de suas *responsabilidades*. Para que o objeto atenda a esta mensagem, ele pode usar informações que ele mesmo armazena, ou enviar mensagens a outros objetos.

### Prática 07

*Com base nos requisitos apresentados no item 1.5, identifique os papéis as responsabilidades e colaborações necessárias para que o nosso sistema possa funcionar. Utilize a notação que achar mais adequada seja ela UML, texto ou qualquer tipo de diagrama.*

## 2.3 - Testes de Unidade

Esse é um ponto extremamente importante do nosso estudo. Os testes têm uma importância muito grande no desenvolvimento de sistemas, qualquer desenvolvedor com alguma experiência sabe disso, é muito comum ver programadores sofrendo por conta de erros que seriam facilmente identificados e corrigidos com bons testes de unidade.

Em geral os testes não são feitos, e raramente são bem feitos. Mas como, se você acabou de dizer que todos programadores com alguma experiência sabem disso? Normalmente atribuo esse fator a falta de planejamento e também ao descaso com a qualidade de software, por isso quase nunca são feitos. Os desenvolvedores, ao planejarem suas datas de entrega não consideram o tempo para a escrita dos testes, ou então subestimam este tempo.

É natural que o teste gaste o mesmo tempo para ser feito que a implementação. Isto é, se existe uma funcionalidade que normalmente é feita em 16 horas, os testes para ela demoram aproximadamente 16 horas também, isso quando bem feitos, o que envolve várias técnicas para escrita de testes de unidade, também conhecidos como testes caixa preta.

Existe uma observação importante, quando os testes são feitos antes do código, o que veremos no próximo tópico, o tempo de implementação da funcionalidade tende a ser



reduzido, e os erros de programação mais ainda, isso por que ao escrever os testes antes, o programador tira todas suas dúvidas quanto ao comportamento daquilo que ele está fazendo, mas isso é assunto para daqui a pouco.

Os testes de unidade geralmente tratam de uma classe específica. Normalmente temos uma classe de testes para uma classe da aplicação, classes do domínio, e dentro desta classe de testes têm vários métodos, que são realmente os testes responsáveis por testar diversos comportamentos das operações desta classe. Esses testes são escritos via código-fonte, na mesma linguagem dos seus sistemas, as plataformas normalmente usadas para aplicações corporativas tem boas bibliotecas para construção de testes (.NET, Java, Groovy, Ruby, etc.)

Para cada operação de uma dada classe, os testes devem ser escritos com base no simples raciocínio: Realizamos diversas chamadas para essas operações, com os parâmetros e configurações conhecidas, e verificamos se o resultado obtido é o que é esperado, de acordo com as regras de negócio. Veja o exemplo abaixo: *(para esse exemplo vamos usar uma classe utilitária, e não uma classe de domínio)*

```
class Calculadora {  
    static soma(a, b) {  
        a + b  
    }  
  
    static multiplique(a,b) {  
        a * b  
    }  
}
```

```
class CalculadoraTests extends GrailsUnitTestCase {  
    protected void setUp() {  
        super.setUp()  
    }  
  
    protected void tearDown() {  
        super.tearDown()  
    }  
  
    void testSoma() {  
        assertEquals 9, Calculadora.soma(2, 7)  
        assertEquals 4.8, Calculadora.soma(4, 0.8)  
    }  
  
    void testMultiplicacao(){  
        assertEquals 14, Calculadora.multiplique(2, 7)  
        assertEquals 3.2, Calculadora.multiplique(4, 0.8)  
    }  
}
```

Vamos a algumas observações importantes:

- A classe que nos fornece a estrutura para testes é *GrailsUnitTestCase*.
- O método *setUp* pode ser usado para colocarmos códigos de inicialização, isto é, um código que será executado antes de cada teste, um código para criar arquivos que serão usados nos testes, por exemplo.
- O método *tearDown* pode ser usado para colocar códigos de finalização, pois este será executado após cada teste, como um código para apagar arquivos por exemplo.
- A verificação do teste é feita pelas assertivas, neste testes usamos assertivas de igualdade, com o método *assertEquals(valorEsperado, valorRetornado)*. Existem diversos outros: *assertNotNull*, *assertNull*, *assertTrue*, *assertContains* etc..

Vale salientar também que os testes dão segurança para que alterações sejam feitas no código, e as alterações vão surgir aos montes!

## Prática 08

Construa testes de unidade para todas as operações da classe *Lista*, fornecida pelo instrutor.

## 2.4 – Cobertura de Testes

Escrever bons testes de software é um desafio tanto para programadores iniciantes, quanto para os mais experientes. Existem dezenas de estratégias que podem ser usadas para buscar uma codificação de testes mais efetivos.

O natural é que à medida que você vai testando seu sistema você vá se deparando com situações mais complexas para serem testadas, é um desafio constante!

Uma das formas de buscar qualidade no seu teste é através da análise de cobertura de código atingida por seus testes de unidade. Mas afinal, o que é essa cobertura de código? Cobertura de código é um relatório que diz quais trechos de seu código foram executados por um determinado conjunto de testes.

Existem diversos frameworks que geram cobertura de código para testes, para nosso estudo iremos utilizar o framework *Cobertura*, que é bastante utilizado em aplicações Java e que está disponível via *plugin* para aplicações Grails.

- Abra o gerenciador de plugins do STS: Alt+G,M
- Selecione o plugin code-coverage e mande instalar

Após instalar o *plugin* basta executar o comando de testes com a opção `-coverage`:

```
grails test-app -coverage
```

Esse comando irá gerar um relatório de cobertura de código em HTML, que estará disponível no diretório `target/test-reports/cobertura`. Através destes relatórios você irá identificar os trechos do seu código que não foram testados e irá, então, poder criar novos testes ou alterar os existentes de modo que esses trechos possam então ser executados durante os testes.

#### 2.4.1 - Cobertura por Linha de Código x Cobertura por *Branch*

No relatório de cobertura nós iremos encontrar informações referentes a dois tipos de cobertura, cobertura por linha e por *branch*, como na figura abaixo:

##### Coverage Report - All Packages

Package /	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	26	20% 	19% 	0
<a href="#">inscricoesesportivas</a>	26	20% 	19% 	0

Report generated by [Cobertura](#) 1.9.4.1 on 08/06/11 14:39.

A cobertura por linha de código simplesmente verifica se a linha em questão foi executada, enquanto que a cobertura por branch irá validar se todos os caminhos possíveis foram executados. Por isso é importante ter poucos condicionais no código e assim manter baixa

a complexidade ciclomática do mesmo, pois quanto menos condicionais menos opções de caminho termos para percorrer e, conseqüentemente, mais fácil será alcançar 100% de cobertura de código por *branch*.

### Prática 09

*Execute a cobertura dos testes criados na Prática 08 e faça as alterações necessárias para que eles cubram 100% das linhas da classe Lista.*

## 2.5 - Desenvolvimento Guiado por Testes (TDD)

Para iniciar este assunto volto ao ponto do item anterior onde foi dito que poucos desenvolvedores escrevem testes de unidade. Vamos aos fatores que causam isso:

- Os sistemas sempre tem uma fila de funcionalidades a serem implementadas, ao acabar uma o desenvolvedor, ou líder de equipe, quer aproveitar a disponibilidade de tempo para pegar outra funcionalidade, e a que acabou de ser feita fica sem testes;
- Escrever testes é difícil, é preciso estudo e treino, se o programador não busca aprender sobre isso na hora do vamos ver a dificuldade irá vencê-lo;

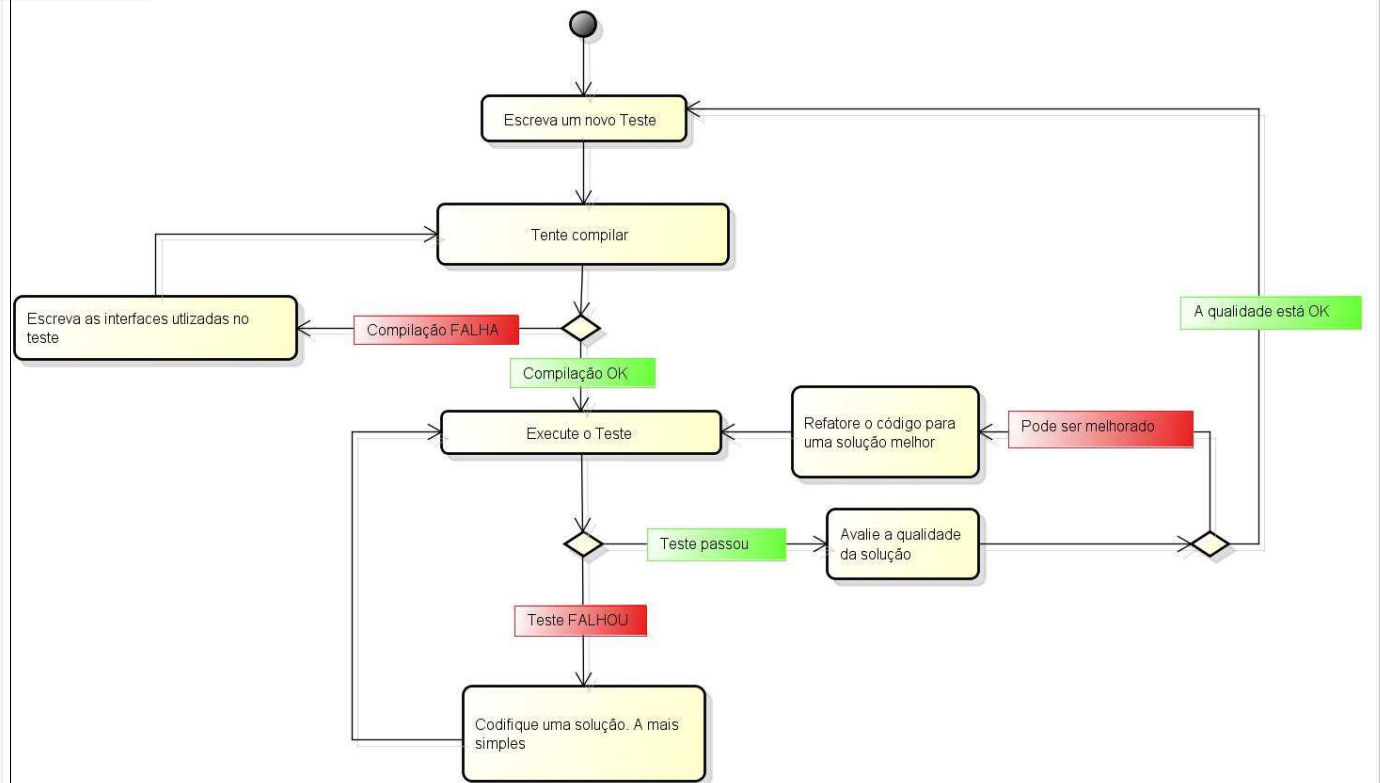
De posse destas informações fica-nos fácil identificar a primeira vantagem de escrever os testes antes de codificar o sistema: **Quando o sistema ficar pronto ele já estará com todos seus testes construídos.**

A outra vantagem, também importantíssima, é que ao escrever os testes, o programador é obrigado a compreender com detalhes, o requisito que está implementando. A escrita de testes serve para refinar os requisitos, à medida que os testes vão sendo escritos, as dúvidas vão sendo esclarecidas, e quando chegar a hora de programar, poucas dúvidas(referentes ao que deve ser feito) restarão na mente do programador.

Como os testes devem validar todo comportamento do sistema, ele também servirá como documentação do projeto, e o melhor de tudo, é uma documentação executável! Pense nisso! Você terá um registro que descreve como suas classes devem se comportar em todas situações, ótimo, e ainda por cima poderá executar para ver se elas realmente estão fazendo o que devem fazer, fantástico!

De forma bem direta, para programarmos de forma guiada por testes devemos seguir os passos do diagrama abaixo: *(antes de fazer os passos abaixo não deve existir código nas suas classes, no máximo o arquivo delas, sem nada dentro)*

actActivity Diagram0



powered by astah™

Esse ciclo deve ser repetido até que você tenha um conjunto de testes bastante completo para a funcionalidade que está implementando. Quando você saberá que está bom o suficiente? Isso depende da sua experiência e dos seus conhecimentos sobre as práticas de escrita para bons testes.

Outra linha tênue é a avaliação que você fará para verificar se seu código precisa de melhoria ou não, isso é sutil, minha dica é a seguinte, quando você achar que está bom o suficiente chame alguém mais experiente para dar uma opinião.

### Prática 10

A partir deste momento iremos direcionar todas nossas práticas para o nosso estudo de caso, o sistema de estacionamento. Essa prática será a mais longa dentre as que já fizemos até este momento. Você deverá construir toda camada de domínio da nossa aplicação, de forma orientada a testes. Construa um teste por vez, e incremente-os gradativamente, conforme o diagrama de TDD apresentado acima, até que tenhamos nossa implementação completa. A cada passo do ciclo você deverá solicitar o acompanhamento do instrutor para que ele possa avaliar os passos que já executou e lhe orientar em relação aos próximos.

*Ainda usando os requisitos definidos no item 1.5, considere os seguintes dados para a construção de seus testes:*

- *Os valores de hora são:*
  - *Moto: R\$ 1,00*
  - *Carro: R\$ 2,00*
  - *Caminhonete: R\$ R\$ 2,50*
- *As configurações de valor reduzido para horas iniciais são:*
  - *Carro, as duas primeiras horas, R\$ 1,00*
  - *Caminhonete, as três primeiras horas, R\$ 2,00*
- *As tolerâncias são:*
  - *Moto: 15 minutos*
  - *Carro: 30 minutos*
  - *Caminhonete: 30 minutos*
- *Os valores de diárias são:*
  - *Moto: R\$ 15,00*
  - *Carro: R\$ 26,00*
  - *Caminhonete: R\$ 35,00*
- *A tolerância vale apenas para os minutos iniciais*
- *Existe também uma tolerância por hora, de 10 minutos para todos*
  - *1h10min => Será cobrada 1 hora*
  - *1h11min => Serão cobradas 2 horas*
- *Deve ser possível criar promoções*
- *Promoções podem valer para um ou mais tipo de veículo*
- *Promoções contêm o valor promocional e a faixa de horários em que ela é válida*
- *A hora promocional vale desde que a hora a receber desconto contenha no mínimo 20 minutos dentro do período promocional*
  - *Promoção, R\$ 0,50, das 11hs às 13hs*
  - *Ex. 01: Entrada 10h30min, Saída 11h30 – Paga o valor promocional*
  - *Ex. 02: Entrada 10h20min, Saída 11h15 – Paga hora normal*

*Obs.: Ao escrever seus testes lembre-se de que eles são uma documentação do seu sistema, através dele deve ser possível compreender tudo que sua aplicação deverá fazer. Esse é o momento de tirar as dúvidas quanto aos requisitos.*

## Capítulo 03: Camada de Persistência

Quando decidimos construir um sistema de informação (*o mesmo que aplicação corporativa*) usando programação orientada a objetos, nós acabamos nos deparando com um problema de compatibilidade, referente ao armazenamento não volátil dos dados. O problema em questão é o seguinte: No armazenamento volátil nos temos os dados representados através de objetos de domínio, enquanto que no armazenamento não volátil (bancos de dados relacionais), nós temos os dados representados através de tabelas, e relações entre as mesmas.

É claro que hoje também temos a opção de usar bancos de dados orientados a objetos, já existem opções bem difundidas no mercado (o que mais ouço falar é o [db4objects](#)).

Mesmo tendo a opção de usar banco de dados orientado a objetos, poucas pessoas passaram por uma experiência real com esses bancos, e eu não sou uma delas. Além disso, os bancos de dados relacionais apresentam algumas vantagens em relação aos bancos relacionais, dentre elas: Vasto conteúdo na internet; atendem bem, comprovadamente, a sistemas com grande fluxo de dados; e existem boas opções grátis.

Deste modo vamos estudar o modelo tradicional que utiliza objetos em memória, e tabelas dos bancos de dados relacionais, como armazenamento estático de dados. Nos próximos tópicos veremos com isso é feito.

### 3.1 - Mapeamento Objeto Relacional

O mapeamento objeto relacional é um conjunto de técnicas usadas para traduzir dados em formato relacional para dados em formato de objetos. Isto é feito através de um framework que fornece arquivos de configuração e uma API para consulta. Através desse framework o programador não precisa se preocupar dos detalhes que ocorrem nessa tradução de dados, isso é tarefa do framework, a manipulação de dados no banco é feita de forma transparente para o desenvolvedor.

O framework mais difundido nos mundos Java e .NET é o Hibernate (que se chama NHibernate para a versão .NET), e é esse mesmo framework que o Grails usa para realizar a comunicação dos objetos de domínio com o banco de dados.

Porém, com o Grails é muito mais simples trabalhar com persistência do que em um sistema Java comum, pois através da convenção sobre configuração, e das características dinâmicas do Groovy os mapeamentos e consultas se tornam automáticos, intuitivos e muito simples. Isto tudo é feito através do GORM (*Grails Object Relational Mapping*).

Vejamos alguns exemplos práticos disto que acabei de falar:

- Para toda classe de domínio é criada uma tabela no banco de dados;
- Para todos os propriedades simples das classes de domínios são criadas colunas para a tabela da classe em questão;
- Para as referências entre objetos o GORM cria uma chave estrangeira da classe contida para a classe que contém;
- De maneira resumida podemos dizer que o GORM cria todos artefatos necessários para o mapeamento dos seus dados nas tabelas, isso vale para associações de listas e de dicionários também;
- A API de persistência é injetada diretamente nos objetos de domínio, através dos *finders* dinâmicos, e demais métodos de persistência (get, delete, save, etc.);
- A sessão e transação são gerenciadas automaticamente através das chamadas aos controladores.

### Prática 11

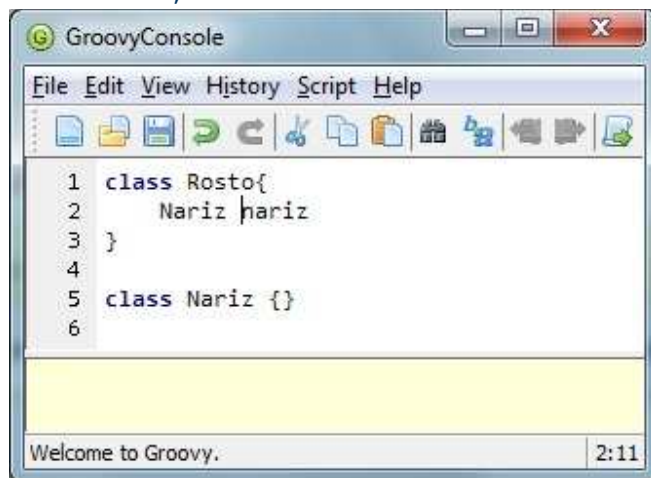
*Altere no arquivo `DataSources.groovy` a url de conexão com o banco de desenvolvimento. Originalmente este banco está configurado para ser em memória, mude para arquivo. Inicie a aplicação e realize alguma operação que altere o estado do banco de dados. Agora abra seu banco com algum cliente e verifique as tabelas e colunas criadas, analise-as comparando-as com o seu modelo de domínio.*

#### 3.1.1 - Mapeamento de Associações

Vejamos alguns exemplos de associações entre objetos e configurações do GORM que nos permitem dizer como desejamos que essas associações sejam refletidas no banco de dados.

##### 3.1.1.1 - One-to-One

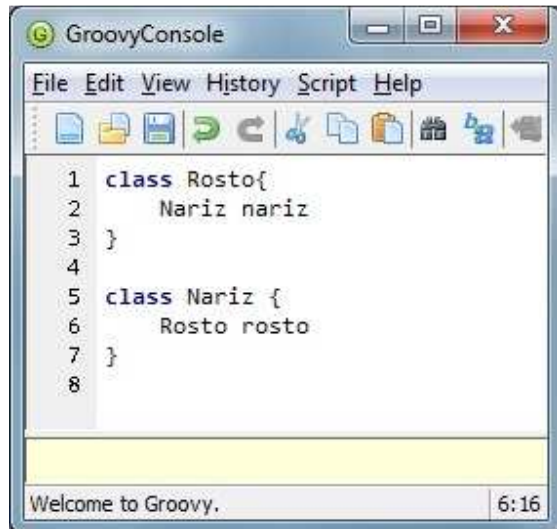
Unidirecional, sem cascata



//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS



Bidirecional, sem cascata

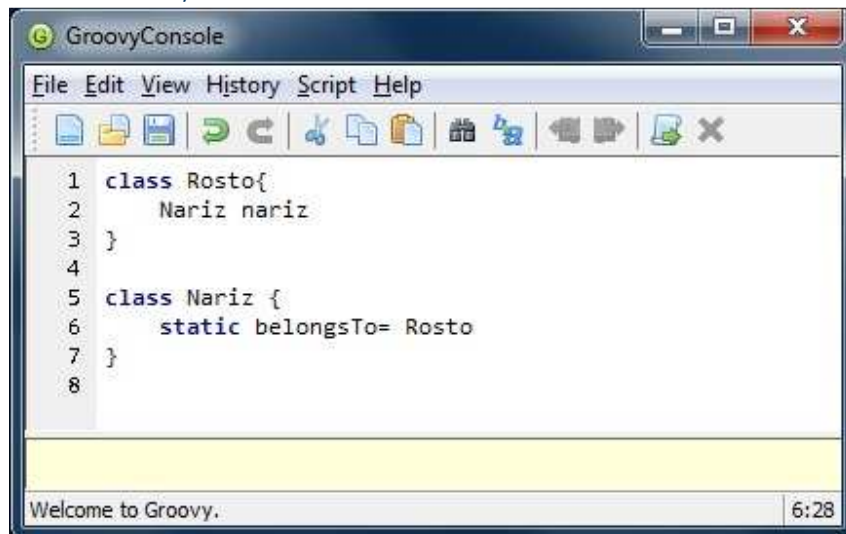


```
1 class Rosto{
2     Nariz nariz
3 }
4
5 class Nariz {
6     Rosto rosto
7 }
8
```

Welcome to Groovy. 6:16

//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS

Unidirecional, com cascata

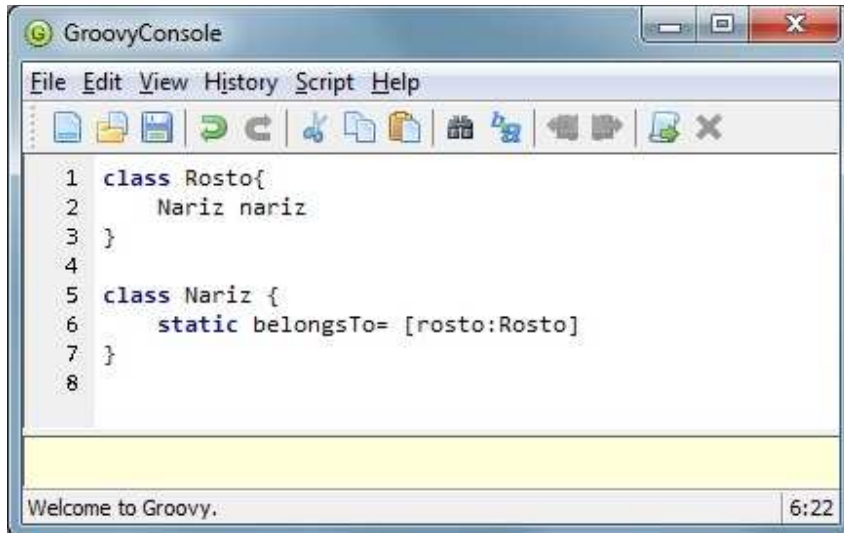


```
1 class Rosto{
2     Nariz nariz
3 }
4
5 class Nariz {
6     static belongsTo= Rosto
7 }
8
```

Welcome to Groovy. 6:28

//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS

Bidirecional, com cascata

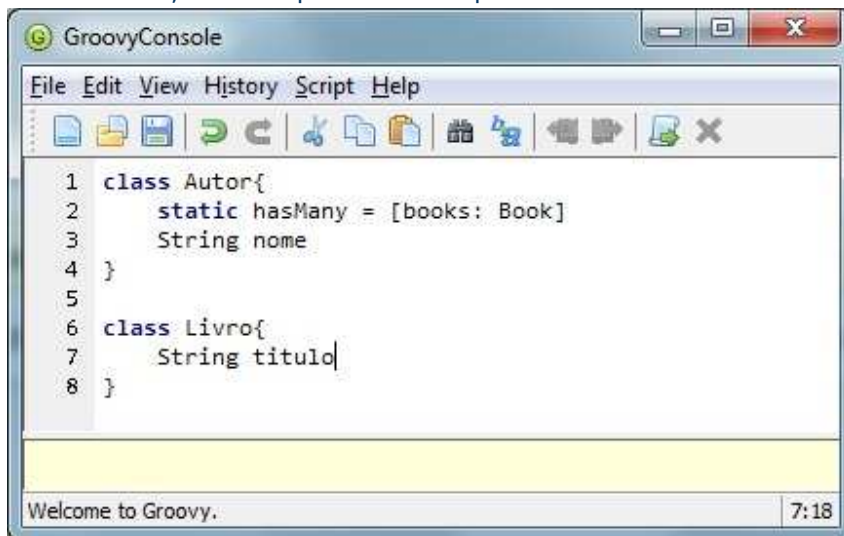


//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS

#### 3.1.1.2 - One-to-Many

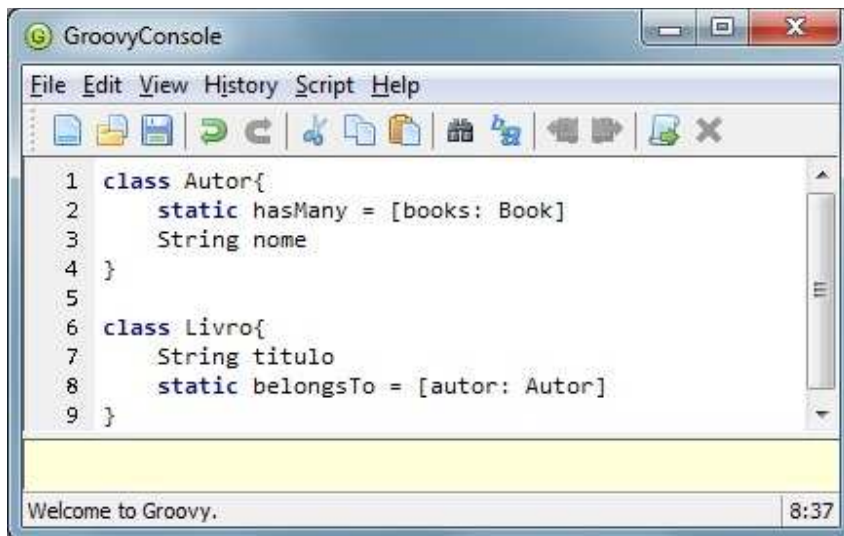
//Por default é criado um list, mas você pode alterar isso, criando uma propriedade do tipo que desejar: Set, SortedSet, Map, etc...

Unidirecional, cascata para save e update



//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS

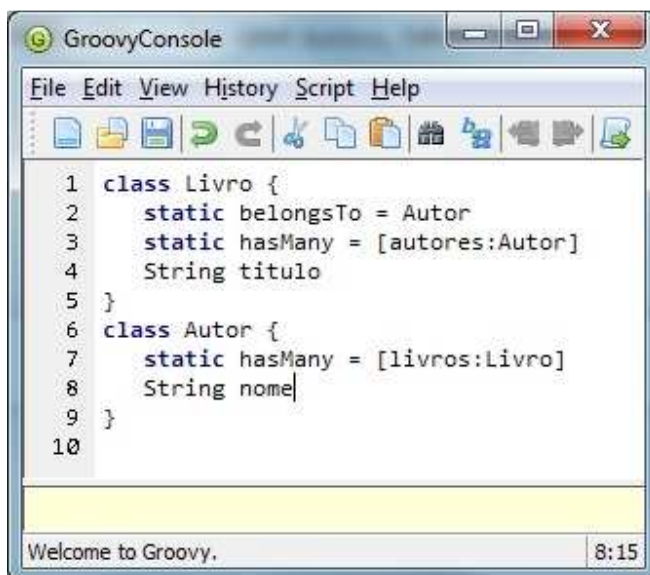
Bidirecional, cascata para save, update e delete



//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS

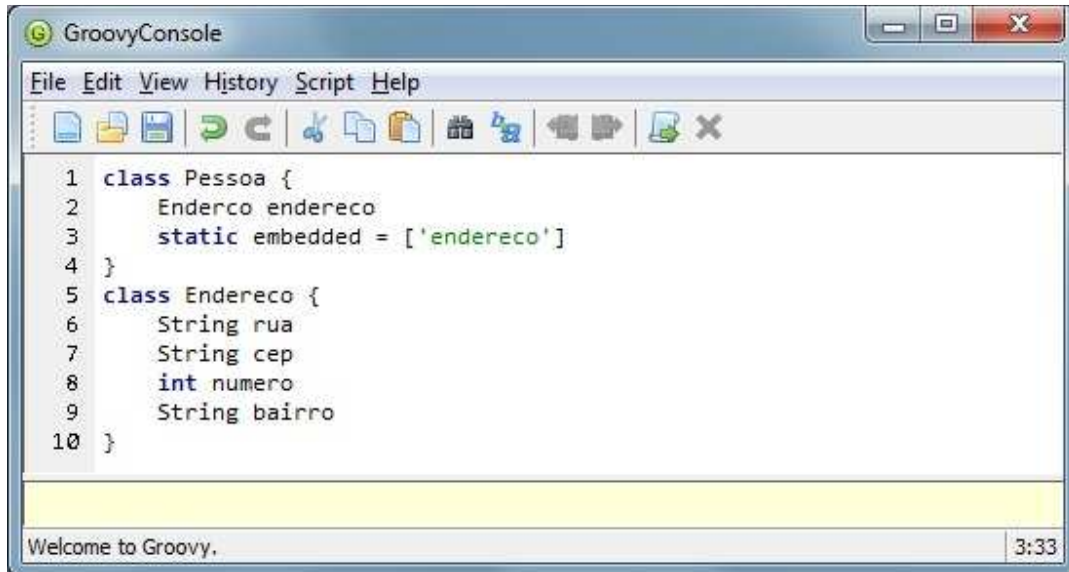
### 3.1.1.3 - Many-to-Many

//OBS: O Scaffolding do Grails não dá suporte a relações do tipo Many-to-Many, para elas é necessário escrever o código manualmente



//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS: A cascata só acontece de um dos lados, do lado pai

### 3.1.2 - Mapeamento de Composições



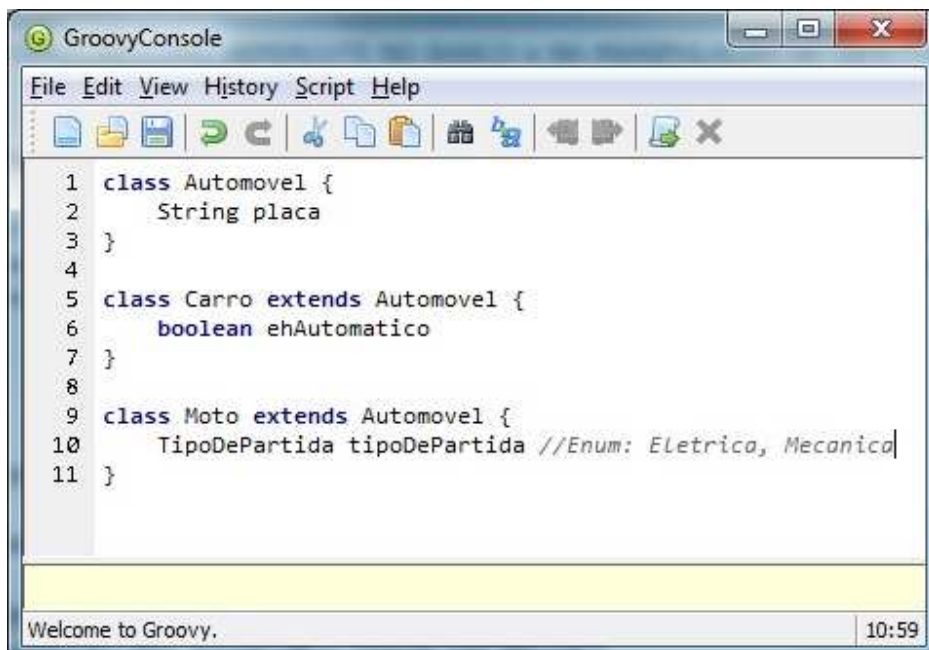
```
1 class Pessoa {
2     Enderco endereco
3     static embedded = ['endereco']
4 }
5 class Endereco {
6     String rua
7     String cep
8     int numero
9     String bairro
10 }
```

Welcome to Groovy. 3:33

//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS

### 3.1.3 - Mapeamento de Herança

Por padrão o GORM irá criar uma tabela única para todas classes de uma dada hierarquia de classes, e nesta tabela existe uma coluna responsável por identificar qual objeto está sendo armazenado em cada linha.



```
1 class Automovel {
2     String placa
3 }
4
5 class Carro extends Automovel {
6     boolean ehAutomatico
7 }
8
9 class Moto extends Automovel {
10     TipoDePartida tipoDePartida //Enum: Eletrica, Mecanica
11 }
```

Welcome to Groovy. 10:59

//EXPLICAR COMO REPERCUTE NO BANCO e NA MANIPULAÇÃO DE OBJETOS

### 3.2 - Realizando operações de banco de dados

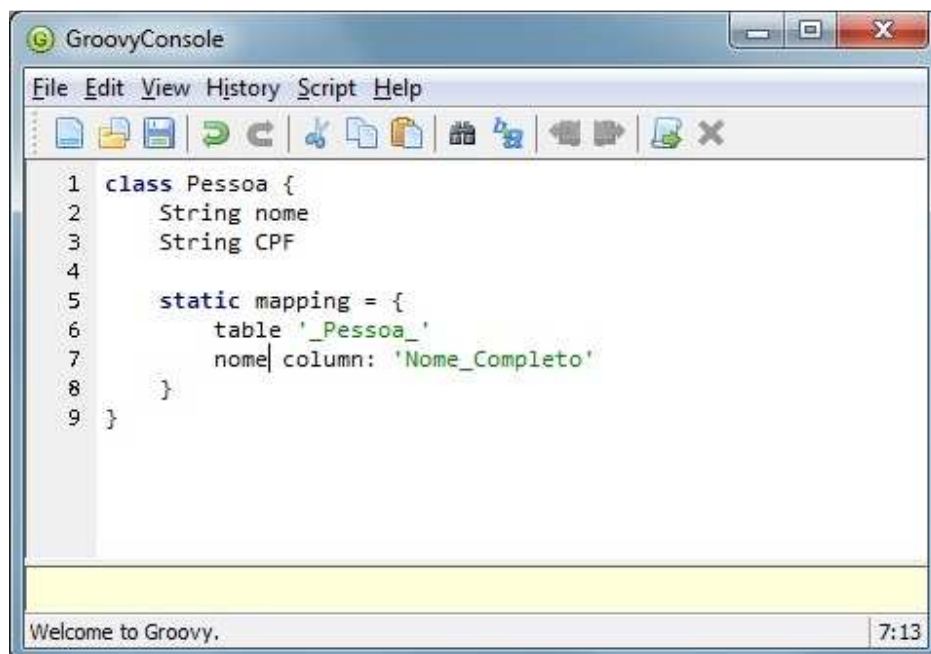
O GORM injeta, em nossos objetos de domínio, uma série de métodos que tornam muito simples as operações em banco de dados. Veja os principais:

- save - Para persistir ou atualizar um objeto no banco;
- delete - Para remover um objeto do banco;
- get - Para recuperar um objeto do banco, dado seu identificador;
- getAll - Para recuperar um objeto do banco, dado seus identificadores;
- list - Para recuperar uma lista de objetos do banco;

### 3.3 - Customizando o Mapeamento

O GORM define as configurações do seu banco de dados de acordo com convenções. Se sua classe se chama Livro, ele irá criar uma tabela no banco com o mesmo nome. Se sua classe livro contém uma propriedade Titulo, ele irá criar uma coluna com o mesmo nome, e assim por diante.

Porém podem haver casos em que seja necessário customizar essa forma de configurar as tabelas do banco de dados, para isso o GORM disponibiliza a GORM DSL. Isso pode ser feito através da notação **static mapping = { //Configurações customizadas }**. Veja um exemplo abaixo:



```
1 class Pessoa {
2     String nome
3     String CPF
4
5     static mapping = {
6         table '_Pessoa_'
7         nome column: 'Nome_Completo'
8     }
9 }
```

Welcome to Groovy. 7:13

//Explicar no que isso reflete

### 3.4 - Consultas via *Finders* Dinâmicos

A forma mais prática de se realizar buscas no banco de dados em Grails é utilizando os *finders* dinâmicos, que nada mais são do que métodos injetados em nossos objetos de domínio e que nos possibilitam consultar objetos através de uma série de restrições. Alguns exemplos:

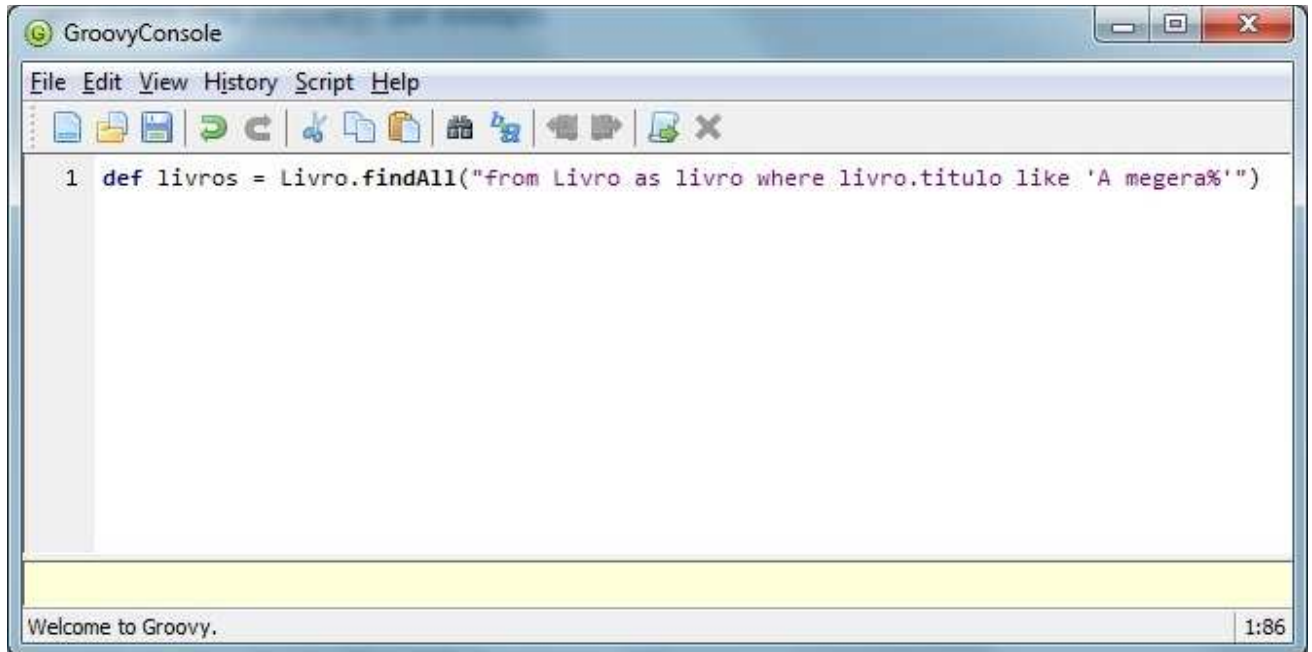
- Buscar uma pessoa por nome
  - `def pessoa = Pessoa.findByName('Roberto Carlos')`
- Buscar todas pessoas que se chamam Roberto
  - `def listaDePessoas = Pessoa.findAllByNameLike('Roberto Carlos%')`
- Buscar pessoas por nome e cidade
  - `def lista = Pessoa.findAllByNameLikeAndCidade('Fernando%', 'Goiânia')`

Essas condições de busca podem ser concatenadas sem restrições, as possibilidades são extensas, confira a lista completa em: <http://inseriraquiiumaurl>

### 3.5 - Consultas via HQL

Os *finders* dinâmicos são suficientes para a grande maioria dos casos (digo isso pois nosso foco são aplicações corporativas, e neste tipo de aplicação as consultas no banco seguem um determinado padrão). Existe casos, todavia, em que utilizar esses métodos pode gerar uma chamada a um método com o nome muito extenso, e fora isso há casos, ainda, em que pode não ser possível determinar nossa consulta através deles, pense em alguma consulta que realize uma *subquery*, por exemplo.

Para essas consultas mais avançadas, temos a opção de realiza-las através da linguagem de consulta de objetos do Hibernate, a HQL (*Hibernate Query Language*).



Essas consultas também podem conter parâmetros.

### Prática 12

**//Mapear e consultar, de diversas formas**  
**//[TODO]**

## 3.6 - Sessão e Transação

As sessões ...

As transações são gerenciadas pela camada de serviço. Através dos *Grails Services*.

### Prática 13

**//[TODO]**

## 3.7 - Configurações de Banco

Além de configurar os mapeamentos e de utilizar as APIs específicas para consultas, existem outras configuração relacionada ao banco de dados, no Grails. Essas são as configurações de acesso ao banco de dados.

As configurações de banco do Grails se encontram no arquivo **DataSources.groovy**, que fica no diretório **grails-app/conf**.

Neste arquivo temos três configurações:

- **dataSource**
  - Configurações do servidor de banco de dados: URL, Driver de Conexão, Usuário, Senha, etc.
- **hibernate**
  - Configurações de cache do Hibernate
- **environments**
  - Configuração individual de cada ambiente disponível
    - **development** - Configurações de banco usadas enquanto o sistema está sendo desenvolvido
    - **test** - Configurações de banco usadas para os testes de integração
    - **production** - Configurações de banco usada quando é feito o deploy do sistema para algum servidor de aplicação/servidor web;

Veja um exemplo: *(essa é a forma que o arquivo é gerado por padrão, quando você cria sua aplicação Grails)*



```
dataSource {
    pooled = true
    driverClassName = "org.hsqldb.jdbcDriver"
    username = "sa"
    password = ""
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.provider_class = 'net.sf.ehcache.hibernate.EhCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop', 'update'
            url = "jdbc:hsqldb:mem:devDB"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:file:prodDb;shutdown=true"
        }
    }
}
```

O arquivo DataSources.groovy deve ser editado de acordo com as configurações que você deseja para seu sistema.

#### Prática 14

// Alterando o DataSources.groovy

### 3.8 - Testes de Integração

Existem vários tipos de testes que são classificados como testes de integração. Em suma o que define um teste de integração é o fato de ele executar mais de um módulo do sistema. Para nossos testes de unidade, por exemplo, a única coisa testada em cada teste é alguma

característica de nossas classes de domínio. Nos testes de integração devemos testar, como o próprio nome diz, a integração de componentes do sistema, e não apenas uma parte isolada.

Uma destas opções para testes de integração são os testes que envolvem acesso ao banco de dados. Vamos fazer um pouco destes testes para que possamos validar, e experimentar, tudo que vimos em relação à camada de persistência até agora.

### **Prática 15**

## Capítulo 04: Camada de Serviços

A essa altura de nosso treinamento nos deparamos com um assunto muito polêmico: O que deve ser colocado na camada de serviços?

Existem arquiteturas em que as regra de negócio são codificadas através de serviços, uma delas é a que citamos anteriormente no tópico *Camada de Domínio*, a arquitetura BO-LO-VO.

Para um desenvolvimento guiado pelo domínio (DDD), a tarefa desta camada é extremamente reduzida. Pense nesta camada como uma camada 'fina', algo que acrescenta alguns detalhes ao comportamento de nosso sistema, mas que não contém código para a implementação dos requisitos funcionais de nosso sistema.

A própria documentação do Grails nos diz para colocar lógica de negócio nos serviços, ao invés de colocá-la nos controladores de tela. Porém no nosso estudo nem os serviços e nem os controladores conterão lógica de negócio, apenas os objetos de domínio.

Para nosso estudo essa camada será responsável por adicionar aspectos à nossa aplicação corporativa.

### 4.1 - O que são aspectos?

Aspectos são comportamentos que adicionamos às chamadas das operações do nosso sistema, e que não estão dentro das regras de negócio do mesmo. Isto é, os aspectos não implementam a lógica de negócio, eles apenas auxiliam algumas características da lógica de aplicação.

### 4.2 - Como criar meus serviços?

Primeiramente você deve identificar quais serviços serão necessários para a nossa aplicação. É importante reiterar que os serviços são identificados pelos requisitos não funcionais, e não pelos requisitos funcionais.

Suponhamos que nossa deva enviar um email para o administrador do sistema sempre que alguma operação de remoção de dados for feita na aplicação. De acordo com esse requisito não funcional, devemos criar um serviço para a remoção de dados, e após a remoção enviar um email:

Esse serviço foi criado pelo comando **grails create-service RemovedorDeDados**. O sufixo **Service** é adicionado pelo Grails, faz parte da convenção de nomes. Esse serviço não precisa ser instanciado pelos controladores da nossa aplicação (ver Camada de Apresentação), pois são injetados pelo Grails, basta declarar um objeto com o mesmo nome do serviço, seguindo o padrão Lower Camel Case.

Veja nosso serviço sendo usado em um controlador:

Vejam que no código do serviço existe uma propriedade estática declarada como *true*, ***transactional***. Essa configuração serve para dizer ao Grails que todas operações deste serviço serão realizadas em escopo transacional, isto é, caso ocorra algum problema as alterações no banco serão revertidas, caso ocorra tudo bem, elas serão *comitadas* no banco de dados.

Os serviços do Grails são criados no diretório **grails-app/services**, e eles são injetados, da forma como foi exemplificado a cima, tanto em controladores quanto em classes de domínio e também em outros serviços.

Já ia me esquecendo de um detalhe importantíssimo: **Serviços não guardam estado!** Isso é uma condição *sine qua non* para nossa arquitetura.

O Grails nos permite definir uma série de escopos para os serviços. Esses escopos estão relacionados ao ciclo de vida destes serviços. Exemplo: é criado um serviço por sessão do banco, é criado um serviço por chamada, dentre outros.

Como faremos o uso mais simples possível dos serviços, para nosso caso iremos usar o escopo padrão, isto é, *singleton*, que cria uma única instância do serviço para todo ciclo de vida da aplicação.

## Prática 16

### // Criar alguns serviços

## Capítulo 05: Camada de Apresentação

Já fizemos bastante coisa, porém você deve estar bastante ansioso para ver a aplicação funcionando, eu estou! Falta pouco para termos esse primeiro contato, então vamos a ele.

O grails possui uma excelente ferramenta de *scaffolding*, que acelera muito o tempo de construção de nossas telas. Scaffold é um procedimento que irá gerar suas telas, em GSP, e também seus controladores, seguindo o padrão MVC.

As telas geradas atendem às operações CRUD (create, retrieve, update, delete) para todos nossos objetos de domínio. Para compreender melhor suponha que seu sistema tenha duas classes: Livro e Autor. Com esse exemplo seriam geradas as seguintes telas:

- Tela de cadastro de livros; (*Create*)
- Tela de edição de livros; (*Update*)
- Tela de exibição de dados do livro; (*Retrieve, Delete*)
- Tela de listagem de livros: (*Retrieve, Delete*)
- E todas as telas acima também para autores;
- As telas para livro são criadas no diretório **grails-app/views/livro** e as telas para autor em **grails-app/views/autor**;

É criado um único controlador para cada objeto de domínio, nele são tratadas todas operações de CRUD, para este caso teríamos o LivroController, e o AutorController, ambos estariam dentro do diretório **grails-app/controllers**.

Para gerar as telas e controladores de nossos objetos de domínio usaremos o comando **grails generate-all nomeDaClasseCompleto**. (*Os IDEs possuem atalhos e opções de menu para executar todos comandos do Grails*)

Para criar essas telas o Grails usa novamente a convenção sobre configuração. Nós não precisamos dizer a ele como queremos nossa tela, pois as telas de CRUD são muito semelhantes em aplicações corporativas, por isso ele tem condição de criá-las para nós.

A definição de como a tela será criada é na verdade a definição da própria camada de domínio, se sua classe tem uma propriedade texto, ele cria um controle de texto, se tem uma propriedade data, ele cria um controle de data, e assim por diante. Novamente notamos que toda construção do nosso sistema parte das classes de domínio, é o DDD em ação, lembra-se?

Nesta introdução, além de falarmos sobre Scaffolding, falamos também de MVC e de GSP, vamos agora estudar esses assuntos com mais detalhes.

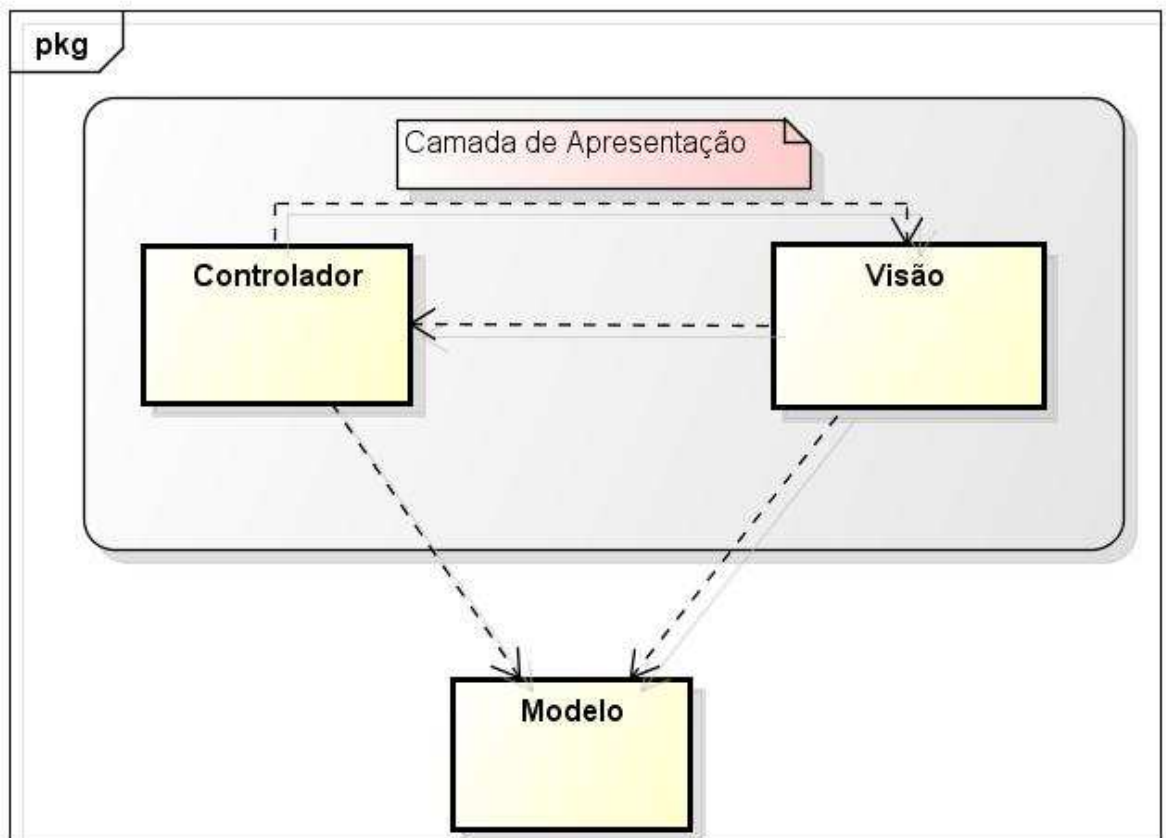
### 5.1 – Model View Controller (MVC)

O padrão MVC é uma proposta para organização de software que tenham interface com usuário. Esta organização sugere que seu sistema seja separado em três partes:

- **Modelo** – Corresponde ao código de regras de negócio do sistema, no nosso caso pode ser um objeto de domínio ou um serviço, que por sua vez irá usar os objetos de domínio para executar as tarefas. O modelo não contém código visual, ele não sabe como os dados serão exibidos ou como o usuário irá interagir com o sistema;

- **Visão** – A visão é formada por componentes que exibem dados ao usuário e que capturam as operações vindas dele. Para um sistema web, por exemplo, a visão é construída com controles HTML, JavaFX, Silverlight, Java Script, Flex, etc. A única missão da visão é a de exibir dados ao usuário e fornecer alguma maneira para que ele interaja com o sistema (através de comandos, botões, etc.). A visão não sabe como o trabalho será feito, ela só sabe que o usuário deseja realizar a tarefa em questão.
- **Controlador** – O controlador é o elo de ligação entre o modelo e a visão. Para cada solicitação do usuário, é o controlador que conhece como repassar esse pedido ao modelo. E dada a resposta do modelo, é o controlador que sabe como repassar isso de volta para a visão.

Vejamos o MVC em um diagrama, para facilitar nossa visualização e compreensão do mesmo:



powered by astah®

Note que o controlador e a visão estão em uma região destacada. Essa separação foi feita para evidenciar que a camada de apresentação é formada pelo controlador e pela visão, enquanto o modelo é constituído pelos serviços, pelo domínio e pela persistência da aplicação.

Para reforçar esses conceitos vamos olhara para cada um destes componentes do MVC do ponto de vista de responsabilidades:

Componente	Responsabilidades
Modelo	<i>Executar as políticas de negócios que irão satisfazer as solicitações do usuário.</i>
Visão	<i>Exibir os dados do domínio de forma visual para o usuário (por isso ele tem uma dependência com o modelo, mais especificamente com a camada de domínio). Receber os comandos do usuário e repassá-los ao controlador. Validar os dados inseridos pelo usuário. Exibir de forma visual a resposta do sistema em relação aos estímulos do usuário. A visão é burra, ela não toma decisões, apenas repassa pedidos e exibe dados da forma que o controlador determinar.</i>
Controlador	<i>Controlar o fluxo das telas. Decidir, dada uma solicitação vinda da tela, com o modelo deverá tratá-la. Decidir, de acordo com a resposta do modelo, como a tela deverá se comportar.</i>

## 5.2 – Controladores

Quando fazemos o comando **grails generate-all nomeDaClasseDeDominio**, tanto o controlador quanto as views em GSP são gerados, porem você tem a opção de criar apenas o controlador com o comando **grails generate-controller nomeDaClasseDeDominio**.

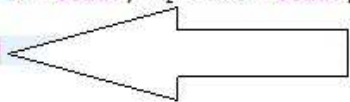
A primeira coisa que podemos compreender nos controladores são as ações. Elas são métodos que são invocados pelas nossas telas da visão, as telas em GSP, ou seja, para cada interação que usuário pode realizar sobre o sistema uma ação de algum controlador deve ser definida.

O comportamento dos controladores, e de suas ações, são definidos pela nossa tão batida convenção sobre configuração. O acesso a eles é dado pela seguinte convenção:

- **UrlDaAplicacao/nomeDoControlador** - para acesso ao controlador, este acesso irá invocar a ação *default* do controlador;
- **UrlDaAplicacao/nomeDoControlador/nomeDaAcao** - para chamar uma ação específica de um controlador;
- Obs.: **nomeDoControlador** - representa o nome do controlador sem a extensão **Controller**
  - Ex.: Se o controlador se chama LivroController, uma das urls seria <http://minhaapp.com.br/livro/list>

A ação default dos controladores é a *index*, caso você deseja explicitar qual deve ser a ação default adicione uma propriedade a seu controlador, conforme o exemplo abaixo:

```
class EventoEsportivoController {  
    static allowedMethods = [save: "POST", update: "POST", delete: "POST"]  
    def defaultAction = "create"  
    def index = {  
        redirect(action: "list", params: params)  
    }  
    def list = {  
        params.max = Math.min(params.max ? params.int('max') : 10, 100)  
        [eventoEsportivoInstanceList: EventoEsportivo.list(params), eventoEsportivoIr  
    }  
    def create = {  
        def eventoEsportivoInstance = new EventoEsportivo()  
        eventoEsportivoInstance.properties = params  
        return [eventoEsportivoInstance: eventoEsportivoInstance]  
    }  
}
```



Veja na ação *list* que na primeira linha de código a propriedade *params* é acessada. É através desta propriedade que a tela passa os dados para o controlador. Essa propriedade é um *map* que contém todas as informações necessárias para o processamento de uma requisição vinda da tela.

O controlador realiza basicamente dois tipos de comandos das telas. Um deles são as mensagens de flash. Essas mensagens são usadas para dar informações ao usuário sobre a operação que ele realizou, para mensagens de erro e de sucesso, e assim por diante. Veja um exemplo, com o trecho de código abaixo:

```
def show = {  
    def eventoEsportivoInstance = EventoEsportivo.get(params.id)  
    if (!eventoEsportivoInstance) {  
        flash.message = "${message(code: 'default.not.found.message', args: [message  
        redirect(action: "list")  
    }  
    else {  
        [eventoEsportivoInstance: eventoEsportivoInstance]  
    }  
}
```





Para simular esse erro eu digitei a seguinte url no browser:

- <http://localhost:8080/inscricoesesportivas/eventoesportivo/show/2>

Como não existia um evento esportivo com id = 2, então meu controlador configurou a mensagem de flash, que foi exibida conforme a imagem acima, e logo após isso o controlador direcionou o fluxo de tela para a ação de listagem.

Note que a mensagem está em inglês. Essa é uma mensagem padrão de objeto não encontrado, você pode retornar a mensagem que quiser. O Ideal é continuar retornando as mensagens padrão, porém você deve alterar as configurações para que o grails use um arquivo com as versões em português destas mensagens. A manipulação de todas configurações serão vistas no capítulo 05.

Além de informar essas mensagens para que as telas possam exibir, nas ações dos nossos controladores podemos também *reenderizar* o fluxo para outra tela ou então redirecionar a chamada para outra ação.

Para repassar a chamada para outra ação usamos o *redirect*, e para exibir uma nova tela usamos o *render*. Veja os dois casos no trecho de código abaixo:

```
def save = {
    def eventoEsportivoInstance = new EventoEsportivo(params)
    if (eventoEsportivoInstance.save(flush: true)) {
        flash.message = "${message(code: 'default.created.message', args: [message(code: '
        redirect(action: "show", id: eventoEsportivoInstance.id)
    }
    else {
        render(view: "create", model: [eventoEsportivoInstance: eventoEsportivoInstance])
    }
}
```

O primeiro caso é um redirecionamento para outra ação, deste mesmo controlador. O método invocado para realizar esta tarefa é o *redirect*, os parâmetros passados são a ação e, neste caso um *id*. Este id será acessado na ação show, através de *params.id*.

O segundo caso, o *else* do código acima, ocorre uma mudança de página. Através do método *render* nós solicitamos que seja exibida a tela de criação. Neste caso passamos também um objeto de domínio para que a tela seja preenchida, no item 5.3 veremos como as telas trabalham com esse objeto.

### 5.3 – Grails Server Pages (GSP)

[TODO]

### 5.4 – Customizando nossa interface gráfica

[TODO]

#### 5.4.1 – Questões de Usabilidade

[TODO]

#### 5.4.2 – Redirecionamento de Páginas

[TODO]

## Capítulo 06: Configurações do Grails

### 6.1 – BootStrap.groovy

//[TODO]

### 6.2 – BuildConfig.groovy

//[TODO]

### 6.3 – DataSources.groovy

//[TODO]

### 6.4 – UrlMappings.groovy

//[TODO]

### 6.5 – Fazendo o deploy de uma aplicação Grails

//[TODO]

### 6.6 – Configurações de Idioma

//[TODO]

### 6.7 – Customizando controles GSP com as taglibs

//[TODO]

### 6.8 – Scripts de customização de Builds

//[TODO]