Testes com objetos mock

Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Análise e Projeto Orientados a Objetos

Introdução

- Testes são essenciais para garantir a qualidade do código.
- No entanto:
  - Escrever testes de unidade pode se tornar um desafio devido ao código de preparação que deve ser escrito: inicialização de objetos, povoamento do banco de dados, comunicação remota, classes colaboradoras, ...
  - A medida que a quantidade de testes cresce, a sua execução pode levar um tempo considerável devido à comunicação com o banco de dados ou outros serviços externos.
- A principal razão dos testes de unidade serem difíceis de escrever e lentos de executar é por causa de suas classes colaboradoras.
  - Esse é um fato relevante em programas fortemente dependentes de sistemas de banco dados. É comum que as regras de negócio dependam de informações armazenadas.

Introdução

- A colaboração é geralmente um elemento bom e necessário, pois mantém a coesão da classe. No entanto, pode acarretar em alguns problemas:
  - Ao implementar a classe a ser testada, o desenvolvedor deve pensar e escrever os detalhes de todos os colaboradores envolvidos.
  - Criar e inicializar os colaboradores torna os testes de uma classe mais complicados pois alguns objetos requerem inicialização complexa até atingirem o estado correto para o teste.
  - Colaboradores introduzem acoplamento indesejado. Por exemplo, utilizar implementações reais de DAOs acoplaria o modelo de domínio ao banco de dados e nos forçaria a resolver questões de persistência.

Benefícios ao utilizar mock objects

- Testes de unidade devem ser isolados dos agentes externos ao código que está sendo testado.
  - O objetivo é testar o método em questão, e não os objetos externos.
- Mock objects permitem o isolamento dos testes de unidade fornecendo implementações falsas dos objetos colaboradores da unidade em teste.
- Quando utilizado com TDD, mock objects permitem que o desenvolvedor concentre-se na implementação do código em questão, livrando-o de escrever o código dos colaboradores.

Mockito

- Existem vários frameworks em Java para facilitar a geração de mock objects
  - **Mockito**, JMock, EasyMock, ...
- Vantagens do Mockito
  - Escrita clara.
  - API simples.
  - Permite a criação de mocks a partir de classes concretas.

Preparando o ambiente

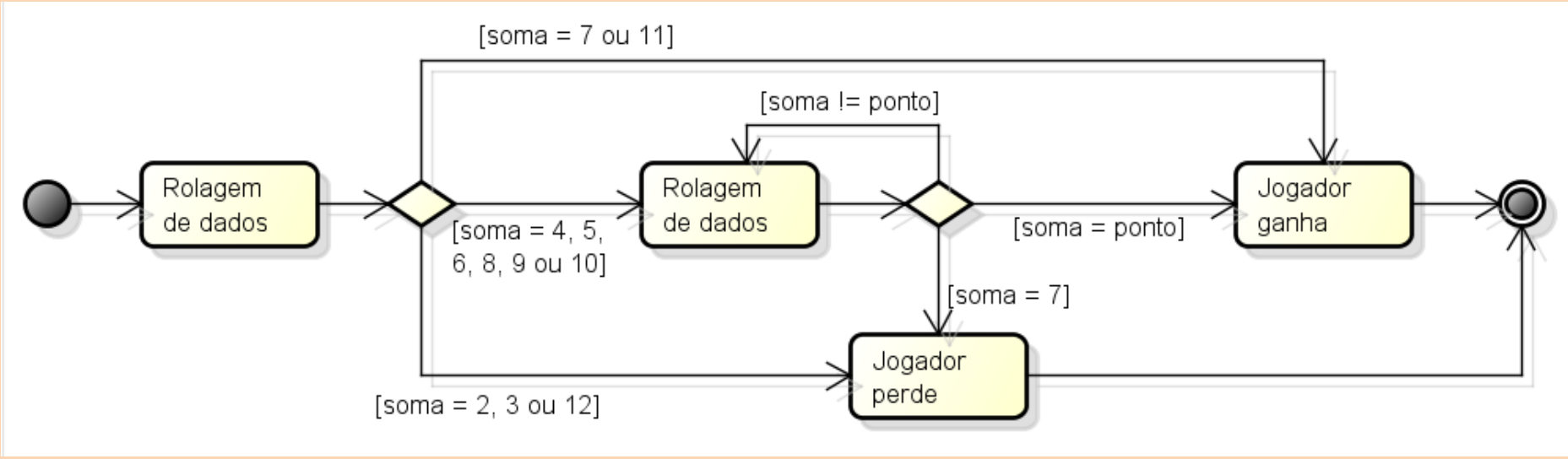
- Criar projeto no Eclipse.
- Adicionar a biblioteca JUnit 4.
- Copiar o arquivo **mockito-all-1.9.0.jar** para o projeto e adicioná-lo ao classpath.
- Criar uma source folder chamada **test**. As classes de teste devem ser criadas nesta pasta.

EXEMPLO 1 – JOGO CRAPS

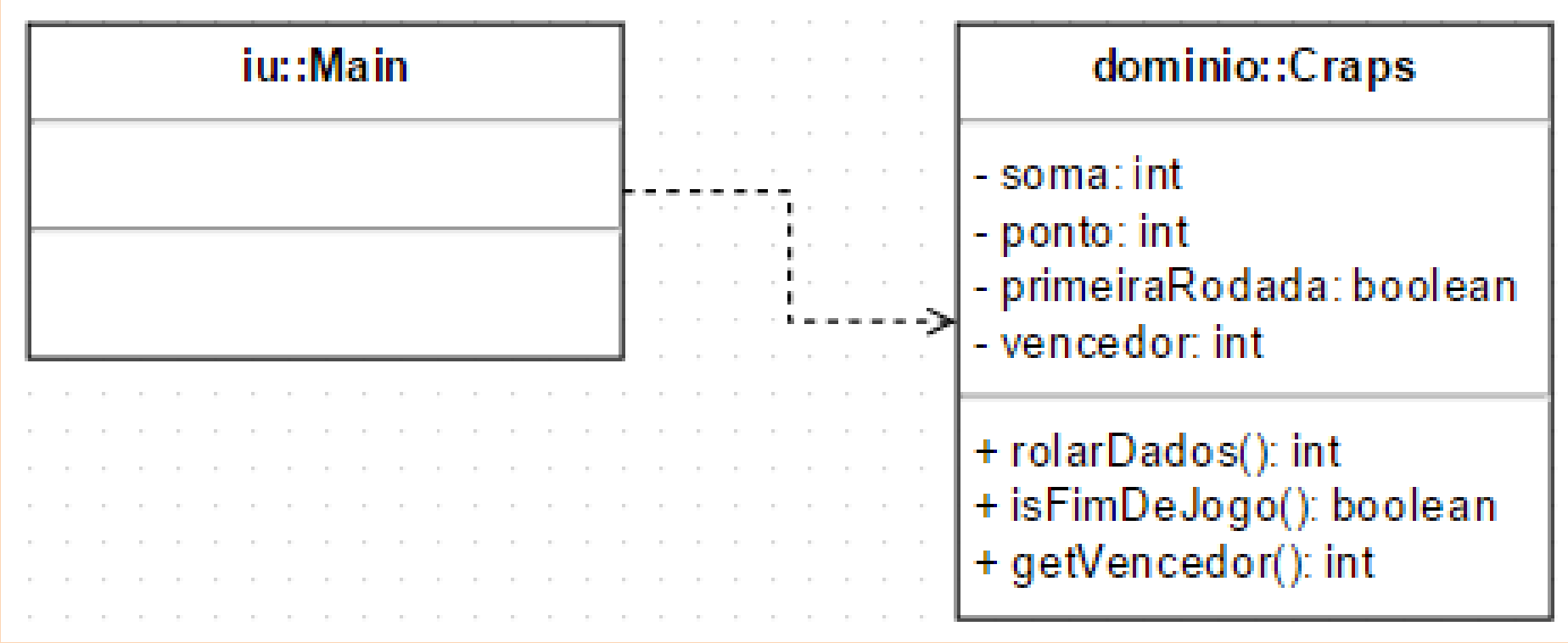
Introdução

- Este é um exemplo simples para demonstrar o uso do Mockito e de objetos mock.
- Craps (jogo de dados):
  - O jogador lança dois dados de seis faces. Depois que os dados param de rolar, calcula-se a soma dos pontos obtidos nos dois dados. Se a soma for 7 ou 11 no primeiro lance, o jogador ganha. Se a soma for 2, 3 ou 12 no primeiro lance, o jogador perde (a mesa ganha).
  - Se a soma for 4, 5, 6, 8, 9 ou 10 no primeiro lance, essa soma torna-se o ponto do jogador. Para ganhar, o jogador deve continuar a rolar os dados até atingir seu ponto (isto é, a soma deve ser igual ao ponto do jogador).
  - O jogador perde se obtiver um 7 antes de atingir seu ponto.

Fluxo do jogo



Projeto – visão estática



Fluxo da aplicação

```
public class Main{
 public static void main(String[] args){
  Craps craps = new Craps();
  Scanner scan = new Scanner(System.in);
   while(!craps.isFimDeJogo()){
   System.out.println("Digite enter para continuar.");
   scan.nextLine(); int soma = craps.rolarDados();
   System.out.println("Soma: " + soma);
  }
  scan.close();
  if(craps.getVencedor() == 1)
   System.out.println("Você ganhou!");
  else if(craps.getVencedor() == 2)
   System.out.println("A banca ganhou!");
  else
   throw new IllegalStateException();
 }
}
```

Classe **Craps** (1)

```
import java.util.Random;
public class Craps{
 private int soma, ponto, vencedor;
 private boolean primeiraRodada = true;
 private Random rand = new Random();
 private int rolarUmDado(){
  return rand.nextInt(6) + 1;
 }
 public boolean isFimDeJogo(){
  return vencedor == 1 || vencedor == 2;
 }
 public int getVencedor(){
  return vencedor;
 }
```

Classe **Craps** (2)

```
 public int rolarDados(){
  soma = rolarUmDado() + rolarUmDado();
  if(primeiraRodada){
   if(soma == 7 || soma == 11)
    vencedor = 1;
   else if(soma == 2 || soma == 3 || soma == 12)
    vencedor = 2;
   else
    ponto = soma;
   primeiraRodada = false;
  }
  else{
   if(soma == ponto)
    vencedor = 1;
   else if(soma == 7)
    vencedor = 2;
  }
  return soma;
 }
}
```

Teste 1: jogador perde na primeira rolagem de dados

```
package craps.dominio;
import static org.junit.Assert.*;
import org.junit.Test;

public class CrapsTest {
 @Test
 public void testRolarDados_1(){
  Craps craps = new Craps();
  craps.rolarDados();
  assertTrue(craps.isFimDeJogo());
  assertEquals(2, craps.getVencedor());
 }
}
```

Problema com o teste 1

- Como garantir a obtenção de 2, 3 ou 12 na primeira rolagem já que a geração dos números é aleatória?
- Solução: substituir o objeto gerador de números aleatórios (**rand**) por um objeto mock.
  - Para tal, é necessário implementar o método **setRand** na classe **Craps**.

Teste 1 usando objeto mock

```
@Test
public void testRolarDados_1(){
 Random randMock = Mockito.mock(Random.class);
 //notar que são feitas duas rolagens de dado
 Mockito.when(randMock.nextInt(6)).thenReturn(0);
 Craps craps = new Craps();
 craps.setRand(randMock); craps.rolarDados();
 assertTrue(craps.isFimDeJogo());
 assertEquals(2, craps.getVencedor());
}
```

Teste 1 usando objeto mock

Criação do objeto mock.

```
@Test
public void testRolarDados_1(){
 Random randMock = Mockito.mock(Random.class);
 //notar que são feitas duas rolagens de dado
 Mockito.when(randMock.nextInt(6)).thenReturn(0);
 Craps craps = new Craps();
 craps.setRand(randMock); craps.rolarDados();
 assertTrue(craps.isFimDeJogo());
 assertEquals(2, craps.getVencedor());
}
```

Teste 1 usando objeto mock

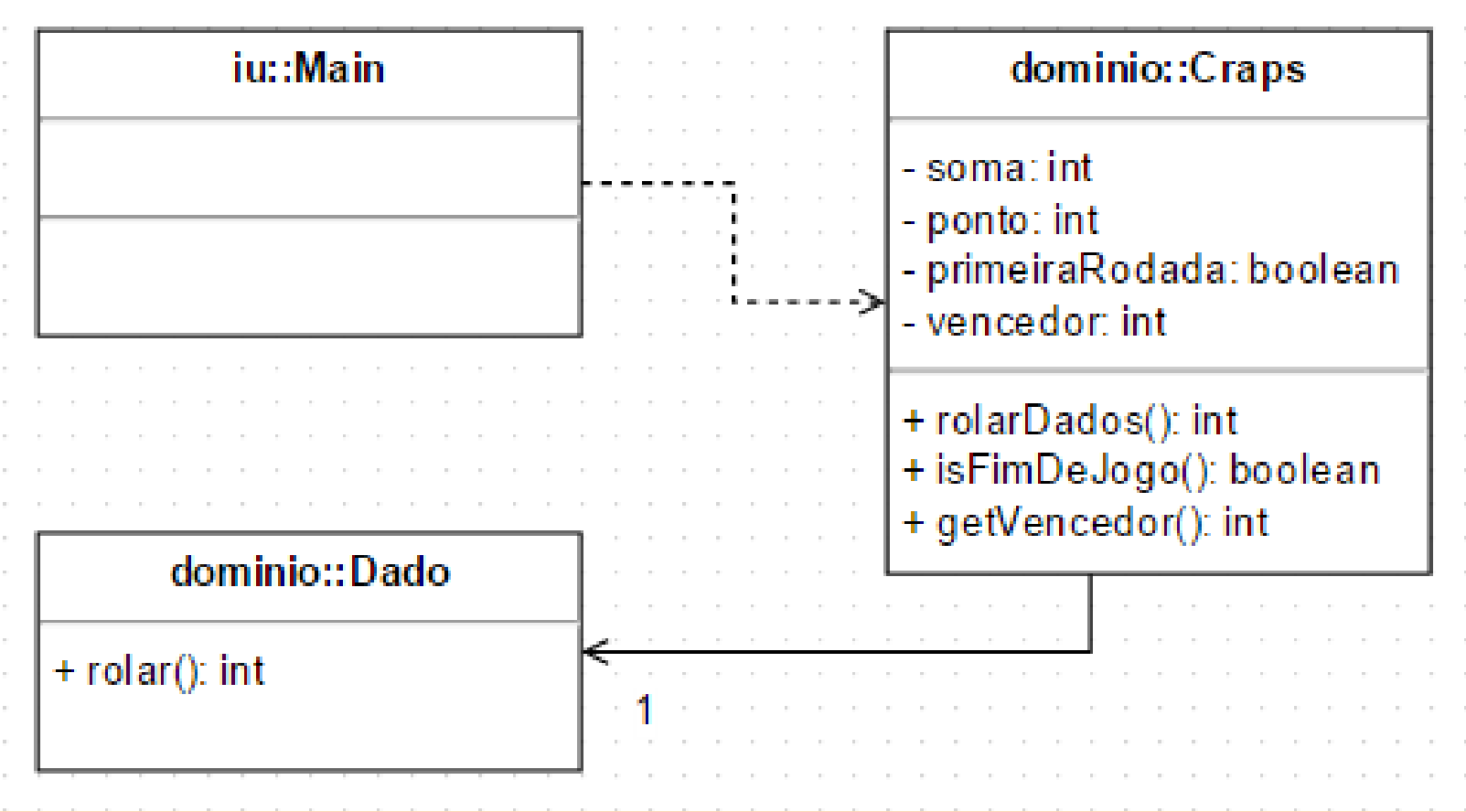
```
@Test
public void testRolarDados_1(){
 Random randMock = Mockito.mock(Random.class);
 //notar que são feitas duas rolagens de dado
 Mockito.when(randMock.nextInt(6)).thenReturn(0);
 Craps craps = new Craps();
 craps.setRand(randMock); craps.rolarDados();
 assertTrue(craps.isFimDeJogo());
 assertEquals(2, craps.getVencedor());
}
```

Configuração de comportamento do objeto mock. Neste caso, foi definido que as chamadas ao método **nextInt** retornem zero como resultado. Note que a configuração do mock foi realizada com base no código do método **rolarUmDado** da classe **Craps** para que a soma resultante seja igual a dois.

Melhorando o projeto

- A configuração de comportamento do mock foi baseada no código do método **rolarUmDado** da classe **Craps**. Além de ferir o encapsulamento, a configuração do mock não ficou intuitiva.
- Vamos melhorar o projeto movendo a geração dos números aleatórios para a classe **Dado**. Esta mudança melhora a coesão da classe **Craps** e melhora a legibilidade do teste.

Visão estática atualizada



Classe **Dado**

```
package craps.dominio;
import java.util.Random;

public class Dado {
 private Random rand = new Random();
 public int rolar(){
  return rand.nextInt(6) + 1;
 }
}
```

Classe **Craps** atualizada (1)

```
public class Craps {
 private int soma, ponto, vencedor;
 private boolean primeiraRodada = true;
 private Dado dado = new Dado();
 public boolean isFimDeJogo(){
  return vencedor == 1 || vencedor == 2;
 }
 public int getVencedor(){
  return vencedor;
 }
 void setDado(Dado novoDado) {
  this.dado = novoDado;
 }
```

Classe **Craps** atualizada (2)

```
 public int rolarDados(){
  soma = dado.rolar() + dado.rolar();
  if(primeiraRodada){
   if(soma == 7 || soma == 11)
    vencedor = 1;
   else if(soma == 2 || soma == 3 || soma == 12)
    vencedor = 2;
   else
    ponto = soma;
   primeiraRodada = false;
  }
  else{
   if(soma == ponto)
    vencedor = 1;
   else if(soma == 7)
    vencedor = 2;
  }
  return soma;
 }
}
```

Teste 1 atualizado

```
@Test
public void testRolarDados_1(){
 Dado dadoMock = Mockito.mock(Dado.class);
 //notar que são feitas duas rolagens de dado
 Mockito.when(dadoMock.rolar()).thenReturn(1);
 Craps craps = new Craps();
 craps.setDado(dadoMock);

 craps.rolarDados();

 assertTrue(craps.isFimDeJogo());
 assertEquals(2, craps.getVencedor());
}
```

Teste 2: jogador ganha na primeira rolagem

```
@Test
public void testRolarDados_2(){
 Dado dadoMock = Mockito.mock(Dado.class);
 //notar que são feitas duas rolagens de dado
 Mockito.when(dadoMock.rolar()).thenReturn(5, 2);
 Craps craps = new Craps();
 craps.setDado(dadoMock);

 craps.rolarDados();

 assertTrue(craps.isFimDeJogo());
 assertEquals(1, craps.getVencedor());
}
```

Teste 2: jogador ganha na primeira rolagem

```
@Test
public void testRolarDados_2(){
 Dado dadoMock = Mockito.mock(Dado.class);
 //notar que são feitas duas rolagens de dado
 Mockito.when(dadoMock.rolar()).thenReturn(5, 2);
 Craps craps = new Craps();
 craps.setDado(dadoMock);

 craps.rolarDados();

 assertTrue(craps.isFimDeJogo());
 assertEquals(1, craps.getVencedo
}
```

Configuração do seguinte comportamento: retorne, respectivamente, 5 e 2 na primeira e segunda chamada ao método **rolar**.

Teste 3: jogador perde na terceira rolagem

```
@Test
public void testRolarDados_3(){
 Dado dadoMock = Mockito.mock(Dado.class);
 Craps craps = new Craps();
 craps.setDado(dadoMock);

 Mockito.when(dadoMock.rolar()).thenReturn(3, 5);
 craps.rolarDados();
 assertFalse(craps.isFimDeJogo());

 Mockito.when(dadoMock.rolar()).thenReturn(6, 5);
 craps.rolarDados();
 assertFalse(craps.isFimDeJogo());

 Mockito.when(dadoMock.rolar()).thenReturn(5, 2);
 craps.rolarDados();
 assertTrue(craps.isFimDeJogo());
 assertEquals(2, craps.getVencedor());
}
```

Teste 4: jogador ganha na segunda rolagem

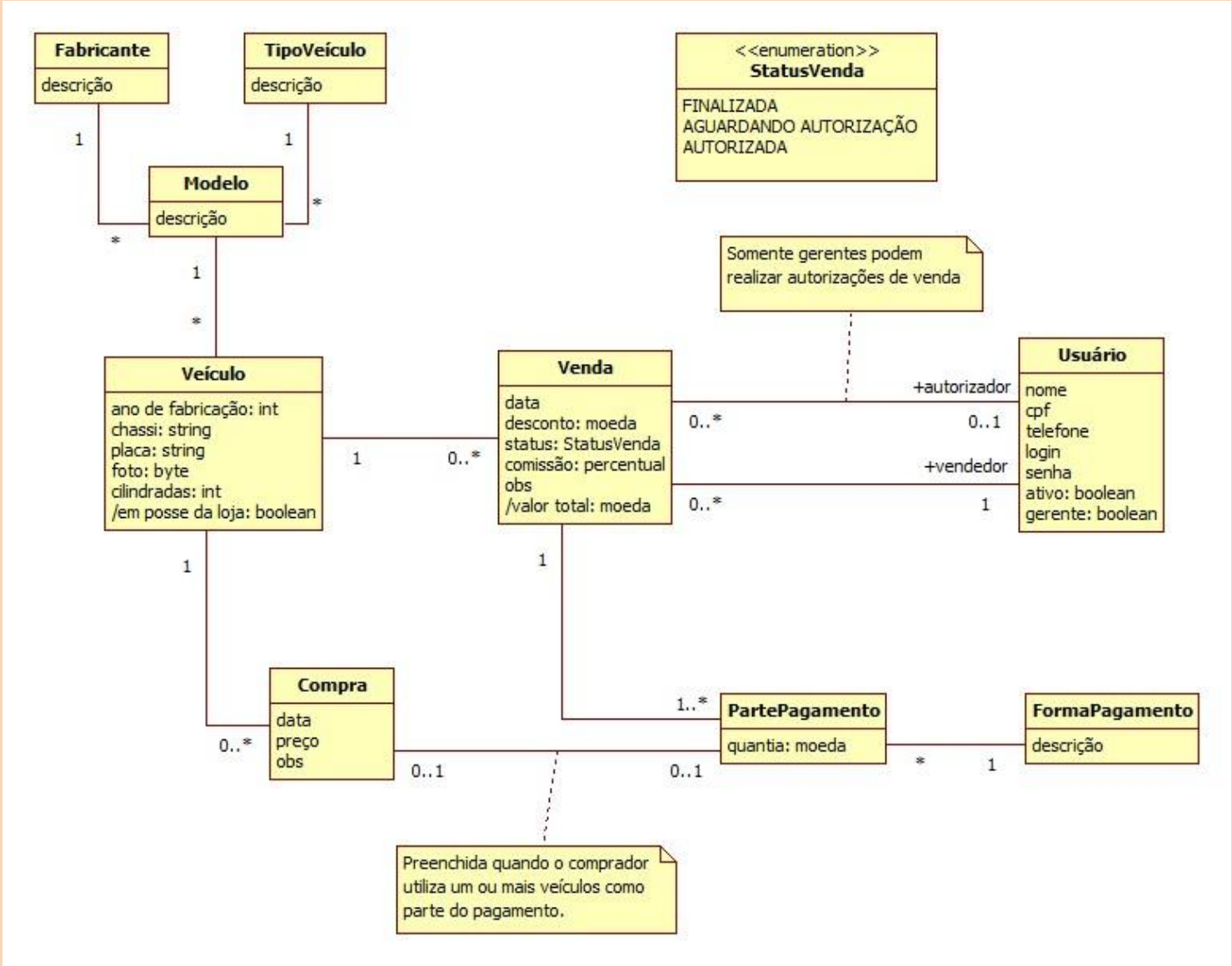
```
@Test
public void testRolarDados_4(){
 Dado dadoMock = Mockito.mock(Dado.class);
 Craps craps = new Craps();
 craps.setDado(dadoMock);

 Mockito.when(dadoMock.rolar()).thenReturn(1, 3);
 craps.rolarDados();
 assertFalse(craps.isFimDeJogo());

 Mockito.when(dadoMock.rolar()).thenReturn(2, 2);
 craps.rolarDados();
 assertTrue(craps.isFimDeJogo());
 assertEquals(1, craps.getVencedor());
}
```

EXEMPLO 2 – SISTEMA DE REVENDA DE VEÍCULOS

Exemplo

- Contexto: sistema exemplo de revenda de veículos – caso de uso *Registrar Compra*.
- **Fluxo básico:**
  1. Usuário (gerente ou vendedor) informa a placa do veículo.
  2. Sistema verifica que o veículo está cadastrado e não está em posse da loja. Em seguida, sistema exibe os dados do veículo.
  3. Usuário informa dados da compra: valor da compra, data da compra, forma de pagamento e observações.
  4. Usuário confirma a operação, sistema registra a compra e o caso de uso termina.
- **Fluxo Alternativo (2):** o veículo está cadastrado no sistema e está em posse da loja.
  - Sistema informa o usuário de que a operação não é permitida, pois o veículo já está em posse da loja, e o caso de uso termina.

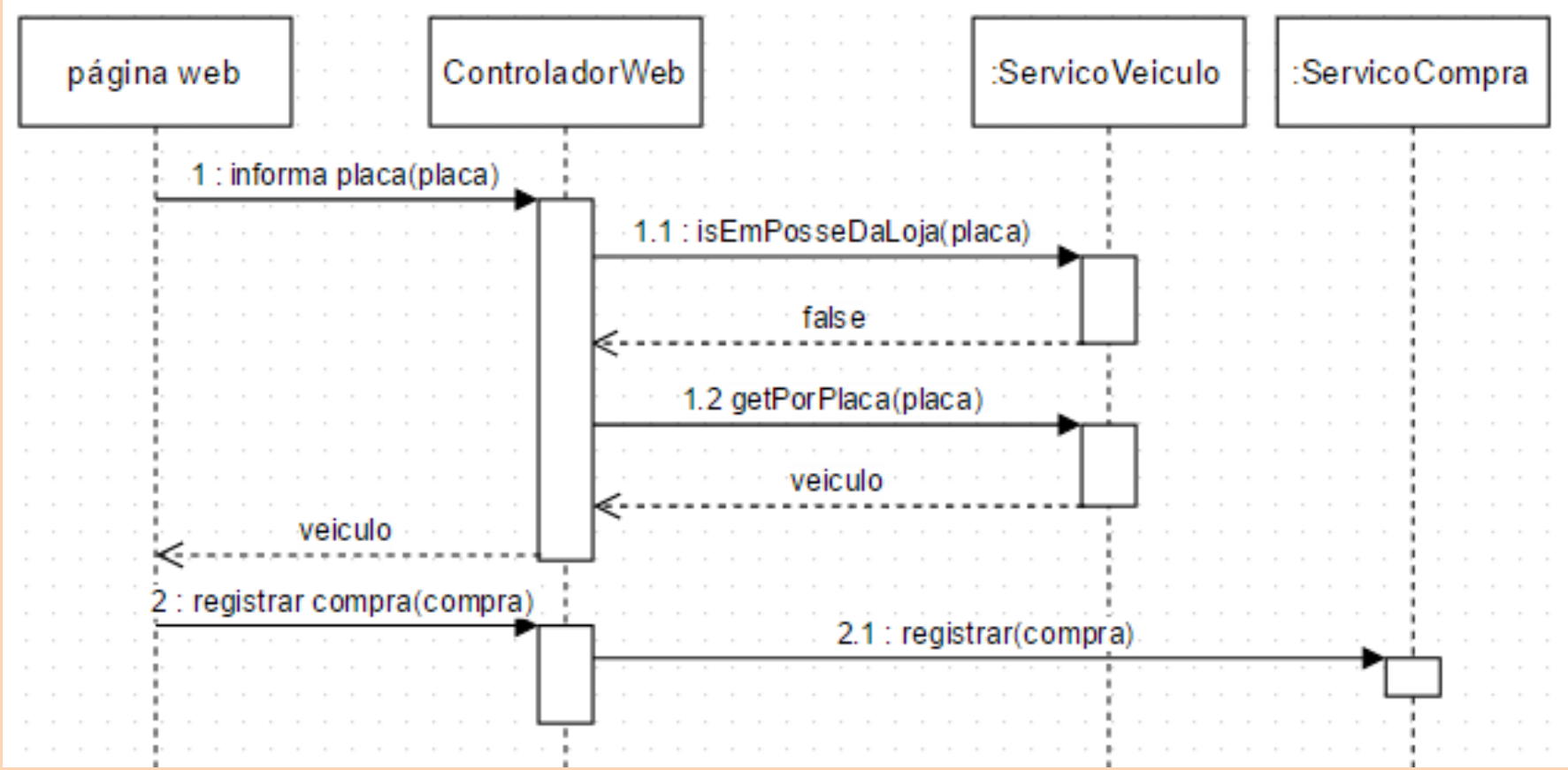
Exemplo - modelo de domínio



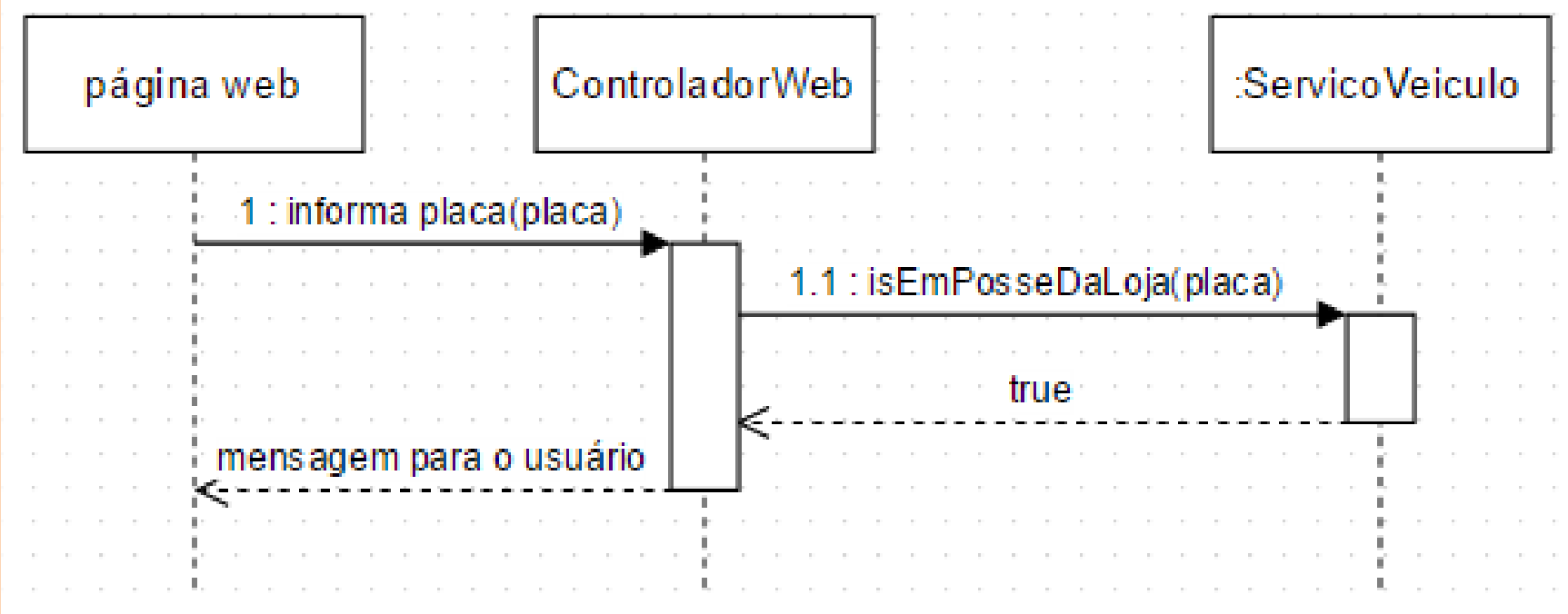
Exemplo – papeis da classes

- **Entidades**: representam conceitos do negócio tais como Compra e Veículo. São tipicamente persistidas em banco de dados.
- **Classes de serviço**: utilizadas pelos controladores web para realizar as operações necessárias aos caso de uso.
- **Interfaces de repositório**: isolam as classes de domínio da implementação de persistência.
- **Classes DAO**: implementação das rotinas de persistência.

Exemplo – projeto de colaboração do fluxo básico



Exemplo – projeto de colaboração do fluxo alternativo

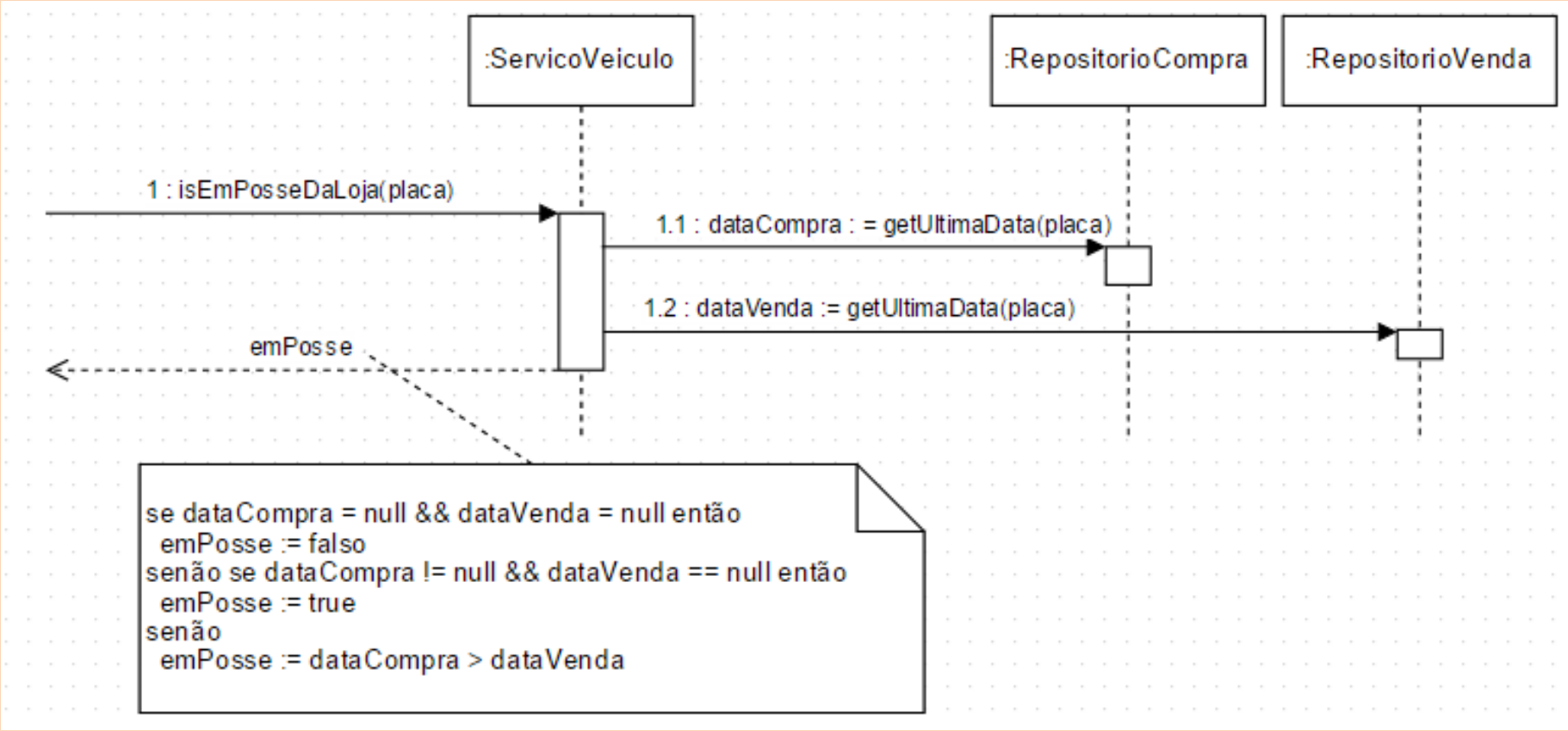


Primeiro teste

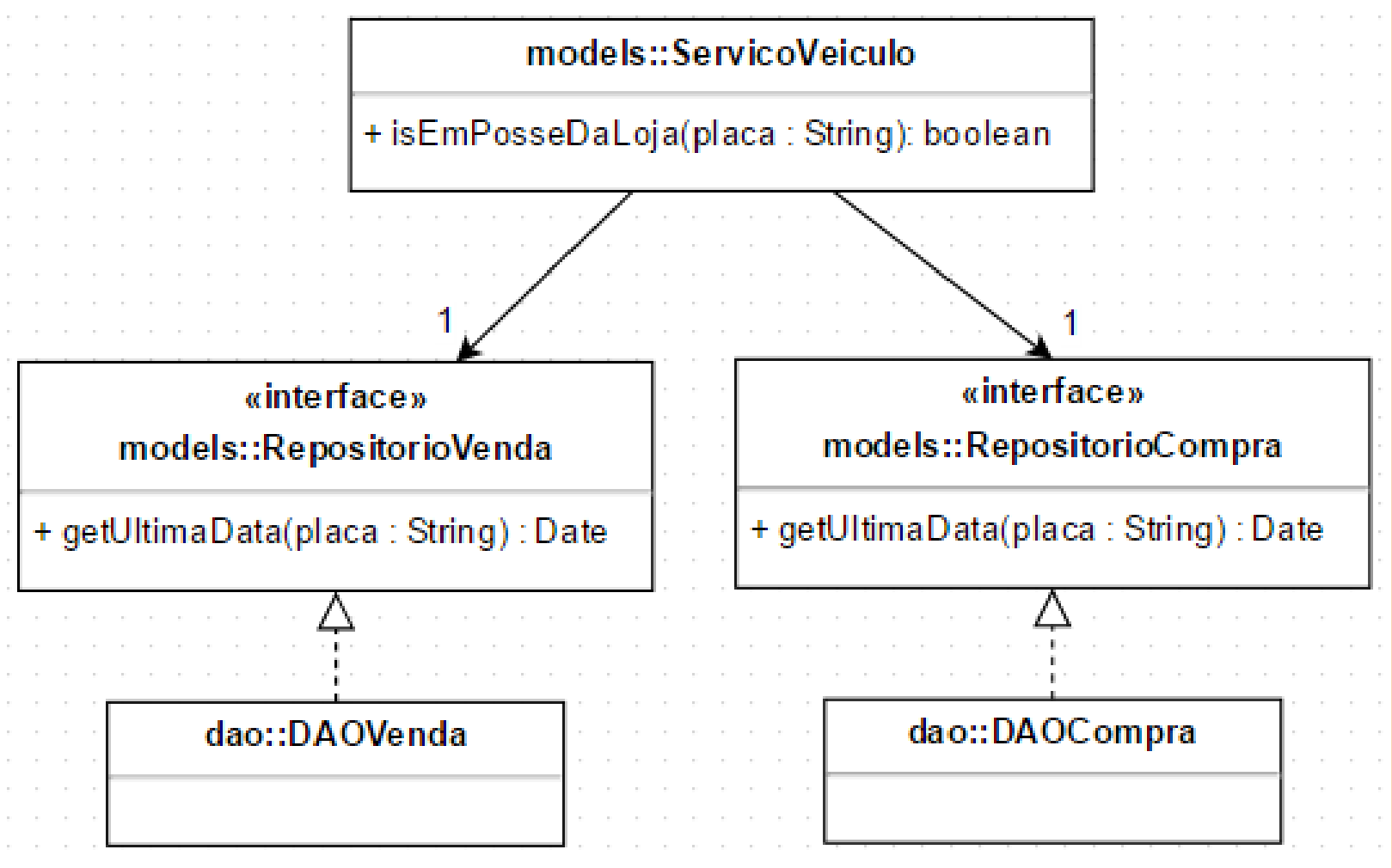
- Vamos começar testando o método **isEmPosseDaLoja** na classe **ServicoVeiculo**.
- A posse do veículo pode ser determinada pela últimas datas de compra e de venda do mesmo:
  - Sem datas: não pertence à loja.
  - Há apenas data da última compra: pertence à loja.
  - Há datas de última compra e de última venda: pertence à loja caso a data da última compra seja posterior à data da última venda.
- É necessário, então, realizar uma consulta ao banco de dados para obter as datas necessárias. Este é uma responsabilidade de **RepositorioCompra** e de **RepositorioVenda**.

Primeiro teste

- Projeto



Primeiro teste – classes



Primeiro teste - **ServicoVeiculo**

```
public class ServicoVeiculo {
 private RepositorioCompra repositorioCompra;
 private RepositorioVenda repositorioVenda;

 public boolean isEmPosseDaLoja(String placa){
  Date ultimaCompra = repositorioCompra.getUltimaData(placa);
  Date ultimaVenda = repositorioVenda.getUltimaData(placa);
  if(ultimaCompra == null && ultimaVenda == null)
   return false;
  else if(ultimaCompra != null && ultimaVenda == null)
   return true;
  return ultimaCompra.after(ultimaVenda);
 }

 public void setRepositorioCompra(RepositorioCompra r){
  this.repositorioCompra = r;
 }
 public void setRepositorioVenda(RepositorioVenda r){
  this.repositorioVenda = r;
 }
}
```

Primeiro teste

- Escrevendo código de teste:
  - **ServiceVeiculo** depende de instâncias de **RepositorioCompra** e **RepositorioVenda**. Ao invés de utilizarmos implementações reais (os DAOs), iremos utilizar objetos mock.
  - Os mocks serão programados para retornar os valores necessários para o funcionamento dos testes.

Primeiro teste – sem data de compra e venda

```
public class ServicoVeiculoTest {

 @Test
 public void testIsEmPosseDaLoja_1(){
  RepositorioCompra rCompra = Mockito.mock(RepositorioCompra.class);
  RepositorioVenda rVenda = Mockito.mock(RepositorioVenda.class);
  ServicoVeiculo servico = new ServicoVeiculo();
  servico.setRepositorioCompra(rCompra);
  servico.setRepositorioVenda(rVenda);

  String placa = "NNX2015";
  Mockito.when(rCompra.getUltimaData(placa)).thenReturn(null);
  Mockito.when(rVenda.getUltimaData(placa)).thenReturn(null);
  boolean emPosse = servico.isEmPosseDaLoja(placa);
  assertFalse(emPosse);
 }
}
```

Primeiro teste – sem data de compra e venda

```
public class ServicoVeiculoTest {

 @Test
 public void testIsEmPosseDaLoja_
  RepositorioCompra rCompra = Moc                    ass);
  RepositorioVenda rVenda = Mocki                    );
  ServicoVeiculo servico = new Se
  servico.setRepositorioCompra(rCompra);
  servico.setRepositorioVenda(rVenda);

  String placa = "NNX2015";
  Mockito.when(rCompra.getUltimaData(placa)).thenReturn(null);
  Mockito.when(rVenda.getUltimaData(placa)).thenReturn(null);
  boolean emPosse = servico.isEmPosseDaLoja(placa);
  assertFalse(emPosse);
 }
}
```

Linha desnecessárias, pois por padrão os métodos de um mock retornam **null**, **zero** ou **false**. O objetivo foi de deixar a intenção do teste clara.

Ampliando o teste

- No teste anterior, testamos a situação em que não há datas de compra ou venda do veículo.
- Agora iremos testar a situação em que há apenas data de compra.
- Já que novamente iremos criar os mocks e configurar **ServicoVeiculo**, faremos uso da anotação **@Before** para evitar a repetição de código.

ServiceVeiculoTest (1)

```
public class ServicoVeiculoTest {

 private RepositorioCompra rCompra;
 private RepositorioVenda rVenda;
 private ServicoVeiculo servico;

 @Before
 public void init(){
  rCompra = Mockito.mock(RepositorioCompra.class);
  rVenda = Mockito.mock(RepositorioVenda.class);
  servico = new ServicoVeiculo();
  servico.setRepositorioCompra(rCompra);
  servico.setRepositorioVenda(rVenda);
 }
```

ServiceVeiculoTest (2)

```
@Test
public void testIsEmPosseDaLoja_1(){
 String placa = "NNX2015";
 Mockito.when(rCompra.getUltimaData(placa))
      .thenReturn(null);
 Mockito.when(rVenda.getUltimaData(placa))
      .thenReturn(null);
 boolean emPosse = servico.isEmPosseDaLoja(placa);
 assertFalse(emPosse);
}
```

ServiceVeiculoTest (3)

```
 @Test
 public void testIsEmPosseDaLoja_2(){
  String placa = "NNX2015";
  Mockito.when(rCompra.getUltimaData(placa))
       .thenReturn(new Date());
  Mockito.when(rVenda.getUltimaData(placa))
       .thenReturn(null);
  boolean emPosse = servico.isEmPosseDaLoja(placa);
  assertTrue(emPosse);
 }
}
```

Ampliando o teste

- Agora inserimos testes para as outras situações em que existem datas de compra e venda.

Teste: última data de venda posterior à última data de compra

```
@Test
public void testIsEmPosseDaLoja_3() throws
  ParseException{
 SimpleDateFormat fmt =
      new SimpleDateFormat("dd/MM/yyyy");
 String placa = "NNX2015";
 Mockito.when(rCompra.getUltimaData(placa))
      .thenReturn(fmt.parse("05/04/2014"));
 Mockito.when(rVenda.getUltimaData(placa))
      .thenReturn(fmt.parse("16/06/2014"));
 boolean emPosse = servico.isEmPosseDaLoja(placa);
 assertFalse(emPosse);
}
```

Teste: última data de venda anterior à última data de compra

```
@Test
public void testIsEmPosseDaLoja_4() throws
  ParseException{
 SimpleDateFormat fmt =
      new SimpleDateFormat("dd/MM/yyyy");
 String placa = "NNX2015";
 Mockito.when(rCompra.getUltimaData(placa))
      .thenReturn(fmt.parse("05/04/2014"));
 Mockito.when(rVenda.getUltimaData(placa))
      .thenReturn(fmt.parse("21/03/2012"));
 boolean emPosse = servico.isEmPosseDaLoja(placa);
 assertTrue(emPosse);
}
```

Outros recursos do Mockito

- Verificar se uma determinada sequência de métodos foi invocada em um mock.
- Personalização de comparadores de parâmetros.
- Lançamento de exceções em métodos **void** presentes em um mock.
- Verificar se determinadas invocações nunca ocorreram.
- Anotação **@Mock** para criar mock objects.
- Injeção de mocks.
- ...
- Consulte a documentação para mais informações.

Exercícios

- Antes de registrar uma compra, **ServiceCompra** deve verificar se o veículo não está em posse da loja. Implementar e testar o método **registrar(Compra c)** de **ServiceCompra**.
- Projetar, implementar e testar os casos de uso Registrar Venda, Autorizar Venda e Finalizar Venda.

Referências

- http://www.playframework.com/documentation/2.1.0/SBTDependencies
- http://code.google.com/p/mockito/
- Lacerda, Raphael. Facilitando seus testes de unidade no Java: um pouco de Mockito. Disponível em <http://blog.caelum.com.br/facilitando-seus-testes-de-unidade-no-java-um-pouco-de-mockito/>