|  |  |
| --- | --- |
| Logo IFRN simples.png | **Instituto Federal de Educação, Ciência e Tecnologia do RN**  **Diretoria Acadêmica de Gestão e Tecnologia da Informação**  **Curso de Tecnologia em Análise e Desenvolvimento de Sistemas** |

Sistema de Revenda de Veículos

**Modelo de Projeto da implementação com Spring MVC**

# Introdução

Este documento tem como objetivo apresentar o projeto de software e as características de implementação do sistema de revenda de veículos. As sessões a seguir apresentam as tecnologias adotadas, a arquitetura lógica do sistema e as realizações de alguns casos de uso. Além disso, é descrita a estrutura de diretórios e de arquivos em tempo de projeto.

Para informações detalhadas a respeito dos requisitos do sistema e dos casos de uso consulte os documentes correspondentes.

# Tecnologias adotadas

Relação de tecnologias adotadas:

* Linguagem e plataforma de programação: Java 7.
* Ambiente de desenvolvimento: Eclipse IDE for JEE Developers, versão Kepler.
* Frameworks de desenvolvimento no lado servidor: Servlets 3.0 e Spring MVC 3.4.
* Tecnologias de páginas dinâmicas: JSP 2.2, JSTL 1.2 e Sitemesh 2.4.2.
* Frameworks do lado cliente: jQuery 1.11 e Bootstrap 3.3.
* Framework de injeção de dependências: Spring 3.4.
* Framework de autenticação e autorização de usuários: Spring Security 3.1.
* SGBDR: H2 database.

# Diretórios e arquivos do projeto Eclipse

* **src**: diretório com códigos fonte Java da aplicação.
* **testes**: diretório com códigos fonte Java de testes automatizados.
* **WebContent**: diretório com códigos fonte e artefatos web (páginas HTML, CSS, JavaScript, imagens, etc).
* **WebContent/WEB-INF/decorators**: diretório com templates de página (Sitemesh).
* **esquema.sql**: esquema do banco de dados e registros de testes.
* **src/applicationContext.xml**: arquivo com configurações do Spring Framework.
* **WebContent/WEB-INF/decorators.xml**: arquivo de configuração do Sitemesh.
* **WebContent/WEB-INF/spring-mvc.xml**: arquivo de configuração do Spring MVC.
* **WebContent/WEB-INF/spring-security.xml**: arquivo de configuração do Spring Security.
* **WebContent/WEB-INF/decorators**: arquivo de configuração da aplicação web (Servlet container).

# Arquitetura Lógica

O sistema está organizado em um conjunto de camadas lógicas ilustradas na Figura 1.



Figura - camadas lógicas.

As camadas foram organizadas para propiciar a separação clara de interesses das classes, levando a um projeto com alta coesão. Note que a camada de domínio independe das outras camadas. Isto permite a criação de testes de unidade para as regras de negócio de forma isolada, ou seja, sem a necessidade de executar o servidor web ou o sistema gerenciador de banco de dados.

As camadas são ditas lógicas por não terem correspondência exata com a organização física dos arquivos presentes no projeto. A seguir são apresentados o significado das camadas e sua correspondência com os diretórios físicos.

## Camada web

A camada web contém o que é relativo à parte web do sistema: controladores web do Spring MVC e arquivos relativos às páginas web (HTML, JavaScirpt, CSS, imagens, etc).

Os controladores web, localizados no pacote Java **web**, têm como responsabilidade interpretar as solicitações dos usuários (requisições HTTP), delegar processamento de negócio para os objetos de domínio e redirecionar a IU de acordo com as respostas do sistema. No Spring MVC, controladores web são classes definidas com a anotação **@Controller**.

As páginas web representam a interface com o usuário (IU) do sistema e, em tempo de projeto, estão localizadas no diretório **WebContent**.

Esta camada corresponde ao que geralmente é denominado camada de visão.

## Camada de domínio

Na camada de domínio encontram-se as classes que representam os objetos relativos ao domínio do problema como **Compra** e **Veiculo**. As responsabilidades desta camada são reunir as classes cujas instâncias devam ser persistidas[[1]](#footnote-1) em banco de dados, concentrar as regras de negócio e as interfaces de repositório. Fisicamente, esta camada corresponde ao pacote Java **dominio**.

Devido à necessidade de serem persistidas em banco de dados, as entidades compartilham os seguintes requisitos: atributo identificador a ser utilizado como chave primária; implementação dos métodos **equals** e **hashCode** baseados no atributo identificador. Como estratégia de aproveitamento de código necessário a atender estas necessidades, foi criada a classe **Entidade** contendo o atributo identificador (**id**), seus métodos **get** e **set** e os métodos **equals** e **hashCode**. Assim, todas as entidades foram definidas como subclasses de **Entidade** (Figura 2).



Figura – exemplo uso da herança para definir uma classe de entidade.

As regras de negócio são distribuídas entre entidades (e.g., método **getStatus()** da classe **Veiculo**) e entre classes de serviço (e.g, **ServiceCompra**). Classes de serviço são utilizadas em regras de negócio que não pertencem a qualquer entidade ou que envolvem mais de um tipo de entidade. Tais classes são uma aplicação do padrão GRASP Invenção Pura. Diferentemente das entidades, as classes de serviço não guardam estado e suas instâncias são gerenciadas pelo Spring framework. Para que sejam reconhecidas por este, as classes de serviço são anotadas com **@Service**.

Repositórios são abstrações do mecanismo de acesso ao meio persistente. Para tal, os repositórios são interfaces Java que têm como responsabilidade isolar as outras classes da implementação real de acesso ao banco de dados. Assim, os objetos de domínio e os controladores web devem realizar as operações de persistência e leitura de dados através dos repositórios. Em alguns casos, a persistência está atrelada a alguma regra de negócio. Nestas situações, a persistência deve ser realizada através de uma classe de serviço, como nos casos de uso Registrar Compra e Registrar Venda, já que os repositórios não podem ter implementação[[2]](#footnote-2). A dependência de interfaces de repositório isola as classes de domínio do código de acesso a banco de dados, facilitando a criação de testes de unidade.



Figura - exemplo de dependência de repositórios.

## Camada de persistência

As classes da camada de persistência concentram a implementação das rotinas de acesso ao banco de dados, as quais utilizam a API JDBC. Basicamente, para cada repositório do pacote **dominio** há uma classe DAO implementadora.



Figura - exemplo de DAO implementando repositório.

Assim como, controladores web e classes de serviço, as classes DAO são beans gerenciados pelo Spring. Entretanto, DAOs são marcados com a anotação **@Repository**. Devido a isto, o objeto **DataSource**, que dá acesso à conexão com o banco de dados, é injetado nos DAOs pelo Spring Framework (Figura 4).

Figura - injeção do objeto DataSource.

@Repository

public class DAOFabricante implements RepositorioFabricante{

**@Autowired**

**private DataSource dataSource;**

...

camada de persistência corresponde ao pacote Java **persistencia**.

## Camada de infraestrutura

Esta camada corresponde ao pacote Java **infra**. Seu objetivo é armazenar classes necessárias ao funcionamento da aplicação e que não se enquadram nas outras camadas. Atualmente contém apenas a classe **AppAuthenticationProvider**, a qual integra a autenticação de usuários do sistema ao Spring Security.

# Guia de casos de uso

Para ilustrar diferentes técnicas de desenvolvimento web, há um diversidade na implementação dos casos de uso. Esta seção apresenta os casos de uso sob a perspectiva de funcionamento e implementação, servindo como um guia de consulta para os exemplos. A Figura 6 apresenta os casos de uso do sistema.

****

Figura - diagrama de casos de uso do sistema.

A seguir, são listados os casos de uso junto das respectivas técnicas.

* **CRUD Fabricante**: implementação tradicional sem Ajax; Não usa o sistema de templates de páginas.
* **CRUD Tipo de veículo**: utiliza Ajax para excluir um registro, mas a atualização da tabela é feita sem dados enviados pelo servidor; Não usa o sistema de templates de páginas.
* **CRUD modelo**: utiliza Ajax para excluir um registro e para atualizar os registros da tabela após a exclusão; Não usa o sistema de templates de páginas.
* **CRUD veículo**: utiliza Ajax para excluir um registro e para obter os registros da tabela; Upload de arquivo; Download, via Ajax, de arquivo armazenado em banco de dados; Usa conversor do Spring para obter objeto selecionado em um campo select.
* **CRUD funcionário**: não implementado.
* **Registrar compra**: não usa Ajax; Processo com páginas sequenciais (wizard); Upload de arquivo; Campo datepicker.
* **Registrar venda**: armazenamento de dados na sessão do usuário; Processo com páginas sequenciais (wizard); Ajax para atualização de registros da tabela; Campo datepicker.
* **Autorizar venda**: não implementado.
* **Finalizar venda**: não implementado.
* **Relatório de veículos mais vendidos**: não implementado.
* **Relatório de comissões**: não implementado.

As subseções a seguir descrevem detalhes de implementação de alguns casos de uso.

## Realização do caso de uso CRUD Fabricante

A implementação deste caso de uso ilustra o tipo mais elementar de CRUD do sistema: o de uma entidade independente (do ponto de vista banco de dados, uma tabela sem chave estrangeira).

A visão estática deste caso de uso, apresentada na Figura 7, mostra as classes envolvidas na implementação. É possível ver que **CRUDFabricante** referencia um objeto do tipo **RepositorioFabricante**. No entanto, o tipo real deste objeto é **DAOFabricante**, o qual é injetado automaticamente em **CRUDFabricante** pelo Spring Framework.



Figura - visão estática do caso de uso CRUD Fabricante.

A Figura 8 ilustra o cenário de entrada do caso de uso, ou seja, o acesso à página que lista os fabricantes cadastrados (**fabricantes/inicio.jsp**) e que oferece as operações (cenários) novo fabricante, alterar fabricante e excluir fabricante.

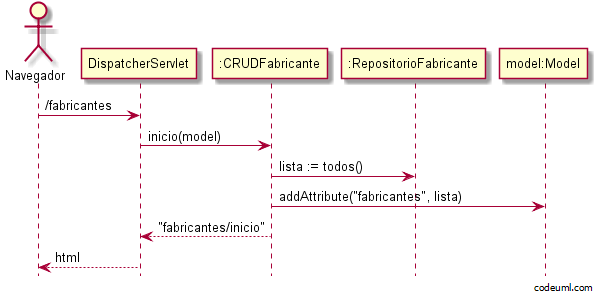


Figura - visão dinâmica do cenário inicial do CRUD Fabricante.

A Figura 9 ilustra o cenário excluir fabricante. Foi omitida a geração de mensagens para o usuário.

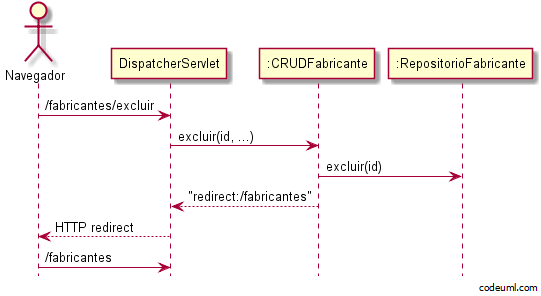


Figura - cenário excluir do caso de uso CRUD Fabricante.

As operações de incluir (Figura 10) e atualizar (Figura 11) são realizadas na mesma página web: **fabricantes/edição.jsp**. O formulário presente na página é submetido à URL **/salvar** nos dois casos e, portanto, os dois casos de uso executam o mesmo método em **CRUDFabricante** quando o formulário é submetido. O que diferencia a operação de persistência a ser executada (inserir ou atualizar) é o valor do campo **id** presente no formulário. O comportamento da URL **/salvar** é ilustrado na Figura 12.

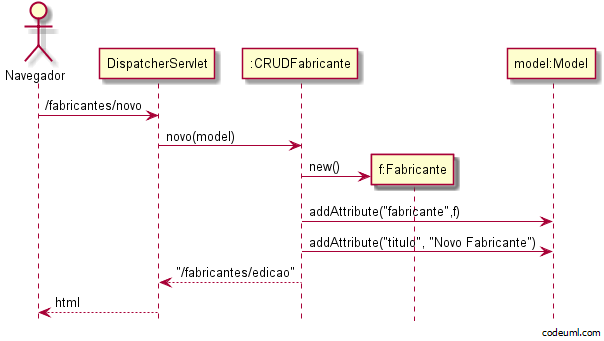


Figura - primeira parte do cenário novo fabricante.

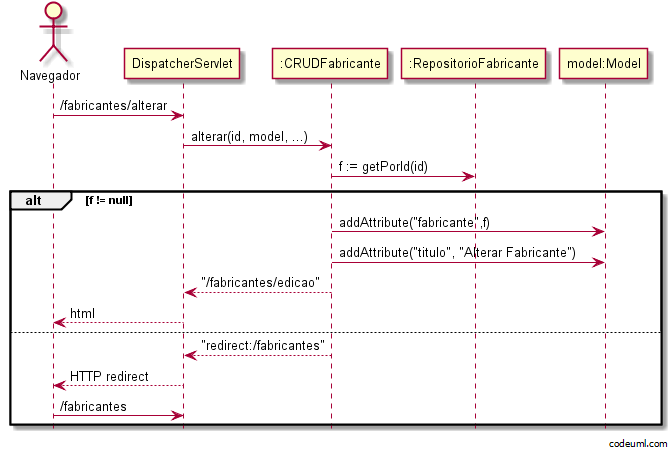


Figura - primeira parte do cenário alterar fabricante.

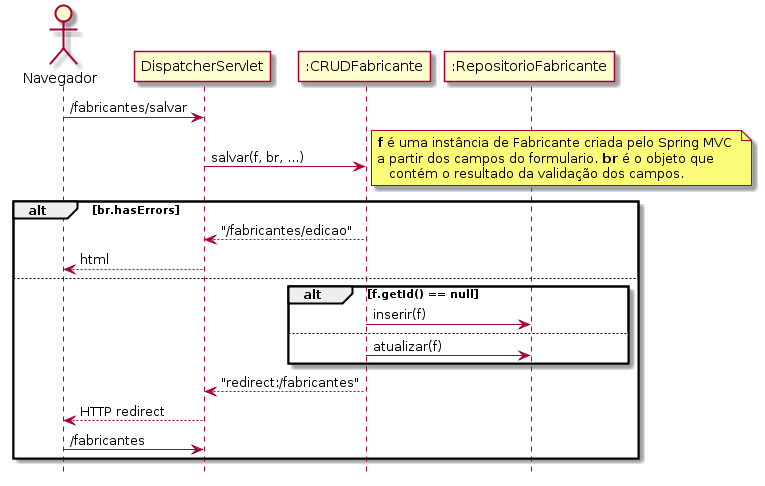


Figura - segunda parte dos cenários novo fabricante e alterar fabricante.

## Realização de caso de uso Registrar Compra

Este é um caso de uso mais complexo do que os casos de uso CRUD. Para que o conteúdo desta subseção fique mais claro para o leitor, é apresentada especificação do caso de uso com seus cenários:

|  |
| --- |
| **Pré-condições:** O usuário deve estar autenticado no sistema.  **Fluxo Básico:**  1. Usuário (Gerente ou Vendedor) informa a placa do veículo.  2. Sistema verifica que o veículo está cadastrado e com status NÃO PERTENCE À 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 com status diferente de NÃO PERTENCE À LOJA.  2.a.1. 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.  **Ponto de Extensão (2):** veículo não está cadastrado no sistema.  2.b.1. Caso de uso segue para CRUD Veículo.  2.b.2. Caso de uso segue para o passo 3. |

A complexidade do caso de uso é determinada pelas regras de negócio: verificar se o veículo está cadastrado e se não está em posse da loja. O registro de compra se resume em persistir uma instância de **Compra**, a qual somente pode ser realizada caso o veículo não esteja em posse da loja. Logo, esta verificação deve ser realizada em dois momentos: na primeira interação do usuário com o caso de uso quando a placa do veículo é informada, e quando o método responsável pela inserção da compra for executado. Esta validação antes da inserção justifica a criação da classe **ServiceCompra**, já que não é possível codificação na interface **RepositorioCompra**. Implementar a validação em **DAOCompra** vai contra o princípio da alta coesão, pois trata-se de uma regra de negócio e **DAOCompra** somente deve ter responsabilidades relativas ao banco de dados. A implementação da validação em **ServiceCompra** também facilita a escrita de testes de unidade, já que **DAOCompra** tem dependências com a API JDBC. Assim, o controlador **RegistrarCompra** deve interagir com **ServiceCompra**, ao invés de **RepositorioCompra**, para garantir a validação. A X exibe as classes e interfaces Java envolvidas neste caso de uso.



Figura - classes e interfaces utilizadas no caso de uso Registrar Compra.

Para indicar se um veículo está em posse da loja ou não, optou-se pela criação do método **getStatus**, na classe **Veiculo**. O status do veículo indica a situação do veículo, a qual pode ser: disponível para a venda, em processo de venda ou não pertence à loja. O status de um veículo é determinado, em tempo de execução, através das ocorrências e datas das últimas compra e venda relativas ao veículo. A última compra e a última venda são obtidas através de consultas ao banco de dados encapsuladas pelos métodos **getUltimaCompraDoVeiculo** e **getUltimaVendaDoVeiculo**, respectivamente implementados em **RepositorioCompra** e **RepositorioVenda**. O diagrama da Figura 14 ilustra a interação entre **Veiculo**, **RepositorioCompra** e **RepositorioVenda**.

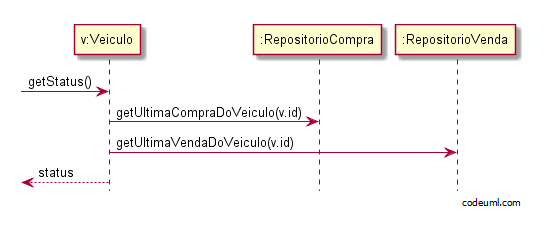


Figura - sequência de operações do método getStatus() da classe Veiculo.

A primeira etapa do caso de uso consiste de uma página com um campo no qual o usuário informa a placa do veículo a ser comprado pela loja. Após a submissão, o sistema encaminha o usuário para a página da segunda etapa caso o veículo esteja cadastrado e não pertença à loja, para a página de cadastro de veículo caso ele não esteja cadastrado no sistema ou para a página inicial caso o status do veículo seja diferente de “não pertence à loja”. Esse procedimento é ilustrado na Figura 15.

Caso o veículo não esteja cadastrado no sistema, o usuário é direcionado para a página de cadastro de veículo (**compras/novo\_veiculo.jsp**). O formulário desta página é submetido à URL **/compras/salvar\_veiculo**. Após uma validação de formulário bem sucedida, o usuário é direcionado para a página que contém o formulário de registro de compra (**compras/registro.jsp**). Este procedimento é ilustrado na Figura 16.

A etapa final do caso de uso ocorre na página **compras/registro.jsp**, cujo formulário é submetido à URL **/compras/salvar**. Caso a validação do formulário seja bem sucedida, controlador persiste a compra por meio de **ServiceCompra** e direciona o usuário para a página principal da aplicação. Este procedimento é ilustrado na Figura 17 e na Figura 18.

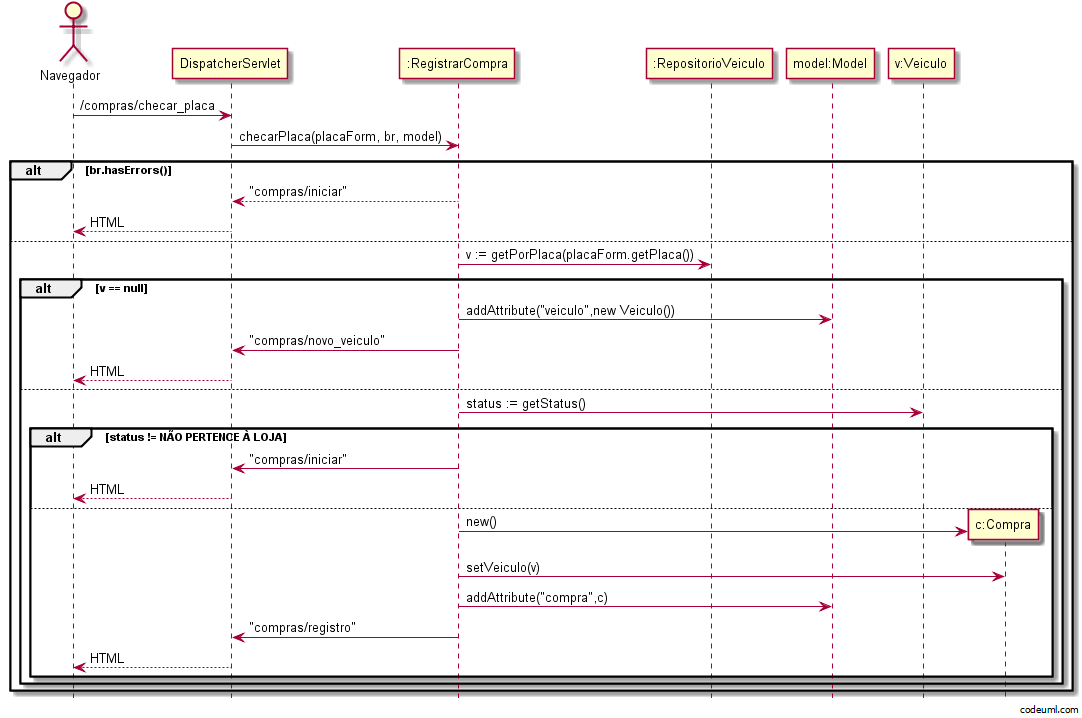


Figura - primeira etapa do caso de uso Registrar Venda.

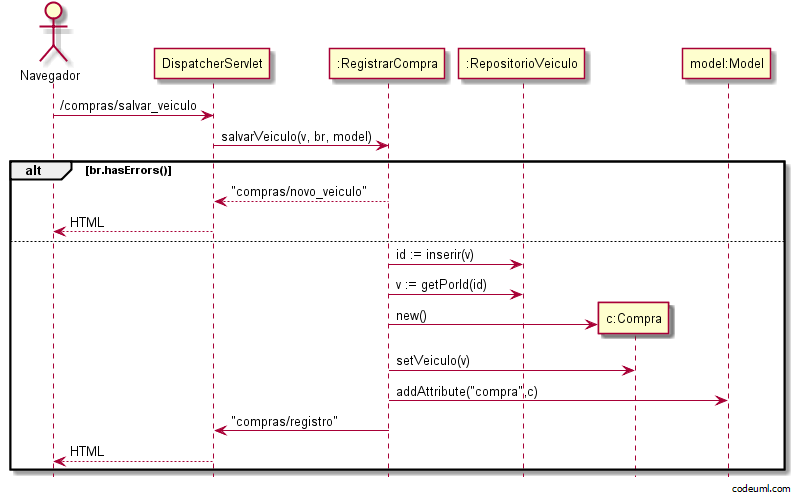


Figura - cadastro de veículo no caso de uso Registrar Compra.

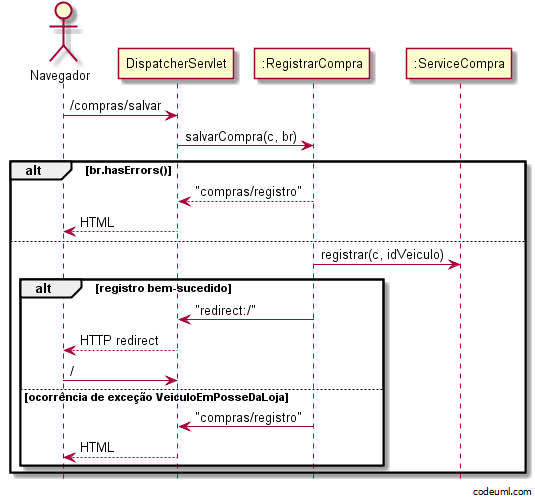


Figura - etapa final do caso de uso Registrar Compra.

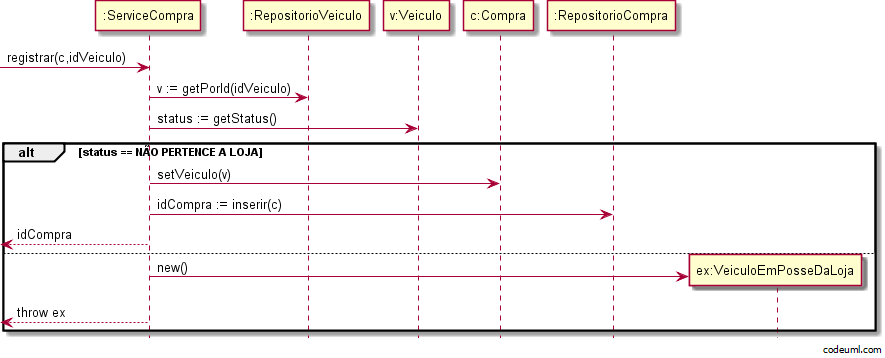


Figura - comportamento do método registrar presente na classe ServiceCompra.

1. Ao longo do texto, o termo entidade é utilizado como sinônimo destas instâncias persistentes. [↑](#footnote-ref-1)
2. A partir do Java 8, é possível a criação de interfaces que possuem implementação. [↑](#footnote-ref-2)