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

Sistema de Revenda de Veículos

**Modelo de Projeto**

# Introdução

Este documento tem como objetivo apresentar o projeto lógico do Sistema de Revenda de Veículos. Assim, são apresentados o diagrama de casos de uso, a arquitetura lógica do sistema e as realizações de alguns casos de uso. Como este documento tem objetivo didático, em determinados trechos do texto são discutidos tópicos de projeto OO. Estas discussões apresentam as implicações da solução adotada e, em alguns casos, também discute alternativas possíveis.

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

# Arquitetura Lógica

O sistema está organizado em uma arquitetura lógica baseada em camadas ilustrada na Figura 1.

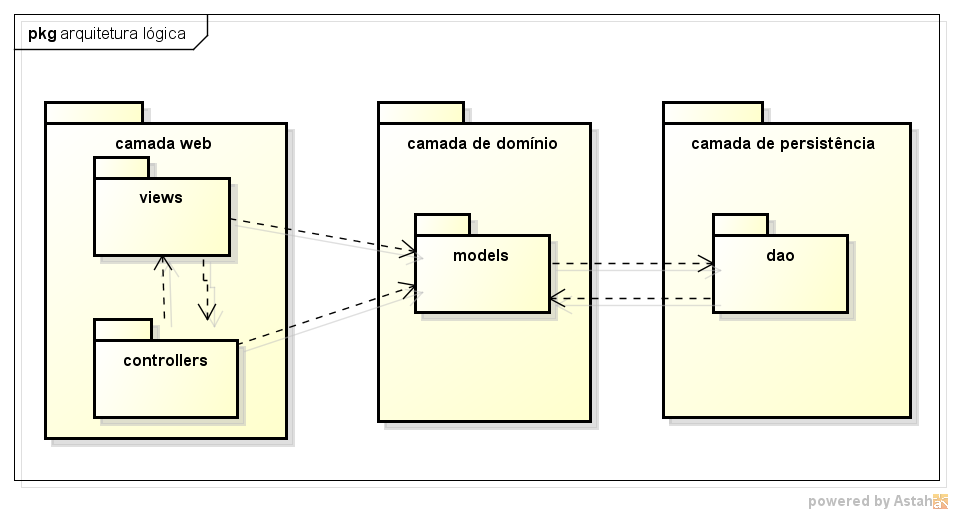


Figura - arquitetura lógica.

As camadas foram organizadas para propiciar a separação clara de interesses dos objetos, levando a um projeto com alta coesão. Note que as camadas são lógicas, não existindo fisicamente na estrutura de arquivos do código-fonte. Além disso, a arquitetura foi organizada de forma que possam ser realizados testes unitários 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.

A camada web envolve os pacotes **views** e **controllers**, os quais seguem a organização padrão do Play Framework. No pacote **views** estão localizadas as páginas web desenvolvidas com o recurso de templates do Play Framework. As páginas web representam a interface com o usuário (IU) do sistema. Já no pacote **controllers**, as classes tem como responsabilidade interpretar as requisições dos usuários, delegar processamento de negócio para os objetos de domínio e redirecionar a IU de acordo com as respostas do sistema. Por tratar-se de recurso do Play Framework, cada classe controladora deve ter **play.mvc.Controller** como superclasse.

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) no banco de dados e concentrar as regras de negócio.

Em um sistema como este, fortemente dependente dos dados persistidos, é comum que existam regras de negócio baseadas em informações existentes ou que devam ser executadas no momento da inserção, alteração ou exclusão de dados (ações CRUD). Infelizmente, nem sempre é possível a utilização do recurso de restrições do banco de dados para implementação de tais regras. Assim, o pacote **models** define uma classe denominada **Repositorio<T>** com métodos para inserção, atualização, exclusão e obtenção de objetos de domínio (figura 2). O tipo genérico é utilizado para definir a classe do objeto de domínio a ser manipulado pelo repositório. Desta forma, para entidades que não necessitam de regras de negócio em ações CRUD, é suficiente instanciar a classe **Repositorio** indicando o tipo de objeto do domínio a ser manipulado (veja a realização do caso de uso CRUD Fabricante). Nos outros casos, é necessária a criação de uma subclasse de **Repositorio** parametrizada com o tipo de objeto de domínio específico e com métodos adicionais, ou sobrescritos, para atender as restrições devidas (veja a realização do caso de uso Registrar Compra).

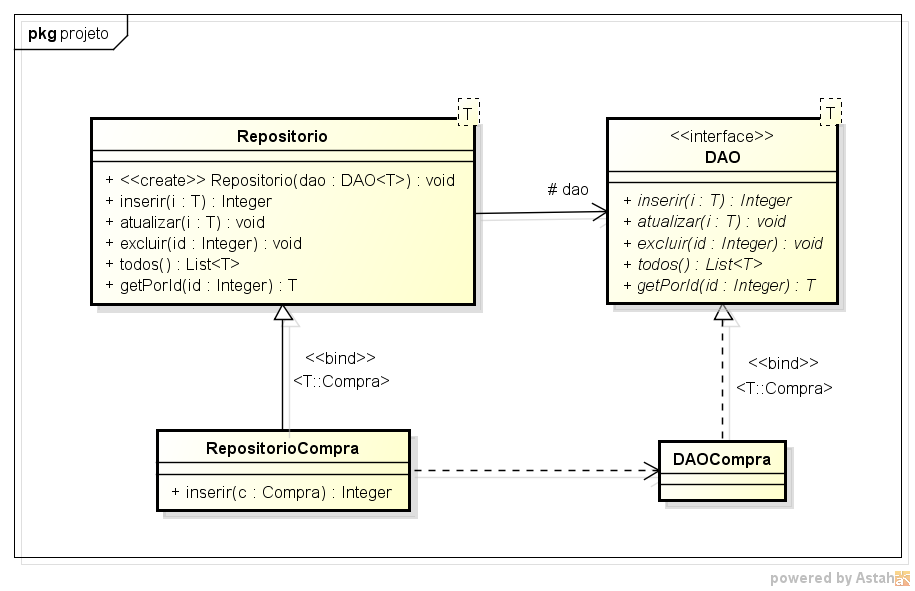


Figura - classe genérica de repositório e exemplo de classe de repositório específica a uma entidade.

As classes de repositório também têm como responsabilidade isolar as outras classes do acesso ao banco de dados. Portanto, os objetos de domínio e as classes controladoras da camada web devem realizar as operações de gravação ou leitura de dados através dos repositórios. Desta forma, temos as regras de negócio isoladas do restante da aplicação, facilitando a implementação e execução dos testes unitários.

Uma classe de entidade deve possuir um atributo identificador e sobrescrever os métodos **equals** e **hashCode**, herdados de **java.lang.Object**, de forma que duas instâncias da mesma entidade sejam consideradas iguais caso ambas possuam o mesmo valor para o atributo identificador. Para evitar a repetição de código, as classes de entidade devem ter **Model** como superclasse (figura 3), a qual define o atributo identificador e os métodos **equals** e **hashCode**. A existência desta classe **Model** permitiu a criação da classe de repositório genérica, já que os métodos **inserir**, **excluir** e **getPorId** de **Repositorio** puderam ser definidos de forma compatível com qualquer classe de entidade, já que todas possuem **Integer** como tipo do identificador.

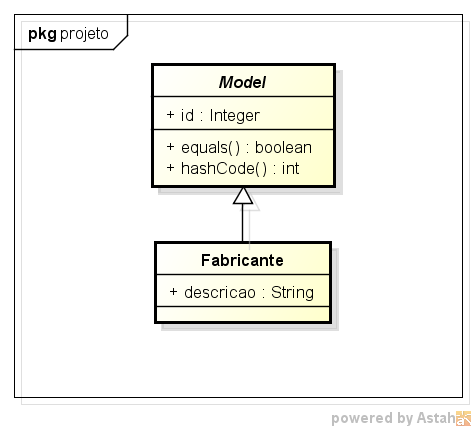


Figura - classe abstrata Model e classe de entidade Fabricante.

As classes da camada de persistência concentram a implementação das rotinas de manipulação de acesso ao banco de dados, incluindo a definição dos comandos SQL. De modo geral, deve ser criada uma classe realizadora da interface **DAO** para cada entidade existente. Assim, para a entidade **Fabricante**, há a respectiva classe **DAOFabricante** implementando os métodos genéricos de operações CRUD relativos aos fabricantes. No caso de consultas específicas, uma classe implementadora pode definir métodos adicionais além dos especificados em **DAO**. A busca de veículo por placa é um exemplo, levando à criação do método **getPorPlaca** na classe **DAOVeiculo**. Ao observar o código das classes implementadoras, repare que a obtenção de conexão com o SGBD é realizada através da classe **play.db.DB**, sendo que o gerenciamento das conexões é realizada pelo Play Framework.

# Diagrama de Casos de Uso

****

Figura - diagrama de casos de uso do sistema.

# Realização de 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).

Apesar da figura 5 sugerir o contrário, não foi implementada uma classe de repositório específica da entidade **Fabricante**. A classe **Repositorio<Fabricante>** está presente na figura apenas para tornar explícita a parametrização de **Repositorio** com o tipo **Fabricante**. Devido a uma limitação da ferramenta UML, não foi possível especificar a parametrização através da associação entre **controller::Fabricantes** e **models::Repositorio**.

A visão dinâmica (figuras 6 e 7) foca na interação entre objetos controladores, objetos de domínio e objetos DAO, abstraindo detalhes tecnológicos ou de implementação das páginas web. São ilustrados dois cenários do caso de uso: incluir e atualizar. Ambos possuem uma sequência de interação de objetos semelhante para realizar a persistência dos dados:

* Obter formulário web com os dados;
* Instanciar **Fabricante** a partir do formulário web;
* Delegar persistência ao repositório (persistir quando não há id e atualizar em caso contrário);
* Delegar persistência a **DAOFabricante**;
* Encaminhar usuário de acordo com o resultado da operação.

Neste momento surge uma questão: já que **Repositorio** não realiza processamento adicional, apenas delega a persistência a **DAOFabricante**, por que não implementar a persistência em **Repositorio** e eliminar **DAOFabricante**, ou eliminar **Repositorio** e permitir que **Fabricantes** comunique-se diretamente com **DAOFabricante**?

Implementar a persistência em **Repositorio** levaria a uma menor coesão, pois esta passaria ter responsabilidades muito distintas. Lembre-se que as classes de repositório devem conter as regras de negócio relativas às operações CRUD, as quais não existem para o CRUD de **Fabricante** (isso explica o uso do repositório genérico). Implementar a persistência em **Repositorio** também aumentaria o seu acoplamento, pois esta passaria a depender das classes da API JDBC.

Eliminar **Repositorio** e permitir a comunicação direta entre **Fabricantes** e **DAOFabricante** vai de encontro ao princípio de que somente os objetos de domínio devem acessar os DAOs. Esta justificativa pode aparentar um purismo arquitetural, mas em casos de uso complexos as regras de negócio teriam que ser implementadas nos DAOs (ou pior, nos controladores!) diminuindo a coesão. Note que, de acordo com estes princípios, as regras de negócio podem ser implementadas (e testadas de forma automatizada) de maneira independente das tecnologias de front-end (web, desktop, etc) e de persistência.

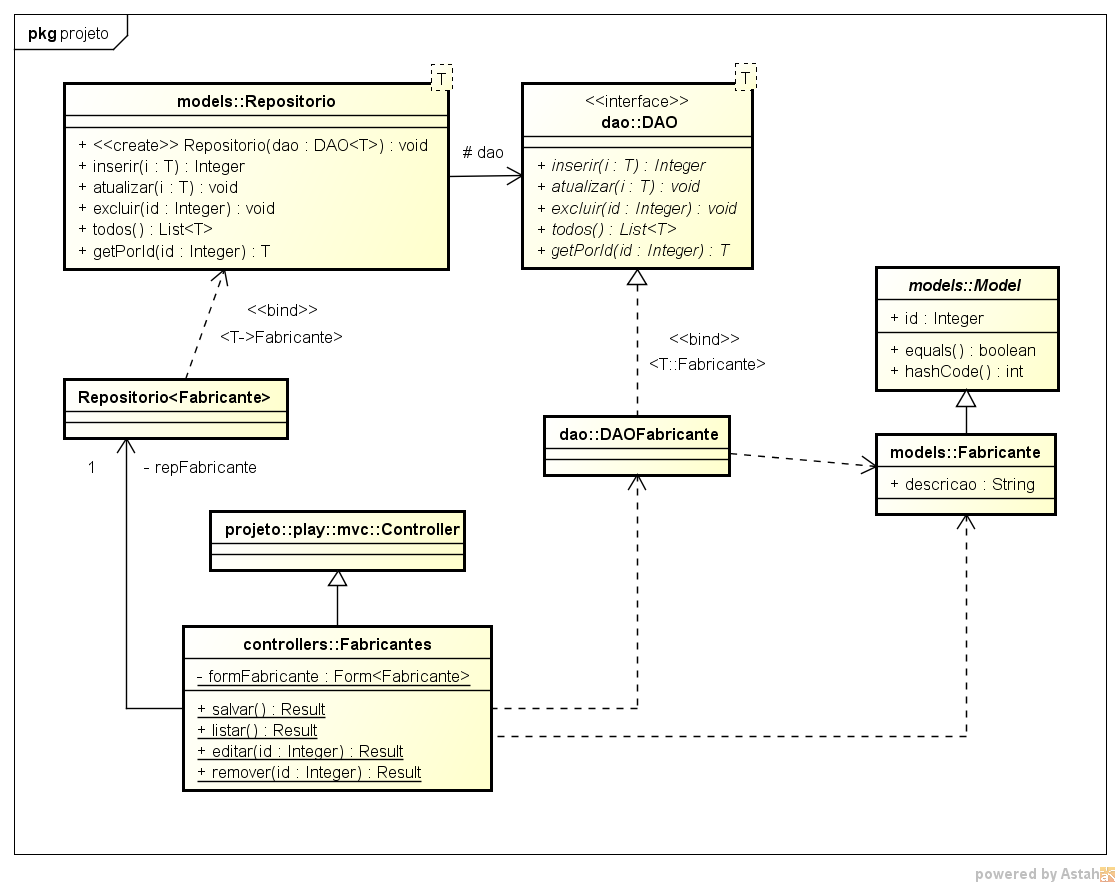


Figura - visão estática do caso de uso CRUD Fabricante.

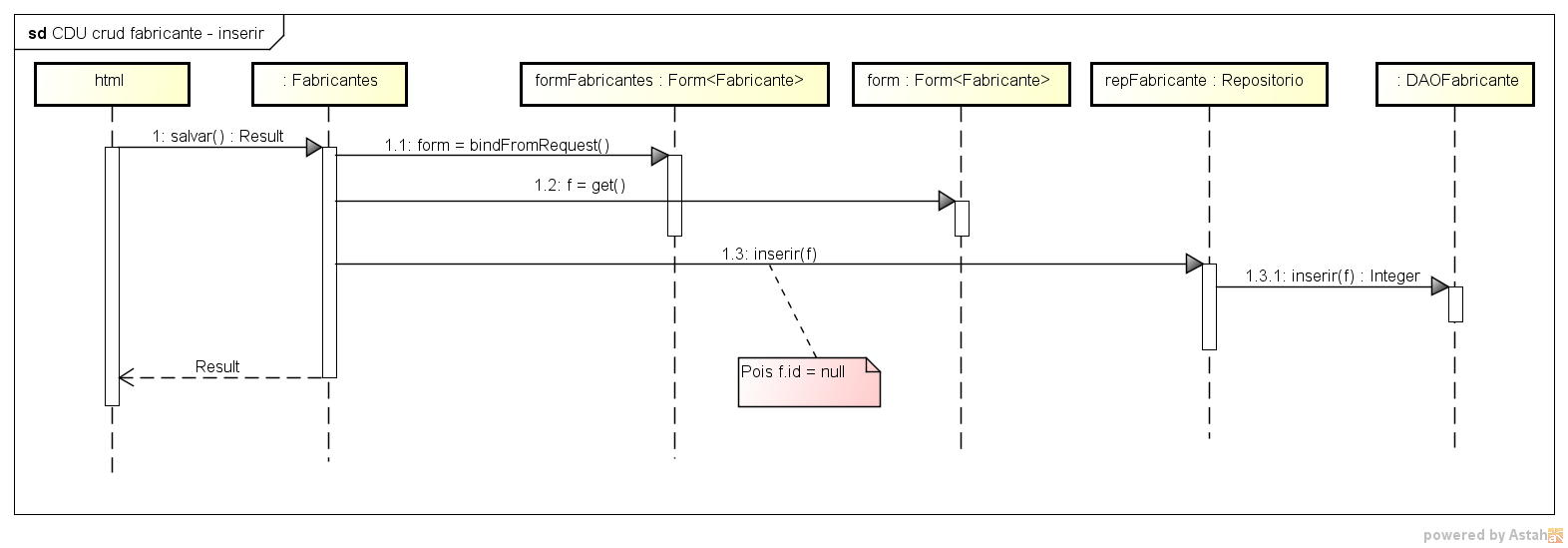


Figura - visão dinâmica do caso de uso CRUD Fabricante (cenário inserir).

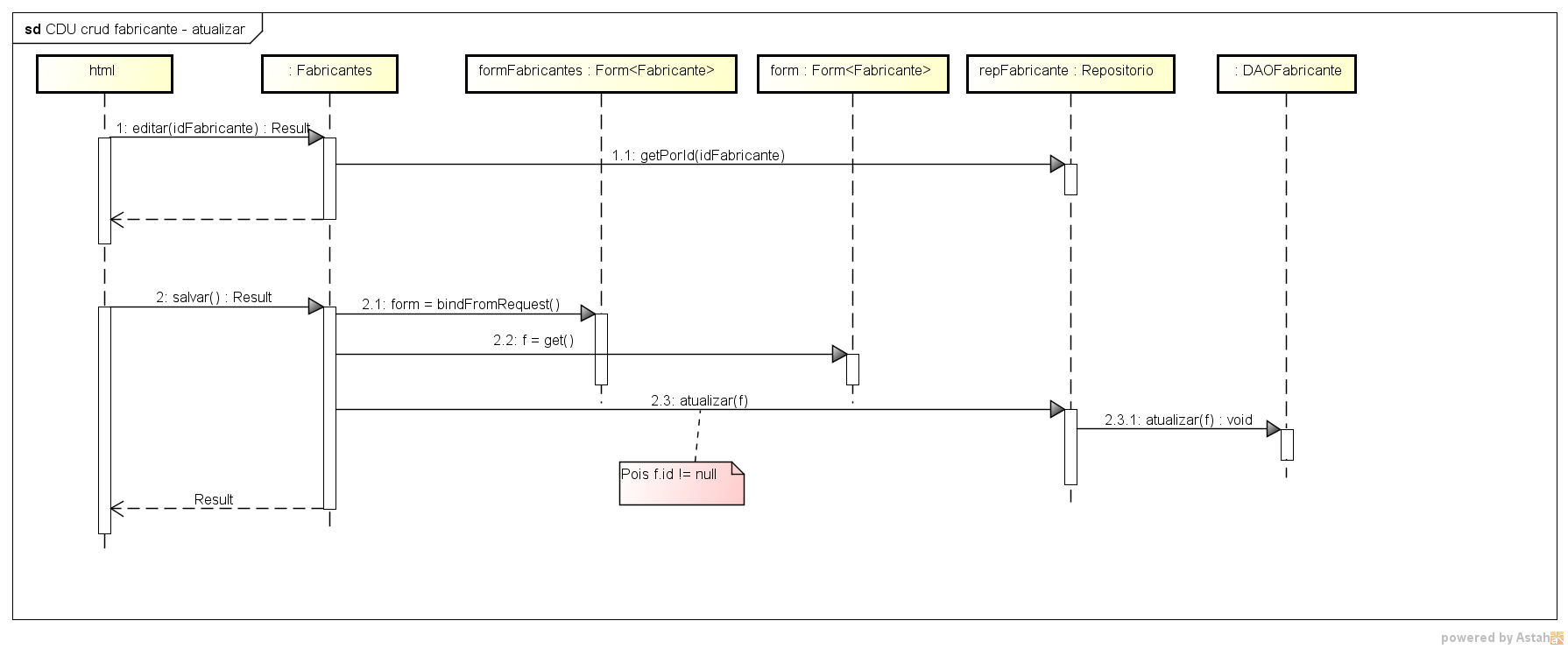


Figura - visão dinâmica do caso de uso CRUD Fabricante (cenário atualizar).

# Realização de caso de uso Registrar Compra

Devido a sua complexidade, este é um caso de uso que justifica a arquitetura adotada. A seguir, são apresentados os seus fluxos (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 não está em posse da loja. Em seguida, sistema exibe os dados 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.  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** (ver figura 8), 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 justifica a criação da classe **RepositorioCompra**, já que o método inserir do repositório genérico não é suficiente[[2]](#footnote-2).

Já a classe **RepositorioVeiculo**, é justificada pela necessidade de buscar um veículo pelo seu número de placa.

Para indicar se um veículo está em posse da loja ou não, optou-se pela criação do método **isEmPosseDaLoja** na classe **Veiculo**. O resultado deste método pode ser determinado pelas últimas datas de compra e de venda relativas ao veículo. Estas datas são atributos das classes **Compra** e **Venda**, podendo ser obtidas através de uma consulta ao banco de dados. Como as classes de repositório devem isolar os outros objetos dos DAOs, foi criado o método **getDatasUltimasTransacoes** em **RepositorioVeiculo**. Por que não criar estes métodos em **RepositorioCompra** já que esta tem que utilizá-los durante a persistência de uma compra? A resposta vem do fato de que esta validação (veículo em posse da loja ou não) também deve ser realizada durante a venda de um veículo.

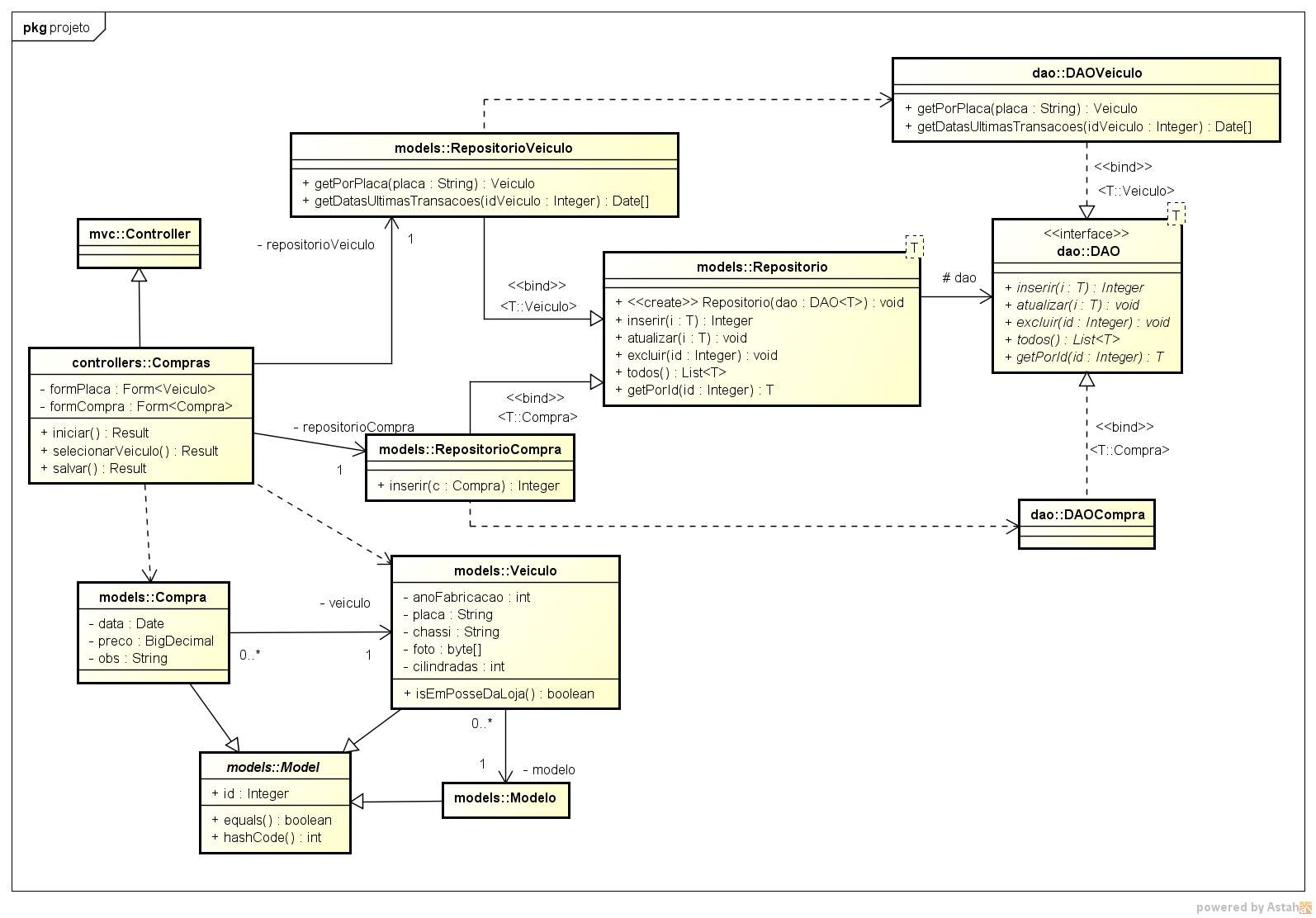


Figura - visão estática do caso de uso Registrar Compra.

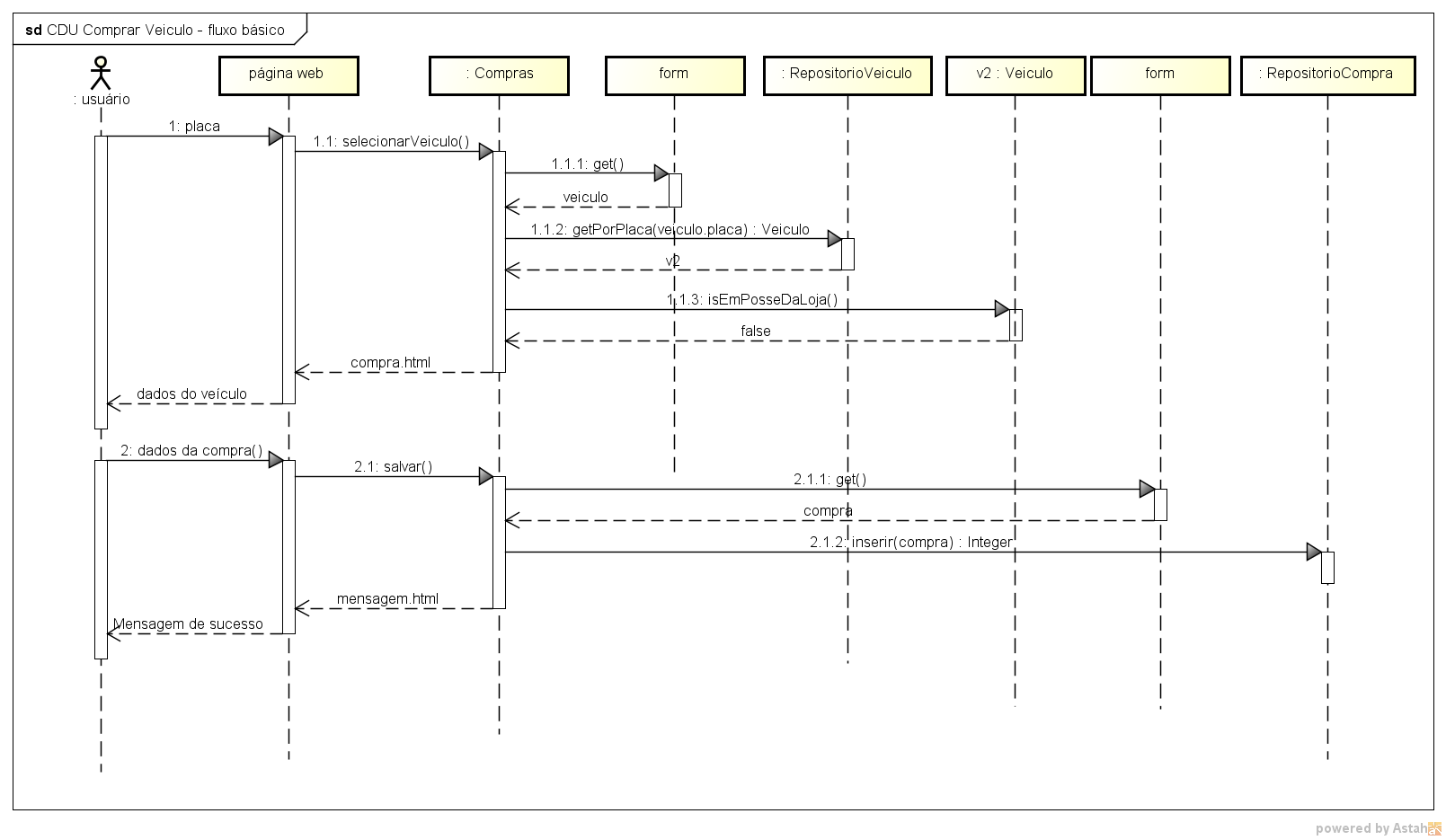


Figura - visão dinâmica do caso de uso Registrar Compra (fluxo básico).

1. Ao longo do texto, o termo entidade é utilizado como sinônimo destas instâncias persistentes. [↑](#footnote-ref-1)
2. Repare que a criação da classe **RepositorioCompra** não impede que a classe **Repositorio** genérica seja utilizada acidentalmente para persistência de uma compra, permitindo a compra de um veículo que já se encontra em posse da loja. [↑](#footnote-ref-2)