

UNIVERSIDADE DO SUL DE SANTA CATARINA

Geison machado da silva

car-crud

Sistema de revenda de carros utilizando java Ssdfwing e persistência em arquivo

Florianópolis/SC

2020

GEISON MACHADO DA SILVA

car-crud

Sistema de revenda de carros utilizando java swing e persistência em arquivo

Trabalho apresentado ao Curso de Sistemas de Informação da Universidade do Sul de Santa Catarina como requisito parcial à aprovação na unidade de aprendizagem de Tópicos Avançados de Programação.

Professor: Ms. Osmar de Oliveira Braz Júnior

Florianópolis/SC

2020

Sumário

[1 introdução 4](#__RefHeading___Toc813_380575225)

[2 Estrutura do projeto 5](#__RefHeading___Toc815_380575225)

[3 Padrão mvc 6](#__RefHeading___Toc817_380575225)

[4 Incluir 8](#__RefHeading___Toc819_380575225)

[5 Listar 10](#__RefHeading___Toc821_380575225)

[6 Consultar (buscar, alterar e excluir) 11](#__RefHeading___Toc823_380575225)

[7 Propriedades 12](#__RefHeading___Toc825_380575225)

[8 Persistência em arquivo 14](#__RefHeading___Toc827_380575225)

[9 Diagrama de classes 18](#__RefHeading___Toc829_380575225)

[10 conclusão 19](#__RefHeading___Toc831_380575225)

[REFERÊNCIAS 20](#__RefHeading___Toc833_380575225)

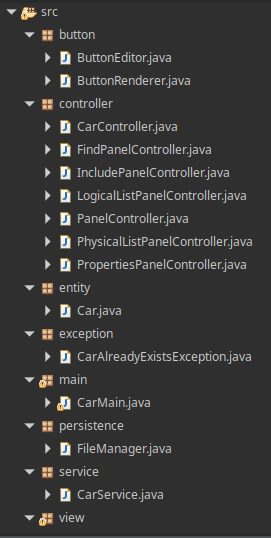
# introdução

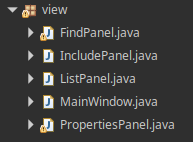
Este trabalho tem como objetivo principal desenvolver um software completo, desde a interface até a persistência em arquivo utilizando a linguagem Java de um CRUD de carros. Ou seja, deve ser permitido pelo programa incluir, alterar, excluir e listar os carros, além de poder visualizar as propriedades do arquivo.

O código deve estar orientado a objetos, utilizando Java Swing na implementação da interface e com persistência em arquivo com a classe RandomAccessFile.java.

Deveríamos escolher um determinado tipo abstrato para manipulação e no meu caso escolhi um carro com os atributos chassi (String, chave), marca (String), modelo (String, obrigatório), ano (inteiro) e preço (real).

# Estrutura do projeto

  
Figura 1: Estrutura dos pacotes e classes (parte 1)

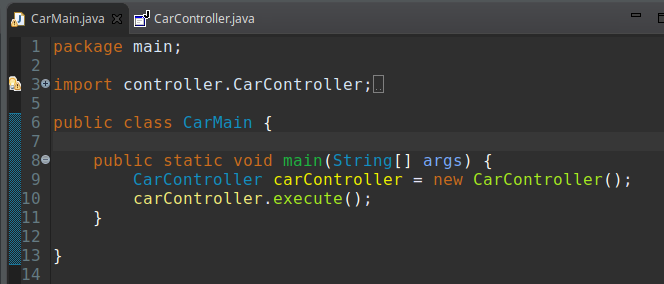
  
Figura 2: Estrutura dos pacotes e classes (parte 2)

Dividi o projeto nos pacotes mostrados na figura 1 e 2 com o seguinte funcionamento: A classe CarMain.java tem a responsabilidade de iniciar o programa e chamar o método execute() de CarController.java. CarController.java age como um controller “mestre” que instancia a classe MainWindow.java, classe da interface principal do programa que mostra os menus e um painel pai que controla qual painel é mostrado no momento.

Cada PanelController então lida com a lógica e validação de cada painel e como todos eles estendem PanelController.java para reusar os atributos e métodos em comum, chamam o CarService.java que faz a intermediação entre o front end e o back end para a manipulação de dados. Nesse caso através da classe FileManager.java que lida com toda a manipulação dos dados em arquivo.

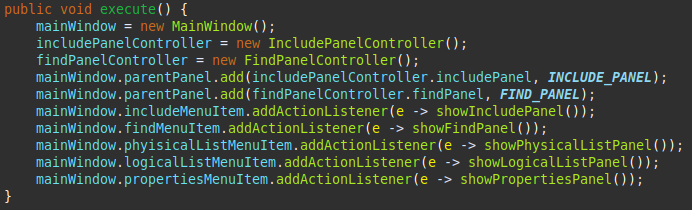
# Padrão mvc

A primeira coisa a fazer foi criar a estrutura do projeto segundo o padrão MVC: Um controller que intermediaria os processos entre a interface, entidade e back-end do projeto. Porém, resolvi separar o método main que inicia o programa em sua própria classe CarMain.java e esse chama CarController que inicia todos or processos:

  
Figura 3: CarMain.java

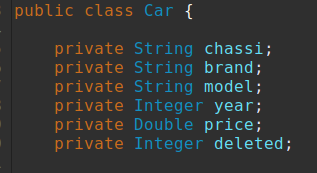
O controller, chamado de CarController.java ficou responsável por criar a primeira tela que foi chamada de MainWindow.java contendo a barra de menus, os próprios menus e um painel pai que controlava qual painel filho deveria aparecer ao ser clicado no menu através do cardLayout.

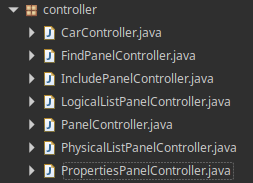
Os painéis mostrados na interface seriam os seguintes:

  
Figura 4: Método execute() de CarController.java

* Incluir, que inclui um carro no sistema,
* Listar, (físico) que lista todos os carros, inclusive os excluídos,
* Listar, (lógico) que lista todos os carros não excluídos,
* Consultar, que permite consultar todos os carros por uma frase contida em seu chassi ou modelo e mostra isso numa tabela onde os dados podem ser alterados ou excluídos
* Propriedades, que mostra um painel com o número de registros e o tamanho do arquivo.

A entidade criada foi chamada de Car.java com os atributos requeridos e o atributo “deleted” (Integer) que caso fosse 1 significava excluído e 0 não excluído.

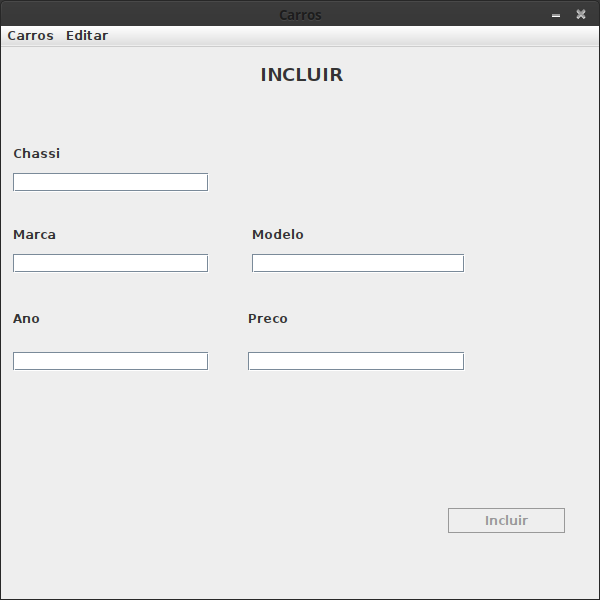
  
Figura 5: Entidade Car.java

  
Figura 6: Pacote com todos os controllers

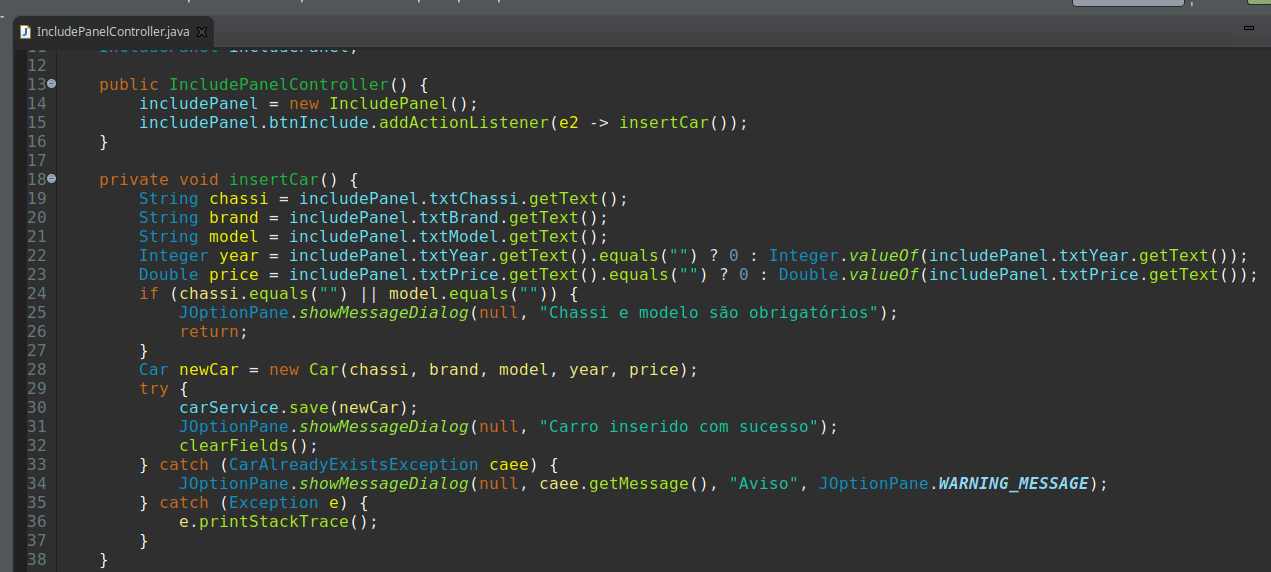
Como toda a lógica do programa ficaria muito extensa e confuso apenas dentro de CarController.java fiz que ele instanciasse um controller próprio para cada painel e esse ficaria responsável por criar a classe do painel e lidar com sua lógica de negócio.

Todos os controllers, exceto por CarController.java estendem PanelController.java que contém atributos e métodos comuns a mais de um panelController.

# Incluir

  
Figura 7: Screenshot do painel incluir

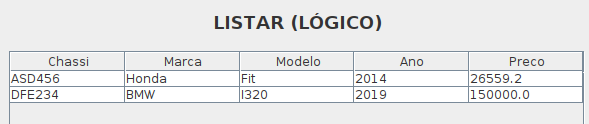
O painel de incluir que é chamado imediatamente ao entrar no programa apenas valida os dados em seu controller, cria um objeto Car.java com os dados da interface e chama carService.save() para persistir os dados.

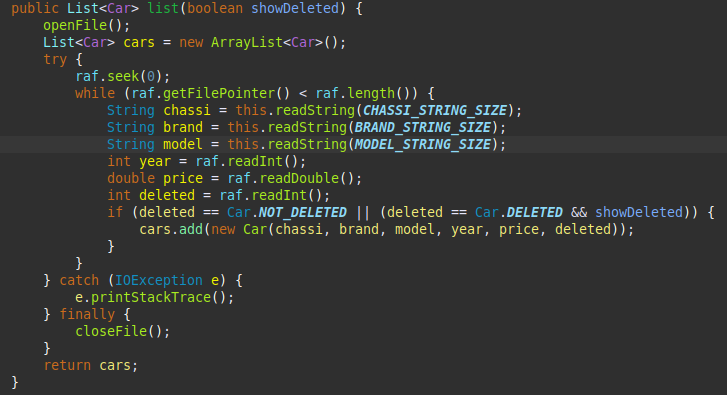
  
Figura 8: IncludePanelController.java

# Listar

Listar físico e lógico são parecidos, com a única diferença sendo que o lógico não mostra os excluídos.

  
Figura 9: Screenshot do painel "Listar (Físico)"

  
Figura 10: Screenshot do painel "Listar (Lógico)"

  
Figura 11: Método list() de FileManager.java

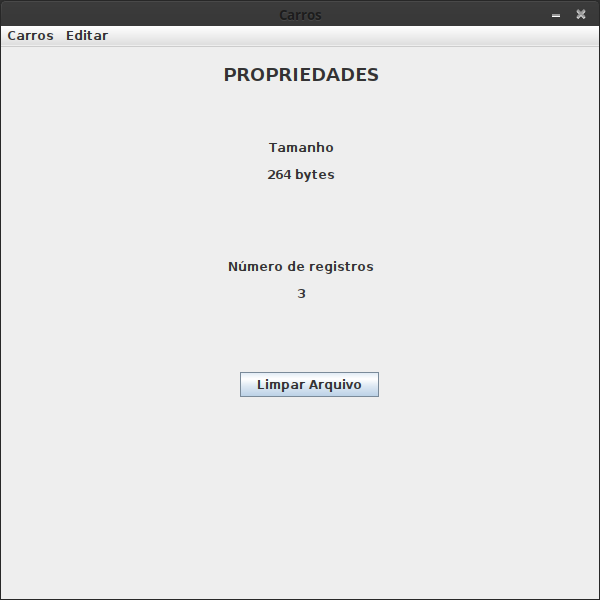
O algorítimo do método listar apenas retorna todos os registros filtrando aqueles que estão excluídos ou não dependendo da flag passada como parâmetro pela painel listar lógico ou físico.

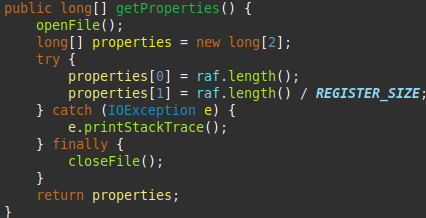
# Consultar (buscar, alterar e excluir)

Consultar busca por chassi ou Modelo e preenche a tabela com os resultados que contenham a String de busca.

  
Figura 12: Screenshot do painel "Buscar"

# Propriedades

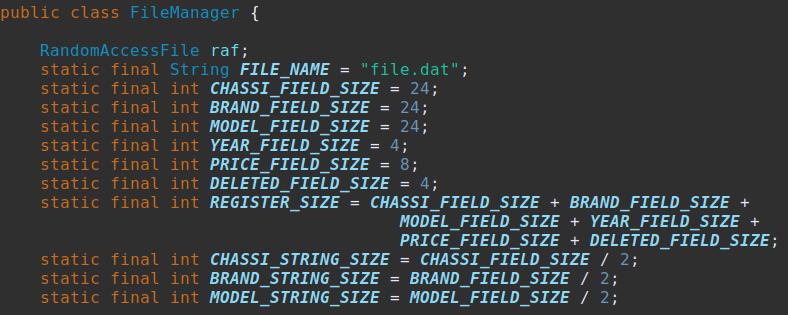
  
Figura 13: Screenshot do painel "Propriedades"

  
Figura 14: Método getProperties() de FileManager.java

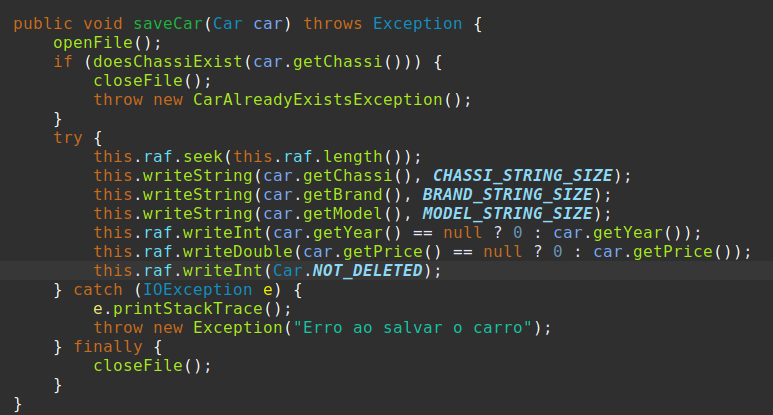
A tela de propriedades mostra o tamanho do arquivo e o número de registros que pode ser conseguindo dividindo o tamanho do arquivo pelo tamanho do registro.

# Persistência em arquivo

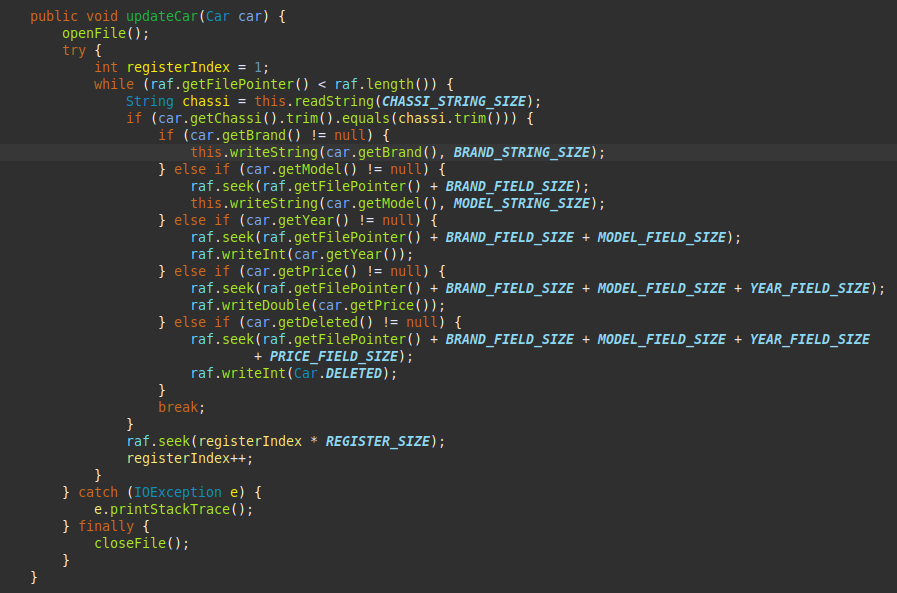
A classe FileManager.java lida com toda a persistência do programa. Ela recebe requisições de CarService.java para salvar, alterar, listar ou excluir a entidade e fica responsável por implementar toda a lógica de persistência.

  
Figura 15: Atributos de FileManager.java

Nela contém um atributo representando o tamanho de cada campo da entidade, um atributo com o tamanho do registro que é a soma de todos os campos e um atributo para o tamanho de cada atributo String que é o tamanho do campo dividido por 2.

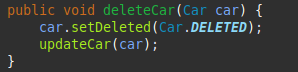
  
Figura 16: Método saveCar() de FileManager.java

O método saveCar abre o arquivo, verifica se o chassi do carro passado como parâmetro existe, se existe fecha o arquivo e joga uma nova exceção. Se não existe ele move o ponteiro até o final do arquivo e escreve todos os atributos do arquivo, colocando 0 no lugar dos números que forem null e não excluído no atributo de exclusão.

  
Figura 17: Método updateCar() de FileManager.java

O método updateCar itera sobre o arquivo verificando se o registro tem o chassi igual ao chassi do objeto “car”, passado como parâmetro. Caso tenha, verifica qual atributo de “car” não é null, move o ponteiro até a posição desse atributo somando os tamanhos dos atributos antes dele e escreve sobre ele.

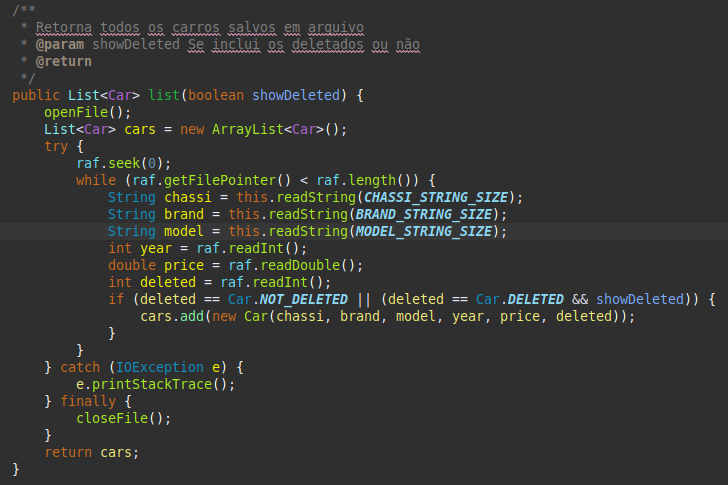
No final da iteração move o ponteiro para o próximo registro através da operação “tamanho de atributo” x registerIndex (contador de registros).

  
Figura 18: Método deleteCar() de FileManager.java

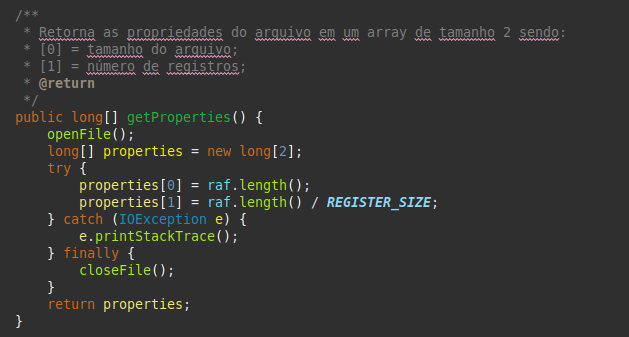
O método deleteCar simplesmente altera o atributo de “car” para deleted e chama o método updateCar(car).

  
Figura 19: Método findCarByChassiOrModel() de FileManager.java

Caso algum tenha adiciona a lista de retorno.

  
Figura 20: Método list() de FileManager.java

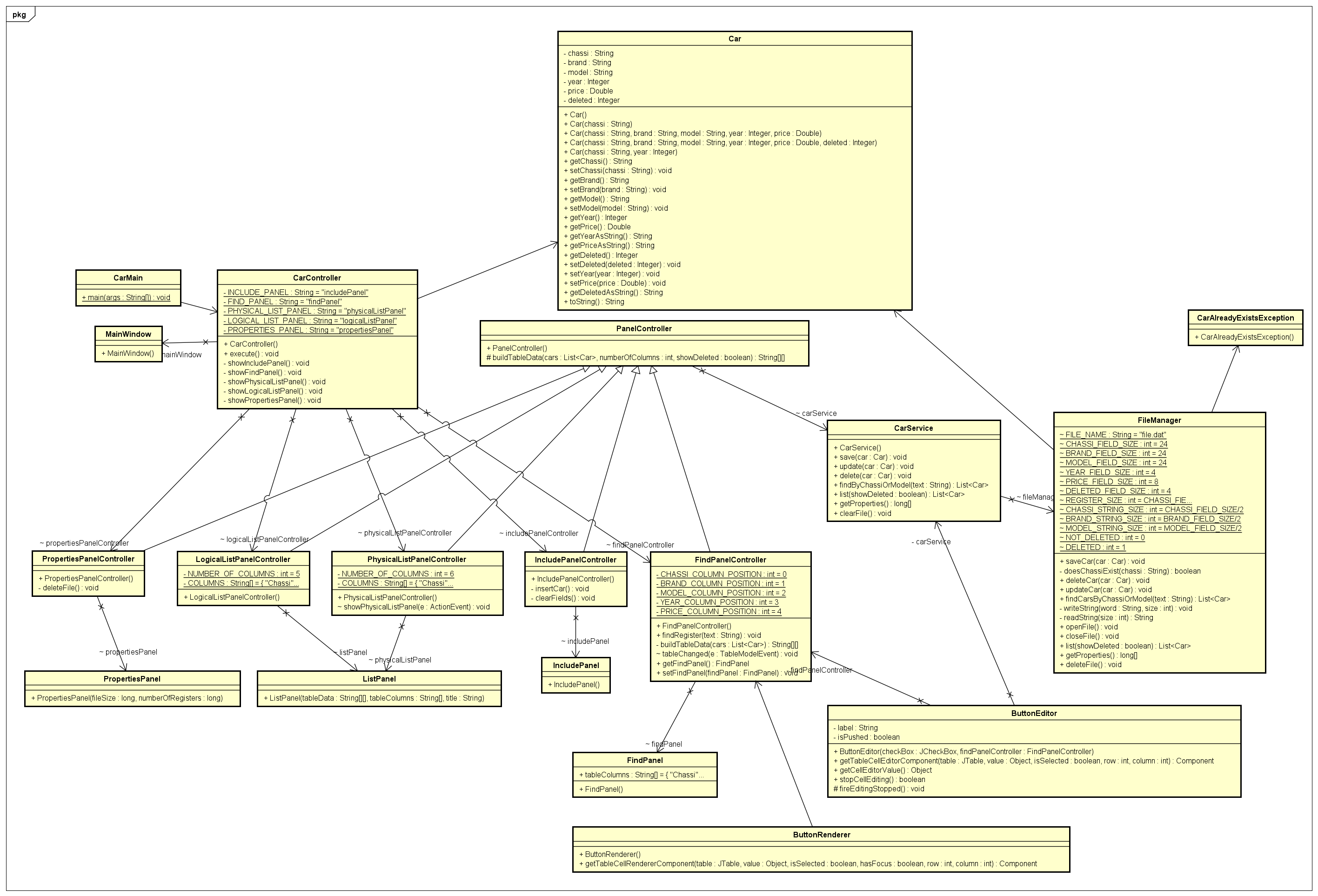
O método listar utilizado em ambos os painéis de listar apenas itera sobre todos os registros adicionando-os a uma lista, apenas verificando se estão excluídos ou não e adicionando de acordo com a flag “showDeleted” passada como parâmetro.

  
Figura 21: Método getProperties() de FileManager.java

O método getProperties retorna um array de tamanho 2 com duas propriedades: o tamanho do arquivo e o número de registros que se dá pelo tamanho do arquivo dividido pelo tamanho do registro.

# Diagrama de classes

No diagrama de classes podemos ver superficialmente a estrutura do projeto: CarMain.java inicia o programa instanciando CarController.java e chamando seu método execute().

  
Figura 22: Diagrama de classes do projeto

O método execute instancia MainWindow.java, todos os controllers e seus painéis.

Podemos ver que todos os painéis estendem PanelController.java e esse guarda uma instãncia de CarService.java que se comunica com FileManager.java

# conclusão

Utilizar persistência em arquivo pede mais cuidados e gera mais complexidade do que utilizar um SGBD que gerencia a persistência para o desenvolvedor. O que se ganha em performance perde-se em tempo, pois a manutenção do arquivo requer atenção a mais detalhes e fica exposta a pequenos bugs devido à complexidade do algoritmo.

No desenvolver do projeto começamos a perceber que a estrutura idealizada inicialmente não suporta mais as funcionalidades propostas e devemos reestruturar e refatorar o projeto ao longo do caminho.

Embora a lógica da persistência demanda muito mais minuciosidade a atenção no desenvolvimento a criação da interface e como os diferente componentes interagem entre si também demandou grande excesso de tempo, pois os bugs, quando aparecem, são mais difíceis de identificar a causa e a depuração das classes do pacote Swing e AWT não é simples como a depuração dos algorítimos criados pelos próprio desenvolvedor.

###### REFERÊNCIAS

JAVA DOCS**. Documentação Oficial da Linguagem Java.** Disponível em: https://docs.oracle.com/javase/8/docs/api/. Acesso em: 2 mai. 2020.