



**UNISUL**

**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

<b>1</b>	<b>INTRODUÇÃO.....</b>	<b>4</b>
<b>2</b>	<b>ESTRUTURA DO PROJETO .....</b>	<b>5</b>
<b>3</b>	<b>PADRÃO MVC.....</b>	<b>6</b>
<b>4</b>	<b>INCLUIR .....</b>	<b>8</b>
<b>5</b>	<b>LISTAR .....</b>	<b>9</b>
<b>6</b>	<b>CONSULTAR (BUSCAR, ALTERAR E EXCLUIR) .....</b>	<b>10</b>
<b>7</b>	<b>PROPRIEDADES .....</b>	<b>11</b>
<b>8</b>	<b>PERSISTÊNCIA EM ARQUIVO .....</b>	<b>12</b>
<b>9</b>	<b>DIAGRAMA DE CLASSES .....</b>	<b>16</b>
<b>10</b>	<b>CONCLUSÃO.....</b>	<b>17</b>
	<b>REFERÊNCIAS .....</b>	<b>18</b>

## **1 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).

## 2 ESTRUTURA DO PROJETO

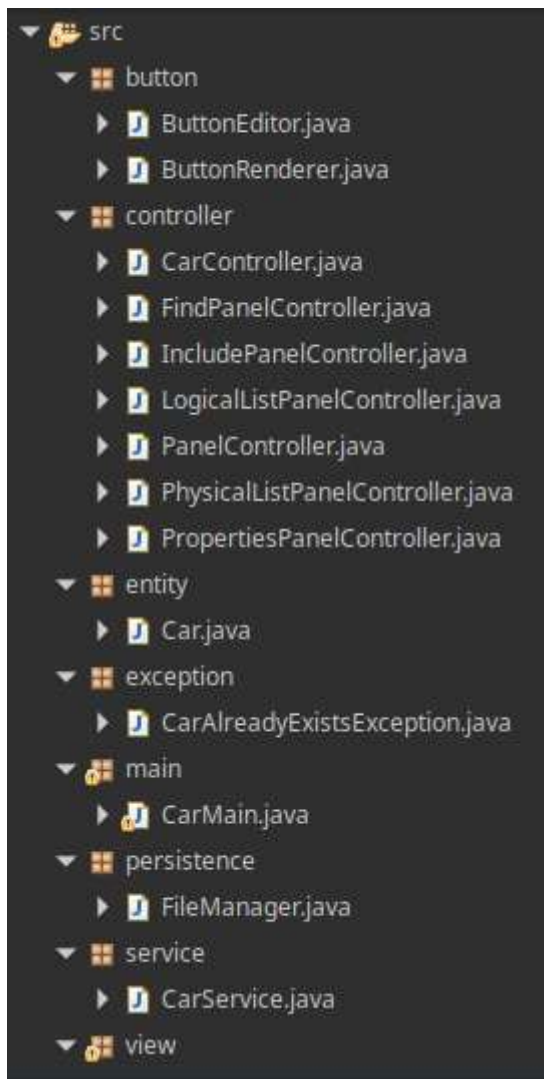


Figura 2: Estrutura dos pacotes e classes

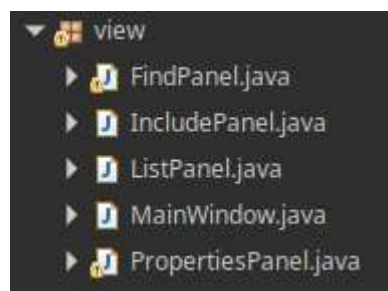


Figura 1: Estrutura dos pacotes e classes (parte 2)

Dividi o projeto em pacotes conforme demonstrado nas figuras 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.

### 3 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 os processos:

A screenshot of a code editor showing the CarMain.java file. The code is in Java and defines a public class CarMain with a main method. The main method creates a new CarController object and calls its execute method. The code is as follows:

```
1 package main;
2
3 import controller.CarController;
4
5
6 public class CarMain {
7
8     public static void main(String[] args) {
9         CarController carController = new CarController();
10        carController.execute();
11    }
12
13 }
14
```

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.

A screenshot of a code editor showing the execute() method of the CarController class. The method initializes several components: a MainWindow object, an IncludePanelController, a FindPanelController, and a JPanel parentPanel. It then sets the layout of the parentPanel to cardLayout and adds the controllers to it. Finally, it adds action listeners to the menu items in the MainWindow, each calling a corresponding show\* method. The code is as follows:

```
public void execute() {
    mainWindow = new MainWindow();
    includePanelController = new IncludePanelController();
    findPanelController = new FindPanelController();
    parentPanel = new JPanel();
    mainWindow.getContentPane().add(parentPanel);
    parentPanel.setLayout(cardLayout);
    parentPanel.add(includePanelController.includePanel, INCLUDE_PANEL);
    parentPanel.add(findPanelController.findPanel, FIND_PANEL);
    mainWindow.includeMenuItem.addActionListener(e -> showIncludePanel());
    mainWindow.findMenuItem.addActionListener(e -> showFindPanel());
    mainWindow.physicalListMenuItem.addActionListener(e -> showPhysicalListPanel());
    mainWindow.logicalListMenuItem.addActionListener(e -> showLogicalListPanel());
    mainWindow.propertiesMenuItem.addActionListener(e -> showPropertiesPanel());
}
```

Figura 4: Método execute() de CarController.java

Os painéis mostrados na interface seriam os seguintes:

- 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.

```
public class Car {  
    private String chassi;  
    private String brand;  
    private String model;  
    private Integer year;  
    private Double price;  
    private Integer deleted;  
}
```

Figura 5: Entidade Car.java

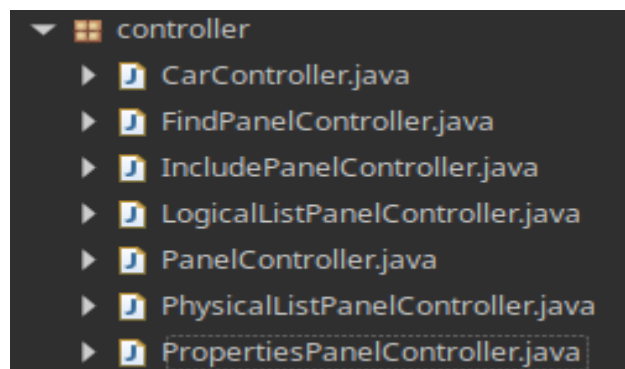
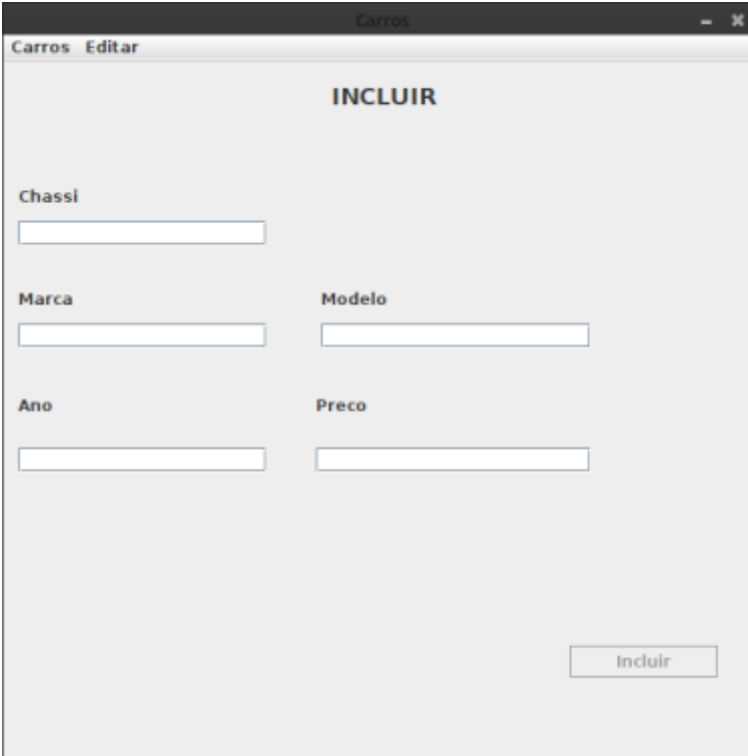


Figura 6: Pacote com todos os controllers

Como toda a lógica do programa ficaria muito extensa e confusa 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.

## 4 INCLUIR



The screenshot shows a Java Swing window titled "Carros". Inside, there's a panel titled "Carros Editar" which contains a sub-panel titled "INCLUIR". This sub-panel has five text input fields arranged in two columns: "Chassi" (top left), "Marca" (middle left), "Ano" (bottom left), "Modelo" (top right), and "Preço" (bottom right). A button labeled "Incluir" is located at the bottom right of the "INCLUIR" sub-panel.

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.

```
12
13
14 public IncludePanelController() {
15     includePanel = new IncludePanel();
16     includePanel.btnAddIncluir.addActionListener(e2 -> insertCar());
17 }
18
19 private void insertCar() {
20     String chassi = includePanel.txtChassi.getText();
21     String brand = includePanel.txtBrand.getText();
22     String model = includePanel.txtModel.getText();
23     Integer year = includePanel.txtYear.getText().equals("") ? 0 : Integer.valueOf(includePanel.txtYear.getText());
24     Double price = includePanel.txtPrice.getText().equals("") ? 0 : Double.valueOf(includePanel.txtPrice.getText());
25     if (chassi.equals("") || model.equals("")) {
26         JOptionPane.showMessageDialog(null, "Chassi e modelo são obrigatórios");
27         return;
28     }
29     Car newCar = new Car(chassi, brand, model, year, price);
30     try {
31         carService.save(newCar);
32         JOptionPane.showMessageDialog(null, "Carro inserido com sucesso");
33         clearFields();
34     } catch (CarAlreadyExistsException caee) {
35         JOptionPane.showMessageDialog(null, caee.getMessage(), "Atenção", JOptionPane.WARNING_MESSAGE);
36     } catch (Exception e) {
37         e.printStackTrace();
38     }
39 }
```

Figura 8: IncludePanelController.java



## 5 LISTAR

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

O algoritmo 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.

LISTAR (FÍSICO)					
Chassi	Marca	Modelo	Ano	Preco	Excluído
ASD456	Honda	Fit	2014	26559.2	Não
DFE234	BMW	I320	2019	150000.0	Não
HJK838	Ford	Fiesta	2016	41000.5	Sim

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

LISTAR (LÓGICO)				
Chassi	Marca	Modelo	Ano	Preco
ASD456	Honda	Fit	2014	26559.2
DFE234	BMW	I320	2019	150000.0

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

```
public List<Car> list(boolean showDeleted) {
    openFile();
    List<Car> cars = new ArrayList<Car>();
    try {
        raf.seek(0);
        while (raf.getFilePointer() < raf.length()) {
            String chassi = this.readString(CHASSI_STRING_SIZE);
            String brand = this.readString(BRAND_STRING_SIZE);
            String model = this.readString(MODEL_STRING_SIZE);
            int year = raf.readInt();
            double price = raf.readDouble();
            int deleted = raf.readInt();
            if (deleted == Car.NOT_DELETED || (deleted == Car.DELETED && showDeleted)) {
                cars.add(new Car(chassi, brand, model, year, price, deleted));
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        closeFile();
    }
    return cars;
}
```

Figura 11: Método list() de FileManager.java

## 6 CONSULTAR (BUSCAR, ALTERAR E EXCLUIR)

O campo “Consultar” permite realizar uma busca por Chassi ou Modelo, e preenche a tabela com os resultados que contenham a String de busca.



**BUSCAR**

Fit

Chassi	Marca	Modelo	Ano	Preço	Delete
ASD456	Honda	Fit	2014	26559.2	<b>Delet...</b>

Figura 12: Screenshot do painel "Buscar"

## 7 PROPRIEDADES



Figura 13: Screenshot do painel "Propriedades"

```
public long[] getProperties() {  
    openFile();  
    long[] properties = new long[2];  
    try {  
        properties[0] = raf.length();  
        properties[1] = raf.length() / REGISTER_SIZE;  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        closeFile();  
    }  
    return properties;  
}
```

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.

## 8 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.

```
public class FileManager {  
  
    RandomAccessFile raf;  
    static final String FILE_NAME = "file.dat";  
    static final int CHASSI_FIELD_SIZE = 24;  
    static final int BRAND_FIELD_SIZE = 24;  
    static final int MODEL_FIELD_SIZE = 24;  
    static final int YEAR_FIELD_SIZE = 4;  
    static final int PRICE_FIELD_SIZE = 8;  
    static final int DELETED_FIELD_SIZE = 4;  
    static final int REGISTER_SIZE = CHASSI_FIELD_SIZE + BRAND_FIELD_SIZE +  
                                     MODEL_FIELD_SIZE + YEAR_FIELD_SIZE +  
                                     PRICE_FIELD_SIZE + DELETED_FIELD_SIZE;  
  
    static final int CHASSI_STRING_SIZE = CHASSI_FIELD_SIZE / 2;  
    static final int BRAND_STRING_SIZE = BRAND_FIELD_SIZE / 2;  
    static final int MODEL_STRING_SIZE = MODEL_FIELD_SIZE / 2;  
}
```

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.

```
public void saveCar(Car car) throws Exception {  
    openFile();  
    if (doesChassiExist(car.getChassi())) {  
        closeFile();  
        throw new CarAlreadyExistsException();  
    }  
    try {  
        this.raf.seek(this.raf.length());  
        this.writeString(car.getChassi(), CHASSI_STRING_SIZE);  
        this.writeString(car.getBrand(), BRAND_STRING_SIZE);  
        this.writeString(car.getModel(), MODEL_STRING_SIZE);  
        this.raf.writeInt(car.getYear() == null ? 0 : car.getYear());  
        this.raf.writeDouble(car.getPrice() == null ? 0 : car.getPrice());  
        this.raf.writeInt(Car.NOT_DELETED);  
    } catch (IOException e) {  
        e.printStackTrace();  
        throw new Exception("Erro ao salvar o carro");  
    } finally {  
        closeFile();  
    }  
}
```

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.

```
public void updateCar(Car car) {
    openFile();
    try {
        int registerIndex = 1;
        while (raf.getFilePointer() < raf.length()) {
            String chassi = this.readString(CHASSI_STRING_SIZE);
            if (car.getChassi().trim().equals(chassi.trim())) {
                if (car.getBrand() != null) {
                    this.writeString(car.getBrand(), BRAND_STRING_SIZE);
                } else if (car.getModel() != null) {
                    raf.seek(raf.getFilePointer() + BRAND_FIELD_SIZE);
                    this.writeString(car.getModel(), MODEL_STRING_SIZE);
                } else if (car.getYear() != null) {
                    raf.seek(raf.getFilePointer() + BRAND_FIELD_SIZE + MODEL_FIELD_SIZE);
                    raf.writeInt(car.getYear());
                } else if (car.getPrice() != null) {
                    raf.seek(raf.getFilePointer() + BRAND_FIELD_SIZE + MODEL_FIELD_SIZE + YEAR_FIELD_SIZE);
                    raf.writeDouble(car.getPrice());
                } else if (car.getDeleted() != null) {
                    raf.seek(raf.getFilePointer() + BRAND_FIELD_SIZE + MODEL_FIELD_SIZE + YEAR_FIELD_SIZE
                        + PRICE_FIELD_SIZE);
                    raf.writeInt(Car.DELETED);
                }
                break;
            }
            raf.seek(registerIndex * REGISTER_SIZE);
            registerIndex++;
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        closeFile();
    }
}
```

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).

```
public void deleteCar(Car car) {
    car.setDeleted(Car.DELETED);
    updateCar(car);
}
```

Figura 18: Método `deleteCar()` de `FileManager.java`



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

```
/**
 * Retornar lista de carros em que o chassi ou modelo dão match com a string
 * @param text
 * @return
 */
public List<Car> findCarsByChassiOrModel(String text) {
    if (text.isEmpty()) return new ArrayList<Car>();
    openFile();
    List<Car> cars = new ArrayList<Car>();
    try {
        raf.seek(0);
        while (raf.getFilePointer() < raf.length()) {
            String chassi = this.readString(CHASSI_STRING_SIZE);
            String brand = this.readString(BRAND_STRING_SIZE);
            String model = this.readString(MODEL_STRING_SIZE);
            int year = raf.readInt();
            double price = raf.readDouble();
            int deleted = raf.readInt();
            if (deleted == Car.NOT_DELETED && (chassi.contains(text) || model.contains(text))) {
                cars.add(new Car(chassi, brand, model, year, price, deleted));
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    closeFile();
    return cars;
}
```

Figura 20: Método findCarByChassiOrModel() de FileManager.java

```
/**
 * Retorna todos os carros salvos em arquivo
 * @param showDeleted Se inclui os deletados ou não
 * @return
 */
public List<Car> list(boolean showDeleted) {
    openFile();
    List<Car> cars = new ArrayList<Car>();
    try {
        raf.seek(0);
        while (raf.getFilePointer() < raf.length()) {
            String chassi = this.readString(CHASSI_STRING_SIZE);
            String brand = this.readString(BRAND_STRING_SIZE);
            String model = this.readString(MODEL_STRING_SIZE);
            int year = raf.readInt();
            double price = raf.readDouble();
            int deleted = raf.readInt();
            if (deleted == Car.NOT_DELETED || (deleted == Car.DELETED && showDeleted)) {
                cars.add(new Car(chassi, brand, model, year, price, deleted));
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        closeFile();
    }
    return cars;
}
```

Figura 19: 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.

```
/**
 * Retorna as propriedades do arquivo em um array de tamanho 2 sendo:
 * [0] = tamanho do arquivo;
 * [1] = numero de registros;
 * @return
 */
public long[] getProperties() {
    openFile();
    long[] properties = new long[2];
    try {
        properties[0] = raf.length();
        properties[1] = raf.length() / REGISTER_SIZE;
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        closeFile();
    }
    return properties;
}
```

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.

## 9 DIAGRAMA DE CLASSES

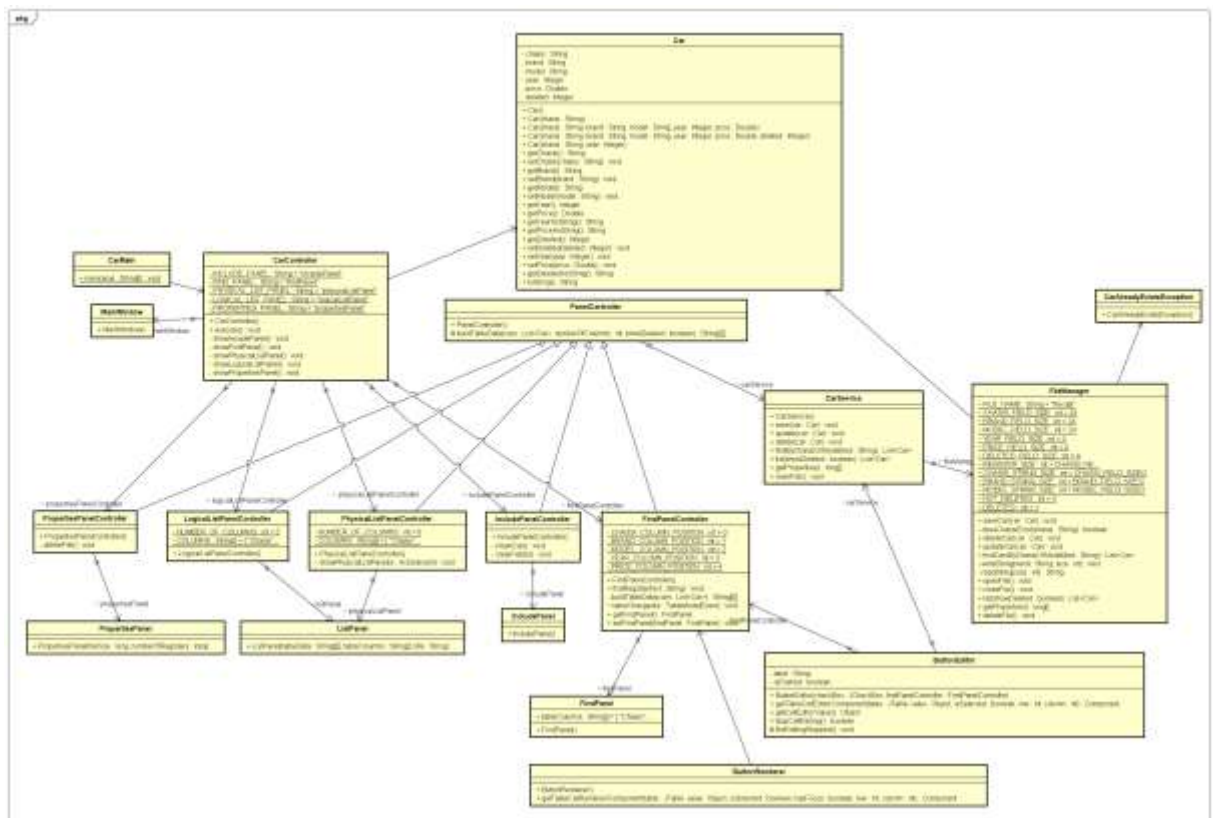


Figura 22: Diagrama de classes do projeto

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

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 instância de CarService.java que se comunica com FileManager.java



## 10 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 diferentes 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 algoritmos criados pelo 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.