Lab 2
Car Showroom Management

Author: SE203334 - Hà Nguyễn Tiến Đạt
Course: Lab211 | Instructor: Ms. Thân Thị Ngọc Vân

Mục lục

I. Phân tích và thiết kế

1. Phân tích dữ liệu

Dựa trên yêu cầu đề bài và ngữ cảnh bài toán là quản lý showroom ô tô BMW, công việc đầu tiên là cần phải xác định các thành phần chính trong hệ thống:

a. Các đối tượng *(entities)*: **Brand**, **Car**, **Menu**. Chi tiết:

- **Brand**: là đối tượng dữ liệu chính, quản lý thông tin thương hiệu xe BMW trong showroom. Sau khi xem xét cấu trúc của tập tin **brands.txt** được cung cấp sẵn kèm theo đề bài của đối tượng dữ liệu này, ta thấy danh sách các thương hiệu xe được mô tả qua nhiều dòng, mỗi dòng là một thương hiệu độc lập với các thuộc tính phụ thuộc bao gồm:
  - **ID** of brand *(là unique key, không thể thay đổi sau khi tạo)*
  - Name of brand
  - Sound Brand
  - **Price** *(giá, tính bằng Billion).*

- **Car**: là đối tượng chính mô tả thông tin chi tiết của từng chiếc xe trong showroom. Tương tự, xem xét cấu trúc của tập tin **cars.txt** được cung cấp sẵn kèm theo đề bài, ta thấy danh sách các xe cũng được mô tả qua nhiều dòng, mỗi dòng là một xe độc lập với các thuộc tính phụ thuộc gồm:
  - **ID** of car
  - **ID** of brand *(liên kết với Brand)*
  - **Color**
  - **Frame ID** *(định dạng Fxxxxx, là duy nhất, ví dụ: F00000)*
  - **Engine ID** *(định dạng Exxxxx, là duy nhất, ví dụ: E00000)*

b. Các thao tác *(operations)*: **Input** *(get data from keyboard by user)*, **Add**, **Update**, **Search**, **Get Info** *(object Information)*, **Display** *(display all data in the list)*, **Remove**, **Filter**, **Load** *(get data from file)*, **Save** *(store data to file)*

Có hai nhóm thao tác chính:

- i. Tác động lên một đối tượng độc lập (Brand hoặc Car): ***Input***, ***Get_Info***
- ii. Tác động lên một nhóm đối tượng (Collections - danh sách Brand hoặc Car): ***Add***, ***Update***, ***Remove***, ***Search***, ***Filter***, ***Load/Save***.

2. Thiết kế

Các class sẽ được thiết kế dựa trên các entity đã xác định. Mỗi class sẽ *đóng gói* các ***thuộc tính*** (attributes) và ***hành vi*** (methods) liên quan.

Theo nguyên tắc *encapsulation*, các attributes được khai báo ***private*** để bảo vệ dữ liệu, chỉ cho phép truy cập thông qua các getter/setter. Điều này là để giúp kiểm soát được cách dữ liệu được truy cập và thay đổi, dễ dàng bảo trì khi có thay đổi logic và tăng tính bảo mật. Ta có các thiết kế như sau:

a. Brand class

Brand
- brandID: String - brandName: String - soundBrand: String - price: double
+ Brand() + Brand(fields) + Getters/Setters + toString()

b. Car class

Car
- carID: String - brand: Brand - color: String - frameID: String - engineID: String
+ Car() + Car(fields) + Getters/Setters + toString()

c. Lựa chọn cấu trúc quản lý dữ liệu

Trong quá trình hoạt động, chương trình cần lưu trữ và thao tác với nhiều đối tượng cùng loại (nhiều thương hiệu, nhiều xe). Do đó, cần có một cấu trúc dữ liệu phù hợp để đáp ứng cho nhu cầu này, các cấu trúc có thể liệt kê như: mảng, linked list,... là những lựa chọn phù hợp.

Mặt khác, có 2 loại quan hệ ta có thể sử dụng là "***has A***" và "***is A***":

- "**has A**": đối tượng chứa một collection bên trong như một thuộc tính. Class sở hữu collection nhưng không phải là collection.
- "**is A**": đối tượng chính là một collection thông qua kế thừa *(inheritance)*. Class vừa là collection, vừa có thể mở rộng thêm behaviors.

  Dự án chọn "**is A**" vì khi kế thừa từ ArrayList, **BrandList** và **CarList** tự động có sẵn các methods như `add()`, `remove()`, `size()`, `get()`... mà không cần phải viết lại. Ta chỉ cần thêm các methods như `searchID()`, `addBrand()`, `updateCar()`... mà sẽ được phân tích ở những phần sau.

Ta có thêm một số class với thiết kế như sau:

BrandList **extends** ArrayList
+ ***BrandList***() + ***searchID***(String bID): int + ***isUnique***(String id): boolean + ***getUserChoice***(): Brand + ***displayBrands***(ArrayList<Brand> brands): void + ***listBrands***(): void + ***addBrand***(): void

+ ***searchBrand***(): void + ***updateBrand***(): void + ***listBrandsLessThanPrice***(): void + ***loadFromFile***(String filename): void + ***saveToFile***(String filename): void

CarList **extends** ArrayList
- brandList: BrandList
+ ***CarList***(BrandList bList) + ***searchID***(String carID): int + ***searchFrame***(String fID): int + ***searchEngine***(String eID): int + ***isUnique***(String id, String frameID, String engineID): boolean + ***displayCars***(ArrayList cars): void + ***displayCarInfo***(Car car): void + ***listCars***(): void + ***printBasedBrandName***(): void + ***addCar***(): void + ***removeCar***(): void + ***updateCar***(): void + ***listCarsWithColor***(): void + ***loadFromFile***(String filename): void + ***saveToFile***(String filename): void

d. Công cụ nhập và kiểm tra dữ liệu

Nên kiểm tra dữ liệu trước khi đưa vào Object để đảm bảo tính đúng đắn và hợp lệ theo yêu cầu của project, ta cần xây dựng interface **iConstants** như thiết kế sau:

<<interface>> iConstants
+ CasualString: String <<final>> + StringButOnlyAlphabetAllowed: String <<final>> + StringButAlphabetAndNumberAllowed: String <<final>> + StringOnlyForFrameFormat: String <<final>> + StringOnlyForEngineFormat: String <<final>> + thePrice: String <<final>>

Giải thích các pattern:

- *CasualString*: chấp nhận chữ cái *(bao gồm cả chữ thường, hoa)*, số, dấu gạch ngang, gạch dưới, ngoặc đơn và khoảng trắng.
- *StringButOnlyAlphabetAllowed*: chỉ chấp nhận chữ cái và khoảng trắng.
- *StringButAlphabetAndNumberAllowed*: chỉ chấp nhận chữ cái và chữ số.
- *StringOnlyForFrameFormat*: format đặc biệt dành cho Frame ID *(Fxxxxx)*.
- *StringOnlyForEngineFormat*: format đặc biệt dành cho Engine ID *(Exxxxx)*.
- *thePrice*: chỉ chấp nhận số thực dương khác 0 với phần thập phân tùy chọn.

Class **Inputter** chứa các phương thức phục vụ cho việc hiển thị thông báo và nhập dữ liệu từ bàn phím, kết hợp với **iConstants** để validation:

Inputter
- sc: Scanner <<static>>
+ ***getString***(String msg, String pattern): String <<static>> + ***readUniqueStringWithPattern***(String msg, String pattern, String formatErrMsg, String uniqueErrMsg, Predicate existsChecker): String <<static>> + ***getStringUpdate***(String msg, String pattern): String <<static>> + ***readString***(String msg): String <<static>> + ***confirmation***(String msg): boolean <<static>> + ***getInt***(String msg, int min, int max): int <<static>> + ***readPositiveDouble***(String msg, String errMsg): double <<static>> + ***capitalizeWords***(String str): String <<static>>

Mô tả các phương thức:

- *getString()*: Nhập chuỗi không rỗng và phải khớp với pattern regex.
- *readUniqueStringWithPattern()*: Nhập chuỗi phải khớp pattern và đảm bảo tính duy nhất (unique).
- *getStringUpdate()*: Dùng cho update, nếu input rỗng thì giữ nguyên giá trị cũ.
- *readString()*: Đọc chuỗi, có thể rỗng.
- *confirmation()*: Yêu cầu xác nhận (Y/N).
- *getInt()*: Nhập số nguyên trong khoảng **[min, max]**.
- *readPositiveDouble()*: Nhập số thực dương.
- *capitalizeWords()*: Viết hoa chữ cái đầu mỗi từ, các chữ cái sau đều là chữ thường.

II. Triển khai kỹ thuật và điều chỉnh thiết kế

1. Kiểm tra dữ liệu

Về nguyên tắc, dữ liệu khi được đưa vào chương trình phải được đảm bảo tính hợp lệ. Vì vậy sau khi được nhập vào bởi người dùng, dữ liệu cần phải được kiểm tra.

a. Kiểm tra format với Regex Pattern

Sử dụng interface **iConstants** để định nghĩa các pattern validation. Ví dụ:

- Kiểm tra id của Brand:

```
// Check Brand ID
String brandID = Inputter.getString("Enter Brand ID: ",
iConstants.CasualString);
```

- Kiểm tra id của Frame:

```
// Check Frame ID format (F00000)
if (frameID.matches(iConstants.StringOnlyForFrameAndEngineFormat)) {
    // Valid format
}
```

b. Kiểm tra tính duy nhất (Uniqueness)

Predicate (hay Vị ngữ) là một *functional interface* trong Java đại diện cho một hàm nhận một tham số và trả về giá trị boolean (true/false). Trong validation, Predicate được dùng để kiểm tra một điều kiện có thỏa mãn hay không.

Ta sẽ sử dụng `Predicate<String>` để kiểm tra xem ID đã tồn tại hay chưa. Ví dụ:

```
String carID = Inputter.readUniqueStringWithPattern(
    "Enter Car ID: ",
    iConstants.StringButAlphabetAndNumberAllowed,
    "Invalid Input!",
    "Car ID already exists!",
    id → searchID(id) ⩾ 0     // lambda expression (Predicate)
);
```

Ở đoạn code trên, `id → searchID(id) ⩾ 0` là một *lambda expression* thay thế cho Predicate, nó có nghĩa là nhận tham số **`id`** *(String)*, sau đó gọi method **`searchID(id)`** để tìm vị trí và cuối cùng trả về một kết quả boolean:

- Nếu `⩾ 0` nghĩa là ID này đã tồn tại trong danh sách → trả về **`true`** *(reject vì không unique)*. Method `readUniqueStringWithPattern()` sẽ tiếp tục yêu cầu nhập lại cho đến khi Predicate trả về false.
- Nếu `< 0` nghĩa là id đó chưa tồn tại → trả về **`false`** *(accept vì unique)*.

c. Kiểm tra ràng buộc dữ liệu

- *Brand ID*: Phải unique, không rỗng, khớp pattern.
- *Brand Name*: Không rỗng.
- *Sound Brand*: Không rỗng, chỉ chứa chữ cái và khoảng trắng
- *Price*: Phải là số thực dương khác 0.
- *Color*: Không rỗng, không chứa số.
- *Frame ID*: Format F00000, phải unique.
- *Engine ID*: Format E00000, phải unique.

2. File I/O Operations

Các thao tác đọc/ghi file luôn tiềm ẩn rủi ro (file không tồn tại, quyền truy cập bị từ chối, lỗi định dạng dữ liệu,...), do đó cần sử dụng **try-catch** để xử lý exception và đảm bảo chương trình không crash.

a. Đọc dữ liệu từ file

Để load data từ file, ta sẽ tạo một method với tên gọi **loadFromFile** và thực hiện theo các bước sau:

1. *Kiểm tra file có tồn tại*: Nếu file không tồn tại, tạo một file rỗng.
2. *Đọc từng dòng*: Sử dụng **`BufferedReader`** để đọc data.
3. *Parse và validate*: Mỗi dòng được tách thành các field, kiểm tra tính hợp lệ trước khi add vào list.
4. *Bỏ qua dòng lỗi*: Nếu dòng nào không hợp lệ (thiếu field, sai format, trùng ID), bỏ qua và tiếp tục.

```
public void loadFromFile(String filename) {
    try {
        File f = new File(filename);
        if (!f.exists()) { // Kiem tra file co ton tai hay khong
          f.createNewFile();
          return;
        }

        BufferedReader r = new BufferedReader(new FileReader(f));
        String line;
        // Doc tung dong trong file cho den khi doc het
        while ((line = r.readLine()) ≠ null) {
            String[] parts = line.split(",");

            // Parse and validate data
            // Neu hop le thì add vao list
        }

        // dong luong sau khi doc de giai phong bo nho
        r.close();
```

```
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

b. Ghi dữ liệu vào file

Để save data vào file, ta sẽ tạo một hàm với tên gọi **saveToFile**:

```
public void saveToFile(String filename) {
    try {
      PrintWriter w = new PrintWriter(new FileWriter(filename));

      for (Brand brand : this) {
        // ghi data
      }
      w.close();
    } catch (Exception e) {
        System.err.println("Error: " + e.getMessage());
    }
}
```

3. Các chức năng chính

3.1. Add

a. Add a new brand

Chức năng này cho phép thêm brand mới vào danh sách:

```
public void addBrand() {
    String id = Inputter.readUniqueStringWithPattern(
        "Enter Brand ID: ",
        iConstants.CasualString,
        "Invalid format!",
        "Brand ID already exists! Please enter a different ID.",
        brandId → searchID(brandId) ⩾ 0
    );

    String name = Inputter.getString("Enter brand name: ",
    iConstants.CasualString);

    String sound = Inputter.getString("Enter Sound Brand: ",
    iConstants.StringButOnlyAlphabetAllowed);

    double price = Inputter.readPositiveDouble(
      "Enter Price (e.g: 3.0): ",
      "Price must be positive!"
    );
```

```
    // Tao object Brand moi
    Brand newBrand = new Brand(id.toUpperCase(), name,
    Inputter.capitalizeWords(sound), price);
    this.add(newBrand); // Them vao danh sach
}
```

Lưu ý:

- Brand ID được chuyển về chữ HOA để đồng nhất.
- Sound Brand được capitalize (vd: Harman Kardon).
- Price phải > 0 (được validate bởi **`readPositiveDouble()`**).

b. Add a new car

Chức năng này phức tạp hơn vì phải chọn brand từ menu và validate nhiều field:

```
public void addCar() {
    // nhap car id (unique, khong rong)

    // Chon brand tu menu
    Brand brand = brandList.getUserChoice();

    // Nhap color (dung ham getString())

    // Nhap Frame ID (format F00000, unique)
    String frameID = Inputter.readUniqueStringWithPattern(
      "Enter Frame ID (F00000 format): ",
      iConstants.StringOnlyForFrameFormat,
      "Invalid Input!",
      "Frame ID must be in F00000 format and unique!",
      fId -> !fId.matches(iConstants.StringOnlyForFrameFormat) ||
      searchFrame(fId) >= 0
    );

    // Nhap engine ID (cach thuc tuong tu nhu frame ID)

    // Tao Object Car moi
    Car newCar = new Car(carID.toUpperCase(), brand,
    color.toLowerCase(), frameID, engineID);

    this.add(newCar);   // Them vao danh sach

    System.out.println("Car added successfully!");
}
```

Lưu ý: cần phải *predicate validation* cho Frame/Engine ID, ví dụ:

```
fId -> !fId.matches(...) || searchFrame(fId) >= 0
```

Đoạn code trên sẽ trả về **true** (reject) nếu: `!fId.matches(...)` *(sai format)* HOẶC `searchFrame(fId) ⩾ 0` *(đã tồn tại)*.

3.2. Search

a. Tìm brand theo ID

Cần tìm kiếm brand theo ID chính xác. Nếu tìm thấy, hiển thị đầy đủ thông tin brand:

```
public void searchBrand() {
    String brandID = Inputter.getString("Enter Brand ID to search: ",
    iConstants.CasualString);

    int pos = searchID(brandID); // tim vi tri brand trong danh sach

    if (pos < 0) System.out.println("This brand does not exist!");
      else {
          Brand found = this.get(pos);
          System.out.printf("%s %s %s %.1fB\n",
          found.getBrandID(), found.getBrandName(),
          found.getSoundBrand(), found.getPrice());
    }
}
```

b. Tìm cars theo 1 phần tên brand

Sau khi người dùng nhập một chuỗi bất kì *(ở đây input của người dùng sẽ được đưa vào chuỗi có tên là partOfBrandName)*. Ta sẽ thực hiện lấy input của người dùng và so sánh nó với từng tên brand có trong danh sách. Để tránh xảy ra trường hợp lỗi không mong muốn, ta nên thực hiện đưa tất cả các chữ về cùng một format *(đều cùng là chữ hoa hoặc thường)*.

```
for (Car c : this) {
    String partOfBrandName = Inputter.getString(
      "Enter part of brand name: ",
      iConstants.CasualString
    );

    ArrayList<Car> cars = new ArrayList<>();

    // Lay ten brand cua xe hien tai va chuyen thanh chu thuong
    String brandName = c.getBrand().getBrandName().toLowerCase();

    // Kiem tra xem ten brand co chua phan text ma user nhap khong
    if (brandName.contains(partOfBrandName.toLowerCase())) {
        cars.add(c);
    }
}
```

3.3. Sorting và Filtering

a. Lọc các brand theo giá

Lọc các brand có giá nhỏ hơn hoặc bằng giá trị price nhập vào:

```
public void listBrandsLessThanPrice() {
    double price = Inputter.readPositiveDouble("Enter Price: ",
      "Price must be positive!"
    );

    ArrayList<Brand> result = new ArrayList<>();

    for (Brand b : this) {
        if (b.getPrice() ⩽ price) result.add(b);
    }

    displayBrands(result);
}
```

b. Sắp xếp cars theo giá trị tăng dần của brand names *(ascending)*, nếu brand names trùng thì sắp xếp theo giá trị price giảm dần *(descending)*.

Chức năng này yêu cầu sắp xếp xe theo ***brand name*** tăng dần (A → Z), nếu trùng brand thì sắp xếp theo ***price*** giảm dần (cao → thấp). Để thực hiện, ta cần tạo bản sao của danh sách gốc nhằm tránh làm thay đổi thứ tự ban đầu, sau đó áp dụng method `sort()` kết hợp lambda expression để định nghĩa logic so sánh hai mức. Mức đầu tiên so sánh brand name bằng **`compareToIgnoreCase`**() (không phân biệt hoa/thường). Nếu brand giống nhau, chuyển sang mức hai là so sánh price bằng **`Double.compare`**() với tham số đảo ngược (c2, c1) để đạt thứ tự giảm dần.

```
public void listCars() {
    // tao bang sao danh sach car
    ArrayList<Car> cars = new ArrayList<>(this);
    cars.sort((c1, c2) → {
        // so sanh ten brand khong phan biet chu hoa, thuong
        int brandCompare = c1.getBrand().getBrandName()
                .compareToIgnoreCase(c2.getBrand().getBrandName());
        // neu brand khac nhau → return ket qua so sanh brand
        if (brandCompare ≠ 0) return brandCompare;
        // Neu brand trung nhau, sap xep theo price giam dan
        return Double.compare(
            c2.getBrand().getPrice(), c1.getBrand().getPrice()
        );
    });
    displayCars(cars);
}
```

c. Hiển thị danh sách cars theo màu được chọn

Tìm và lọc tất cả các xe có màu khớp với màu mà user nhập vào. Ví dụ, nếu người dùng nhập màu đỏ (Red) thì sẽ trả về tất cả những chiếc xe có màu đỏ.

```
public void listCarsWithColor() {
    String color = Inputter.getString(
      "Enter color: ",
      iConstants.StringButOnlyAlphabetAllowed
    );

    ArrayList<Car> cars = new ArrayList<>();

    for (Car c : this) {
        if (c.getColor().equalsIgnoreCase(color)) cars.add(c);
    }

    if (cars.isEmpty()) {
        System.out.print("No car found with color: " + color);
    } else displayCars(cars);
}
```

3.4. Update

Khi cập nhật, sẽ có trường hợp user muốn thay đổi giá trị của một hay nhiều dữ liệu nào đó (không nhất thiết phải là thay đổi toàn bộ). Thay vì bắt buộc người dùng nhập giá trị mới cho tất cả các field, ta nên cho phép họ nhấn Enter (chuỗi rỗng) để giữ nguyên giá trị hiện tại của những field không muốn thay đổi.

Ví dụ: Cập nhật field color trong hàm **updateCar**.

```
String newColor = Inputter.getStringUpdate(
    "Enter new Color: ", pattern);

if (!newColor.isEmpty()) {
    currentCar.setColor(Inputter.capitalizeWords(newColor));
}

// Neu rong thi khong thay doi gi
```

Cơ chế hoạt động của hàm **getStringUpdate()**:

```
public static String getStringUpdate(String msg, String pattern) {
    while (true) {
        System.out.print(msg);

        // doc va cat khoang trang
        String input = sc.nextLine().trim();
```

```
        if (input.isEmpty()) {
            System.out.println("No change!");
            return input; // tra ve chuoi rong (khong thay doi)
        } else {
            if (input.matches(pattern)) {
                System.out.println("New data has been updated.");
                return input;
            } else System.err.println("Input not valid.");
        }
    }
}
```

- Nếu người dùng nhấn Enter *(chuỗi rỗng)*: trả về "" và in "No change!".
- Nếu là một *chuỗi hợp lệ*: trả về chuỗi đó và in "New data updated".
- Nếu *không hợp lệ:* yêu cầu nhập lại.

a. Update brand

Trước khi update, cần tìm brand theo ID và kiểm tra sự tồn tại. Nếu không tồn tại, dừng ngay để tránh chương trình báo lỗi. Mỗi field vẫn nên được validate qua pattern, đảm bảo dữ liệu mới (nếu có) vẫn hợp lệ. Sound Brand nên được capitalize để đồng nhất format (vd: "harman kardon" → "Harman Kardon").

```
public void updateBrand() {
    String brandID = Inputter.getString(
    "Enter Brand ID to update: ", iConstants.CasualString);

    int pos = searchID(brandID); // tim vi tri brand trong danh sach

    if (pos < 0) {
        System.out.println("\nThis brand does not exist!");
        return;
    }

    // Update Brand Name
    String name = Inputter.getStringUpdate(
      "Enter new Brand Name: ",
      iConstants.CasualString
    );
    if (!name.isEmpty()) {
        this.get(pos).setBrandName(name);
    }

    // Update Sound Brand
    String sound = Inputter.getStringUpdate(
      "Enter new Sound Brand: ",
      iConstants.StringButOnlyAlphabetAllowed
    );
```

```
    if (!sound.isEmpty()) {
        this.get(pos).setSoundBrand(Inputter.capitalizeWords(sound));
    }

    // Update Price
    String priceStr = Inputter.getStringUpdate(
      "Enter new Price: ",
      iConstants.thePrice
    );
    if (!priceStr.isEmpty()) {
        this.get(pos).setPrice(Double.parseDouble(priceStr));
    }

    System.out.println("\nBrand updated successfully!");
}
```

b. Update car

Tương tự với chức năng Update của brand, nhưng Update car *phức tạp hơn* do phải xử lý nhiều yếu tố.

Brand là một **relational data** *(object reference)*, không phải là *primitive type* như **String** hay **double**, nên việc thay đổi brand cần có sự xác nhận từ user. Ngoài ra, Frame ID và Engine ID phải là *duy nhất* trong toàn bộ danh sách, ngoại trừ chính xe đang được update để tránh báo lỗi khi user giữ nguyên giá trị cũ.

Để thực hiện việc cập nhật, đầu tiên hệ thống cần hiển thị thông tin hiện tại của xe để user có thể đối chiếu và quyết định xem cần thay đổi những gì. Tiếp theo, vì brand là thuộc tính quan trọng ảnh hưởng đến giá trị xe, hệ thống sẽ hỏi trước "*Bạn có muốn update brand không?*". Nếu user chọn "Không" thì chương trình thoát. Ngược lại, nếu chọn "Có", hệ thống gọi **`getUserChoice`**() để hiển thị menu cho phép chọn brand mới.

```
public void updateCar() {
    String carID = Inputter.readString("Enter Car ID to update: ");
    int pos = searchID(carID);

    if (pos < 0) {
        System.err.println("Car ID '" + carID + "' does not exist!");
        return;
    }

    Car currentCar = this.get(pos);

    System.out.println("Current Car Information:");
    displayCarInfo(currentCar);
```

```
    boolean updateBrand = Inputter.confirmation(
      "Do you want to update the brand? (Y/N): "
    );

    if (!updateBrand) {
        System.out.println("Car update cancelled.");
        return;
    }

    // Update brand
    Brand newBrand = brandList.getUserChoice();
    if (newBrand ≠ null) currentCar.setBrand(newBrand);

    // Update Color
    String newColor = Inputter.getStringUpdate(
      "Enter new Color: ",
      iConstants.StringButOnlyAlphabetAllowed
    );
    if (!newColor.isEmpty())
    currentCar.setColor(Inputter.capitalizeWords(newColor));

    // Update Frame ID
    String newFrameID = Inputter.getStringUpdate(
      "Enter new Frame ID (F00000 format): ",
      iConstants.StringOnlyForFrameFormat
    );

    if (!newFrameID.isEmpty()) {
        int framePos = searchFrame(newFrameID);
        if (framePos ⩾ 0 && framePos ≠ pos) {
          System.err.println("Frame ID already exists!");
        } else currentCar.setFrameID(newFrameID.toUpperCase());
    }

    // Update Engine ID
    String newEngineID = Inputter.getStringUpdate(
      "Enter new Engine ID (E00000 format): ",
      iConstants.StringOnlyForEngineFormat
    );

    if (!newEngineID.isEmpty()) {
        int enginePos = searchEngine(newEngineID);
        if (enginePos ⩾ 0 && enginePos ≠ pos) {
          System.err.println("Engine ID already exists!");
      } else currentCar.setEngineID(newEngineID.toUpperCase());
    }

    System.out.println("Car updated successfully!");
}
```

3.5. Remove a car by ID

Ta nên yêu cầu confirmation để tránh xóa nhầm dữ liệu quan trọng:

```
public void removeCar() {
    String removeID = Inputter.readString("Enter car id to remove: ");

    // Lay thong tin car can xoa
    Car removedCar = this.get(pos);

    // Hien thi thong tin de xac nhan va yeu cau xac nhan
    boolean confirm = Inputter.confirmation(
        "Are you sure you want to remove this car? (Y/N): "
    );

    if (confirm) this.remove(pos); // Thuc hien xoa neu confirm
      else System.out.println("Car removal cancelled.");
}
```

Tại sao ta lại cần phải confirmation?

Chúng ta cần **confirmation** là để tránh phải xóa nhầm dữ liệu quan trọng và để cho user có cơ hội kiểm tra lại thông tin trước khi xóa.

4. Menu-driven Architecture

Class Manager chính là nơi để điều khiển flow chính của chương trình. Nó cũng sẽ đảm nhiệm hai vai trò: tải dữ liệu từ các tệp ***brands.txt***, ***cars.txt*** lúc khởi động và quản lý việc lưu dữ liệu vào các tệp này khi cần thiết.

Manager
+ main(String[] args): void <<static>>

III. Mô hình triển khai mã nguồn

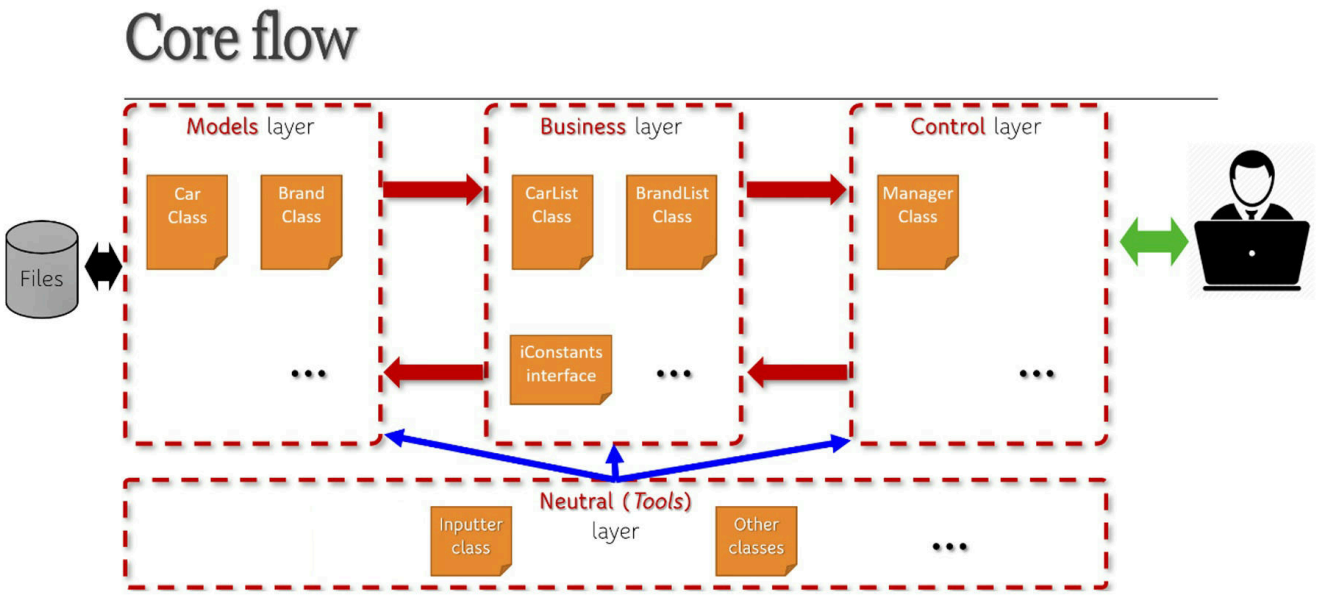
Trong các dự án phần mềm thực tế, việc quản lý source code sẽ trở nên ngày càng phức tạp khi số lượng files mã nguồn dần tăng lên cùng với việc nhiều nhà phát triển cùng làm việc chung với những file mã nguồn đấy. Vấn đề không chỉ dừng lại ở giai đoạn phát triển mà còn ảnh hưởng đến khả năng bảo trì và mở rộng hệ thống về sau.

Để giải quyết vấn đề này, project áp dụng ***n-Layer Architecture*** (mô hình tổ chức code theo các tầng), mỗi tầng đảm nhiệm vai trò cụ thể và được triển khai bằng một Java package để gom nhóm các lớp có liên quan.

Dự án Car Showroom này được tổ chức thành 4 tầng chính:

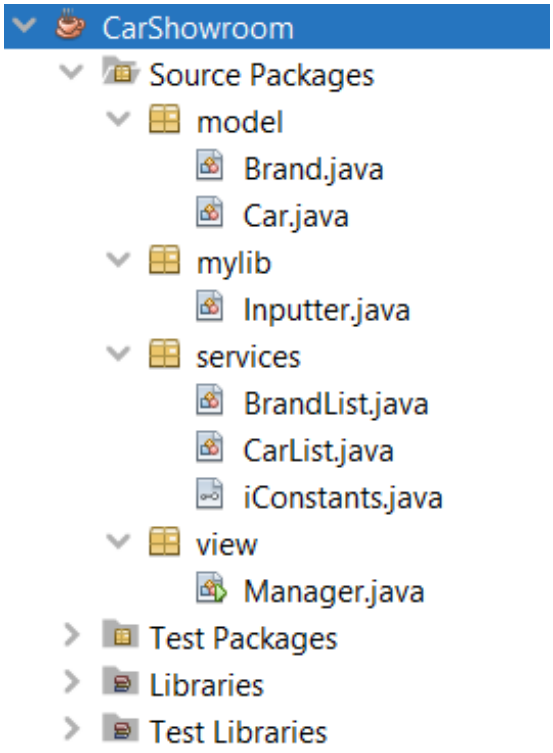
- Những mã nguồn chỉ phục vụ cho việc mô tả dữ liệu chính như: **Brand** class, **Car** class thì được quản lý bởi **model** pakage, sẽ được gọi là tầng thứ nhất: ***models layer***.
- Những mã nguồn chuyên thực hiện các nghiệp vụ (add, showAll, searchByName, update, delete, filter, …) có liên quan đến **models layer** như Brand class, Car class thì được quản lý bởi **services** package, sẽ được gọi là tầng thứ hai: ***business layer***.
- **Manager** class có nhiệm vụ xây dựng giao diện, xử lý tương tác với người dùng đồng thời gọi **business layer** để thực thi yêu cầu thì được quản lý bởi **view** package, tầng thứ ba này sẽ được gọi là ***control layer***.
- Những mã nguồn còn lại không tham gia vào quy trình xử lý theo các tầng đã thiết kế nhưng có thể tham gia ở bất kỳ nơi nào trong chương trình để phục vụ cho quá trình xử lý thì được quản lý bởi **mylib** package và tầng này sẽ được gọi là ***neutral layer***.

Sơ đồ thiết kế được dùng cho project sẽ có dạng như sau:



Với **Car Showroom** project, áp dụng thiết kế trên sẽ có cấu trúc mô tả sau:

```
src
├── model (Models layer)
│   ├── Brand.java
│   └── Car.java
├── services (Business layer)
│   ├── BrandList.java
│   ├── CarList.java
│   └── iConstants.java
├── view (Control layer)
│   └── Manager.java
└── mylib (Neutral layer)
    └── Inputter.java
```



IV. Kết luận

Chương trình **Car Showroom Management** đã được triển khai thành công với đầy đủ **13 chức năng** theo yêu cầu đề bài. Hệ thống áp dụng các nguyên tắc OOP, đảm bảo tính mở rộng và bảo trì. Validation được thực hiện nghiêm ngặt ở mọi điểm nhập liệu và đảm bảo tính toàn vẹn dữ liệu.