TURKCELL GYGY 3.0

HOMEWORK REPORT DATED 20.03.2024

APPLICATION OF SOLID SOFTWARE DEVELOPMENT PRINCIPLES

Görkem Rıdvan ARIK

March 22, 2024

Table of Contents

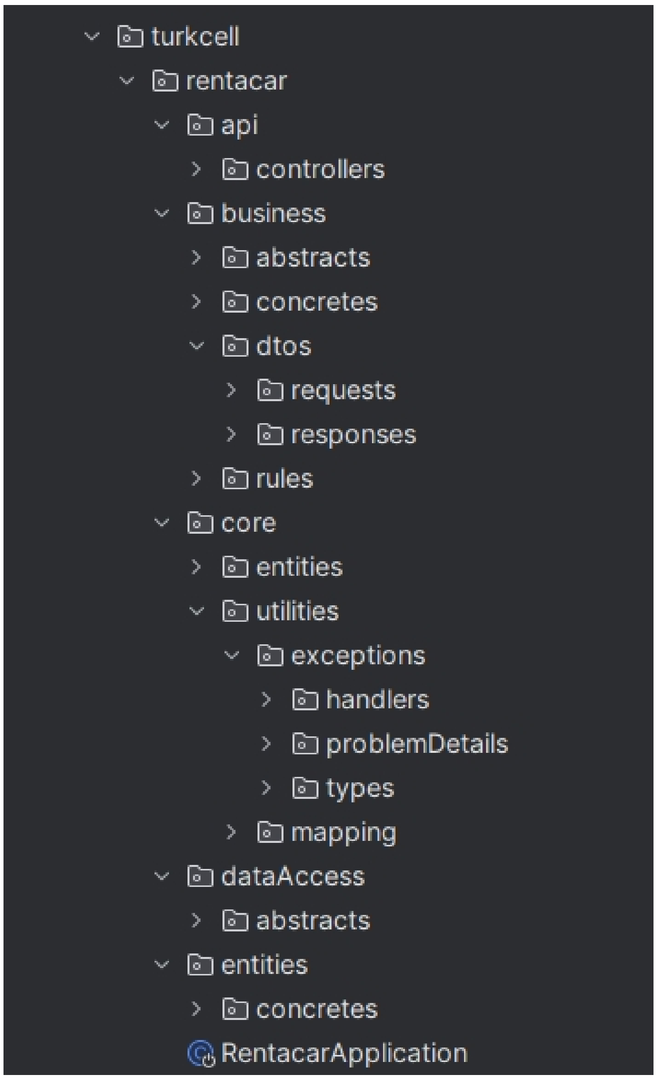
1. Introduction

SOLID is the initials of 5 software development principles. The aim of these principles is to make the developed software sustainable, flexible and more understandable.

In this report, **Rentacar**, which is being developed under Turkcell GYGY 3.0 program SOLID software development principles in the project are discussed.

2. Single Responsibility

2.1 Package Structure



```
turkcell
  rentacar
    api
      controllers
    business
      abstracts
      concretes
      dtos
        requests
        responses
      rules
    core
      entities
      utilities
        exceptions
          handlers
          problemDetails
          types
        mapping
    dataAccess
      abstracts
    entities
      concretes
    RentacarApplication
```

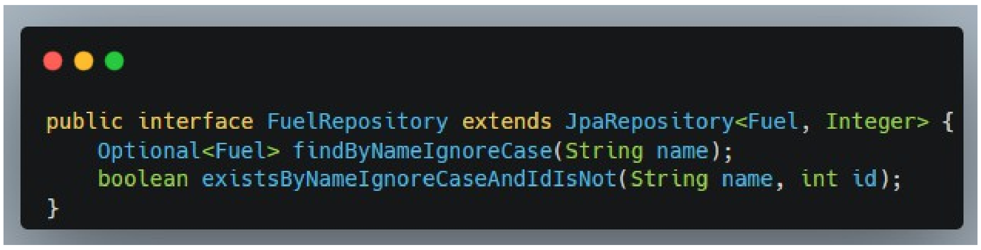
Figure 1

The project code is divided into 4 basic packages as shown in *Figure 1,* each of which is responsible for its own task. In short, the *entities* layer is responsible for database models, the *dataAccess* layer for database operations, the *business* layer for business code and the *api* layer for API operations.

As can be seen from the subfolders, the packages are also divided into responsibility parts. For example, in the *business* layer, abstract structures are grouped under *abstracts* folder, concrete structures are grouped under *concretes* folder, business rules are grouped under *rules* folder and request-response classes are grouped under *dtos* folder with their own folders.

Each layer is thus assigned a responsibility.

2.2 DataAccess Layer and Repositories



```java
public interface FuelRepository extends JpaRepository<Fuel, Integer> {
    Optional<Fuel> findByNameIgnoreCase(String name);
    boolean existsByNameIgnoreCaseAndIdIsNot(String name, int id);
}
```

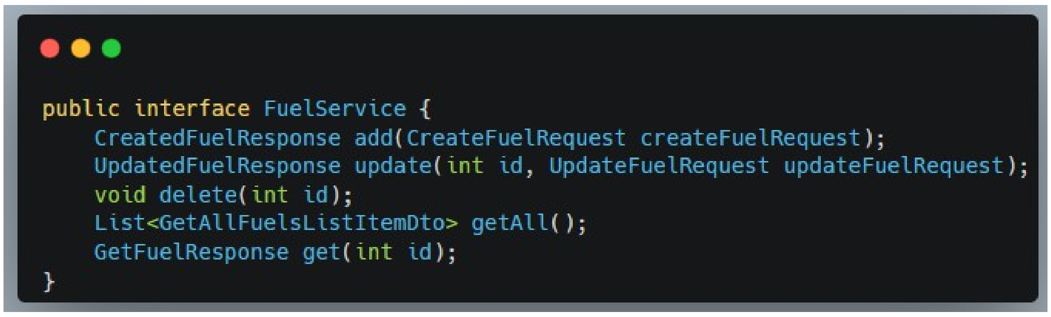
Figure 2

The repository interfaces under the *abstracts* package in the *dataAccess* layer are created for each entity. The interface named FuelRepository in *Figure 2* should only contain the database operations of the Fuel model.

For example, since it is possible to write the queries of other entities here with methods such as *@Query* annotation during the development phase, it will be in accordance with the Single Responsibility principle that each entity has its own repository.

2.3 Business Tier

2.3.1 Service - Manager Structure



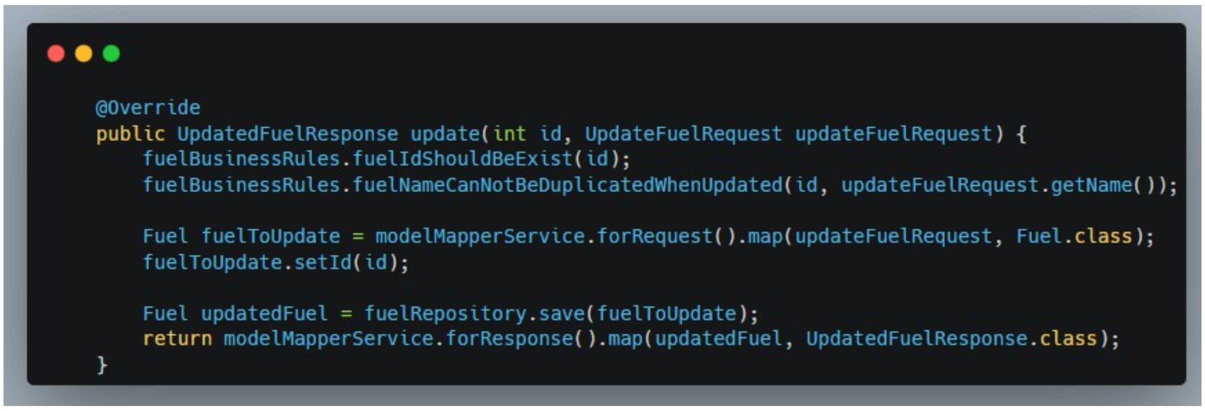
```java
public interface FuelService {
    CreatedFuelResponse add(CreateFuelRequest createFuelRequest);
    UpdatedFuelResponse update(int id, UpdateFuelRequest updateFuelRequest);
    void delete(int id);
    List<GetAllFuelsListItemDto> getAll();
    GetFuelResponse get(int id);
}
```

Figure 3

Just like the repository interfaces in the *dataAccess* layer, the *abstracts* package in the *business* layer contains the interfaces implemented by the *Manager* classes that contain the implementations of the business code. Each service and manager is named according to its entity or property name and contains only operations in that scope.

For example, the FuelService in *Figure 3* contains only method signatures for the Fuel model, and its concrete class contains method implementations.

2.3.2 Methods



```java
@Override
public UpdatedFuelResponse update(int id, UpdateFuelRequest updateFuelRequest) {
    fuelBusinessRules.fuelIdShouldBeExist(id);
    fuelBusinessRules.fuelNameCanNotBeDuplicatedWhenUpdated(id, updateFuelRequest.getName());

    Fuel fuelToUpdate = modelMapperService.forRequest().map(updateFuelRequest, Fuel.class);
    fuelToUpdate.setId(id);

    Fuel updatedFuel = fuelRepository.save(fuelToUpdate);
    return modelMapperService.forResponse().map(updatedFuel, UpdatedFuelResponse.class);
}
```

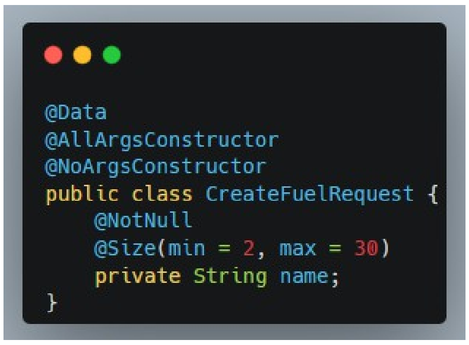
Figure 4

Figure 4 shows a snippet of the FuelManager class, the *update* method. This method makes sense hierarchically with the class name and is responsible for updating a Fuel object. Since no other operations are performed in it, it has the potential to be used in every part of the project.

If the method is analyzed, 2 business rules called from another class are executed first. This structure is examined in detail in section 2.3.4 Rules.

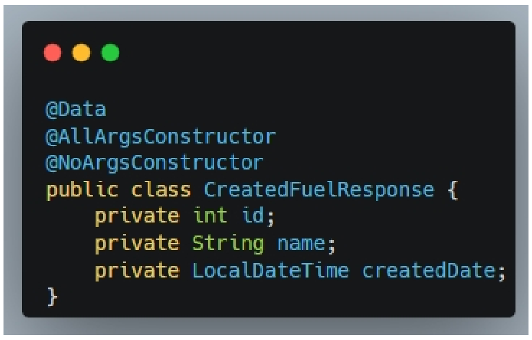
After the business rules, the process is completed by performing database operations with the relevant mapping operations.

2.3.3 Request Response Pattern



```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CreateFuelRequest {
    @NotNull
    @Size(min = 2, max = 30)
    private String name;
}
```

Figure 5



```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class CreatedFuelResponse {
    private int id;
    private String name;
    private LocalDateTime createdDate;
}
```

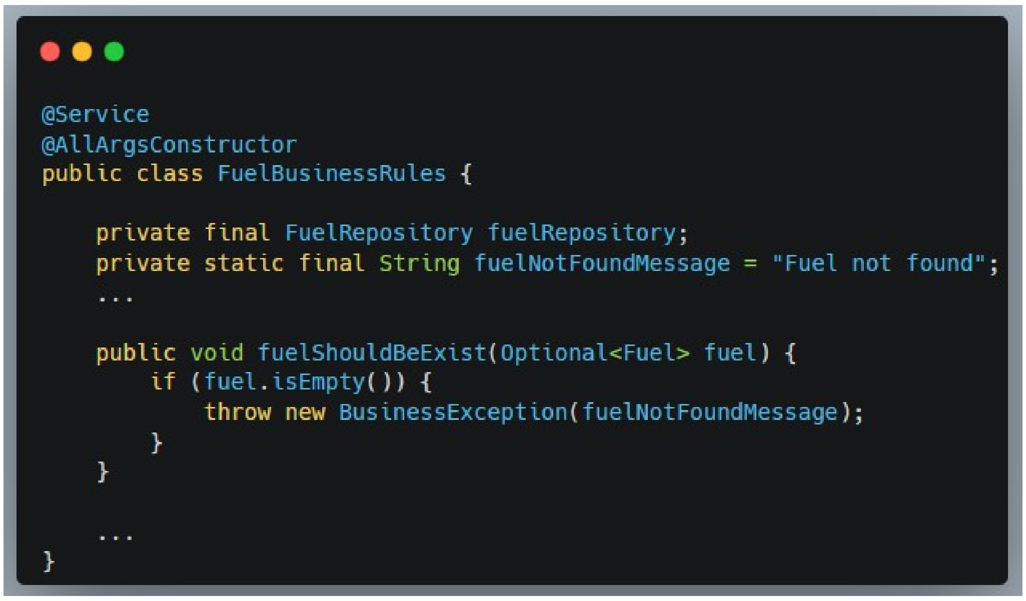
Figure 6

2.1 Package Structure As examined, there are 2 packages named *requests* and *responses* in the *dtos* package in the business layer. *requests* package contains the parameters to the methods in the services. *responses* package contains the types returned by these methods.

Figure 5 and *Figure 6 show* examples of classes representing the request to create Fuel and the result returned from this request. Although creating a single DTO class and using it without separating it into *request* and *response seems to be* a useful method, the 2 images above are the clearest example that this will not be a healthy solution.

There may be fields in the Response class that we do not want to be in the Request. Likewise, there may be various validations etc. in the Request class that we do not want to be in the Response. Therefore, it would be more appropriate to separate these 2 structures as shown in the figures by applying the Single Responsibility principle.

2.3.4 Rules

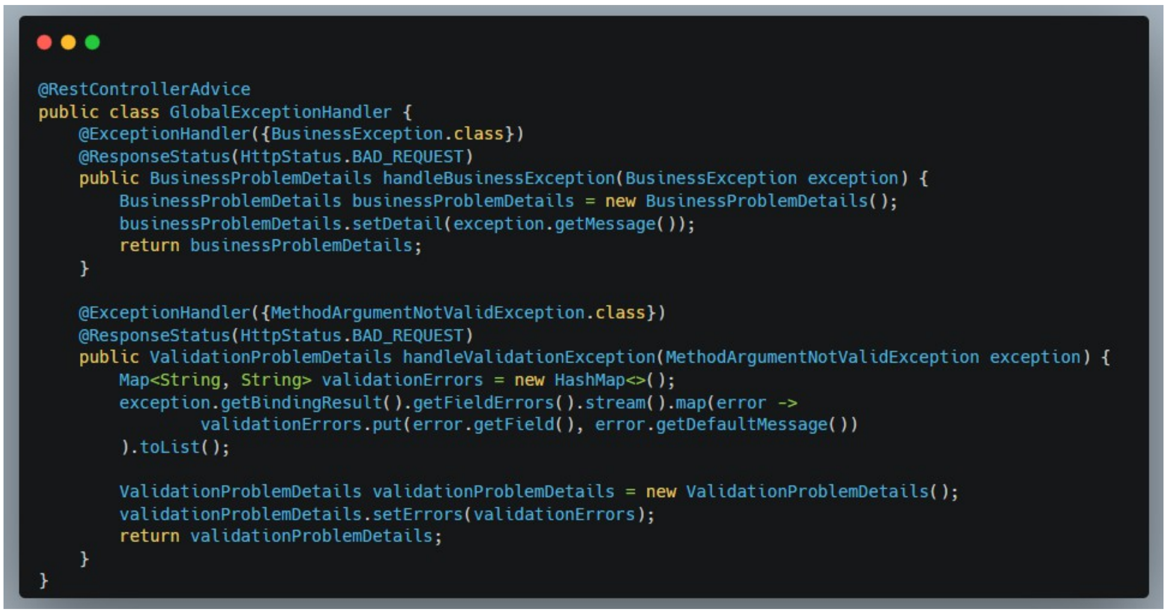


```java
@Service
@AllArgsConstructor
public class FuelBusinessRules {

    private final FuelRepository fuelRepository;
    private static final String fuelNotFoundMessage = "Fuel not found";
    ...

    public void fuelShouldBeExist(Optional<Fuel> fuel) {
        if (fuel.isEmpty()) {
            throw new BusinessException(fuelNotFoundMessage);
        }
    }

    ...
}
```

Figure 7

The rules package contains classes that cover business rules in manager or other needed places. These classes are also written in such a way that they do not contain the rules of more than one context in a class or feature specific way. The Single Responsibility principle comes into play here again.

2.4 Global Exception Handler



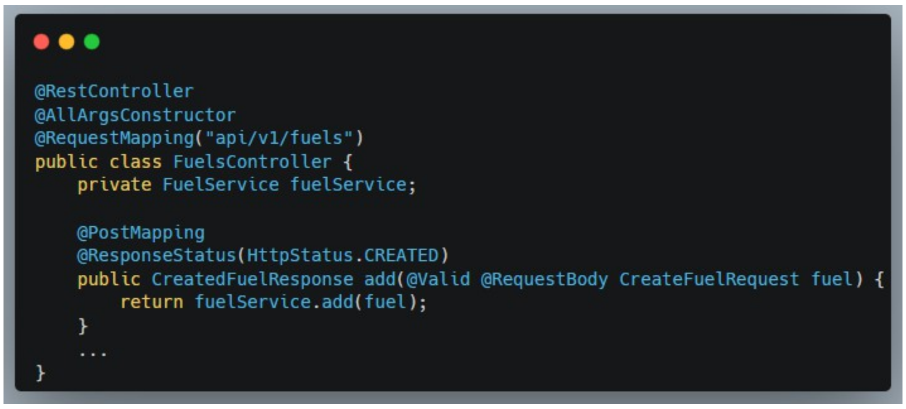
```java
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler({BusinessException.class})
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public BusinessProblemDetails handleBusinessException(BusinessException exception) {
        BusinessProblemDetails businessProblemDetails = new BusinessProblemDetails();
        businessProblemDetails.setDetail(exception.getMessage());
        return businessProblemDetails;
    }

    @ExceptionHandler({MethodArgumentNotValidException.class})
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ValidationProblemDetails handleValidationException(MethodArgumentNotValidException exception) {
        Map<String, String> validationErrors = new HashMap<>();
        exception.getBindingResult().getFieldErrors().stream().map(error ->
                validationErrors.put(error.getField(), error.getDefaultMessage())
        ).toList();

        ValidationProblemDetails validationProblemDetails = new ValidationProblemDetails();
        validationProblemDetails.setErrors(validationErrors);
        return validationProblemDetails;
    }
}
```

Figure 8

In the global error handler in the Core layer, instead of writing multiple exception types in a single exception handler by separating them with commas, 1 exception handler is written per exception. Thus, each handler has one responsibility.

2.5 API Layer

2.5.1 Controller Classes



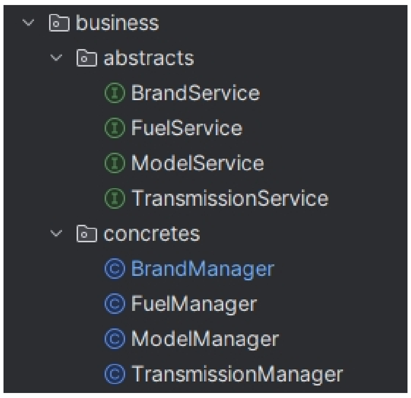
```java
@RestController
@AllArgsConstructor
@RequestMapping("api/v1/fuels")
public class FuelsController {
    private FuelService fuelService;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public CreatedFuelResponse add(@Valid @RequestBody CreateFuelRequest fuel) {
        return fuelService.add(fuel);
    }
    ...
}
```

Figure 9

The API layer is the presentation layer and is responsible for exposing the application as a Restful API. The task of the FuelsController class given as an example in *Figure 8 is* to expose the methods of the Fuel property in the *api/v1/fuels* endpoint. According to Single Responsibility and Restful API standards, operations of other scopes should not be exposed under this endpoint.

3. Open Closed

3.1 Service-Manager Relationship in the Business Layer

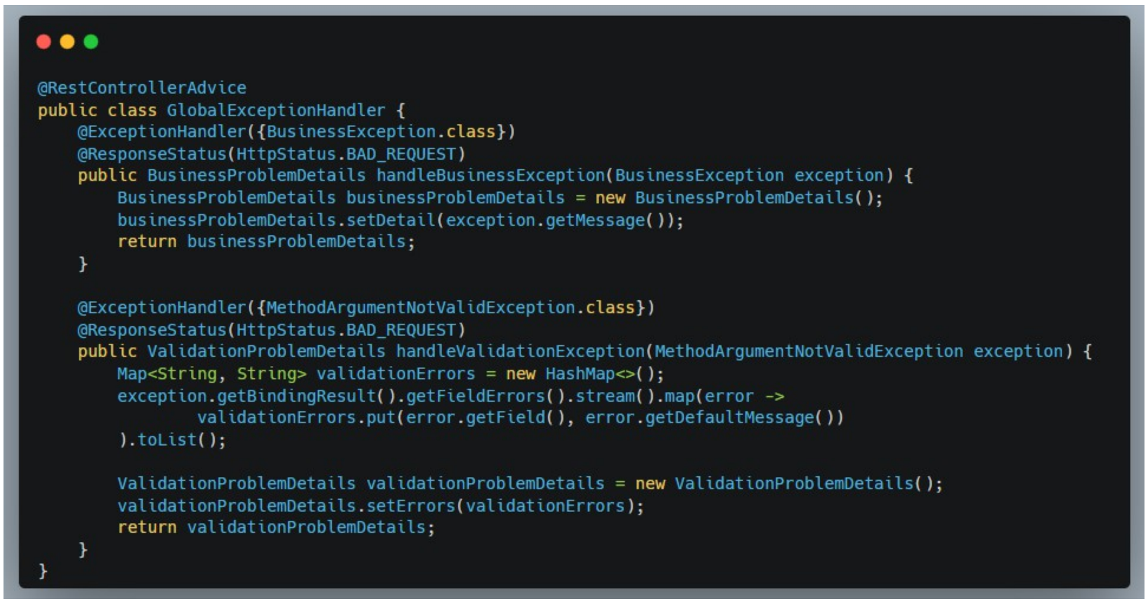


```
business
  abstracts
    BrandService
    FuelService
    ModelService
    TransmissionService
  concretes
    BrandManager
    FuelManager
    ModelManager
    TransmissionManager
```

Figure 10

Service interfaces and Manager classes in the Business layer are written in accordance with the open closed principle. When we want to update the API to v2 version in the future, we can disable BrandManager by creating a new version like BrandV2Manager. Since BrandService will still be implemented, we will be able to update our business code without the need to make any changes to the references.

3.2 Global Exception Handler



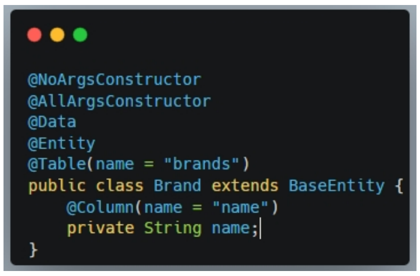
```java
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler({BusinessException.class})
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public BusinessProblemDetails handleBusinessException(BusinessException exception) {
        BusinessProblemDetails businessProblemDetails = new BusinessProblemDetails();
        businessProblemDetails.setDetail(exception.getMessage());
        return businessProblemDetails;
    }

    @ExceptionHandler({MethodArgumentNotValidException.class})
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ValidationProblemDetails handleValidationException(MethodArgumentNotValidException exception) {
        Map<String, String> validationErrors = new HashMap<>();
        exception.getBindingResult().getFieldErrors().stream().map(error ->
                validationErrors.put(error.getField(), error.getDefaultMessage())
        ).toList();

        ValidationProblemDetails validationProblemDetails = new ValidationProblemDetails();
        validationProblemDetails.setErrors(validationErrors);
        return validationProblemDetails;
    }
}
```

Figure 11

The handlers that we have separated according to the exception type with Single Responsibility actually paves the way for the open closed principle. For example, when we add a new exception of the NotFoundException type, we will be able to develop the new feature without changing the existing code.

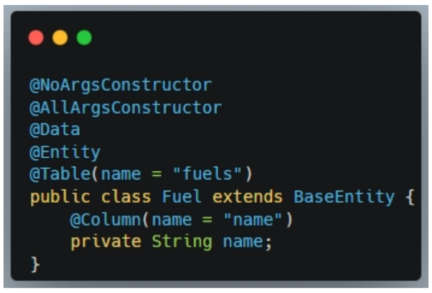
4. Liskov's Substitution

4.1 Entity - Base Entity Relationships



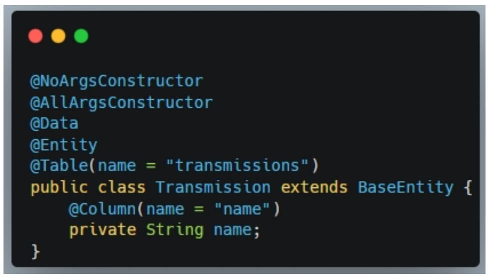
```java
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@Table(name = "brands")
public class Brand extends BaseEntity {
    @Column(name = "name")
    private String name;
}
```

Figure 12



```java
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@Table(name = "fuels")
public class Fuel extends BaseEntity {
    @Column(name = "name")
    private String name;
}
```

Figure 13



```java
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@Table(name = "transmissions")
public class Transmission extends BaseEntity {
    @Column(name = "name")
    private String name;
}
```

Figure 14

All of the database models in the project extend the BaseEntity class. If we examine the Brand, Fuel and Transmission models, all of them currently only have a *name* field.

Since they are similar to each other here, it is technically possible to gather 3 tables under a single table with a name similar to Lookup and control them with temporary solutions such as enum. But let's imagine that we want to add a photo field to brands, which is a possible request in the future. The photo field that only brands need will also exist for Fuel and Transmission, and the photo field for these 2 models will be assigned a null value for each record. With the formation of incorrect data in the database, control will also become difficult.

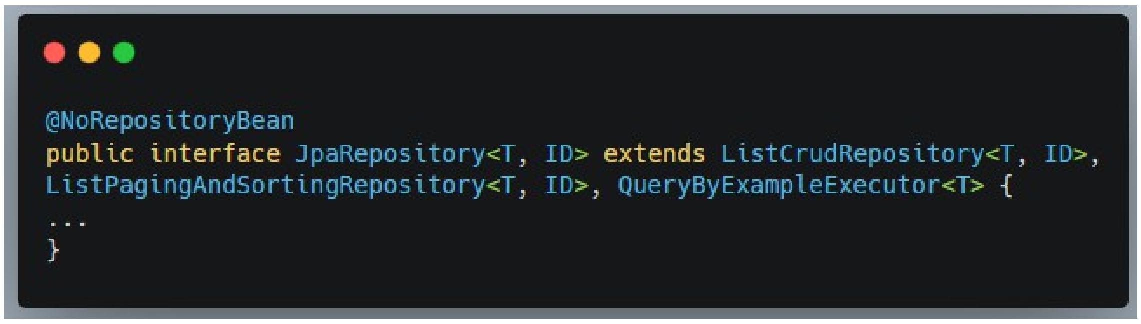
As a result, the importance of Liskov's Substitution principle is understood at this point. These 3 models are not used interchangeably as they are similar and all inherit BaseEntity.

4.2 Separation of Get and GetAll Method Return Types (?)

Although we return the same fields in the *get* and *getAll* methods in the Manager classes, unique response objects are created for both of them.

5. Interface Segregation

5.1 Global Exception Handler



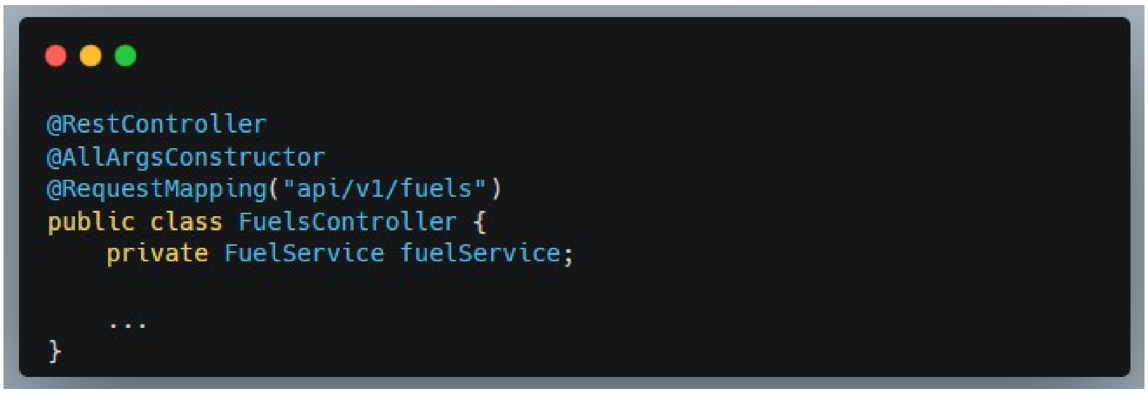
```
@NoRepositoryBean
public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID>,
ListPagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {
...
}
```

Figure 15

We did not find a direct application of the Interface Segregation principle in the project. However, we examined the JPA Repository we used and discovered that the Interface Segregation principle is used here. A repository interface is defined for CRUD operations, a repository for paging and sorting operations, and a separate repository interface for QueryByExample query executions.

6. Dependency Inversion

6.1 Controllers - Services

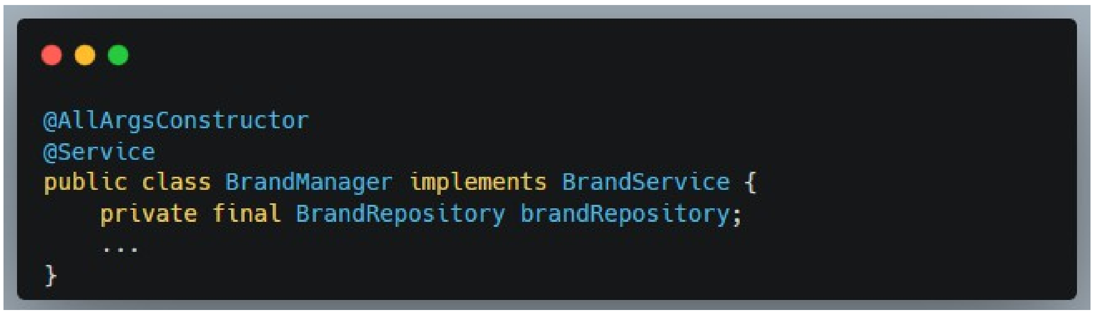


```
@RestController
@AllArgsConstructor
@RequestMapping("api/v1/fuels")
public class FuelsController {
    private FuelService fuelService;

    ...
}
```

Figure 16

The structure between the service interfaces we inject in the controllers and the controllers applies the principle of Dependency Inversion. The superclass FuelsController is loosely coupled to the subclass FuelService interface.

6.2 Services - Repositories



```
@AllArgsConstructor
@Service
public class BrandManager implements BrandService {
    private final BrandRepository brandRepository;
    ...
}
```

Figure 17

The relationship between Controllers and Services also exists between Managers and Repositories. The superclass BrandManager is loosely coupled to the subclass BrandRepository.