

TURKCELL GYGY 3.0

20.03.2024 TARİHLİ DERSE AİT ÖDEV RAPORU

SOLID YAZILIM GELİŞTİRME  
PRENSİPLERİNİN UYGULANMASI

Görkem Rıdvan ARIK

22 Mart 2024

## İçindekiler

1. Giriş.....	3
2. Single Responsibility .....	4
2.1 Package Yapısı .....	4
2.2 DataAccess Katmanı ve Repository'ler .....	5
2.3 Business Katmanı.....	5
2.3.1 Service - Manager Yapısı .....	5
2.3.2 Methodlar .....	6
2.3.3 Request Response Pattern .....	6
2.3.4 Rules .....	7
2.4 Global Exception Handler.....	8
2.5 API Katmanı .....	8
2.5.1 Controller Sınıfları .....	8
3. Open Closed.....	9
3.1 Business Katmanındaki Service-Manager İlişkisi .....	9
3.2 Global Exception Handler.....	9
4. Liskov's Substitution .....	10
4.1 Entity - Base Entity İlişkileri .....	10
4.2 Get ve GetAll Method Dönüş Tiplerinin Ayrılması (?) .....	10
5. Interface Segregation .....	11
5.1 Global Exception Handler.....	11
6. Dependency Inversion .....	11
6.1 Controllers – Services .....	11
6.2 Services – Repositories .....	12

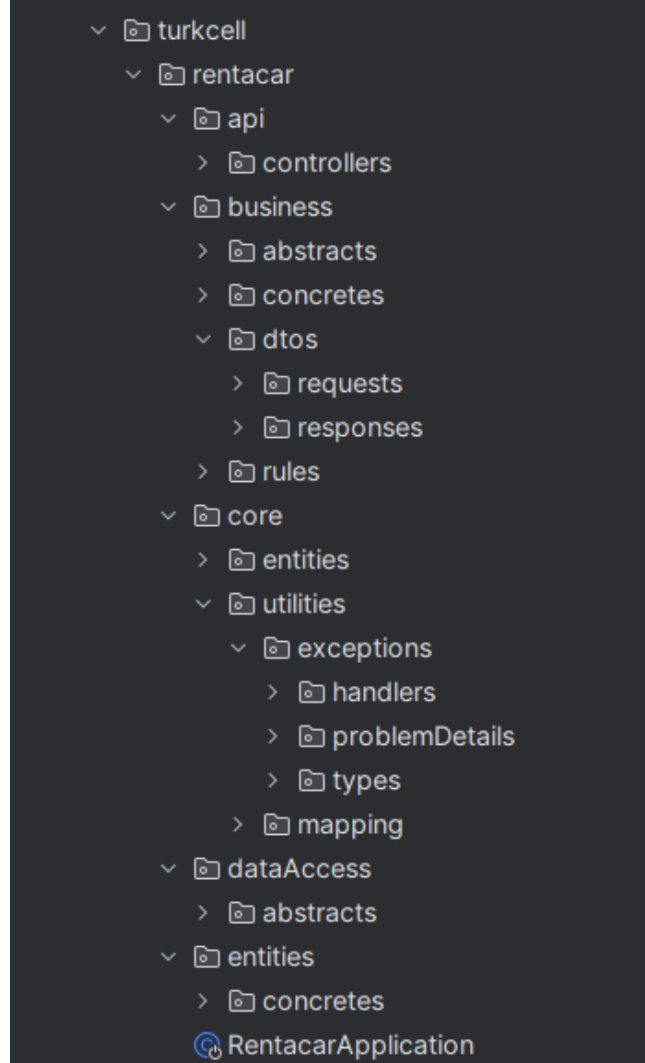
# 1. Giriş

SOLID, 5 adet yazılım geliştirme prensibinin baş harflerinden oluşan kelimedir. Bu prensiplerin amacı geliştirilen yazılımın sürdürülebilir, esnek ve daha anlaşılır hale getirilmesidir.

Bu raporda Turkcell GYGY 3.0 programı kapsamında geliştirilmekte olan **Rentacar** projesindeki SOLID yazılım geliştirme prensipleri ele alınmaktadır.

## 2. Single Responsibility

### 2.1 Package Yapısı



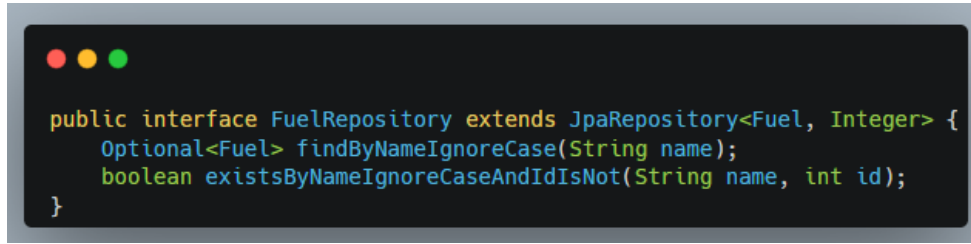
Şekil 1

Proje kodları 4 Şekil 1’de görüldüğü üzere temel pakete bölünmüş olup hepsi kendi görevini yerine getirmekten sorumludur. Kısaca *entities* katmanı veri tabanı modellerinden, *dataAccess* katmanı veri tabanı operasyonlarından, *business* katmanı iş kodlarından ve *api* katmanı API işlemlerinden sorumludur.

Alt klasörlerden de görüleceği üzere paketler de kendi içerisinde sorumluluk parçalarına bölünmüştür. Örneğin *business* katmanında soyut yapılar *abstracts* klasörüne, somut yapılar *concretes* klasörüne, iş kuralları *rules* klasörüne ve request-response sınıfları kendilerine özel klasörler ile *dtos* klasörü altında toplanmıştır.

Böylelikle her katmana bir sorumluluk yüklenmiştir.

## 2.2 DataAccess Katmanı ve Repository'ler



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

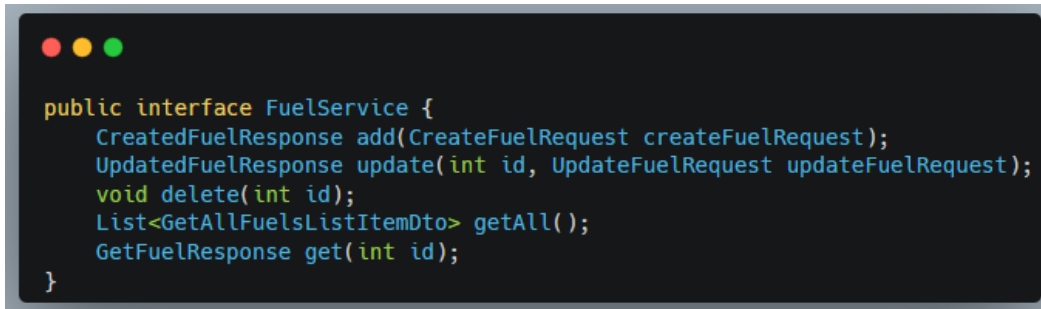
Şekil 2

*dataAccess* katmanında yer alan *abstracts* paketi altındaki repository interface'leri her entity özelinde oluşturulmaktadır. Şekil 2'de yer alan FuelRepository isimli interface yalnızca Fuel modelinin veri tabanı operasyonlarını içermelidir.

Örneğin geliştirme aşamasında *@Query* anotasyonu gibi yöntemlerle diğer entity'lerin sorgularını buraya yazma imkanı olduğundan her entity'nin kendine ait bir repository'si olması Single Responsibility prensibine uygun olacaktır.

## 2.3 Business Katmanı

### 2.3.1 Service - Manager Yapısı



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

Şekil 3

Tıpkı *dataAccess* katmanında yer alan repository interface'leri gibi *business* katmanında da var olan *abstracts* paketi, iş kodlarının implementasyonlarını içeren *Manager* sınıflarının implement ettiği interface'leri içermektedir. Her service ve manager, entity veya özellik ismine göre adlandırılmaktadır ve yalnızca bu kapsamdaki operasyonları içerirler.

Örneğin Şekil 3'teki FuelService içerisinde yalnızca Fuel modeline ait method imzaları bulunmaktadır ve buna ait somut sınıf da method implementasyonlarını içermektedir.

### 2.3.2 Methodlar

```
@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);
}
```

Şekil 4

Şekil 4'te FuelManager sınıfından bir kesit olan *update* methodu verilmiştir. Bu method hiyerarşik olarak sınıf adıyla birlikte anlamlı hale gelir ve bir Fuel nesnesini güncellemekten sorumludur. İçerisinde başka bir işlem gerçekleşmediğinden dolayı projede ihtiyaç olan her kısımda kullanılabilecek potansiyeldedir.

Method incelenecek olursa ilk olarak başka bir sınıftan çağrılan 2 adet iş kuralı çalıştırılmaktadır. Bu yapı ayrıntılı olarak [2.3.4 Rules](#) bölümünde incelenmiştir.

İş kurallarının ardından ilgili map'leme işlemleri ile veritabanı operasyonları gerçekleştirilerek işlem tamamlanmaktadır.

### 2.3.3 Request Response Pattern

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

Şekil 5

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

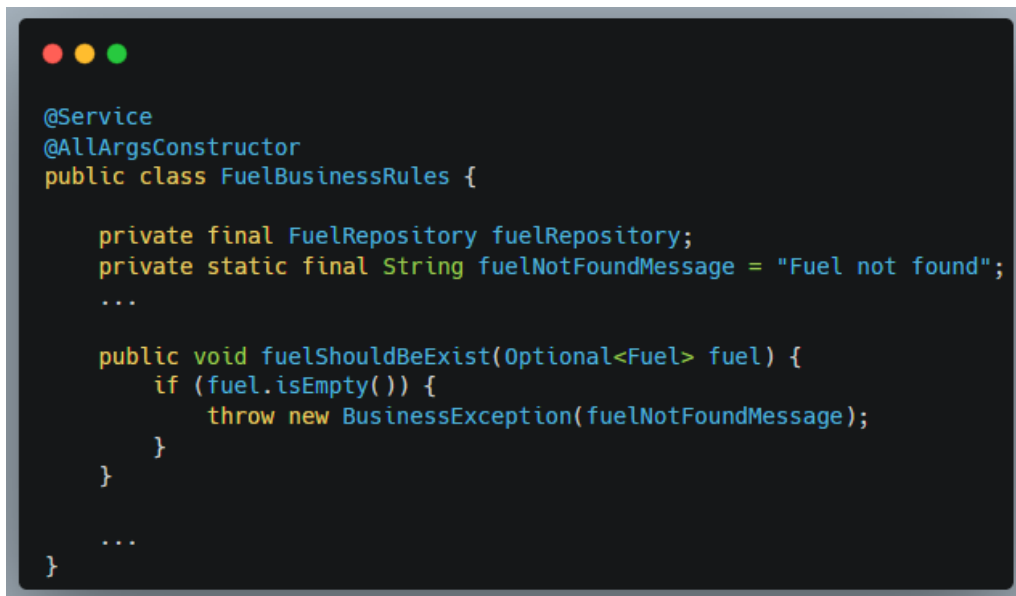
Şekil 6

[2.1 Package Yapısı](#) incelendiği üzere business katmanındaki *dtos* paketi içerisinde *requests* ve *responses* adında 2 adet paket bulunmaktadır. *requests* paketi servislerdeki methodlara gelen parametreleri içerir. *responses* paketi ise bu methodların geriye döndürdüğü tipleri içermektedir.

Şekil 5 ve Şekil 6'da Fuel oluşturma isteği ve bu istekten dönen sonucu temsil eden sınıflar örnek olarak verilmiştir. Tek bir DTO sınıfı oluşturarak *request* ve *response* şeklinde ayırmadan kullanmak da kullanışlı bir yöntem gibi gözükse de bunun sağlıklı bir çözüm olmayacağının en net örneği yukarıdaki 2 görüntüdür.

Response sınıfında Request'te olmasını istemediğimiz field'lar olabilir. Aynı şekilde Request sınıfı içerisinde de Response'ta olmasını istemediğimiz çeşitli validasyonlar vb. yer alabilir. Dolayısıyla Single Responsibility prensibi uygulanarak bu 2 yapıyı şekillerdeki gibi ayırmak daha uygun olacaktır.

### 2.3.4 Rules

A screenshot of a code editor showing a Java class named FuelBusinessRules. The class is annotated with @Service and @AllArgsConstructor. It contains a private final FuelRepository field, a private static final String fuelNotFoundMessage, and a public void method fuelShouldBeExist that throws a BusinessException if the fuel is empty. The code is as follows:

```
@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);
        }
    }
    ...
}
```

Şekil 7

*rules* paketi, manager veya diğer ihtiyaç duyulan yerlerdeki iş kurallarını kapsayan sınıfları içermektedir. Bu sınıflar da ihtiyaca göre sınıf veya özellik özelinde birden fazla bağlamın kurallarını içermeyecek şekilde yazılmaktadır. Single Responsibility prensibi yine burada devreye girmektedir.

## 2.4 Global Exception Handler

```
@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;
    }
}
```

Şekil 8

Core katmanında yer alan global hata işleme işleyicisinde birden fazla exception türünü virgüller ile ayırarak tek exception handler içinde yazmak yerine exception başına 1 adet exception handler yazılmıştır. Böylelikle her handler'ın bir sorumluluğu bulunmaktadır.

## 2.5 API Katmanı

### 2.5.1 Controller Sınıfları

```
@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);
    }
    ...
}
```

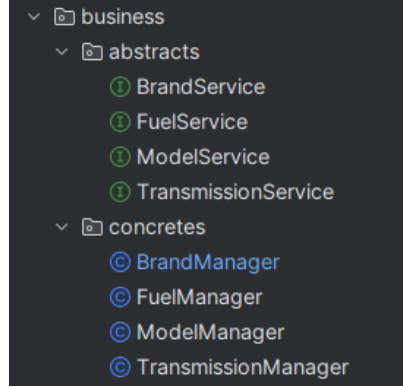
Şekil 9

API katmanı sunum katmanıdır ve uygulamayı bir Restful API olarak dış ortama sunmaktan sorumludur. Şekil 8'de örnek olarak verilen FuelsController sınıfının görevi *api/v1/fuels* endpoint'inde Fuel özelliğine ait methodları dışarıya sunmaktır. Single Responsibility ve Restful API standartlarına göre bu endpoint altında diğer kapsamlara ait işlemler sunulmamalıdır.



## 3. Open Closed

### 3.1 Business Katmanındaki Service-Manager İlişkisi



Şekil 10

Business katmanındaki Service interface'leri ve Manager class'ları open closed prensibine uygun olarak yazılmıştır. İlerleyen süreçte API'yi v2 versiyonuna güncellemek istediğimizde BrandV2Manager gibi yeni bir versiyon oluşturarak BrandManager'ı devre dışı bırakabiliriz. BrandService yine implement edileceğinden dolayı referans alan yerlerde bir değişiklik yapmaya gerek kalmadan iş kodlarımızı güncelleyebileceğiz.

### 3.2 Global Exception Handler

```
@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;
    }
}
```

Şekil 11

Single Responsibility ile exception türüne göre ayırmış olduğumuz handler'lar aslında open closed prensibinin yolunu da açar. Örneğin NotFoundException türünde yeni bir exception ekleyeceğimiz zaman var olan kodları değiştirmeden yeni özelliği geliştirebileceğiz.

## 4. Liskov's Substitution

### 4.1 Entity - Base Entity İlişkileri

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

Şekil 12

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

Şekil 13

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

Şekil 14

Projedeki veri tabanı modellerinin hepsi BaseEntity sınıfını extend etmektedir. Brand, Fuel ve Transmission modellerini inceleyecek olursak şu anlık hepsinde yalnızca *name* field'ı bulunmaktadır.

Burada birbirlerine benzedikleri için 3 tabloyu Lookup benzeri bir isimle tek bir tablo altında toplayarak enum gibi geçici çözümlerle kontrolünü sağlamak teknik olarak bir yere kadar mümkündür. Fakat ilerleyen süreçte olası bir ister olan markalara fotoğraf alanı eklemek istediğimizi düşünelim. Yalnızca markaların ihtiyaç duyduğu fotoğraf alanı Fuel ve Transmission için de var olacak ve bu 2 model için fotoğraf alanına her kayıttaki null değeri atanacak. Veri tabanında hatalı verilerin oluşmasıyla birlikte kontrol de zorlaşacaktır.

Sonuç olarak bu noktada Liskov's Substitution prensibinin önemi anlaşılmaktadır. Bu 3 model birbirlerine benzedikleri için birbirlerinin yerine kullanılmamıştır ve hepsi BaseEntity'yi inherit etmektedir.

### 4.2 Get ve GetAll Method Dönüş Tiplerinin Ayrılması (?)

Manager sınıflarında yer alan *get* ve *getAll* methodlarında aynı field'ları döndürmemize rağmen her ikisi için kendilerine özgü response nesneleri oluşturulmuştur.

## 5. Interface Segregation

### 5.1 Global Exception Handler



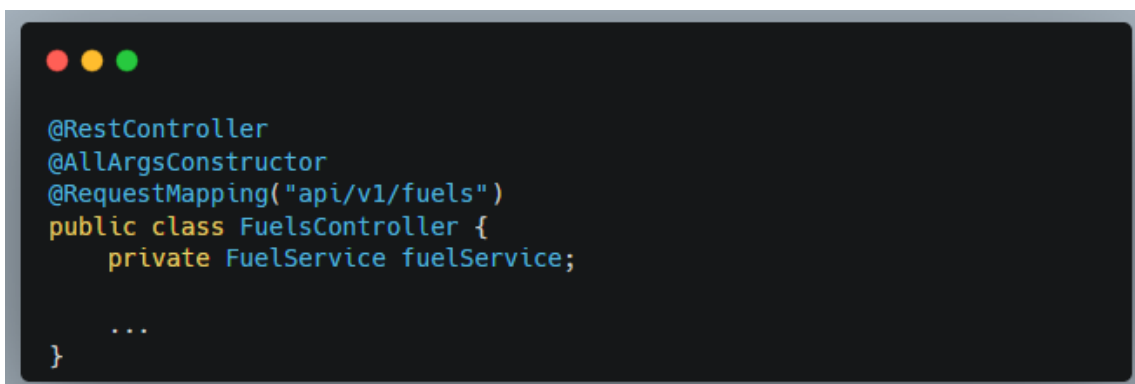
```
@NoRepositoryBean
public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID>,
ListPagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {
    ...
}
```

Şekil 15

Proje içerisinde doğrudan bir Interface Segregation prensibi uygulaması tespit edemedik. Fakat kullandığımız JPA Repository'yi inceledik ve burada Interface Segregation prensibinin kullanıldığını keşfettik. CRUD işlemleri için bir repository, sayfalama ve sıralama işlemleri için bir repository ve QueryByExample sorgu yürütmeleri için ayrı bir repository interface'i tanımlanmıştır.

## 6. Dependency Inversion

### 6.1 Controllers – Services



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

    ...
}
```

Şekil 16

Controller'larda inject ettiğimiz service interface'leri ve controller'lar arasındaki yapı Dependency Inversion prensibini uygulamaktadır. Üst sınıf olan FuelsController sınıfı alt sınıf olan FuelService interface'ine gevşek bağımlıdır (Loosly Coupled).

## 6.2 Services – Repositories

A screenshot of a code editor with a dark background and light-colored text. The code is in Java and defines a class named BrandManager. The code is as follows:

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

Şekil 17

Controllers – Services arasındaki ilişki Managers – Repositories arasında da bulunmaktadır. Üst sınıf olan BrandManager alt sınıf olan BrandRepository'ye gevşek bağımlıdır (Loosly Coupled).