Software Engineering Design Patterns

Victor Padurean

January 2020

1 Strategy

The Strategy Design Pattern was used to distribute orders among couriers in a more dynamic way, easy to change. The UML diagram of this design pattern, customized for the project, is found in Figure 1, whereas the relevant code, in Listing 1.

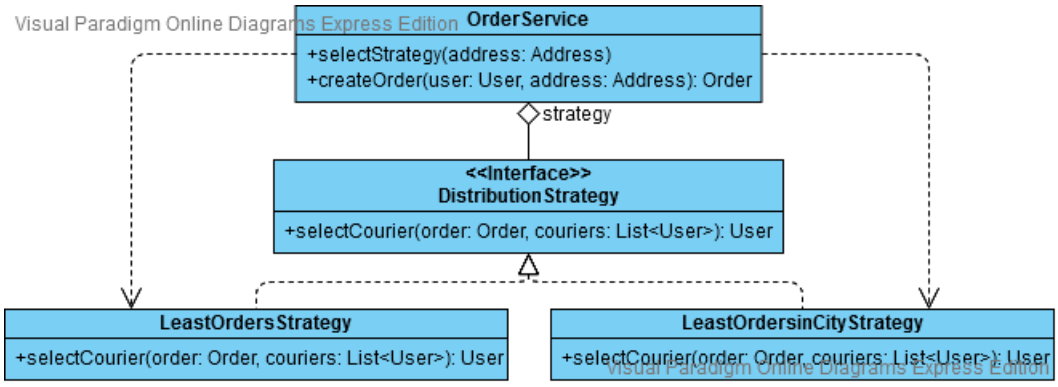

Figure 1: Strategy Design Pattern UML

```
1  public interface DistributionStrategy {
2
3    public User selectCourier(Order order, List<User> couriers);
4
5  }
6
7
8
9
10 //take the courier found in the same city as the order, with the
       least orders
11 public class LeastOrdersinCityStrategy implements
       DistributionStrategy {
```

```
12
13    @Override
14    public User selectCourier(Order order, List<User> couriers) {
15      return couriers.stream().filter(x -> x.getAddress().getCity().
      equals(order.getAddress().getCity()))
16          .sorted(Comparator.comparing((User x) -> x.getCourierOrders
      ().stream()
17                  .filter(y -> y.getStatus().equals(OrderStatus.
      PROCESSING)).collect(Collectors.toList()).size(),
18                  Comparator.reverseOrder()))
19          .findFirst().orElse(null);
20    }
21
22  }
23
24
25
26
27
28  //take the courier with the least orders
29  public class LeastOrdersStrategy implements DistributionStrategy {
30
31    @Override
32    public User selectCourier(Order order, List<User> couriers) {
33      return couriers.stream()
34          .sorted(Comparator.comparing((User x) -> x.getCourierOrders
      ().stream()
35                  .filter(y -> y.getStatus().equals(OrderStatus.
      PROCESSING)).collect(Collectors.toList()).size()))
36          .findFirst().orElse(null);
37    }
38
39  }
40
41
42
43
44
45  public class OrderService implements OrderServiceI {
46
47  private DistributionStrategy strategy;
48
49    @Override
50    public Order createOrder(User user, Address address) {
51      if (user == null)
52        return null;
53      if (address == null)
54        return null;
55      Order o;
56      if (address.getCountry() == null || address.getCountry().equals
      ("") || address.getCity() == null
57          || address.getCity().equals("") || address.getStreet() ==
      null || address.getStreet().equals("")
58          || address.getNumber() == null || address.getZipCode() ==
      null || address.getZipCode().equals(""))
59        o = new Order(user, user.getAddress());
60      else
```

```
61        o = new Order(user, address);
62      o.setStatus(OrderStatus.PROCESSING);
63      selectStrategy(address);
64      o.setCourierName(strategy.selectCourier(o, userDao.getByRole(
        UserType.COURIER)).getUsername());
65      o = orderRepo.save(o);
66      userDao.addCourierOrder(strategy, o);
67      return o;
68    }
69
70    private void selectStrategy(Address address) {
71      if (userDao.getAll().stream().anyMatch(x -> x.getAddress().
        getCity().equals(address.getCity()))) {
72        strategy = new LeastOrdersinCityStrategy();
73      } else {
74        strategy = new LeastOrdersStrategy();
75      }
76    }
77  }
```

Listing 1: Strategy Pattern

2 Data Access Object

This is used to separate the database access from the rest of the application. Spring provides implementations for the classes that implement interfaces customized by the users. The provided interface should extend the MongoRepository interface. The schematic is presented in Figure 2, whereas the relevant code is present in Listing 2.

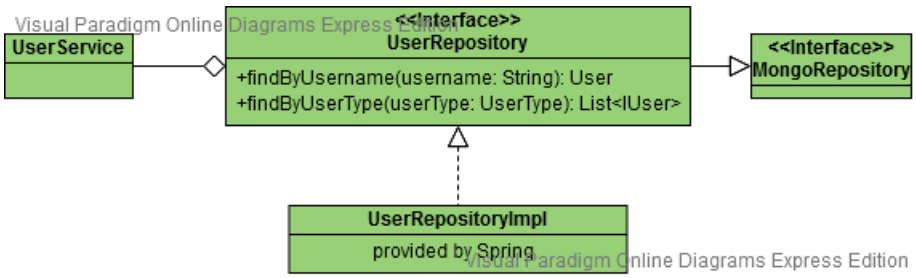

Figure 2: DAO Design Pattern UML

```
1
2  @Repository
3  public interface UserRepository extends MongoRepository<User,
       String> {
4
5    public User findByUsername(String username);
```

```
6
7   public List<User> findByUserType(UserType userType);
8
9
10
11
12 @Service
13 public class UserService implements UserServiceI {
14   @Autowired
15   private UserRepository userRepo;
16
17   @Autowired
18   private ProductServiceI productDao;
19
20   @Autowired
21   private OrderServiceI orderDao;
22
23   @Override
24   public List<User> getByRole(UserType role) {
25     return userRepo.findByUserType(role);
26   }
27
28   @Transactional
29   @Override
30   public User saveUser(User user) {
31     if (user == null)
32       return null;
33     user.setUsername(user.getUsername().trim());
34     if (userRepo.findByUsername(user.getUsername()) != null)
35       return null;
36     if (user.getUserType() == null)
37       user.setUserType(UserType.CUSTOMER);
38     if (user.getUserType().equals(UserType.CUSTOMER) && user.
       getShoppingCart() == null)
39       user.setShoppingCart(new ShoppingCart());
40     user.setUserStatus(Status.ACTIVE);
41     if (user.getUserType().equals(UserType.COURIER) && user.
       getCourierOrders() == null) {
42       user.setCourierOrders(new ArrayList<Order>());
43     }
44     user.setPassword(new BCryptPasswordEncoder().encode(user.
       getPassword().trim()));
45     return userRepo.save(user);
46   }
47
48   public List<User> getAll() {
49     return userRepo.findAll();
50   }
51
52   @Transactional
53   @Override
54   public User updateUser(User user) {
55     if (user == null)
56       return null;
57     User found = getByUsername(user);
58     if (found == null) {
59       return null;
```

```
60      } else {
61        if (user.getPassword() == null || user.getPassword().equals("
        ")) {
62          user.setPassword(found.getPassword());
63        }
64        user.setId(found.getId());
65        if (user.getUserType() == null) {
66          user.setUserType(found.getUserType());
67          user.setShoppingCart(found.getShoppingCart());
68          user.setCourierOrders(found.getCourierOrders());
69        }
70        if (user.getUserType().equals(UserType.CUSTOMER)) {
71          if (user.getShoppingCart() == null)
72            user.setShoppingCart(new ShoppingCart());
73          user.setCourierOrders(null);
74        }
75        if (user.getUserType().equals(UserType.ADMIN)) {
76          user.setShoppingCart(null);
77          user.setCourierOrders(null);
78        }
79        if (user.getUserType().equals(UserType.COURIER)) {
80          user.setShoppingCart(null);
81          if (user.getCourierOrders() == null)
82            user.setCourierOrders(new ArrayList<Order>());
83        }
84        if (!user.getPassword().equals(found.getPassword()))
85          user.setPassword(new BCryptPasswordEncoder().encode(user.
        getPassword()));
86        if (user.getUserStatus() == null)
87          user.setUserStatus(found.getUserStatus());
88        if (user.getUserType() == null)
89          user.setUserType(found.getUserType());
90        return userRepo.save(user);
91      }
92    }
93
94    @Transactional
95    @Override
96    public User getById(User user) {
97      userRepo.findById(user.getId());
98      return null;
99    }
100
101   @Transactional
102   @Override
103   public User getByUsername(User user) {
104     return userRepo.findByUsername(user.getUsername());
105   }
106
107   @Transactional
108   @Override
109   public User deleteUser(User user) {
110     if (user == null)
111       return null;
112     User found = getByUsername(user);
113     if (found == null) {
114       return null;
```

```
115       } else {
116         found.setUserStatus(Status.DELETED);
117         return userRepo.save(found);
118       }
119     }
120
121     @Override
122     public ShoppingCart addToCart(User user, Product product, Integer
          quant) throws InvalidArgumentsException {
123       Product found = productDao.getByName(product);
124       if (quant < 1)
125         throw new InvalidArgumentsException("The quantity should be a
          number greater than zero");
126       if(product.getStock() < quant)
127         throw new InvalidArgumentsException("The quantity exceeds the
          stock");
128       removeFromCart(user, product);
129       found.setStock(found.getStock() - quant);
130       User u = getByUsername(user);
131       ShoppingCart sc = u.addProductToCart(found, quant);
132       updateUser(u);
133       productDao.updateProduct(found);
134       return sc;
135     }
136
137     @Override
138     public ShoppingCart removeFromCart(User user, Product product) {
139       if (product == null)
140         return null;
141       Product found = productDao.getByName(product);
142       User u = getByUsername(user);
143       Integer q = null;
144       if (u.getShoppingCart().getProducts().contains(new
          ProductQuantity(product)))
145         q = u.getShoppingCart().getProductQuantity(product);
146       if (u.removeProduct(product))
147         found.setStock(found.getStock() + q);
148       updateUser(u);
149       productDao.updateProduct(found);
150       return u.getShoppingCart();
151     }
152
153     @Override
154     public void discardCart(User user) {
155       user.setShoppingCart(new ShoppingCart());
156       updateUser(user);
157     }
158
159     @Override
160     public void discardCartAndRestock(User user) {
161       for (ProductQuantity p : user.getShoppingCart().getProducts())
          {
162         removeFromCart(user, p.getProduct());
163       }
164     }
165
166     @Override
```

```
167   public Order placeOrder(User user, Address address) {
168     if (user == null)
169       return null;
170     if (address == null)
171       return null;
172     user = getByUsername(user);
173     Order o = orderDao.createOrder(getByUsername(user), address);
174     discardCart(user);
175     return o;
176   }
177
178   @Override
179   public ShoppingCart updateQuantityCart(User user, Product product
        , Integer quant) throws InvalidArgumentsException {
180     Product found = productDao.getByName(product);
181     // removeFromCart(user, product);
182     found.setStock(found.getStock() - quant);
183     return addToCart(user, product, quant);
184   }
185
186   @Override
187   public ShoppingCart updateQuantityCart(User user, Integer[] quant
        ) {
188     User u = getByUsername(user);
189     for (int i = 0; i < quant.length; i++) {
190       int diff = -u.getShoppingCart().getProducts().get(i).getQuant
      () + quant[i];
191       Product found = productDao.getByName(u.getShoppingCart().
      getProducts().get(i).getProduct());
192       found.setStock(found.getStock() - diff);
193       productDao.updateProduct(found);
194       u.getShoppingCart().getProducts().get(i).setQuant(quant[i]);
195     }
196     updateUser(u);
197     return u.getShoppingCart();
198   }
199
200   @Override
201   public User addCourierOrder(DistributionStrategy strategy, Order
        order) {
202     List<User> couriers = getByRole(UserType.COURIER);
203     User selected = strategy.selectCourier(order, couriers);
204     selected.getCourierOrders().add(order);
205     order.setCourierName(selected.getUsername());
206     return updateUser(selected);
207   }
208
209 }
210
211 }
```

Listing 2: Data Access Object and Relevant Code

3 Service

The Service design pattern is used to separate the application functionalities from the presentation layer. It encapsulates the application logic.

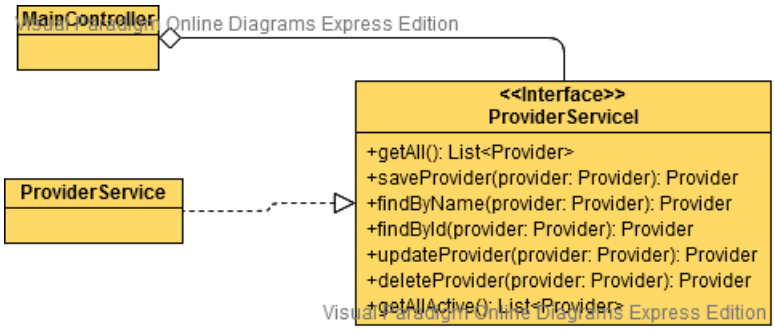

Figure 3: Service Design Pattern UML

```
1
2  public interface ProviderServiceI {
3
4    List<Provider> getAll();
5
6    Provider saveProvider(Provider provider);
7
8    Provider findByName(Provider provider);
9
10   Provider findById(Provider provider);
11
12   Provider updateProvider(Provider provider);
13
14   Provider deleteProvider(Provider provider);
15
16   List<Provider> getAllActive();
17
18 }
19
20
21
22
23 @Service
24 public class ProductService implements ProductServiceI {
25
26   @Autowired
27   private ProductRepository prodRepo;
28
29   @Autowired
30   private CategoryServiceI catDao;
31
32   @Autowired
33   private ProviderServiceI provDao;
```

```
34
35  @Override
36  public Product saveProductWithImage(Product product,
      MultipartFile file) throws IOException {
37    if (product == null)
38      return null;
39    if (getByName(product) != null)
40      return null;
41    if (product.getProductStatus() == null)
42      product.setProductStatus(Status.ACTIVE);
43    product.setImage(new Binary(BsonBinarySubType.BINARY, file.
      getBytes()));
44    return prodRepo.save(product);
45  }
46
47  @Override
48  public Product saveProduct(Product product) {
49    if (product == null)
50      return null;
51    if (getByName(product) != null)
52      return null;
53    if (product.getProductStatus() == null)
54      product.setProductStatus(Status.ACTIVE);
55    return prodRepo.save(product);
56  }
57
58  @Override
59  public Product getByName(Product product) {
60    if (product == null)
61      return null;
62    return prodRepo.findByName(product.getName());
63  }
64
65  @Override
66  public Product getById(Product product) {
67    if (product == null)
68      return null;
69    return prodRepo.findById(product.getId()).orElse(null);
70  }
71
72  @Override
73  public Product updateProductImage(Product product, MultipartFile
      file) throws IOException {
74    if (file == null)
75      return updateProduct(product);
76    if (product == null)
77      return null;
78    Product found = getByName(product);
79    if (found == null) {
80      return null;
81    } else {
82      if (file == null || file.isEmpty())
83        product.setImage(found.getImage());
84      else
85        product.setImage(new Binary(BsonBinarySubType.BINARY, file.
      getBytes()));
86      product.setId(found.getId());
```

```
87        return prodRepo.save(product);
88      }
89    }
90
91    @Override
92    public Product updateProduct(Product product) {
93      if (product == null)
94        return null;
95      Product found = getByName(product);
96      if (found == null) {
97        return null;
98      } else {
99        product.setId(found.getId());
100       return prodRepo.save(product);
101     }
102   }
103
104   @Override
105   public Product deleteProduct(Product product) {
106     if (product == null)
107       return null;
108     Product found = getByName(product);
109     if (found == null) {
110       return null;
111     } else {
112       found.setProductStatus(Status.DELETED);
113       return prodRepo.save(found);
114     }
115   }
116
117   @Override
118   public List<Product> getAll() {
119     return prodRepo.findAll();
120   }
121
122   @Override
123   public List<Product> getAllActiveByNameLike(Product product) {
124     if (product == null)
125       return null;
126     String name = Arrays.stream(product.getName().split("\\s+"))
127         .map(t -> t.substring(0, 1).toUpperCase() + t.substring(1))
      .collect(Collectors.joining(" "));
128     return prodRepo.findByNameLikeAndProductStatus(name, Status.
      ACTIVE);
129   }
130
131   @Override
132   public List<Product> getAllActiveByCatergory(Category category) {
133     if (category == null)
134       return null;
135     Category fCat = catDao.findByName(category);
136     if (fCat == null)
137       return null;
138     return prodRepo.findAllByCategoryAndProductStatus(fCat, Status.
      ACTIVE);
139   }
140
```

```
  @Override
  public List<Product> getAllActiveByProvider(Provider provider) {
    if (provider == null)
      return null;
    Provider fProv = provDao.findByName(provider);
    if (fProv == null)
      return null;
    return prodRepo.findAllByProviderAndProductStatus(fProv, Status
    .ACTIVE);
  }

  @Override
  public List<Product> getAllByCategory(Category cat) {
    if (cat == null)
      return null;
    Category fCat = catDao.findByName(cat);
    if (fCat == null)
      return null;
    return prodRepo.findAllByCategory(fCat);
  }

  @Override
  public List<Product> getAllByProvider(Provider provider) {
    if (provider == null)
      return null;
    Provider fProv = provDao.findByName(provider);
    if (fProv == null)
      return null;
    return prodRepo.findAllByProvider(fProv);
  }

  @Transactional
  @Override
  public List<Product> getAllActive() {
    return prodRepo.findByProductStatus(Status.ACTIVE);
  }
}



public class MainController {

@Autowired
  private ProductServiceI productDao;

}
```

Listing 3: Data Access Object and Relevant Code