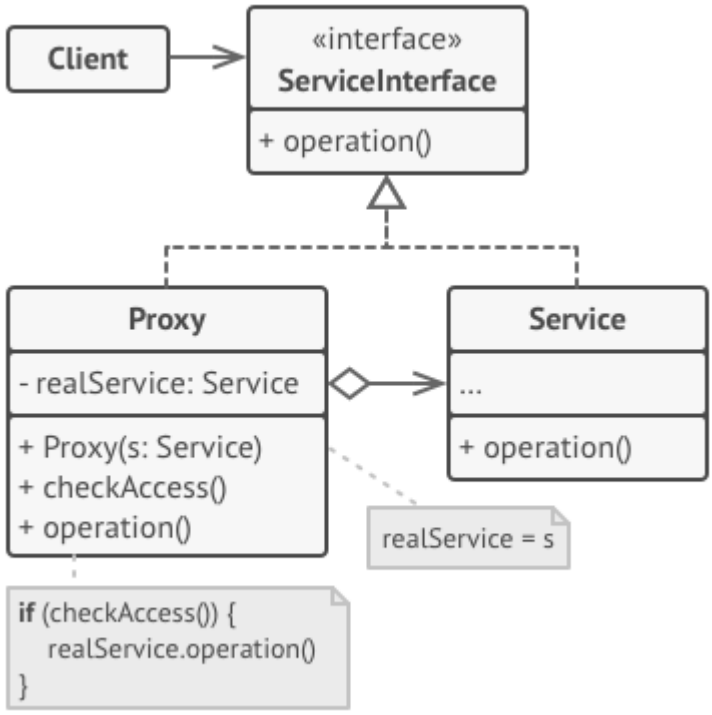
Spring Framework’ de Kullanılan Design Patternler

Proxy Design Pattern

Proxy Factory Bean

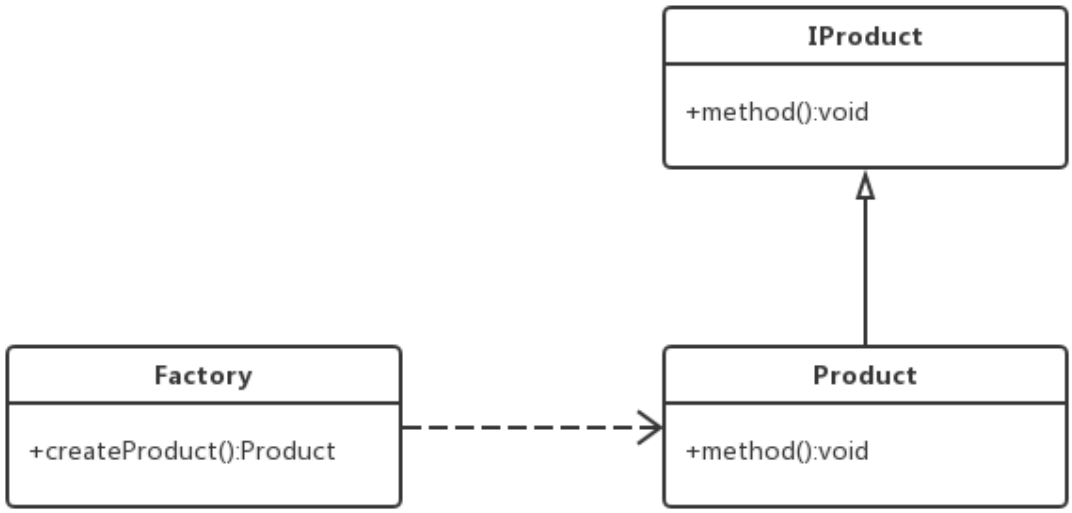
Yüksek oranda sistem kaynağı tüketen ve arada sırada ihtiyaç duyduğumuz nesneleri her zaman bellekte tutmak istemeyiz, istek dahilinde bu nesneyi oluşturup işlem tamamlandığında da nesneyi bellek dışı bırakmak isteriz. Bu problemi Proxy Design Pattern ile çözebiliriz. Aşağıdaki diyagramda da görüldüğü gibi bir interface oluşturup, bu interface’ i implemente edecek Proxy ve Service sınıflarımızı oluştururuz. Bu sınıflar aynı interface’ i implemente ettiğinden ortak methodlara sahip olurlar. Proxy ve Service, türleri aynı olduğundan Proxy’ deki bir method içinden Service’ deki method çağrılır. Proxy’ deki methoda Client erişir ancak Service’ e direkt erişemez. Proxy aracılığı ile erişmiş olur. Proxy’ e bu özelliğinden dolayı erişimi kontrol eden sınıf denir. Service’ den nesne oluşturma işlemi Proxy sınıfında yapılır.



Factory Design Pattern

Spring BeanFactory Container

Factory Design Pattern, Spring içine entegre edilmiş, FactorBean anotasyonu ile kullanılabilen design patterndir. Aşağıdaki diyagramda da görüldüğü gibi, bir nesneyi, Factory aracılığı ile oluştururuz yani bir nesneyi oluşturma görevi artık Factory' e aittir. Factory Pattern, kalıtımsal ilişkileri olan nesnelerin üretilmesi amacı ile kullanılır.



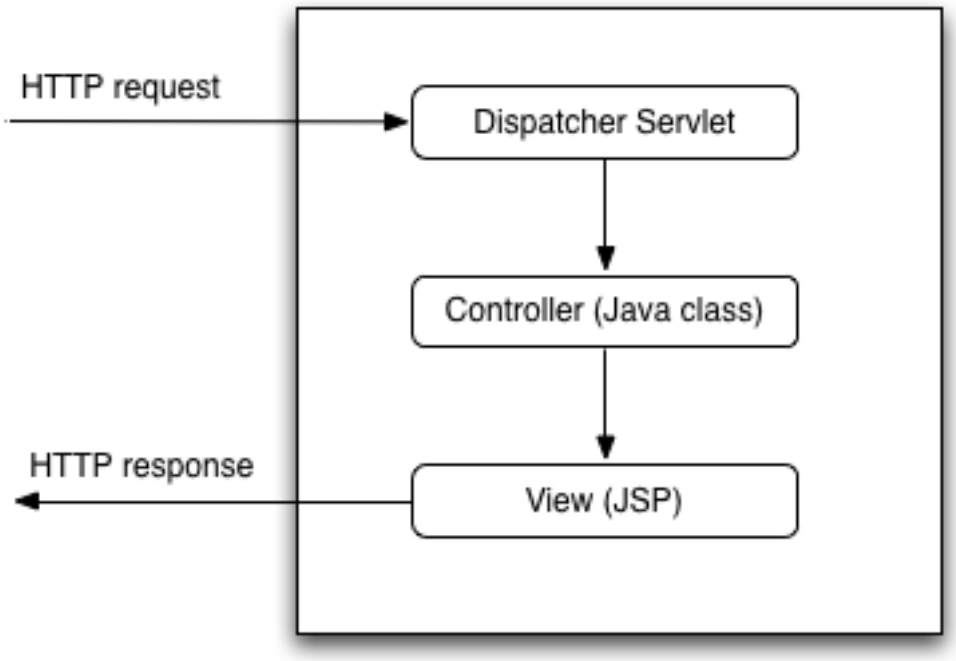

```java
public interface FactoryBean {
    T getObject() throws Exception;
    Class<?> getObjectType();
    boolean isSingleton();
}
```

```java
public class Tool {

    private int id;

    // standard constructors, getters and setters
}
```

```java
public class ToolFactory implements FactoryBean<Tool> {

    private int factoryId;
    private int toolId;

    @Override
    public Tool getObject() throws Exception {
        return new Tool(toolId);
    }

    @Override
    public Class<?> getObjectType() {
        return Tool.class;
    }

    @Override
    public boolean isSingleton() {
        return false;
    }

    // standard setters and getters
}
```

Model View Contoller Pattern

Spring Model View Controller, Spring framework’ ün desteklediği bir modül ve bir design pattern’ dir. Bu patternde, aşağıdaki diyagramda görüldüğü gibi üç katman bulunur. Spring, bu katmanların birinde, DispatcherServlet kullanarak aslında front controller pattern’ i de kullanmış olur. Gereken nesne tanımlamalarının yapıldığı katman Model katmanı, bu nesnelerin arayüz olarak kullanıcıya sunulduğu yer View katmanı ve isteklerin yapılıp modelde güncellemelerin yapıldığı katman da Controller katmanıdır.



Controller Class

```
@Controller
public class HomeController {

	/**
	 * Simply selects the home view to render by returning its name.
	 */
	@RequestMapping(value = "/", method = RequestMethod.GET)
	public String home(Locale locale, Model model) {
		System.out.println("Home Page Requested, locale = " + locale);
		Date date = new Date();
		DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG,
DateFormat.LONG, locale);

		String formattedDate = dateFormat.format(date);

		model.addAttribute("serverTime", formattedDate);

		return "home";
	}

	@RequestMapping(value = "/user", method = RequestMethod.POST)
	public String user(@Validated User user, Model model) {
		System.out.println("User Page Requested");
		model.addAttribute("userName", user.getUserName());
		return "user";
	}
}
```

Model Class

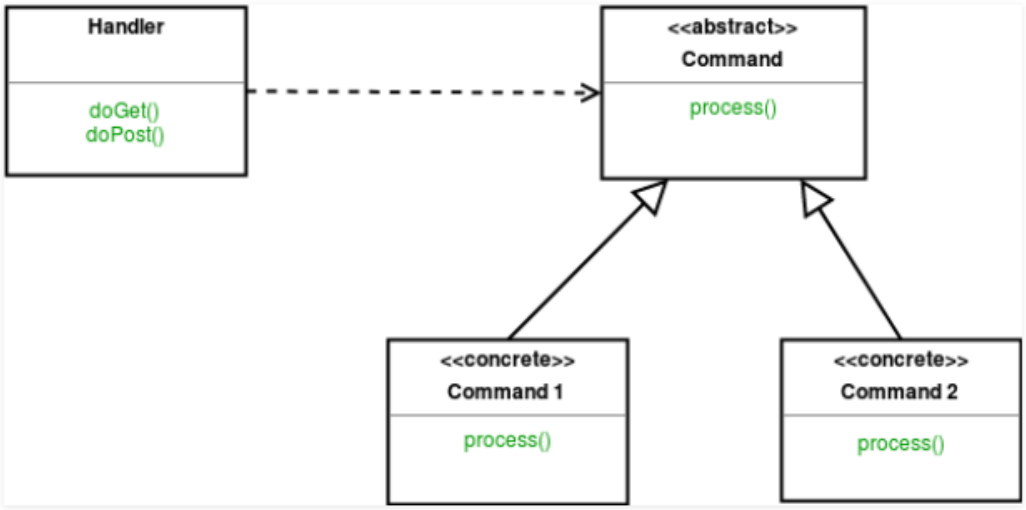
```
public class User {
	private String userName;

	public String getUserName() {
		return userName;
	}

	public void setUserName(String userName) {
		this.userName = userName;
	}
}
```

View Pages

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"https://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>User Home Page</title>
</head>
<body>
<h3>Hi ${userName}</h3>
</body>
</html>
```

Front Controller Pattern

Spring framework' de Dispatcher Servlet olarak adlandırılır. MVC' de View katmanında kullanılır. Front Controller' da amaç, aşağıdaki diyagramda görüldüğü gibi istekleri merkezi bir yerde karşılayıp, ilgili yere yönlendirmektir. Kullanıcıdan gelen her istek farklı yetkiler gerektirebilir. Her isteği kontrol etmektense, gelen istekleri merkez noktada toplayıp sonrasında ilgili yere yönlendirmek kodumuzda tekrarlamanın önüne geçer. Gelen bütün istekleri karşılayan sınıf FrontController' dır. Sonrasında aşağıdaki Front Controller Pattern' i basitçe anlatan programdaki gibi, istekler kategorilerine göre FrontController' da dağılır.

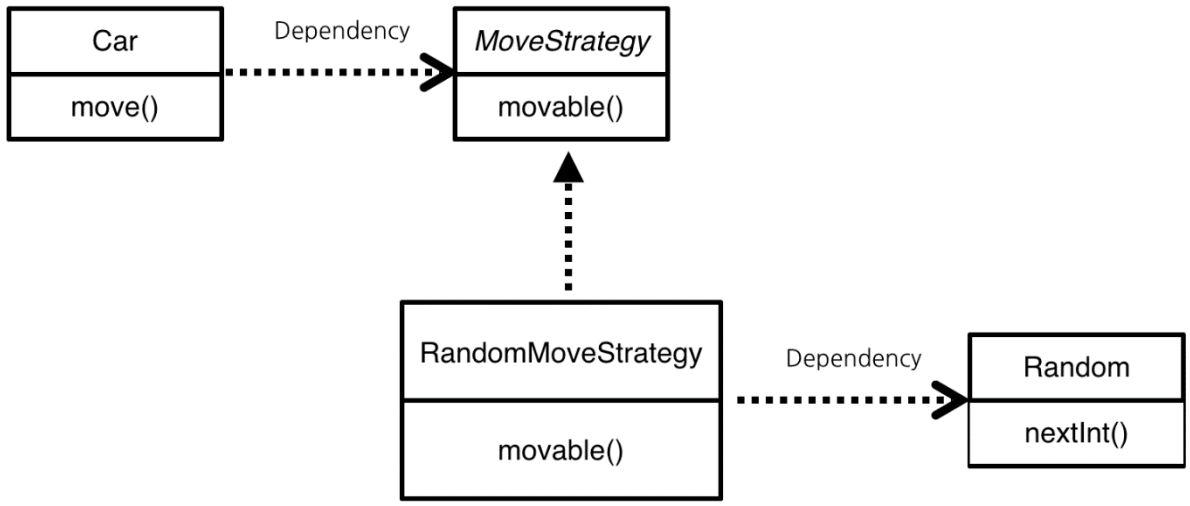


```java
class TeacherView
{
    public void display()
    {
        System.out.println("Teacher View");
    }
}

class StudentView
{
    public void display()
    {
        System.out.println("Student View");
    }
}


class Dispatching
{
    private StudentView studentView;
    private TeacherView teacherView;

    public Dispatching()
    {
        studentView = new StudentView();
        teacherView = new TeacherView();
    }

    public void dispatch(String request)
    {
        if(request.equalsIgnoreCase("Student"))
        {
            studentView.display();
        }
        else
        {
            teacherView.display();
        }
    }
}
```

```
class FrontController
{
    private Dispatching Dispatching;

    public FrontController()
    {
        Dispatching = new Dispatching();
    }

    private boolean isAuthenticUser()
    {
        System.out.println("Authentication successful.");
        return true;
    }

    private void trackRequest(String request)
    {
        System.out.println("Requested View: " + request);
    }

    public void dispatchRequest(String request)
    {
        trackRequest(request);

        if(isAuthenticUser())
        {
            Dispatching.dispatch(request);
        }
    }
}

class FrontControllerPattern
{
    public static void main(String[] args)
    {
        FrontController frontController = new FrontController();
        frontController.dispatchRequest("Teacher");
        frontController.dispatchRequest("Student");
    }
}
```

Dependency Injection

Dependency Injection, bağımlılıkların kontrolü için kullanılır. Inversion of Control' ün bir uygulama methodudur. Bir sistemi tasarlarken ve bu sistemi yazılıma döktüğümüzde, kullandığımız bir class da başka bir class a mutlaka ihtiyaç duyarız. Buna bağımlılık denir. Aşağıdaki diyagramda bağımlılıklar basitçe açıklanmış.




```java
@RestController
@RequestMapping(value="/users", method = RequestMethod.GET)
public class UsersController {

    private UserDao userDao;
    private TodoDao todoDao;

    @Autowired
    public UsersController(UserDao userDao, TodoDao todoDao){
        this.userDao=userDao;
        this.todoDao=todoDao;
    }
```

Bağımlı olunan classı, içinde kullanmak istediğimiz classa enjekte ederiz.

Dependency Injection’ da başka bir classda kullanmak istediğimiz başka bir classa ait nesneyi, new ile oluşturmak yerine constructor yardımı ile ya da getter setter methodları ile parametre olarak almamız gerektiği anlatılır. Böylece iki sınıfı birbirinden izole edeceğimiz ve düşük seviyede bir bağımlılık elde edeceğimiz savunulur.

Yukarıda görülen kod parçasındaki gibi, Controller classında Data Access Object nesnesine ihtiyaç duyulmuş ve bu class dan bir bir obje oluşturup, UsersController’ a verilmiş.

KAYNAKLAR

1) https://www.baeldung.com/
2) https://refactoring.guru/
3) https://www.javaguides.net/
4) https://www.tutorialspoint.com/index.htm
5) http://www.javaturk.org/
6) https://www.geeksforgeeks.org/
7) https://www.journaldev.com/
8) https://www.injavawetrust.com/

Not: Çok fazla siteden okudum bu yüzden tüm sitelere yer veremedim. Ancak en çok baktığım, kodlarından yararlandığım siteler bunlar oldu.

Şeyma Tezcan