

Паттерны проектирования и GoF паттерны в Spring приложении








Евгений Манько

Java-разработчик, создатель данного курса

- ✨ Разрабатывал бэкенд для Яндекс, Тинькофф, МТС;
- ✨ Победитель грантового конкурса от “Росмолодежь”
- ✨ Руководил IT-Департаментом “Студенты Москвы”
- ✨ И т.д.



Что будет на уроке сегодня

-  Паттерны проектирования
-  GoF паттерны
-  Архитектурные паттерны
-  Паттерны интеграций, безопасности
-  Singleton, Sharding, Leader Election



GoF Паттерны



GoF или "**Gang of Four**" паттерны — это четверка умных парней, которые собрали в одной книге "Design Patterns" базовые принципы и паттерны, которые можно использовать при проектировании программ.

- Кодовая База Становится Чище.
- Облегчает Командную Работу.
- Повторное Использование Кода.
- Повышение Производительности.





Архитектурные Паттерны



- Следуя архитектурным паттернам, ты можешь создать Spring приложение, которое не только хорошо спроектировано внутри, но и отлично впишется в любую экосистему, будь то микросервисы или монолитная архитектура.
- Это сделает твою жизнь гораздо проще, когда дело доходит до тестирования, деплоя и масштабирования.





Паттерны Интеграций

Spring Framework предоставляет множество инструментов для работы с интеграцией, начиная от Spring Integration и заканчивая поддержкой различных протоколов и стандартов.

Поэтому знание паттернов интеграции сильно упрощает работу с Spring в реальных проектах, где интеграция с внешними системами — это не роскошь, а необходимость.





Паттерны Безопасности

Spring Security — это один из модулей Spring, который позволяет эффективно решать задачи в области безопасности. Он предоставляет инструменты, которые поддерживают множество паттернов безопасности из коробки.





Структурные GoF Паттерны



Adapter: Переводчик между двумя Мирами

Представьте, что у вас есть два класса с разными интерфейсами, которые нужно как-то заставить работать вместе. Adapter в этом случае выступает как переводчик между ними.

```
1 // Сторонний класс, который мы не можем изменить
2 public class ThirdPartyLibrary {
3     public void doSomethingSpecific() {
4         // логика
5     }
6 }
7
8 // Наш интерфейс
9 public interface OurInterface {
10     void doSomething();
11 }
12
13 // Adapter
14 public class Adapter implements OurInterface {
15     private ThirdPartyLibrary thirdPartyLibrary;
16
17     public Adapter(ThirdPartyLibrary thirdPartyLibrary) {
18         this.thirdPartyLibrary = thirdPartyLibrary;
19     }
20
21     public void doSomething() {
22         thirdPartyLibrary.doSomethingSpecific();
23     }
24 }
```





Bridge и Composite

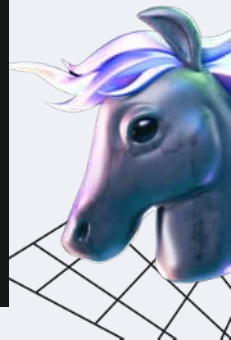


Bridge: Разделяй и Властвуй

Этот паттерн помогает разделить абстракцию от реализации, так что обе стороны могут изменяться независимо друг от друга.

В Spring это полезно, когда у вас есть разные реализации одного и того же интерфейса и вы хотите динамически переключаться между ними.

```
1 public interface Logger {
2     void log(String message);
3 }
4
5 public class ConsoleLogger implements Logger {
6     public void log(String message) {
7         System.out.println("Console: " + message);
8     }
9 }
10
11 public class FileLogger implements Logger {
12     public void log(String message) {
13         // логика записи в файл
14     }
15 }
16
17 public class App {
18     private Logger logger;
19
20     public App(Logger logger) {
21         this.logger = logger;
22     }
23
24     public void doSomething() {
25         logger.log("Doing something");
26     }
27 }
```





Bridge и Composite

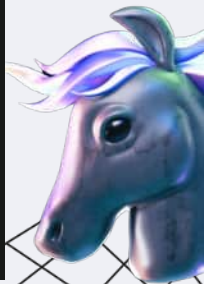


Composite: Один за Всех и Все за Одного

Этот паттерн позволяет считать единичный объект и композицию объектов одинаково.

Круто для работы с деревьями или графами в Spring!

```
1 public interface Graphic {
2     void draw();
3 }
4
5 public class Circle implements Graphic {
6     public void draw() {
7         // рисуем круг
8     }
9 }
10
11 public class GraphicComposite implements Graphic {
12     private List<Graphic> graphics = new ArrayList<>();
13
14     public void addGraphic(Graphic graphic) {
15         graphics.add(graphic);
16     }
17
18     public void draw() {
19         for (Graphic graphic : graphics) {
20             graphic.draw();
21         }
22     }
23 }
```





Singleton

Он гарантирует, что класс имеет только один экземпляр и предоставляет глобальную точку доступа к этому экземпляру.

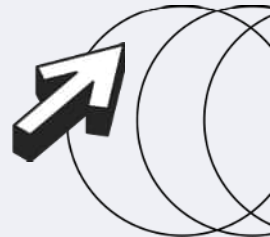
```
1 @Service
2 public class SingletonService {
3     private static SingletonService instance;
4
5     private SingletonService() {
6     }
7
8     public static SingletonService getInstance() {
9         if (instance == null) {
10             instance = new SingletonService();
11         }
12         return instance;
13     }
14 }
```





Factory Method

```
1 public interface PaymentService {
2     void pay();
3 }
4
5 @Service("paypal")
6 public class PaypalPaymentService implements PaymentService {
7     // ...
8 }
9
10 @Service("creditCard")
11 public class CreditCardPaymentService implements PaymentService {
12     // ...
13 }
14
15 @Component
16 public class PaymentServiceFactory {
17     @Autowired
18     private Map<String, PaymentService> services;
19
20     public PaymentService getService(String method) {
21         return services.get(method);
22     }
23 }
```

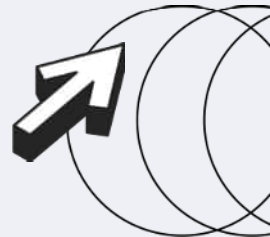




Prototype

Противоположность Singleton — это Prototype, паттерн, который создаёт новый экземпляр класса каждый раз, когда вы его запрашиваете.

```
1 @Service
2 @Scope("prototype")
3 public class PrototypeService {
4     // Каждый раз при запросе будет создан новый экземпляр этого бина
5 }
```





Observer и Сила Реактивности в Spring



1. Создадим событие:

```
1 public class TaskUpdatedEvent extends ApplicationEvent {  
2     private Task task;  
3  
4     public TaskUpdatedEvent(Object source, Task task) {  
5         super(source);  
6         this.task = task;  
7     }  
8  
9     // getters  
10 }
```

2. Теперь реализуем слушатель:



```
1 @Component  
2 public class TaskUpdatedListener implements ApplicationListener<TaskUpdatedEvent> {  
3     @Override  
4     public void onApplicationEvent(TaskUpdatedEvent event) {  
5         // Сделать что-то с event.getTask()  
6     }  
7 }
```




Observer и Сила Реактивности в Spring

3. И, наконец, опубликуем событие, когда задача обновляется:

```
1 @Service
2 public class TaskService {
3     @Autowired
4     private ApplicationEventPublisher publisher;
5
6     public void updateTask(Task task) {
7         // обновляем задачу
8         publisher.publishEvent(new TaskUpdatedEvent(this, task));
9     }
10 }
```



Strategy

1. Создаём интерфейс для стратегии:

```
1 public interface PaymentStrategy {  
2     void pay(int amount);  
3 }
```

2. Реализуем конкретные стратегии:

```
1 public class CardPayment implements PaymentStrategy {  
2     public void pay(int amount) {  
3         // Реализация оплаты через карточку  
4     }  
5 }  
6  
7 public class PayPalPayment implements PaymentStrategy {  
8     public void pay(int amount) {  
9         // Реализация оплаты через PayPal  
10    }  
11 }
```





Strategy

3. Используем их:

```
1 @Service
2 public class PaymentService {
3     private PaymentStrategy paymentStrategy;
4
5     public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
6         this.paymentStrategy = paymentStrategy;
7     }
8
9     public void pay(int amount) {
10         paymentStrategy.pay(amount);
11     }
12 }
```





Command



```
1 public interface Command {  
2     void execute();  
3 }  
4  
5 public class StartCommand implements Command {  
6     public void execute() {  
7         // Запуск чего-либо  
8     }  
9 }  
10  
11 public class StopCommand implements Command {  
12     public void execute() {  
13         // Остановка чего-либо  
14     }  
15 }
```

```
15 }  
16  
17 @Service  
18 public class CommandExecutor {  
19     private Command command;  
20  
21     public void setCommand(Command command) {  
22         this.command = command;  
23     }  
24  
25     public void runCommand() {  
26         command.execute();  
27     }  
28 }
```





MVC: Как Сложить Всё По Полочкам



1. **Model** — это ваши сущности и сервисы.

```
1 public class User {  
2     private String name;  
3     // getters and setters  
4 }
```

2. **View** — это ваш front-end.

3. **Controller**

```
1 @Controller  
2 public class UserController {  
3     @Autowired  
4     private UserService userService;  
5  
6     @GetMapping("/users")  
7     public String listUsers(Model model) {  
8         model.addAttribute("users", userService.findAll());  
9         return "users";  
10    }  
11 }
```





Microservices

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class UserServiceApplication {
4     public static void main(String[] args) {
5         SpringApplication.run(UserServiceApplication.class, args);
6     }
7 }
```





Message Bus

```
1 @EnableIntegration
2 public class AppConfig {
3
4     @Bean
5     public MessageChannel messageChannel() {
6         return new DirectChannel();
7     }
8
9     @ServiceActivator(inputChannel = "messageChannel")
10    @Bean
11    public MessageHandler messageHandler() {
12        return message → System.out.println("Received message: " + message);
13    }
14 }
```



Publish/Subscribe

```
1 @Component
2 public class Publisher {
3
4     @Autowired
5     private ApplicationEventPublisher publisher;
6
7     public void doStuffAndPublishEvent() {
8         // Делаем что-то полезное
9         publisher.publishEvent(new MyCustomEvent(this, "Event message"));
10    }
11 }
12
13 @Component
14 public class Subscriber implements ApplicationListener<MyCustomEvent> {
15
16     @Override
17     public void onApplicationEvent(MyCustomEvent event) {
18         System.out.println("Received: " + event.getMessage());
19     }
20 }
```





Singleton Security Context

```
1 import org.springframework.security.core.context.SecurityContextHolder;
2
3 public class SecurityService {
4     public void performSecureAction() {
5         var authentication = SecurityContextHolder.getContext().getAuthentication();
6
7         if (authentication != null && "ROLE_ADMIN".equals(authentication.getAuthorities())) {
8             // Выполняем какое-то действие
9         } else {
10             throw new SecurityException("Недостаточно прав!");
11         }
12     }
13 }
```



Sharding



```
1 // Код только для иллюстрации, Spring Data JPA не предоставляет нативную поддержку шардинга
2 public class OrderService {
3
4     public void saveOrder(Order order) {
5         // Определение шарда на основе ID пользователя
6         String shard = shardResolver.resolveShard(order.getUserId());
7
8         // Сохранение заказа в соответствующем шарде
9         orderRepository.saveToShard(shard, order);
10    }
11 }
```





Leader Election

```
1 @Service
2 public class LeaderService implements SmartLifecycle {
3
4     private boolean isRunning = false;
5
6     @Autowired
7     private LeaderInitiator leaderInitiator;
8
9     @Override
10    public void start() {
11        isRunning = true;
12        leaderInitiator.start(); // Инициация процесса выбора лидера
13    }
14
15    // ...
16 }
```





Заключение

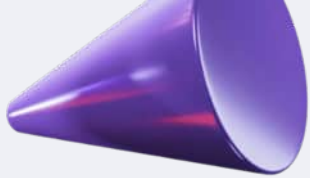
Плюсы паттернов

1. Эффективность.
2. Читаемость и Поддержка.
3. Карьерный Рост.
4. Spring Love.

Что мы получили?

1. Практические Навыки.
2. Понимание Архитектуры.
3. Безопасность и Масштабирование.
4. Тестирование и Мониторинг.





Спасибо за внимание

