

Chapter-4:

Spring context: Using abstractions

Upcode Software
Engineer Team



CONTENT

1. Using interfaces to define contracts
 - 1.1. Using interfaces for decoupling implementations
 - 1.2. Implementing the requirement without using a framework
2. Using dependency injection with abstractions
 - 2.1. Deciding which objects should be part of the Spring context
 - 2.2. Choosing what to auto-wire from multiple implementations of an abstraction
3. Focusing on object responsibilities with stereotype annotations
4. Conclusion
5. References

1.1. Using interfaces to define contracts



1.1. Using interfaces to define contracts

```
public class DeliveryDetailsPrinter {  
    private SorterByAddress sorter;  
  
    public DeliveryDetailsPrinter (SorterByAddress sorter)  
    {  
        this.sorter = sorter;  
    }  
  
    public void printDetails () {  
        sorter.sortDetails();  
        //печать сведения о доставке  
    }  
}
```

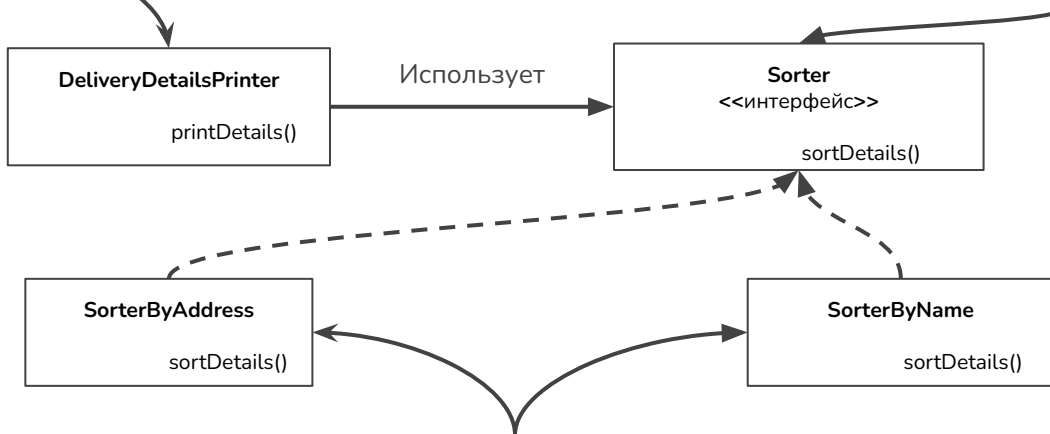
При необходимости изменить
сортировки придется
модифицировать код в этих
местах

код №1.

1.1. Using interfaces to define contracts

В объекте **DeliveryDetailsPrinter** указано только то, что нужно для реализации обязанности объекта. Теперь объект **DeliveryDetailsPrinter** зависит не от реализации, а от интерфейса.

Интерфейс **Sorter** определяет, что нужно объекту **DeliveryDetailsPrinter**.



Теперь можно создать несколько объектов, реализующих один интерфейс. Так мы сможем изменять реализацию ("то, как это именно это должно случиться"), не затрагивая объект, который получает результаты от одной реализации(**DeliveryDetailsPrinter**)

1.1. Using interfaces to define contracts

Применение интерфейса для разделения обязанностей. Теперь объект **DeliveryDetailsPrinter** зависит не от самой реализации, а от интерфейса (контракта). **DeliveryDetailsPrinter** больше не привязан к определенной реализации и может использовать любой объект, реализующий интерфейс

```
public class DeliveryDetailsPrinter {  
    private Sorter sorter;  
  
    public DeliveryDetailsPrinter (Sorter sorter) {  
        this.sorter = sorter;  
    }  
  
    public void printDetails () {  
        sorter.sortDetails();  
        //печать сведений о доставке  
    }  
}
```

Теперь можно применять любую реализацию интерфейса **Sorter**, и не потребуется изменять объект, использующий эту обязанность

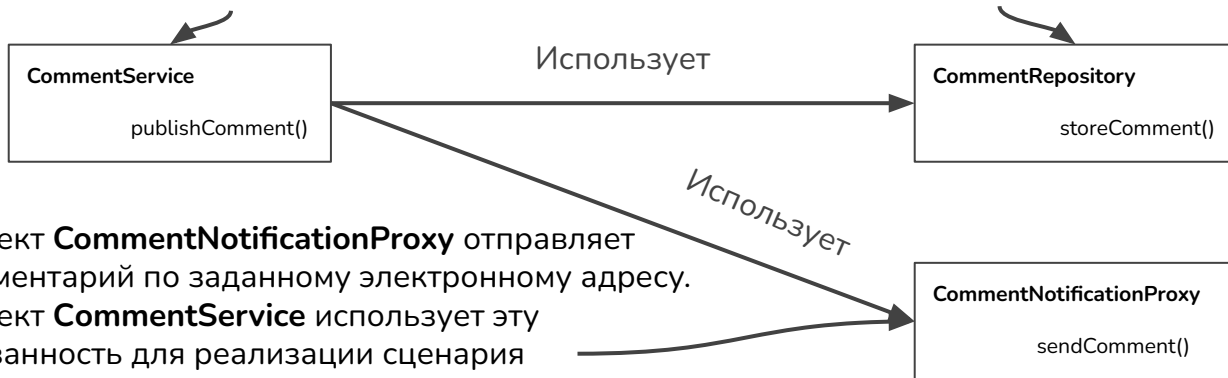
Объект **DeliveryDetailsPrinter** зависит от интерфейса **Sorter**. Можно изменить реализацию интерфейса **Sorter**, не меняя объект, использующий эту обязанность (**DeliveryDetailsPrinter**)

1.2. Implementing the requirement without using a framework

Реализация сценариев использования без применения фреймворка

Объект **CommentService** реализует сценарий использования «опубликовать комментарий»

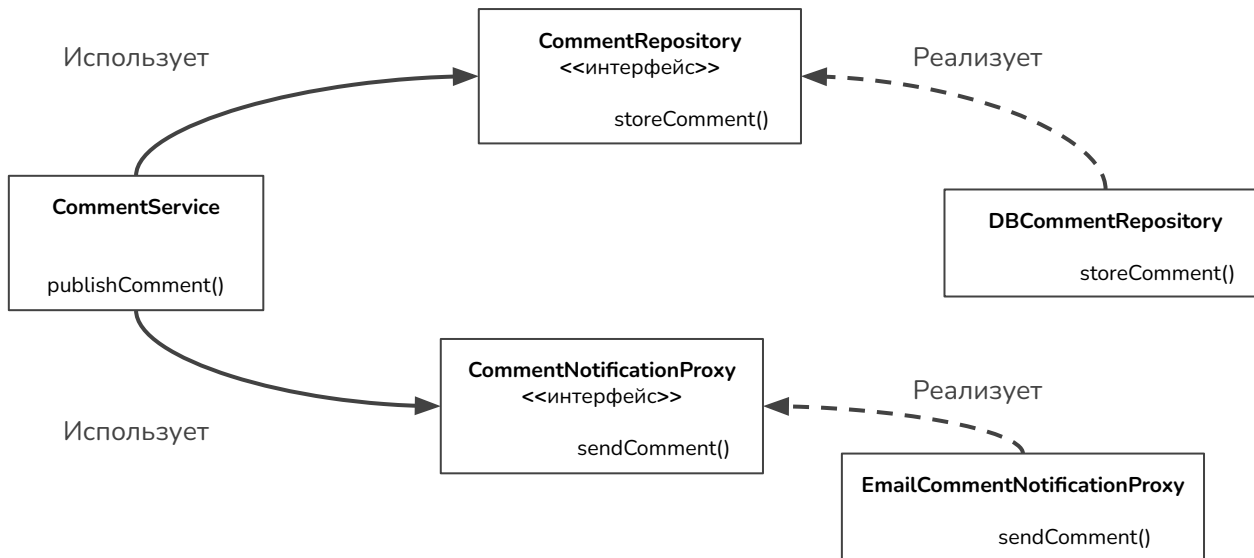
Объект **CommentRepository** сохраняет комментарий в базе данных. Объект **CommentService** использует эту обязанность для реализации сценария «опубликовать комментарий»



Объект **CommentNotificationProxy** отправляет комментарий по заданному электронному адресу. Объект **CommentService** использует эту обязанность для реализации сценария «опубликовать комментарий»

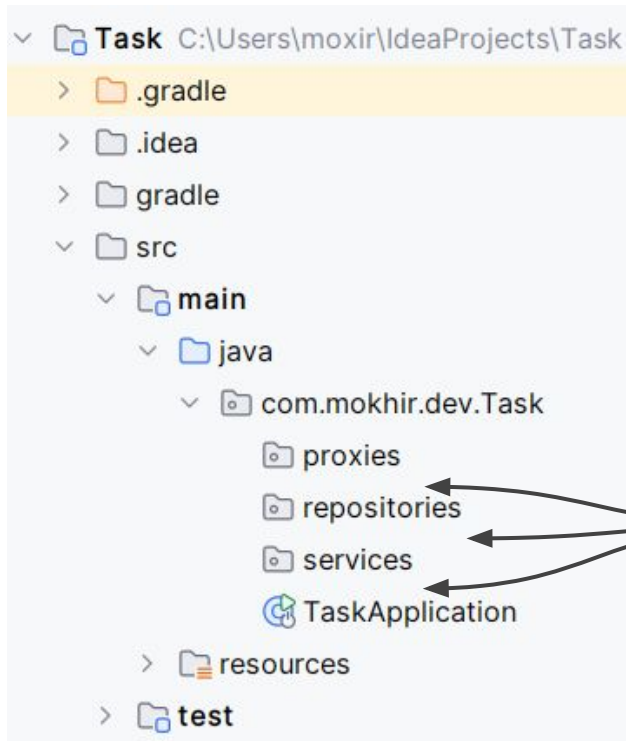
Объект **CommentService** выполняет сценарий использования «опубликовать комментарий». Для этого он делегирует обязанности, реализованные в объектах **CommentRepository** и **CommentNotificationProxy**

1.2. Implementing the requirement without using a framework



Объект **CommentService** зависит от абстракций, представленных интерфейсами **CommentRepository** и **CommentNotificationProxy**. Эти интерфейсы, в свою очередь, реализуются классами **DBCommentRepository** и **EmailCommentNotificationProxy**. В такой структуре выполнение сценария использования «опубликовать комментарий» отделено от зависимостей, благодаря чему будет проще изменять приложение в ходе дальнейшей разработки

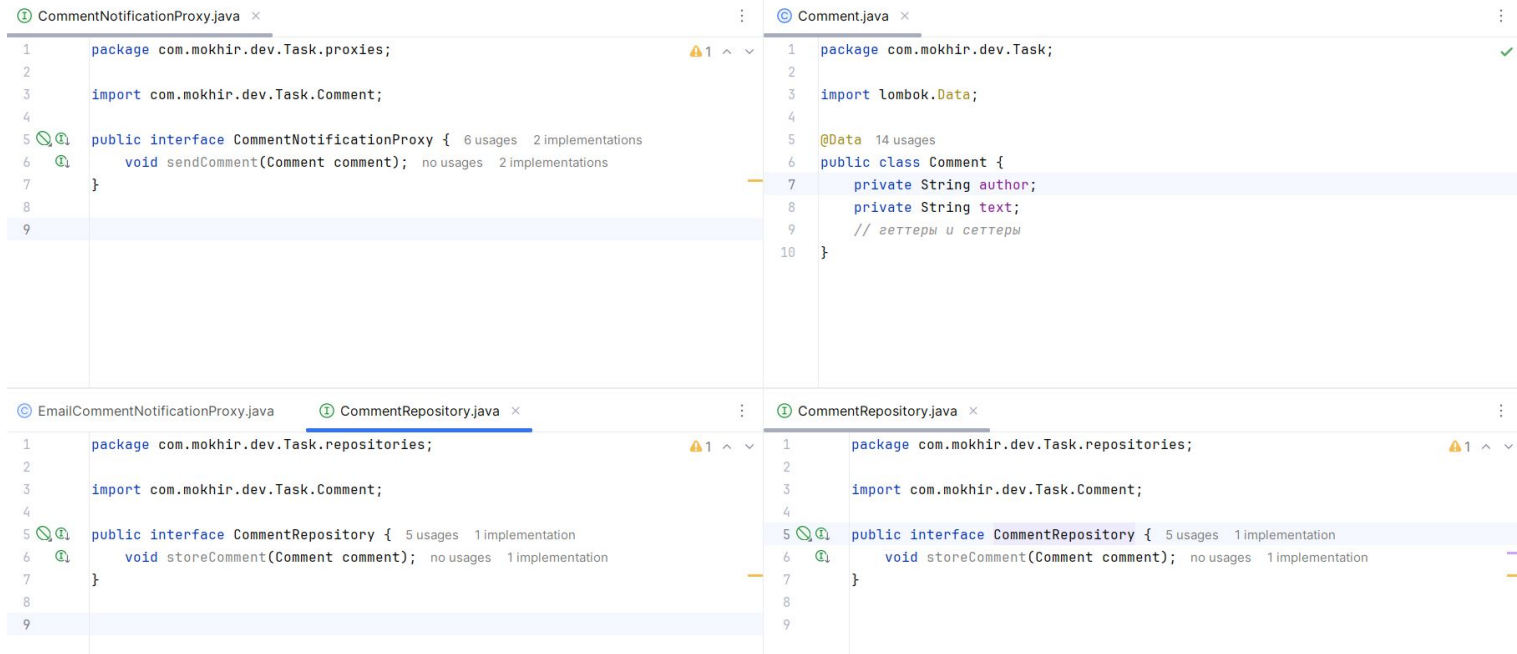
1.2. Implementing the requirement without using a framework



Структура проекта. Мы создали отдельные пакеты для каждой обязанности, чтобы вся конструкция была более понятной и удобной для чтения

Чтобы было проще разобраться в структуре проекта, мы построим его так, чтобы обязанности размещались в отдельных пакетах

1.2. Implementing the requirement without using a framework



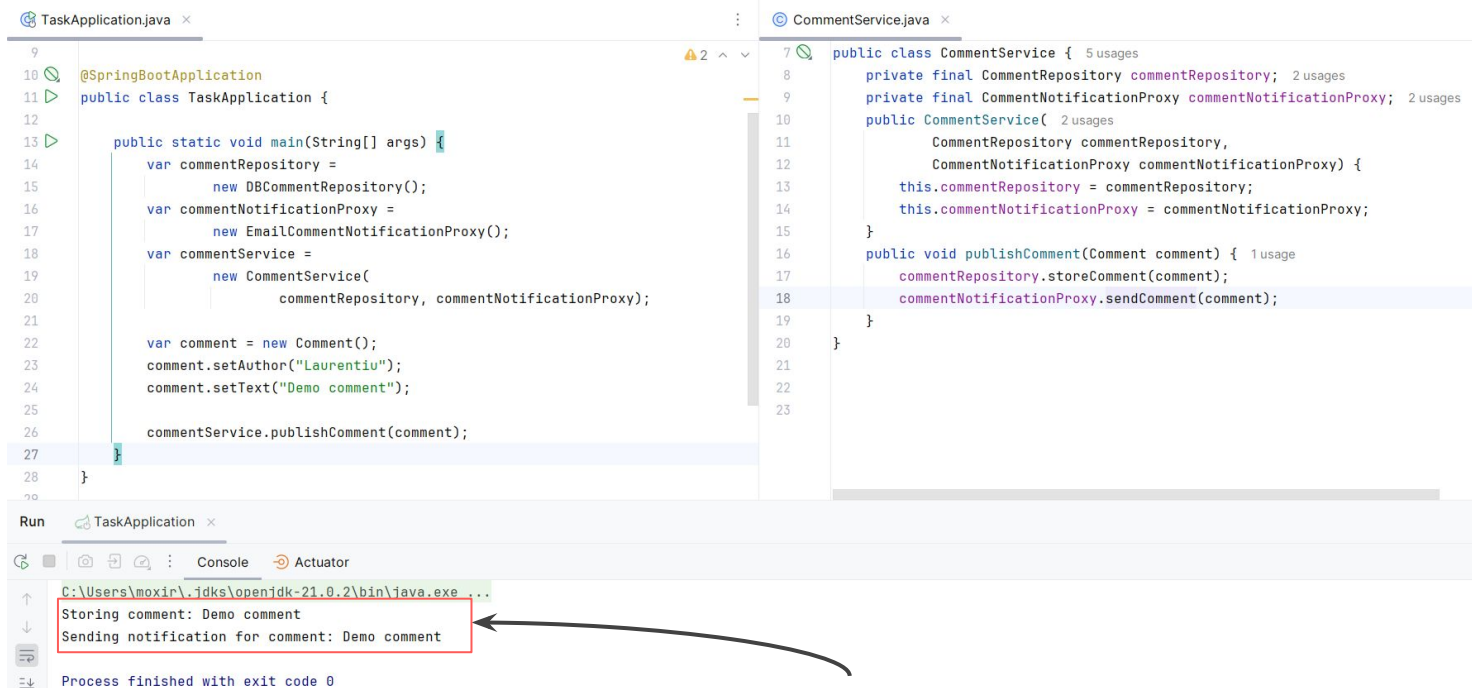
```
CommentNotificationProxy.java
1 package com.mokhir.dev.Task.proxies;
2
3 import com.mokhir.dev.Task.Comment;
4
5 public interface CommentNotificationProxy { 6 usages 2 implementations
6     void sendComment(Comment comment); no usages 2 implementations
7 }
8
9

Comment.java
1 package com.mokhir.dev.Task;
2
3 import lombok.Data;
4
5 @Data 14 usages
6 public class Comment {
7     private String author;
8     private String text;
9     // геттеры и сеттеры
10 }

EmailCommentNotificationProxy.java
1 package com.mokhir.dev.Task.repositories;
2
3 import com.mokhir.dev.Task.Comment;
4
5 public interface CommentRepository { 5 usages 1 implementation
6     void storeComment(Comment comment); no usages 1 implementation
7 }
8
9

CommentRepository.java
1 package com.mokhir.dev.Task.repositories;
2
3 import com.mokhir.dev.Task.Comment;
4
5 public interface CommentRepository { 5 usages 1 implementation
6     void storeComment(Comment comment); no usages 1 implementation
7 }
8
9
```

1.2. Implementing the requirement without using a framework



The screenshot shows an IDE with two Java files open: `TaskApplication.java` and `CommentService.java`. The `TaskApplication` class is annotated with `@SpringBootApplication` and contains a `main` method. Inside `main`, it creates instances of `DBCommentRepository`, `EmailCommentNotificationProxy`, and `CommentService`, then creates a `Comment` object and calls `publishComment` on the `CommentService` instance. The `CommentService` class has two private final fields, `commentRepository` and `commentNotificationProxy`, and a constructor that takes them as arguments. It also has a `publishComment` method that calls `storeComment` on the repository and `sendComment` on the proxy. Below the code editor, the Run tab is active, showing the console output. The output consists of two lines: `Storing comment: Demo comment` and `Sending notification for comment: Demo comment`. A red box highlights these two lines, and an arrow points from the text below to the box. The console also shows the command `C:\Users\moxir\jdk8\openjdk-21.0.2\bin\java.exe ..` and the message `Process finished with exit code 0`.

```
TaskApplication.java
9
10 @SpringBootApplication
11 public class TaskApplication {
12
13     public static void main(String[] args) {
14         var commentRepository =
15             new DBCommentRepository();
16         var commentNotificationProxy =
17             new EmailCommentNotificationProxy();
18         var commentService =
19             new CommentService(
20                 commentRepository, commentNotificationProxy);
21
22         var comment = new Comment();
23         comment.setAuthor("Laurentiu");
24         comment.setText("Demo comment");
25
26         commentService.publishComment(comment);
27     }
28 }
29
```

```
CommentService.java
7 public class CommentService { 5 usages
8     private final CommentRepository commentRepository; 2 usages
9     private final CommentNotificationProxy commentNotificationProxy; 2 usages
10     public CommentService( 2 usages
11         CommentRepository commentRepository,
12         CommentNotificationProxy commentNotificationProxy) {
13         this.commentRepository = commentRepository;
14         this.commentNotificationProxy = commentNotificationProxy;
15     }
16     public void publishComment(Comment comment) { 1 usage
17         commentRepository.storeComment(comment);
18         commentNotificationProxy.sendComment(comment);
19     }
20 }
21
22
23
```

Run TaskApplication

Console

```
C:\Users\moxir\jdk8\openjdk-21.0.2\bin\java.exe ..
Storing comment: Demo comment
Sending notification for comment: Demo comment
Process finished with exit code 0
```

Запустив это приложение, увидим в консоли следующие две строки — они выводятся объектами `CommentRepository` и `CommentNotificationProxy`.

1.2. Implementing the requirement without using a framework

Реализация объекта CommentService

```
public class CommentService {  
    private final CommentRepository commentRepository;  
    private final CommentNotificationProxy  
commentNotificationProxy;  
    public CommentService (  
        CommentRepository commentRepository ,  
        CommentNotificationProxy commentNotificationProxy ) {  
        this.commentRepository = commentRepository;  
        this.commentNotificationProxy = commentNotificationProxy;  
    }  
    public void publishComment (Comment comment) {  
        commentRepository.storeComment (comment);  
        commentNotificationProxy.sendComment (comment);  
    }  
}
```

Определяем две
зависимости в виде
атрибутов класса

Предоставляем эти
зависимости в
момент создания
объекта
посредством
параметров
конструктора

Реализуем сценарий
использования, который
делегировать зависимостям
обязанности «сохранить
комментарий» и «отправить
уведомление»

1.2. Implementing the requirement without using a framework

Реализация объекта `CommentService`

```
public class TaskApplication {
```

```
    public static void main(String[] args) {
```

```
        var commentRepository =
```

```
            new DBCommentRepository();
```

```
        var commentNotificationProxy =
```

```
            new EmailCommentNotificationProxy();
```

```
        var commentService =
```

```
            new CommentService(  
                commentRepository, commentNotificationProxy);
```

```
        var comment = new Comment();
```

```
        comment.setAuthor("Laurentiu");
```

```
        comment.setText("Demo comment");
```

```
        commentService.publishComment(comment);
```

```
    }
```

```
}
```

Создаем
экземпляры
для
зависимостей

Создаем
экземпляр класса
сервиса и
предоставляем
ему зависимости

Создаем экземпляр комментария,
чтобы передать его сценарию
использования “опубликовать
комментарий” в качестве параметра

Вызываем сценарий
использования
«опубликовать
комментарий»



2. Using dependency injection with abstractions

2. ИСПОЛЬЗОВАНИЕ ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ ДЛЯ АБСТРАКЦИЙ

2.1. Выбор объектов для добавления в контекст Spring

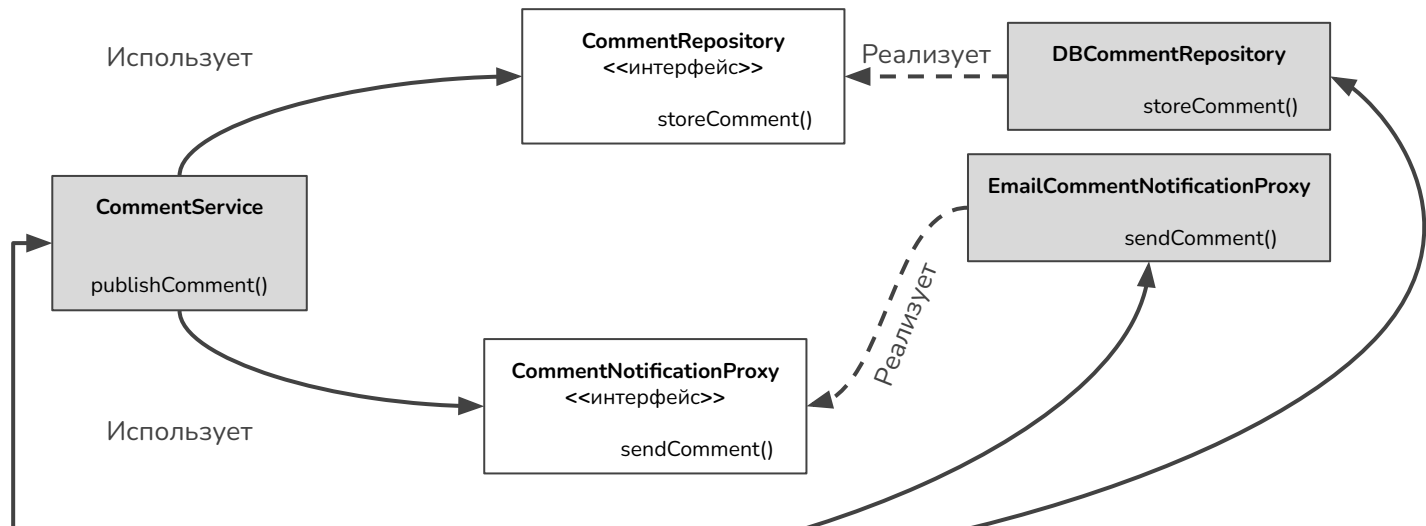
Зависимости объектов друг от друга:

- **CommentService** — имеет две зависимости, от **CommentRepository** и **CommentNotificationProxy**;
- **DBCommentRepository** — реализует интерфейс **CommentRepository** и является зависимостью для **CommentService**;
- **EmailCommentNotificationProxy** — реализует интерфейс **CommentNotificationProxy** и представляет собой зависимость для **CommentService**.

2.1. Deciding which objects should be part of the Spring context

2.1. Выбор объектов для добавления в контекст Spring

Классы со стереотипными аннотациями **@Component** выделены серым цветом. При загрузке контекста Spring создаст экземпляры этих классов и добавит их в контекст



Для этих классов использованы
стереотипные аннотации

2.1. Deciding which objects should be part of the Spring context

```
TaskApplication.java
9
10 @SpringBootApplication
11 public class TaskApplication {
12
13     public static void main(String[] args) {
14         var context =
15             new AnnotationConfigApplicationContext(
16                 ProjectConfiguration.class);
17         var comment = new Comment();
18         comment.setAuthor("Laurentiu");
19         comment.setText("Demo comment");
20         var commentService = context.getBean(CommentService.class);
21         commentService.publishComment(comment);
22     }
23
24
25
26

```

```
CommentService.java
7
8 @Component 2 usages
9 public class CommentService {
10     private final CommentRepository commentRepository; 2 usages
11     private final CommentNotificationProxy commentNotificationProxy; 2 usages
12
13     public CommentService(
14         CommentRepository commentRepository,
15         CommentNotificationProxy commentNotificationProxy) {
16         this.commentRepository = commentRepository;
17         this.commentNotificationProxy = commentNotificationProxy;
18     }
19
20     public void publishComment(Comment comment) { 1 usage
21         commentRepository.storeComment(comment);
22         commentNotificationProxy.sendComment(comment);
23     }
24
25

```

```
ProjectConfiguration.java
1 package com.mokhir.dev.Task;
2
3 > import ...
4
5 @Configuration
6 @ComponentScan(
7     basePackages = {"com"}
8 )
9
10 public class ProjectConfiguration {
11 }
12

```

```
EmailCommentNotificationProxy.java
3 > import ...
4
5 @Component
6 public class EmailCommentNotificationProxy
7     implements CommentNotificationProxy {
8
9     @Override 1 usage
10     @Override
11     public void sendComment(Comment comment) {
12         System.out.println("Sending notification for comment: "
13             + comment.getText());
14     }
15

```

```
Run TaskApplication
Console
C:\Users\moxir\.jdk\openjdk-21.0.2\bin\java.exe ...
Storing comment: Demo comment
Sending notification for comment: Demo comment
Process finished with exit code 0

```


2.1. Deciding which objects should be part of the Spring context

2.1. Выбор объектов для добавления в контекст Spring

Добавление аннотации **@Component** к классу **DBCommentRepository**

```
@Component  
public class DBCommentRepository implements CommentRepository {
```

Для класса с аннотацией **@Component**
Spring создаст экземпляр и добавит этот
экземпляр в контекст как бин

```
    @Override  
    public void storeComment (Comment comment) {  
        System.out.println("Storing comment: " + comment.getText());  
    }  
}
```

Где нужно использовать стереотипные аннотации?



2.1. Deciding which objects should be part of the Spring context

Класс EmailCommentNotificationProxy с аннотацией **@Component**

```
@Component
public class EmailCommentNotificationProxy
    implements CommentNotificationProxy {
    @Override
    public void sendComment (Comment comment) {
        System.out.println("Sending notification for comment: "
            + comment.getText());
    }
}
```

2.1. Deciding which objects should be part of the Spring context

Создание компонента из класса `CommentService`

```
@Component
public class CommentService {
    private final CommentRepository commentRepository;
    private final CommentNotificationProxy commentNotificationProxy;

    public CommentService (
        CommentRepository commentRepository ,
        CommentNotificationProxy commentNotificationProxy ) {
        this.commentRepository = commentRepository;
        this.commentNotificationProxy = commentNotificationProxy;
    }

    public void publishComment (Comment comment) {
        commentRepository.storeComment ( comment );
        commentNotificationProxy.sendComment ( comment );
    }
}
```

← Spring создает бин этого класса и добавляет его в контекст

← Если бы у этого класса было несколько конструкторов, нужно было бы использовать аннотацию `@Autowired`

← Spring использует конструктор для создания бина и при создании экземпляра внедряет в параметры ссылки из контекста

2.1. Deciding which objects should be part of the Spring context

Использование аннотации `@ComponentScan` в классе конфигурации

```
@Configuration  
@ComponentScan (   
    basePackages = { "proxies", "services", "repositories" }  
)  
public class ProjectConfiguration {  
}
```

← Класс конфигурации отмечен аннотацией
← `@Configuration`

С помощью аннотации `@ComponentScan` мы сообщаем Spring, в каких пакетах находятся классы со стереотипными аннотациями.



2.1. Deciding which objects should be part of the Spring context

Создание DI, с помощью аннотации **@Autowired**

```
@Component
public class CommentService {

    @Autowired
    private CommentRepository commentRepository;

    @Autowired
    private CommentNotificationProxy commentNotificationProxy;

    public void publishComment (Comment comment) {
        commentRepository.storeComment (comment);
        commentNotificationProxy.sendComment (comment);
    }
}
```

2.1. Deciding which objects should be part of the Spring context

Теперь меняем класс конфигурации

```
@Configuration
public class ProjectConfiguration {

    @Bean
    public CommentRepository commentRepository () {
        return new DBCommentRepository();
    }

    @Bean
    public CommentNotificationProxy commentNotificationProxy () {
        return new EmailCommentNotificationProxy();
    }

    @Bean
    public CommentService commentService (CommentRepository commentRepository ,
    CommentNotificationProxy commentNotificationProxy ) {
        return new CommentService( commentRepository,
        commentNotificationProxy );
    }
}
```

Поскольку стереотипные аннотации больше не используются, аннотация `@ComponentScan` тоже не нужна

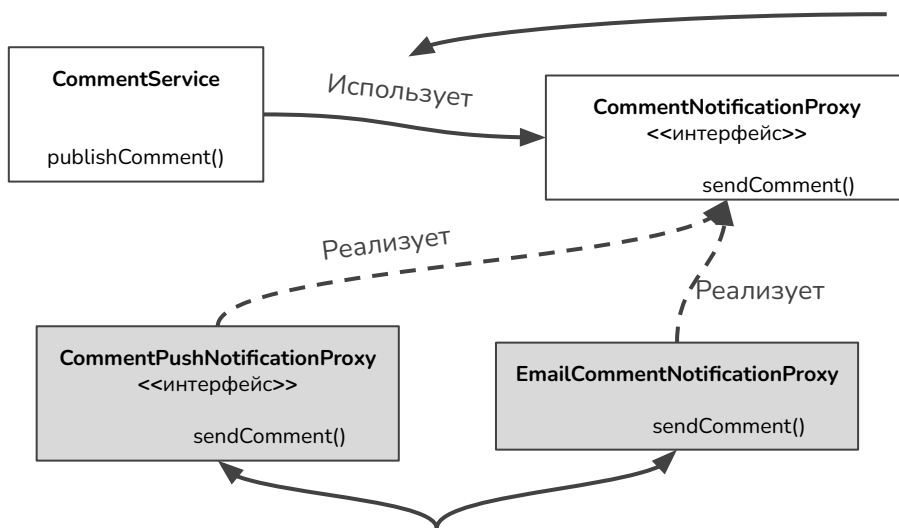
Создаем бин для каждой из двух зависимостей

С помощью параметров метода `@Bean` (которые теперь имеют тип интерфейса) сообщаем Spring, что нужно предоставить ссылки на бины из контекста;

2.2. Choosing what to auto-wire from multiple implementations of an abstraction

2.2. Выбор одной из реализаций абстракции для автомонтажа(auto wiring)

Когда CommentService запрашивает зависимость типа CommentNotificationProxy, Spring должен выбрать, какую из нескольких существующих реализаций следует внедрить



Иногда в реальных приложениях встречается несколько реализаций одного интерфейса. При внедрении зависимостей через интерфейс необходимо сообщить Spring, какую именно из реализаций следует использовать

2.2. Choosing what to auto-wire from multiple implementations of an abstraction

2.2. Выбор одной из реализаций абстракции для
автомонтажа
Новая реализация интерфейса
CommentNotificationProxy

```
@Component
public class CommentPushNotificationProxy
    implements CommentNotificationProxy {
    @Override
    public void sendComment (Comment comment) {
        System.out.println(
            "Sending push notification for
comment: "
            + comment.getText());
    }
}
```

Этот класс реализует
интерфейс
CommentNotificationProxy

Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException: **No qualifying bean of type** 'proxies.CommentNotificationProxy' **available: expected single matching bean but found 2:** commentPushNotificationProxy,emailCommentNotificationProxy

2.2. Choosing what to auto-wire from multiple implementations of an abstraction

2.2. Выбор одной из реализаций абстракции для автомонтажа

Внедрение реализации, используемой по умолчанию, с помощью аннотации

`@Primary`

```
@Component
@Primary
public class CommentPushNotificationProxy
    implements CommentNotificationProxy {
    @Override
    public void sendComment(Comment comment) {
        System.out.println(
            "Sending push notification for comment:
            "
            + comment.getText());
    }
}
```

Результат:

Storing comment: Demo comment
Sending push notification for comment: Demo comment

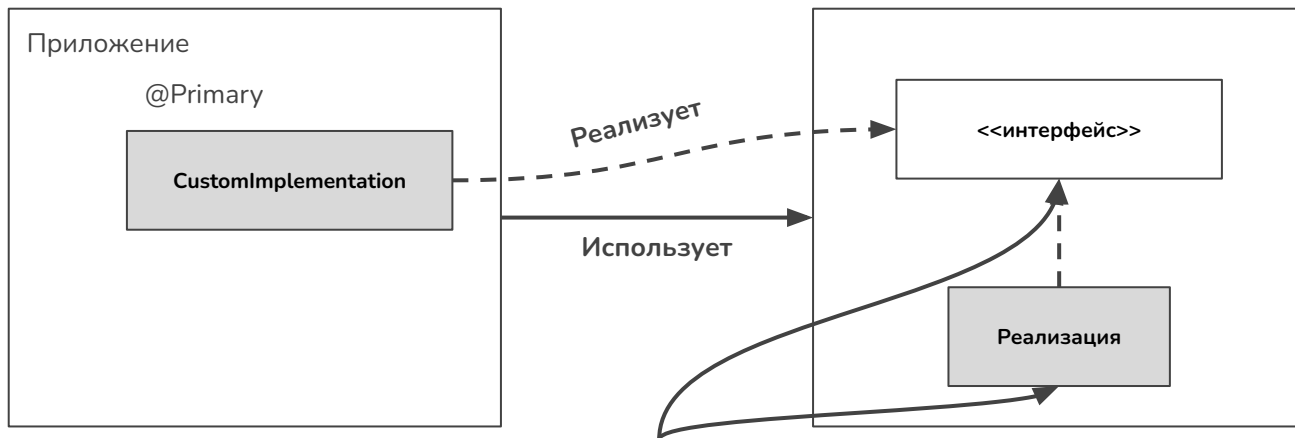
Аннотация `@Primary` отмечает реализацию, которая используется по умолчанию при внедрении зависимости

Spring внедряет новую реализацию, поскольку мы поместили ее аннотацией `@Primary`

2.2. Choosing what to auto-wire from multiple implementations of an abstraction

2.2. Выбор одной из реализаций абстракции для автомонтажа

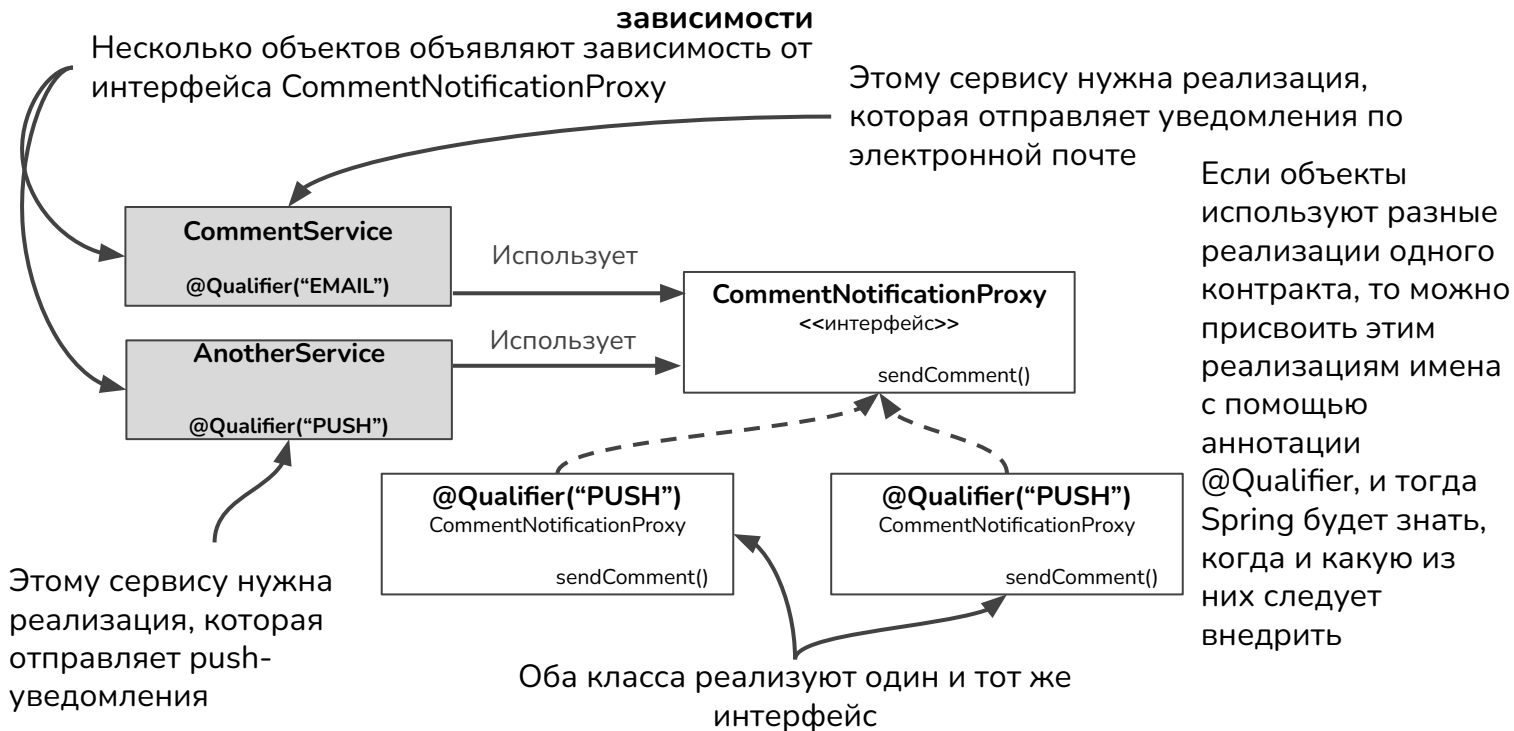
Иногда приходится использовать зависимости для интерфейсов, у которых уже есть реализации. И необходимо подключить аннотацию `@Primary`, чтобы ваша собственная реализация использовалась при DI по умолчанию. Тогда Spring будет знать, что нужно внедрять именно ее, а не ту, определенную зависимостью



В приложении есть зависимость. Она определяет интерфейс, а также его реализацию

2.2. Choosing what to auto-wire from multiple implementations of an abstraction

Создание именованной реализации с помощью аннотации `@Qualifier` при внедрении



2.2. Choosing what to auto-wire from multiple implementations of an abstraction

```
DBCommentRepository.java
5
6 @Repository
7 public class DBCommentRepository implements CommentRepository {
8     @Override
9     public void storeComment(Comment comment) {
10         System.out.println("Storing comment: " + comment.getText());
11     }
12 }
13
14

EmailCommentNotificationProxy.java
1 package com.mokhir.dev.Task.proxies;
2
3 import ...
4
5 @Component
6 @Qualifier("EMAIL")
7 public class EmailCommentNotificationProxy
8     implements CommentNotificationProxy {
9     @Override
10     public void sendComment(Comment comment) {
11         System.out.println("Sending notification for comment: "
12             + comment.getText());
13     }
14 }
15
16
17
18


CommentService.java
10 @Service
11 public class CommentService {
12
13     private final CommentRepository commentRepository;
14     private final CommentNotificationProxy commentNotificationProxy;
15
16     @Autowired
17     public CommentService(CommentRepository commentRepository,
18         @Qualifier("PUSH") CommentNotificationProxy commentNotificationProxy) {
19         this.commentRepository = commentRepository;
20         this.commentNotificationProxy = commentNotificationProxy;
21     }
22
23     public void publishComment(Comment comment) {
24         commentRepository.storeComment(comment);
25         commentNotificationProxy.sendComment(comment);
26     }
27 }

CommentPushNotificationProxy.java
6
7 @Component
8 @Qualifier("PUSH")
9 public class CommentPushNotificationProxy
10     implements CommentNotificationProxy {
11     @Override
12     public void sendComment(Comment comment) {
13         System.out.println(
14             "Sending push notification for comment: "
15             + comment.getText());
16     }
17 }
18

Run
TaskApplication
Console
C:\Users\moxir\j\jdk\openjdk-21.0.2\bin\java.exe ...
Storing comment: Demo comment
Sending push notification for comment: Demo comment
Process finished with exit code 0
```


2.2. Choosing what to auto-wire from multiple implementations of an abstraction

С помощью аннотации @Qualifier
присваиваем реализации имя PUSH



```
@Component
@Qualifier("PUSH")
public class
CommentPushNotificationProxy
    implements
CommentNotificationProxy {
    @Override
    public void sendComment (Comment
comment) {
        System.out.println(
            "Sending push
notification for comment: "
            +
comment.getText());
    }
}
```

С помощью аннотации @Qualifier
присваиваем реализации имя EMAIL



```
@Component
@Qualifier("EMAIL")
public class
EmailCommentNotificationProxy
    implements
CommentNotificationProxy {
    @Override
    public void sendComment (Comment
comment) {
        System.out.println("Sending
notification for comment: "
            + comment.getText());
    }
}
```

2.2. Choosing what to auto-wire from multiple implementations of an abstraction

```
@Component
public class CommentService {
    @Autowired
    private CommentRepository commentRepository;
    @Autowired
    private CommentNotificationProxy commentNotificationProxy;
    public CommentService (CommentRepository commentRepository ,
                           @Qualifier("PUSH") CommentNotificationProxy
commentNotificationProxy) {
        this.commentRepository = commentRepository;
        this.commentNotificationProxy = commentNotificationProxy;
    }
    public void publishComment (Comment comment) {
        commentRepository.storeComment (comment);
        commentRepository.sendComment (comment);}}
}
```

Каждый параметр, для которого
нужно использовать специальную
реализацию, сопровождается
аннотацией **@Qualifier**

**Обратите внимание: Spring
внедрил реализацию push-
сообщений**

Результат
Storing comment: Demo comment
Sending push notification for comment: Demo comment

3. Focusing on object responsibilities with stereotype annotations

3. Подробнее об обязанностях объектов со стереотипными аннотациями:

@Service, @Repository

С помощью аннотации @Repository мы показываем, что этот объект является компонентом, исполняющим обязанность репозитория

```
@Repository
public class DBCommentRepository implements CommentRepository {
    @Override
    public void storeComment (Comment comment) {
        System.out.println("Storing comment: " + comment.getText());
    }
}
```

3. Focusing on object responsibilities with stereotype annotations

@Service

public class CommentService {

private CommentRepository commentRepository;

private CommentNotificationProxy commentNotificationProxy;

@Autowired

public CommentService (CommentRepository commentRepository ,
@Qualifier ("PUSH") CommentNotificationProxy

commentNotificationProxy) {

this.commentRepository = commentRepository;

this.commentNotificationProxy = commentNotificationProxy;

}

public void publishComment (Comment comment) {

commentRepository.storeComment (comment);

commentNotificationProxy.sendComment (comment);

}

}

С помощью аннотации @Service мы показываем, что этот объект является компонентом, исполняющим обязанность сервиса

3. Focusing on object responsibilities with stereotype annotations

С помощью аннотации `@Service` мы отмечаем класс `CommentService` как компонент и четко обозначаем его обязанность



Использует

С помощью аннотации `@Repository` мы отмечаем класс `DBCommentRepository` как компонент и четко обозначаем его обязанность



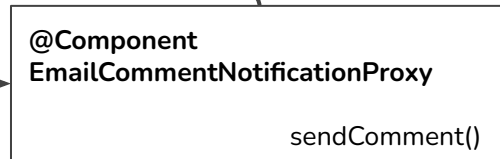
Реализует



Использует

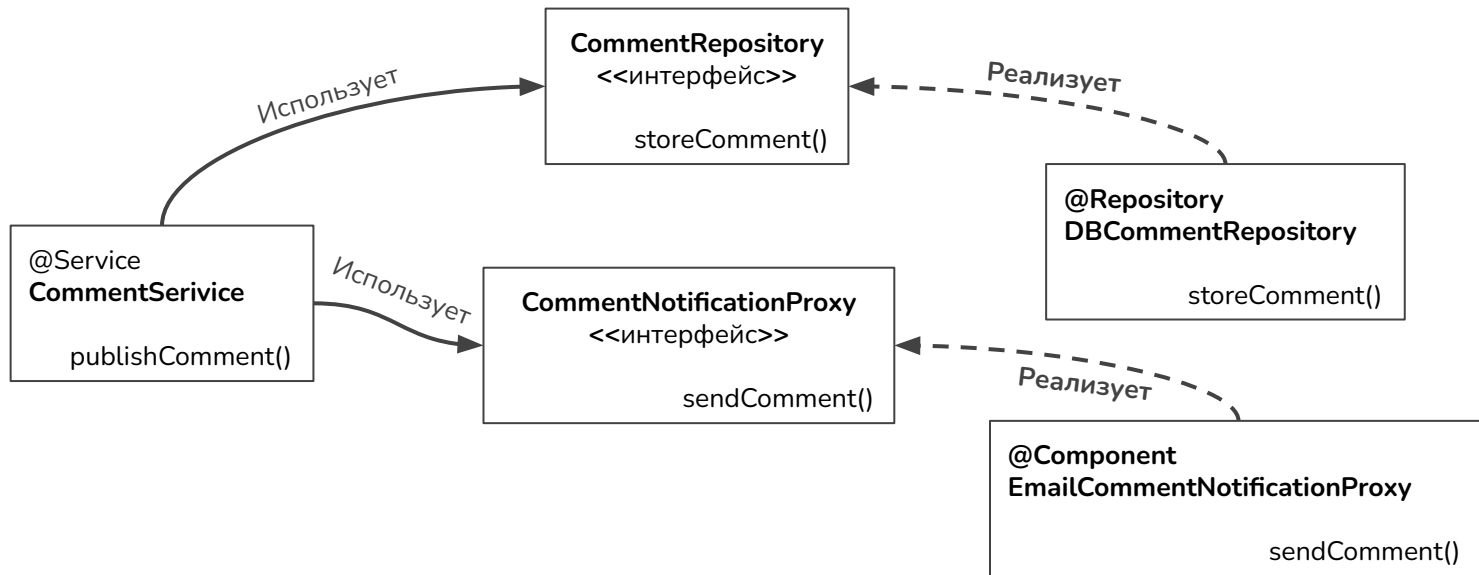


Реализует



Если в Spring нет отдельной аннотации для данной обязанности, продолжаем использовать `@Component`

3. Focusing on object responsibilities with stereotype annotations



Аннотации `@Service` и `@Repository` позволяют явно указать обязанности компонентов в структуре классов. Если в Spring нет специальной аннотации для данной обязанности, продолжаем использовать `@Component`



Conclusion (1/3)

- Разделение реализаций посредством абстракций — это хороший принцип проектирования классов. Он позволяет легко изменять реализации, не затрагивая при этом слишком много частей приложения. Благодаря этому приложение становится проще в расширении и поддержке.
- В Java для разделения реализаций действительно часто используются интерфейсы. Интерфейсы определяют контракты, которые классы должны реализовывать. Это означает, что класс, реализующий интерфейс, обязуется предоставить реализацию всех методов, определенных в этом интерфейсе. Использование интерфейсов помогает создавать гибкую и расширяемую архитектуру приложений, поскольку позволяет легко изменять поведение компонентов, подменяя одну реализацию интерфейса на другую.
- При использовании абстракции с внедрением зависимостей Spring знает, где искать бин, который ее реализует.



Conclusion (2/3)

- Если нужно создать экземпляр класса и поместить его в контекст Spring, то такой класс сопровождается стереотипной аннотацией. Стереотипные аннотации никогда не применяются к интерфейсам.
- Если в контексте Spring есть несколько бинов, представляющих собой разные реализации одной и той же абстракции, существует несколько способов обозначить для Spring, какой из этих бинов следует внедрить:
 - Отметить один из бинов как используемый по умолчанию с помощью аннотации `@Primary`;
 - Создать именованный бин с помощью аннотации `@Qualifier` и затем дать Spring инструкцию внедрить именно этот бин, указав его имя



Conclusion (3/3)

- Для компонентов с обязанностями сервиса вместо стереотипной аннотации `@Component` можно применить аннотацию `@Service`. Аналогичным образом для компонентов с обязанностью репозитория вместо `@Component` можно использовать аннотацию `@Repository`. Таким образом мы явно обозначаем обязанность компонента, благодаря чему структура классов становится более понятной и удобной для чтения.



Reference

- 1: [Spring Start Here](#)
- 2: [Spring Annotations](#)
- 3: [Spring MVC](#)
- 4: [Spring MVC tutorial](#)
- 5: [Spring Annotations tutorial](#)



Thank you!
Presented by
Moxirbek Maxkamov
(mokhirbek.makhkam@gmail.com)