

完成日期：2024/1/1

报告撰写人：全体成员

**软件设计模式**

**期末课程项目**

TONGJI SSE

**JiCode**

**——敏捷开发管理工具**

2150266 时天逸

2152590 王琳

2151396 张靖凯

2152034 吴杭

2150276 沈卓成

2154298 王颖

2154059 樊佳怡

2152216 蒋钰萱

2151641 王佳垚

2154016 沙木哈尔·阿吾森

目录

[一、 项目简介 2](#_Toc30537)

[二、 设计模式汇总表 3](#_Toc31301)

[三、 设计模式详述 4](#_Toc5303)

[3.1 工厂模式 4](#_Toc12767)

[3.2 建造者模式 8](#_Toc20701)

[3.3 适配器模式 12](#_Toc7346)

[3.4 原型模式 14](#_Toc23091)

[3.5 调停者模式 17](#_Toc11362)

[3.6 单例模式 21](#_Toc3250)

[3.7 责任链模式 23](#_Toc16987)

[3.8 外观模式 26](#_Toc19460)

[3.9 装饰器模式 29](#_Toc8231)

[3.10 策略模式 31](#_Toc14895)

[3.11脏标记模式 35](#_Toc12334)

[3.12代理模式 39](#_Toc11674)

[附录：项目分工说明与成员贡献 43](#_Toc21511)

## 

## 项目简介

软件研发过程管理，是一条复杂的管理链条，在当今的商业环境中，敏捷项目管理 方法已经成为了高效的项目管理方式之一。敏捷方法特征包括：强调小规模的可交付结 果为导向，采用迭代和增量的开放方式，鼓励与客户紧密联系以理解需求、获取反馈， 具有高度适应性和灵活性。因此，越来越多的团队和组织采用敏捷方法组织、协调和管 理项目。

JiCode是一个管理Scrum敏捷开发方法的平台，具有账号管理、项目管理、产品管理三个模块功能，确保工具提供敏捷项目管理的核心功能，如需求获取、迭代管理、团队协作等，帮助产研团队：

· 快速建立标准化、规范化研发管理工作流程，有节奏的持续交付价值；

· 轻松规划和应对需求变化，提高项目可预测性，降低风险；

· 提升项目信息透明度，协作更顺畅，过程统计和改进有据可依。

通过小步快跑的方式，能够有效帮助企业持续规划和交付，打破协作壁垒和信息孤岛，让项目管理更加轻松且高效。

## 设计模式汇总表

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 设计模式名称 | 涉及的文件 | 何处功能/函数体现 |
| 1 | 工厂模式 | BacklogitemFactory.java | class BacklogitemFactory |
| 2 | 建造者模式 | ComResponse.java | function build() |
| 3 | 适配器模式 | adpator文件夹 | 数据格式转化，服务调用。 |
| 4 | 原型模式 | ResponseCode.java  ComResponse.java | 在responseCode中预定义一些对象，需要的时候从ResponseCode中拷贝。 |
| 5 | 调停者模式 | Repository.java  RepositoryImpl.java | 定义抽象中介者，提供操作的抽象方法，然后定义具体中介者实现中介者，操作具体逻辑。 |
| 6 | 单例模式 | ComResponse.java | builder当中通过getInstance的方法实现创建类的实例化。 |
| 7 | 责任链模式 | ProductDev.domain/service/correlation文件夹 | 基类当中维护successor对象标识责任链当中的下一个处理者。 |
| 8 | 外观模式 | Application.java  ApplicationImpl.java | 客户端只需要与一个外观类交互，使得客户端只需要和一个外观类交互。 |
| 9 | 装饰器模式 | ComResponse.java | ResponseCode用于处理HTTP相关请求信息的类，这里进行包装，添加了data相关的处理。 |
| 10 | 策略模式 | ProductDev.domain/service/correlation文件夹 | 抽象策略给出所有具体策略类需要的接口，具体策略类实现了抽象策略定义的接口，并提供具体的行为。 |
| 11 | 脏标记模式 | RequirementAggregationm.java | 聚合整体更新数据库写优化 |
| 12 | 代理模式 | ProductDev.domain/service/correlation文件夹 | RelateServiceImpl类实现了对RelateStrategy的代理，在访问真实主题类之前执行了一些额外的逻辑来进行访问控制。 |

## 设计模式详述

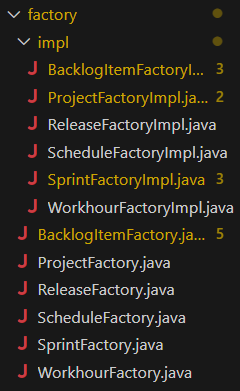
### 3.1 工厂模式

#### 3.1.1 实现描述

我们的项目Jicode是一个微服务的课程项目，使用了领域驱动设计（Domain-Driven Design, DDD）。在领域驱动设计中，工厂模式是一种常用的创建型设计模式，它负责创建和初始化领域模型中的聚合根和实体。工厂模式允许将对象的创建逻辑与使用逻辑分离，从而提供一种灵活的方式来创建复杂对象，同时保持客户端的简洁和领域的完整性。

在DDD中，领域模型是应用程序的核心，它反映了业务领域的概念和规则。领域模型中的聚合（Aggregates）是一组相关对象的集合，它们可以作为一个单元被访问和修改。聚合根（Aggregate Roots）是聚合的入口点，通常是不可变的，并且负责确保整个聚合的一致性。 所以在数据初始化方面，数据访问层的反向工程生成mapper提供的服务不能满足领域聚合初始化的需求，聚合中对其他聚合、实体等的引用也无法在简单的构造函数中完成，需要将整个聚合的初始化整合成一个完整的流程，封装在一个工厂中实现。另外由于领域的仓储实现为springboot中的service，对其的依赖注入不应该在一个需要手动实例化的实体当中实现，应经由springboot管理，所以在聚合中实现工厂方法模式并不能满足需求，另外实现一个工厂来处理初始化可以解决该问题。

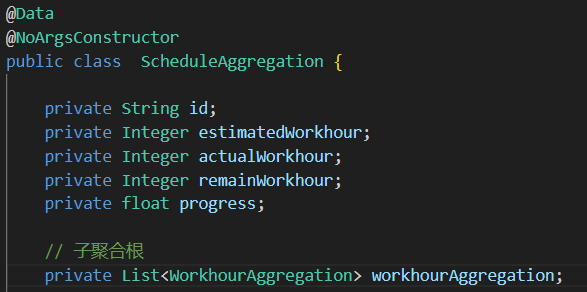
下面是本项目中的【项目开发-微服务】中的工厂接口及其实现的目录：



下面，以Schedule工厂为例，介绍工厂模式应用的步骤及其核心代码：

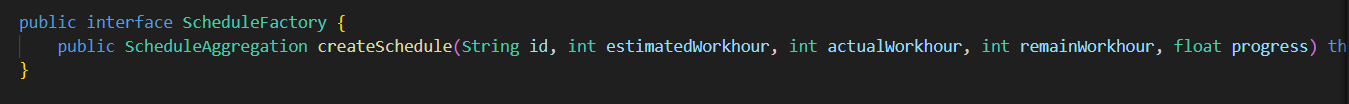
**1. 定义聚合根和实体：**

首先，定义领域模型中的聚合根和实体。聚合根通常包含对实体的引用，并负责聚合的生命周期和不变量。



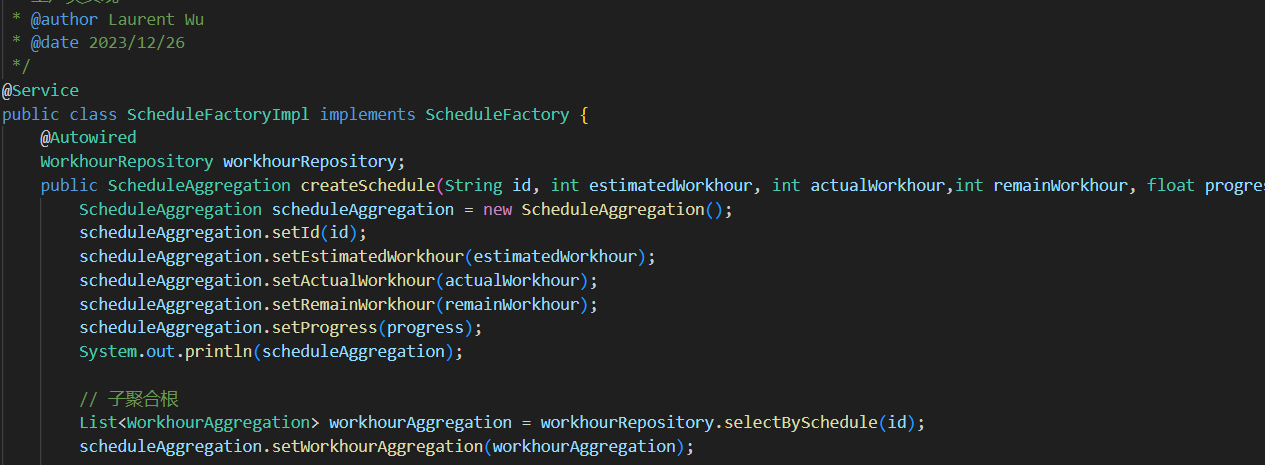
1. **提供统一的接口：**

为每个聚合根创建一个工厂类。工厂类通常提供一个统一的接口来创建聚合。另外类似schedule的创建需要调用其他聚合、工厂、仓储的方法，工厂向客户端隐藏具体的创建细节。



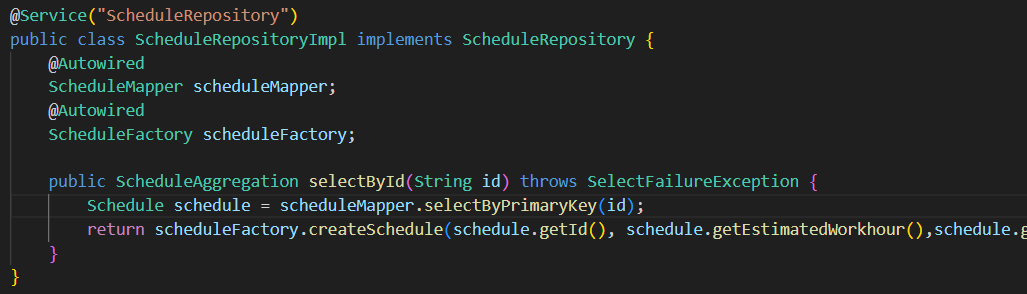
1. **创建工厂类并实现创建逻辑：**

这个工厂类负责创建和初始化聚合根，以及与之相关的实体和值对象。工厂类实现了创建逻辑，它可以根据传入的参数或依赖关系来创建和组装聚合。这可能包括调用其他工厂或服务来获取必要的依赖。



**4. 客户端使用工厂：**

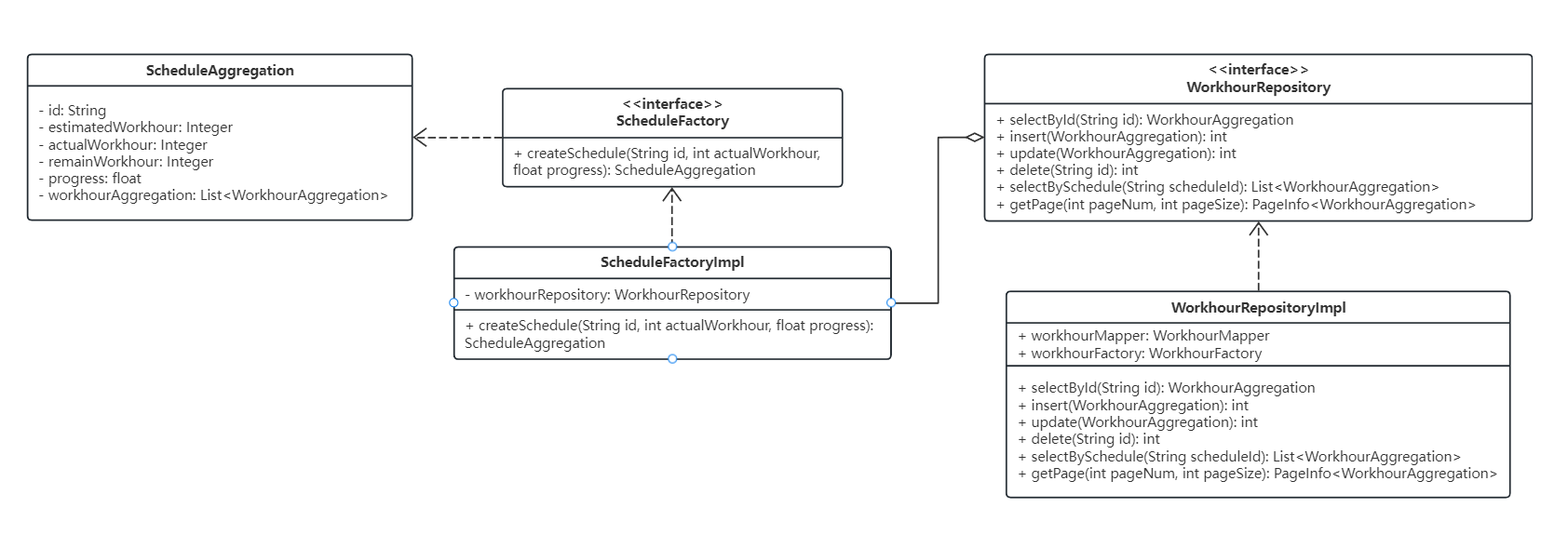
客户端代码通过工厂类来获取聚合根实例，而不是直接创建。这样可以确保客户端与领域模型之间的解耦。客户端只需要关注如何使用对象，而不需要了解对象的创建细节。下面是ScheduleRepository的实现类中的一个方法在返回聚合时使用工厂模式来创建Schedule聚合的实例。



在DDD中，工厂模式有助于保持领域模型的纯净和不可变性，同时提供了创建和初始化聚合的统一方式。这种模式有助于实现领域驱动设计的目标，即创建一个反映业务领域本质的模型，并确保应用程序的灵活性和可维护性。

领域驱动设计中所有聚合的初始化由工厂负责，工厂作为一个服务（使用springboot的service注解），他的依赖注入是聚合中的工厂方法模式不能实现的。

#### 3.1.2 UML建模



#### 3.1.3 代价分析

工厂模式优化了代码结构和扩展性，使得添加新产品类成为一件简易无碍的任务并显著地降低了代码耦合度。然而，这通常以增加系统复杂性为代价，可能导致初始的开发和后续维护成本的上升。该模式的实施需要额外的设计工作和对现有代码的重构，同时可能引入一些性能负担，尽管在大多数情形下这种负担几乎可以忽略不计。总体而言，工厂模式的引入提供了长远的益处，特别是在动态环境中或者需要频繁地引入新类型对象时。在协调初期的时间与资源投入与长期的系统灵活性和可维护性之间，工厂模式能够作为一个承前启后的桥梁。

在本项目的领域驱动设计中的应用具有以下优缺点：

**优点：**

1. 封装性：工厂模式通过将对象的创建逻辑与使用逻辑分离，提高了领域的封装性。客户端只需要关注如何使用对象，而不需要了解对象的创建细节。

2. 可维护性：当领域模型发生变化时，只需要修改工厂类的实现，而不需要修改客户端代码。这有助于减少维护成本，并提高代码的可维护性。

3. 灵活性：工厂模式允许在不修改现有代码的情况下，通过替换工厂类或修改工厂逻辑来引入新的领域模型或实现。这为应用程序提供了很大的灵活性。

4. 一致性：工厂模式确保了领域模型的一致性。通过工厂创建的聚合根和实体遵循相同的规则和约定，这有助于保持整个领域模型的一致性。

**缺点：**

1. 复杂性：对于复杂的领域模型，工厂模式可能会引入额外的复杂性。创建和管理工厂类本身可能需要大量的代码和设计工作。

2. 性能问题：如果工厂模式的使用不当，可能会导致性能问题。例如，频繁地创建和销毁对象可能会对性能产生负面影响。

3. 过度设计：在某些情况下，工厂模式可能会导致过度设计。如果领域模型相对简单，使用工厂模式可能会增加不必要的复杂性。

4. 依赖管理：工厂模式可能需要管理大量的依赖关系。这可能导致代码变得难以理解和维护，特别是当领域模型中的依赖关系变得复杂时。

5. 难以追踪：在大型项目中，工厂模式可能导致对象的创建和生命周期管理变得难以追踪。这可能会对调试和性能优化造成困难。

6. 运行时性能开销：使用工厂模式会引入额外的运行时性能开销。创建聚合对象需要通过工厂类进行实例化，会涉及额外的方法调用和对象创建的过程。这可能会在一定程度上影响性能，尤其是在频繁创建对象的场景中。

### 3.2 建造者模式

#### 3.2.1 实现描述

建造者模式（Builder Pattern）是一种创建型设计模式，它用于构建复杂对象。这种模式将对象的构造与表示进行分离，使得相同的构造过程可以产生不同的表示。建造者模式通常用于那些需要大量参数才能创建的对象，这些参数可能随着时间的推移而变化。

在建造者模式中，有一个称为“建造者”的接口，它定义了构建复杂对象的方法。建造者接口通常包含一个名为“build”的方法，该方法负责返回最终构建的对象。建造者接口的实现类（ConcreteBuilder）会提供具体的构建逻辑，并使用“Builder”模式来逐步构建对象。

在HTTP调用中，建造者模式可以用于封装HTTP响应。例如，可以将HTTP响应封装在一个名为“Comeresponse”的模板类中，该类的构造由一个名为“Builder”的类负责。这种模式允许客户端以统一的方式构建HTTP响应，而不需要关心具体的构建细节。

具体实现如下：

1. ComResponse类作为产品类（Product Class），表示要构建的对象。
2. 在ComResponse类中自定义了一个内部类Builder，它作为建造者类（Builder Class）,提供了一系列的方法用于设置 ComResponse 对象的属性值，并通过 build 方法返回构建好的对象。



1. 编写了一些静态工厂方法，利用了链式调用的方式设置了对象的属性，并返回了一个构建好的对象。



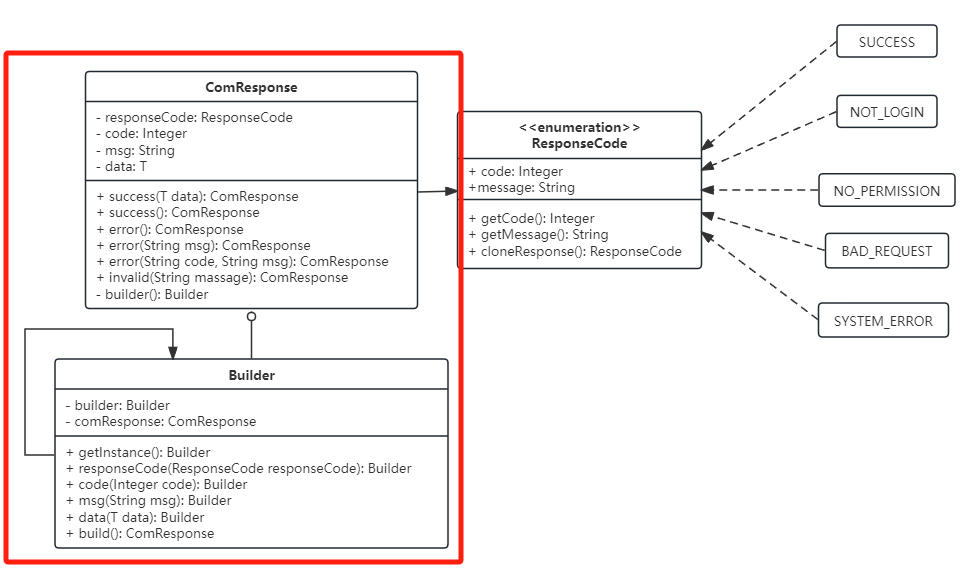
1. **与其他模式结合使用。**
   1. 其中Builder 类中的 getInstance() 方法使用了单例模式（Singleton Pattern），确保只有一个 Builder 实例存在。



* 1. 与装饰器模式结合使用。



#### 3.2.2 UML建模



#### 3.2.3 代价分析

建造者模式（Builder Pattern）的优缺点如下：

**优点：**

1. 封装性：建造者模式将对象的构建逻辑与表示分离，使得客户端只需要关注如何使用对象，而不需要了解对象的构建细节。

2. 可维护性：当对象的构建逻辑发生变化时，只需要修改建造者类的实现，而不需要修改客户端代码。这有助于减少维护成本，并提高代码的可维护性。

3. 灵活性：建造者模式允许在不修改现有代码的情况下，通过替换建造者类或修改建造逻辑来引入新的对象或实现。这为应用程序提供了很大的灵活性。

4. 一致性：建造者模式确保了对象构建的一致性。通过建造者创建的对象遵循相同的规则和约定，这有助于保持整个对象模型的一致性。

5. 可测试性：由于建造者模式将对象的构建与使用分离，可以更容易地对对象进行单元测试和集成测试。

**缺点：**

1. 复杂性：对于复杂的对象，建造者模式可能会引入额外的复杂性。创建和管理建造者类本身可能需要大量的代码和设计工作。

2. 性能问题：如果建造者模式的使用不当，可能会导致性能问题。例如，频繁地创建和销毁对象可能会对性能产生负面影响。

3. 过度设计：在某些情况下，建造者模式可能会导致过度设计。如果对象模型相对简单，使用建造者模式可能会增加不必要的复杂性。

4. 依赖管理：建造者模式可能需要管理大量的依赖关系。这可能导致代码变得难以理解和维护，特别是当对象模型中的依赖关系变得复杂时。

5. 难以追踪：在大型项目中，建造者模式可能导致对象的创建和生命周期管理变得难以追踪。这可能会对调试和性能优化造成困难。

总的来说，建造者模式在提高领域模型的封装性、可维护性和灵活性方面具有优势，但也可能引入额外的复杂性和性能问题。在使用建造者模式时，需要权衡其带来的好处和潜在的缺点，并根据具体的项目需求和环境来决定是否采用。

### 3.3 适配器模式

#### 3.3.1 实现描述

在六边形架构（Hexagonal Architecture）中，适配器层（Adapters）是连接应用程序的核心业务逻辑与外部系统或基础设施的关键部分。这种架构风格由Alistair Cockburn在2005年提出，它强调将应用程序的业务逻辑与外部依赖解耦，以便应用程序可以更容易地适应变化和进行测试。

六边形架构中的适配器层主要包括以下两种类型的适配器：

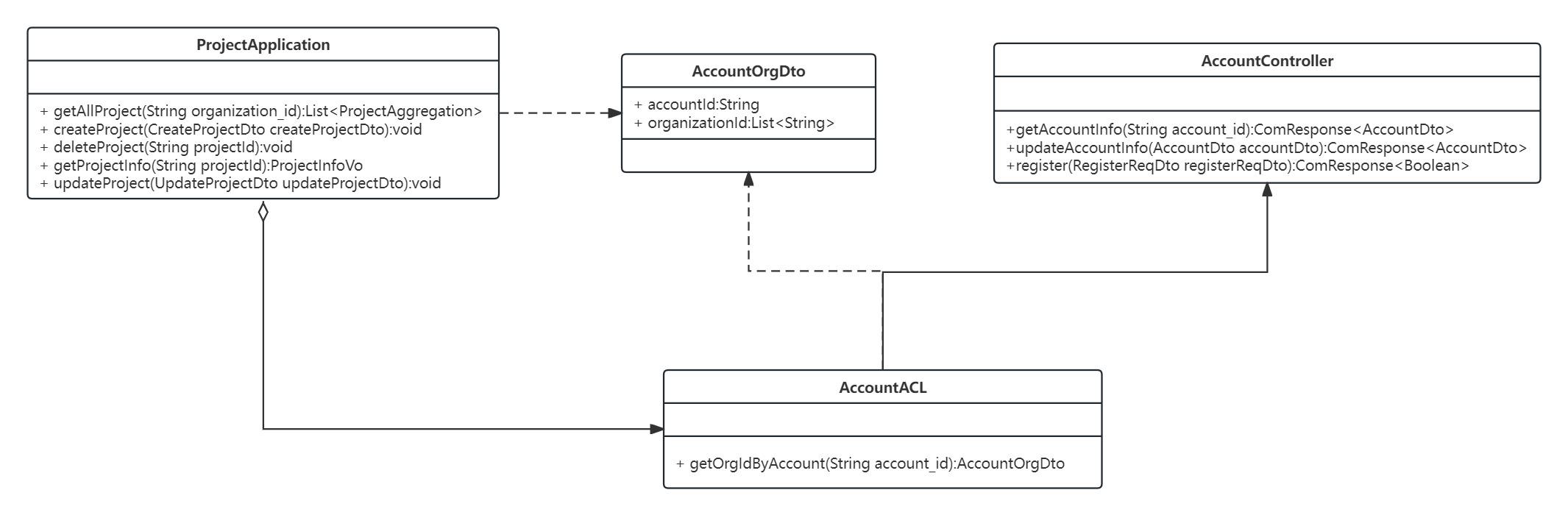
1. 适配器（Driving Adapters）： 主适配器负责接收来自外部系统的请求，并将其转换为应用程序内部可以理解的形式。这些适配器通常实现了一组标准的接口，用于与外部系统（如用户界面、消息队列、数据库等）进行通信。主适配器可以看作是外部系统到应用程序的“入口”。
2. 次适配器（Driven Adapters）： 次适配器负责将应用程序内部的响应或状态更新转换为外部系统可以理解的形式。这些适配器通常实现了应用程序内部定义的接口，用于将数据持久化到数据库、发布消息到消息队列或响应外部系统的请求。次适配器可以看作是应用程序到外部系统的“出口”。

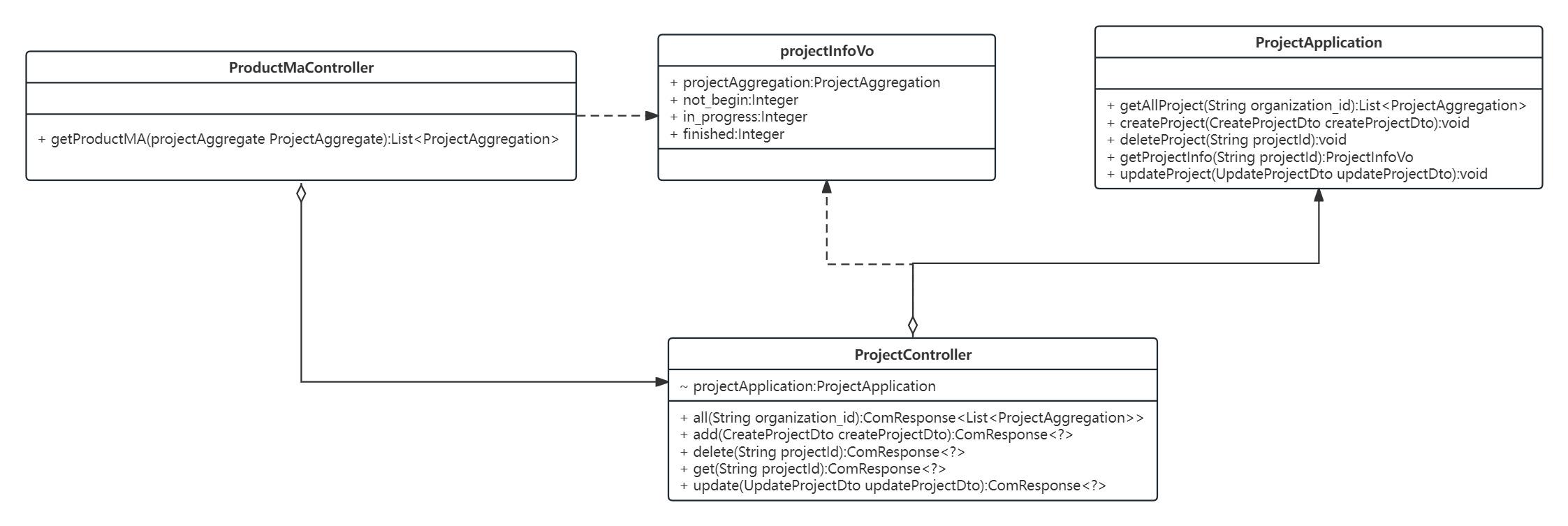
在本例中，六边形架构（端口适配器模式）适配器层体现：

1. http调用（controller）：将application所提供的应用服务封装，controller作为适配器将application返回的dto等业务对象转换成vo展示给前端或者开放给其他微服务调用。
2. anticorruption layer：封装本地上下文对外部接口的调用，把外部接口返回的对象转换为本地上下文的entity，再从六边形架构外层向内层依次传入给领域层业务逻辑实现使用。
3. 数据访问：把sql语句使用orm封装成各个mapper，便于领域层的仓储调用直接读取数据表转换的业务对象。

在六边形架构中，适配器层的作用是提供一种机制，使得业务逻辑层（Domain）与外部系统之间的交互变得尽可能简单和一致。通过定义清晰的接口和实现适配器，可以确保业务逻辑层不直接依赖于任何外部系统或技术，从而提高了应用程序的可测试性、可维护性和可扩展性。

#### 3.3.2 UML建模



3.3.3 代价分析

在六边形架构的端口适配器当中，适配器层负责系统内外部的交互，将系统内部、外部很好的解耦，其中主要三个方面使用适配器模式实现。

1. http调用（controller）：将application所提供的应用服务封装，controller作为适配器将application返回的dto等业务对象转换成vo展示给前端或者开放给其他微服务调用。
2. anticorruption layer：封装本地上下文对外部接口的调用，把外部接口返回的对象转换为本地上下文的entity，再从六边形架构外层向内层依次传入给领域层业务逻辑实现使用。
3. 数据访问：把sql语句使用orm封装成各个mapper，便于领域层的仓储调用直接读取数据

**缺点：**

1. 开发成本：适配器设计和代码编写需要特定的专业知识，可能导致初期人力成本增加。实现各种外部系统或基础设施的接口可能因其复杂性而成本较高。

2. 维护和更新成本：随着外部系统更新或内部需求变更，适配器需要持续维护和更新。维护多个适配器的兼容性可能导致成本累积。

3. 性能开销：适配器额外的处理层可能引入延迟，特别是在高频率调用的场景中。数据转换和校验逻辑可能占用计算资源。

4. 测试成本：每个适配器都需要经过严格的测试以保证正确性，这可能会消耗大量的测试资源。随着外部系统的升级，需要重新进行兼容性和集成测试。

5. 知识和技术学习成本：团队需要培训以理解适配器模式及其在六边形架构中的应用。适配器的开发可能需要特定的技术栈知识。

6. 集成复杂性：在现有系统中集成新适配器可能会面临技术上的挑战和阻力。

**优点：**

1. 业务逻辑与外部系统的解耦：适配器使得核心逻辑与IO操作解耦，业务逻辑更加纯粹。隔离变化，当外部系统变化时，只需修改对应的适配器而不影响内部逻辑。
2. 可测试性提升：独立的适配器面向接口编程，易于模拟和测试。外部系统的变动不会影响到业务逻辑层的测试。
3. 扩展性和灵活性：为新的接入或第三方系统提供服务时，添加新适配器更为方便。系统能更好地适配未来需求的变化。
4. 提高系统稳定性：适配器提供一层缓冲，将外部系统的异常隔离。适配器可以实现重试机制、错误处理等，增强系统鲁棒性。
5. 复用性：标准化的适配器可以在相似的场景或不同项目中复用，提高开发效率。对于常见的基础设施，如数据库或消息队列，标准适配器减少了重复工作。

### 3.4 原型模式

#### 3.4.1 实现描述

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在软件系统中,有时候需要多次创建某一类型的对象，为了简化创建过程,可以只创建一个对象,然后再通过克隆的方式复制出多个相同的对象,这就是原型模式的设计思想。

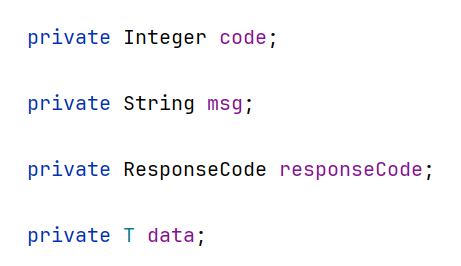
原型模式的基本工作原理是通过将一个原型对象传给那个要发动创建的对象,这个要发动创建的对象通过请求原型对象复制原型自己来实现创建过程。

具体如下：

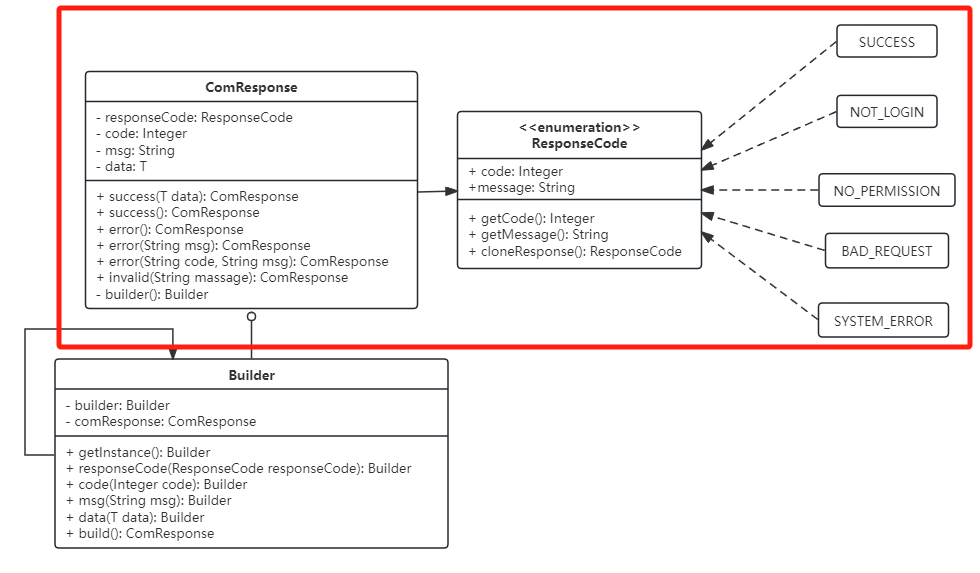
1. 在 ResponseCode 类中预先定义了一些对象。



1. 在 ComResponse 类中，每次需要时，从ResponseCode 类中拷贝。主要是实现只读的，预定好的常量，便于封装http请求的返回。



#### 3.4.2 UML建模



#### 3.4.3 代价分析

ResponseCode存在五种常规返回情况，将枚举类预先定义五种常量，定义clone方法进行深拷贝，http请求返回时在ComResponse的builder中使用预定义的常量原型clone并返回，可以加快ResponseCode的生成速度。

**缺点：**

1. 实现成本：设计合理的原型接口和实现克隆/复制逻辑需要时间和精力。

复制复杂对象或对象图可能需要深度拷贝，逻辑可能复杂。

1. 内存使用：如果不当地使用，原型模式可能导致大量不必要的副本，占用过多内存。
2. 维护成本：当原型对象的结构有变化时，克隆方法也需要相应的修改。

对象复制后保持依赖关系一致性可能导致维护上的困难。

1. 性能考虑：深拷贝大型或复杂对象结构时可能会导致性能问题。
2. 代码复杂性：管理原型注册库（如果使用）会增加代码复杂度。引入了一个新的抽象层次，增加了学习和理解的难度。
3. 非直观行为：使用不当可能会导致对象之间共享状态，引发难以追踪的错误。

**优点：**

1. 性能优化：在需要重复创建类似对象时，克隆可能比重新创建性能更高。能有效地复制对象状态，减少对象创建的开销。
2. 抽象层次：客户端代码可以操作抽象接口，而不必关心具体的类。提高了代码的可复用性。
3. 动态性：可以在运行时通过克隆生成新的对象，增加系统的动态性和灵活性。有助于减少与具体类的耦合。
4. 易于实现撤销操作：通过保存对象状态的克隆，可简化撤销或恢复的实现。

### 3.5 调停者模式

#### 3.5.1 实现描述

调停者模式（Mediator Pattern）是一种设计模式，用于降低多个对象之间的耦合，通过引入一个中介对象来封装对象之间的交互。在调停者模式中，对象之间的直接依赖关系被转换为对中介对象的依赖，从而使得对象之间的交互变得更加松散和可管理。

在领域驱动设计（Domain-Driven Design, DDD）中，仓储（Repository）是一个用于封装对数据库（或任何其他持久化存储）的操作的层。仓储的目的是提供一种与领域模型（Domain Model）紧密相连的数据访问方式，同时隐藏数据访问的细节，使得领域逻辑与数据存储的物理实现解耦。

在本项目的领域驱动设计中，仓储可以体现调停者模式，具体表现在以下几个方面，并给出相关代码：

1. **封装数据访问：**

仓储作为中介层，封装了领域模型与数据库之间的交互。它提供了一个抽象的接口来执行数据访问操作，如增加、删除、更新和查询实体。这样，领域模型不需要直接与数据库交互，而是通过仓储来间接访问数据。

1. **解耦领域逻辑与数据存储：**

通过使用仓储，领域逻辑与数据存储的物理实现被解耦。这样，领域模型可以专注于业务逻辑的实现，而不必关心数据是如何被持久化的。

1. **统一的数据访问方式：**

仓储提供了一种统一的数据访问方式，无论数据最终是如何存储的（例如，关系数据库、NoSQL数据库、内存中的数据结构等）。这有助于在领域模型中保持一致的数据访问抽象，从而简化了代码的编写和维护。

1. **支持事务操作：**

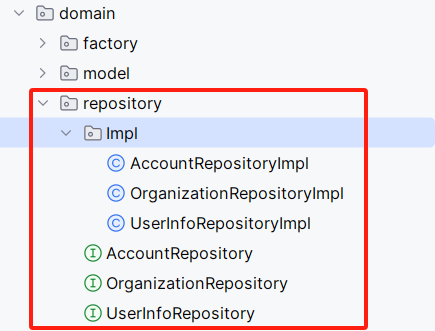
仓储通常支持事务操作，这有助于确保领域模型中的操作要么完全成功，要么在遇到错误时完全回滚，保持数据的一致性。

1. **可替换的数据存储实现：**

由于仓储封装了数据访问，因此可以很容易地替换底层的数据存储实现，而不会影响到领域模型和业务逻辑。这种替换可以是由于技术栈的变更，也可以是为了优化性能或实现其他业务需求。

总之，在领域驱动设计中，仓储通过封装数据访问和提供事务支持，体现了调停者模式的精神，即通过引入中介层来降低耦合，封装交互，并允许更灵活地管理和扩展数据存储的实现。

下面是本项目中的【账号管理-微服务】中的仓储及其实现的目录：



下面，以AccountRepository账号仓储为例，介绍调停者模式应用的步骤及其核心代码：

* 首先定义**抽象中介者（Mediator）**角色——**账号仓储类AccountRepository**，它是调停者的接口，提供了对数据库实体操作的抽象方法。 如根据账号ID查找账号、插入账号、更新账号信息和删除账号等方法。



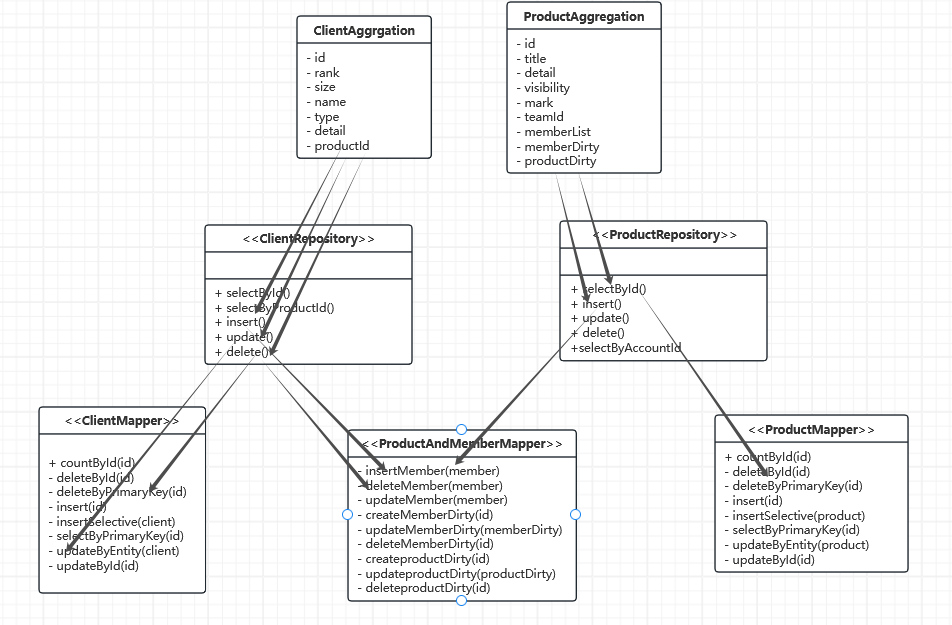
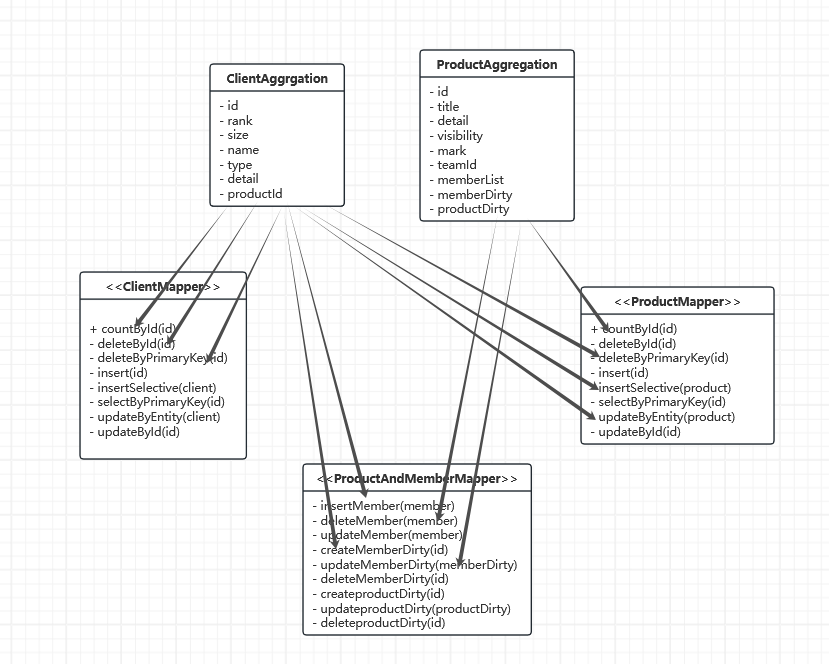
* 其次**具体中介者（ConcreteMediator）角色——账号仓储实现类AccountRepositoryImpl**：实现中介者接口，实现了对数据库实体操作的具体逻辑。



在本例子中，AccountRepositoryImpl就充当了中介者，将客户端（应用层）和数据库实体之间的交互进行了解耦，客户端只需要通过调用AccountRepository接口定义的方法来进行数据库实体的操作，而无需直接与底层数据库进行交互。

#### 3.5.2 UML建模

下图左侧是本项目使用了调停者模式的类图示意，右侧是假设不是用调停者模式的类图示意。显然，使用调停者模式可以将客户端和数据库实体之间的交互进行了解耦。

#### 3.5.3 代价分析

在领域驱动设计的仓储中，其封装了整个聚合的持久化操作。如果不使用仓储充当数据访问层的中介，聚合的工厂、领域服务等都需要直接调用各个mapper，对一个聚合中需要修改数据库多张表的情况非常繁琐，也会产生大量重复代码；同时在其他聚合引用到该聚合时，其他聚合的持久化需要调用该聚合的对应mapper，可能也会出现聚合间的互相调用，仓储作为一个中介者的设计很好的解决了繁琐的调用流程。

**优点：**

1. 解耦：调停者模式可以将不同的仓储实现解耦，使得它们不需要直接相互通信。这样可以降低各个仓储之间的耦合度，提高系统的模块化。

2. 统一接口：通过调停者模式，可以为不同的仓储提供一个统一的接口，这样可以简化客户端代码，并确保所有的持久化操作都遵循相同的模式。

3. 扩展性：当需要引入新的仓储实现时，只需要扩展调停者模式，而不需要修改现有的仓储代码。这有助于提高系统的扩展性。

4. 可维护性：由于仓储之间的交互通过调停者模式进行，因此可以更容易地管理和维护这些交互，有助于减少维护成本。

**缺点：**

1. 性能开销：每次持久化操作都需要通过调停者模式进行转发，这可能会导致额外的性能开销。

2. 复杂性：调停者模式可能会增加系统的复杂性，尤其是在处理多种仓储实现和复杂的交互时。

3. 开发成本：编写和维护调停者模式需要额外的开发资源，这可能会增加项目的总体开发成本。

4. 抽象泄漏：如果调停者模式的实现不够健壮，可能会导致仓储实现的特定细节泄漏到业务逻辑中，从而违反了解耦的原则。

5. 集成挑战：在集成新的仓储实现时，可能会遇到与调停者模式不兼容的接口或协议，这需要额外的开发和调整。

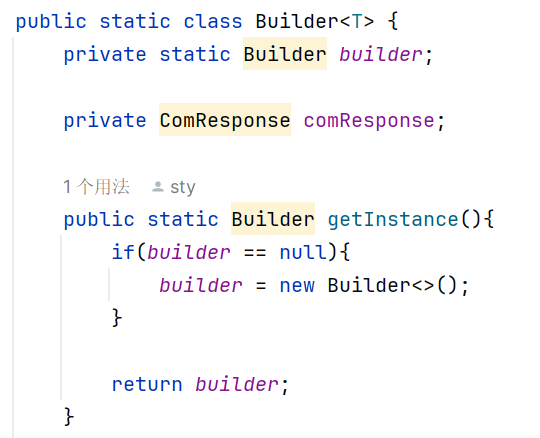
### 3.6 单例模式

#### 3.6.1 实现描述

单例模式，属于创建类型的一种常用的软件设计模式。通过单例模式的方法创建的类在当前进程中只有一个实例。它采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法(静态方法)。

实现方法：

1. **懒汉式**（在需要调用的时候再创建类的例化）。在 ComResponse 类的 Builder 类中，通过 getInstance 方法来实现在需要调用的时候再创建类的实例化。此处将单例模式与建造者模式结合使用。



1. **依赖注入。**在一个类上面标注@Service或者@Controller或@Component或@Repository注解后，容器启动时，Spring组件扫描时就会发现它，并且会直接将其创建好变成Spring应用上下文中bean。在需要使用时通过@Autowired注解把容器创建好的bean从容器获取过来赋予给使用类的成员属性。



#### 3.6.2 代价分析

类似builder的服务类可以使用依赖注入等注解来完成，使用单例模式实现同样功能，管理服务类的单一示例。

**优点：**

1. 管理单一性：确保只有一个实例，这减少了手动创建和管理单例的复杂性。

2. 配置简化：简化了代码中的依赖管理。

3. 依赖解耦：依赖注入允许将依赖关系从组件中解耦，使得组件更专注于业务逻辑，而不是依赖的管理。

**缺点：**

1. 抽象泄漏：如果单例模式的实现不够健壮，可能会导致依赖实现的特定细节泄漏到业务逻辑中，从而违反了解耦的原则。

2. 集成挑战：在集成新的依赖时，可能会遇到与单例模式不兼容的接口或协议，这需要额外的开发和调整。

总的来说，在Spring Boot中通过依赖注入实现的单例模式在管理单一性、配置简化、依赖解耦和测试方便方面具有显著优势，但也可能引入额外的性能开销和复杂性。在设计系统时，需要仔细权衡这些优缺点，并根据实际需求来决定是否采用单例模式。

### 3.7 责任链模式

#### 3.7.1 实现描述

责任链模式（Chain of Responsibility Pattern）是一种行为型设计模式，它定义了对象处理请求的顺序。在责任链模式中，每个处理者都有责任链的下一个处理者，它们按照一定的顺序依次处理请求，每个对象都有机会处理请求，或者将请求沿着链传递给下一个对象。

职责链模式主要包含以下角色:

**抽象处理者（Abstract Handler）角色**：定义一个处理请求的接口，包含抽象处理方法和一个后继连接。

**具体处理者（Concrete Handler）角色：**实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。

**客户类（Client）角色：**创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程。

在本项目的领域服务“关联”中，由于关联的对象有2个，因此在【产品开发-微服务】中实现了双层嵌套的责任链模式。每个RelateStrategy都具有一个successor字段，用于指向下一个处理者。当请求到达一个处理者时，该处理者会检查是否满足自己的处理条件，如果满足则处理请求，否则将请求传递给下一个处理者。在这个场景中，RelateStrategy是责任链的基础，而BacklogRelateStrategy、SprintRelateStrategy和ReleaseRelateStrategy是责任链中的不同层级。其中，RelateStrategy是第一层责任链，ReleaseRelateStrategy是第二层责任链，而它们共同组成了一个嵌套的责任链。 使得责任链中的每个处理者具有多个策略，当请求到达RelateStrategy时，它会根据特定的条件选择下一个处理者，即BacklogRelateStrategy、SprintRelateStrategy或ReleaseRelateStrategy中的一个。然后，当请求到达ReleaseRelateStrategy时，它会再次根据特定的条件选择下一个处理者，即BacklogReleaseStrategy或SprintReleaseStrategy中的一个。 这样可以根据不同的情况选择不同的策略，具体分析如下：

1. **抽象处理者角色：**

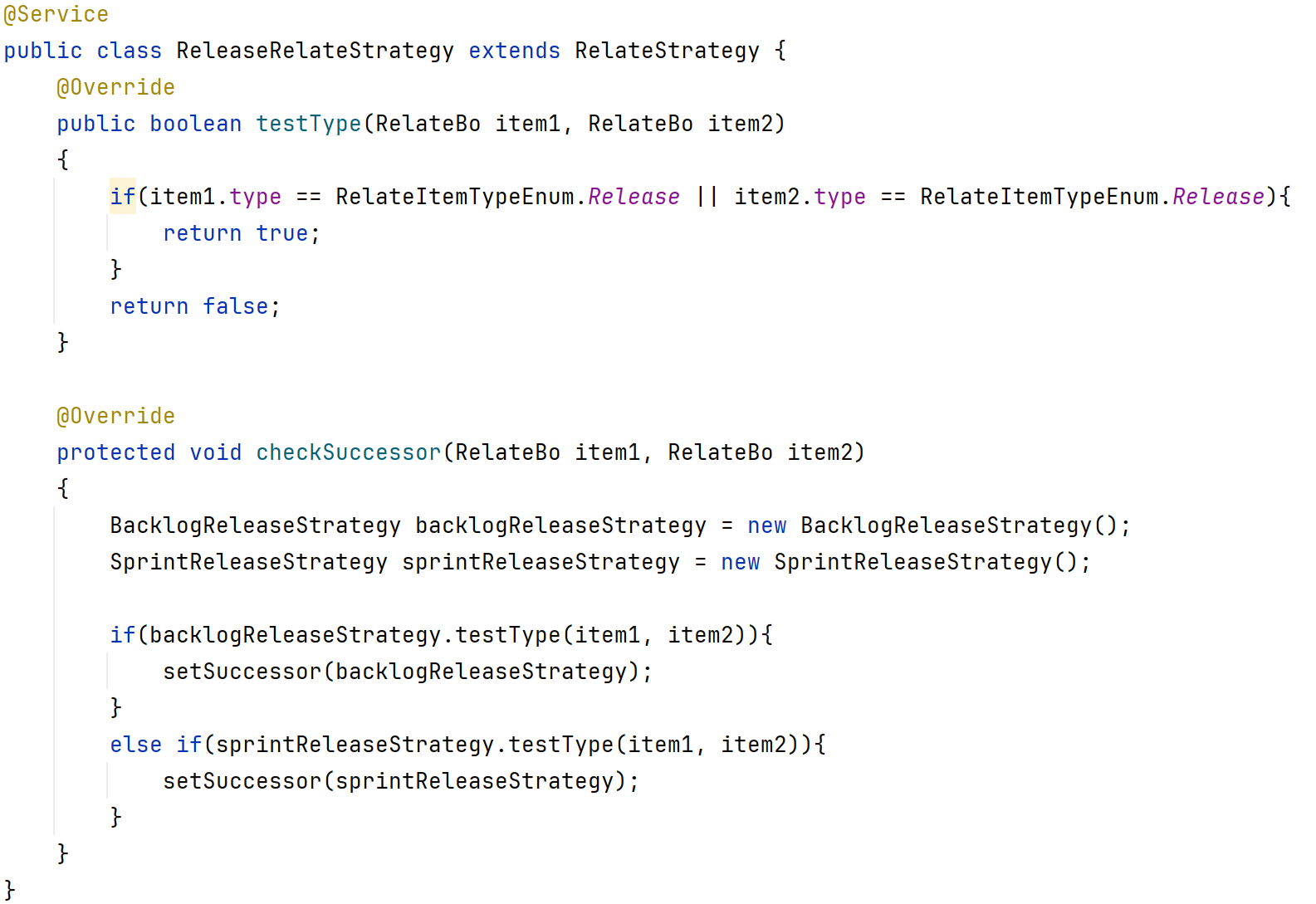
在本项目中基类RelateStrategy维护了一个successor字段表示责任链中的下一个处理者；checkSuccessor方法根据不同的类型处理下一个处理者，形成责任链；relate方法则通过调用checkSuccessor方法来检查下一个处理者，并将请求委托给下一个处理者进行处理。



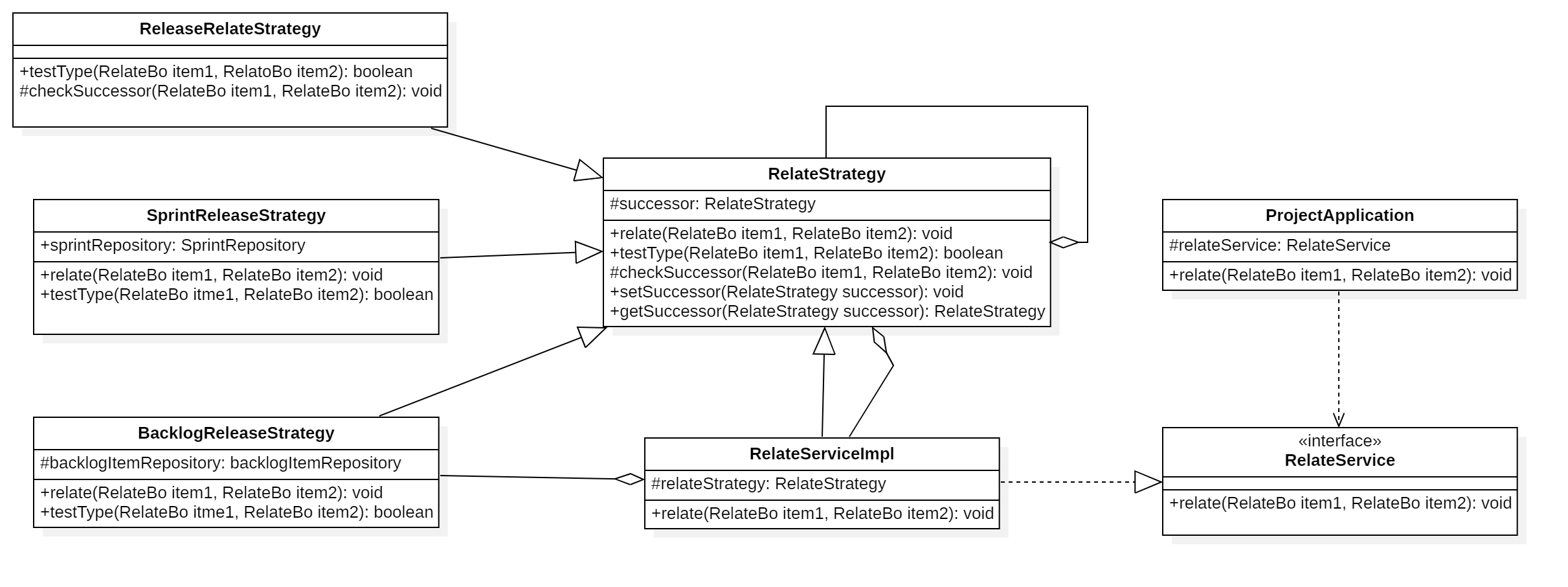
1. **具体处理者角色：**

实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。在本项目中，相对于基类RelateStrategy而言，具体处理者角色有BacklogRelateStrategy、ReleaseRlateStrategy、SprintRelateStrategy等类，它们都继承自抽象处理角色类RelateStrategy，分别用来实现与Backlog、Release、Sprint有关的关联操作；而对于它们下一层的处理者而言，它们又作为抽象处理者角色，将根据testType的结果来检查它的下一个处理者。

以ReleaseRelateStrategy类为例，它的checkSuccessor方法首先检查待关联的两个对象是否有Release类型的对象，若有则分别调用BacklogReleaseStrategy和SprintReleaseStrategy两个策略类的testType方法来判断是否能够处理给定的类型，如果backlogReleaseStrategy能够处理类型，则将它设置为下一个处理者，即调用setSuccessor(backlogReleaseStrategy)，否则将sprintReleaseStrategy设置为下一个处理者，如下图代码所示：



#### 3.7.2 UML建模



#### 3.7.3 代价分析

关联业务逻辑中有大量的两两关联，如果直接实现在service当中会出现大量的if-else语句，使用责任链模式自动在各个strategy当中判断后继处理策略，消除了大量的条件判断，大大提高了系统的可维护性、扩展性。当出现新的关联需求的时候只需要在现有的责任链当中加入新策略的责任和后继策略判断即可。

**优点：**

1. 解耦：责任链模式将请求的发送者和接收者解耦，请求的发送者不需要知道具体哪个对象会处理请求，只需要将请求发送到链中的第一个处理者即可。

2. 扩展性：责任链模式支持在运行时动态地添加或删除处理者，不需要修改现有代码，只需在链中添加或移除处理者对象。

3. 分配请求：请求可以在处理者之间按照预定的顺序传递，每个处理者都可以根据自己的能力决定是否处理请求，或者将请求传递给链中的下一个处理者。

4. 单一职责原则：每个处理者对象通常只负责一项特定的任务，这样可以保持处理者的单一职责，便于维护和测试。

5. 灵活的请求处理：责任链模式允许请求在处理者之间灵活传递，处理者可以根据请求的类型、内容或其他条件来决定是否处理请求。

**缺点：**

1. 性能开销：请求在处理者之间传递可能会导致性能开销，尤其是在处理者数量较多或请求频繁的情况下。

2. 复杂性：责任链模式的实现相对复杂，特别是在需要管理多个处理者对象时。此外，如果处理者之间的传递逻辑没有得到恰当管理，可能会导致逻辑错误。

3. 违反开闭原则：在某些责任链模式的实现中，为了添加新的处理者，可能需要修改现有代码，这违反了开闭原则中对扩展开放但对修改封闭的要求。

4. 过度使用：如果在不必要的场景下使用责任链模式，可能会增加系统的复杂性，导致过度设计。

5. 请求丢失：在责任链模式中，请求可能会在没有被任何处理者处理的情况下消失，这可能会导致请求丢失。

### 3.8 外观模式

#### 3.8.1 实现描述

外观模式（Facade Pattern）是一种软件设计模式，属于结构型模式。它提供了一个统一的接口，用来访问一组接口或类的集合，从而隐藏了客户端需要与多个子系统交互的复杂性。外观模式通常用于简化客户端与系统内部的交互，使得客户端只需要与一个外观类交互，而不需要与系统中的多个部分直接交互。

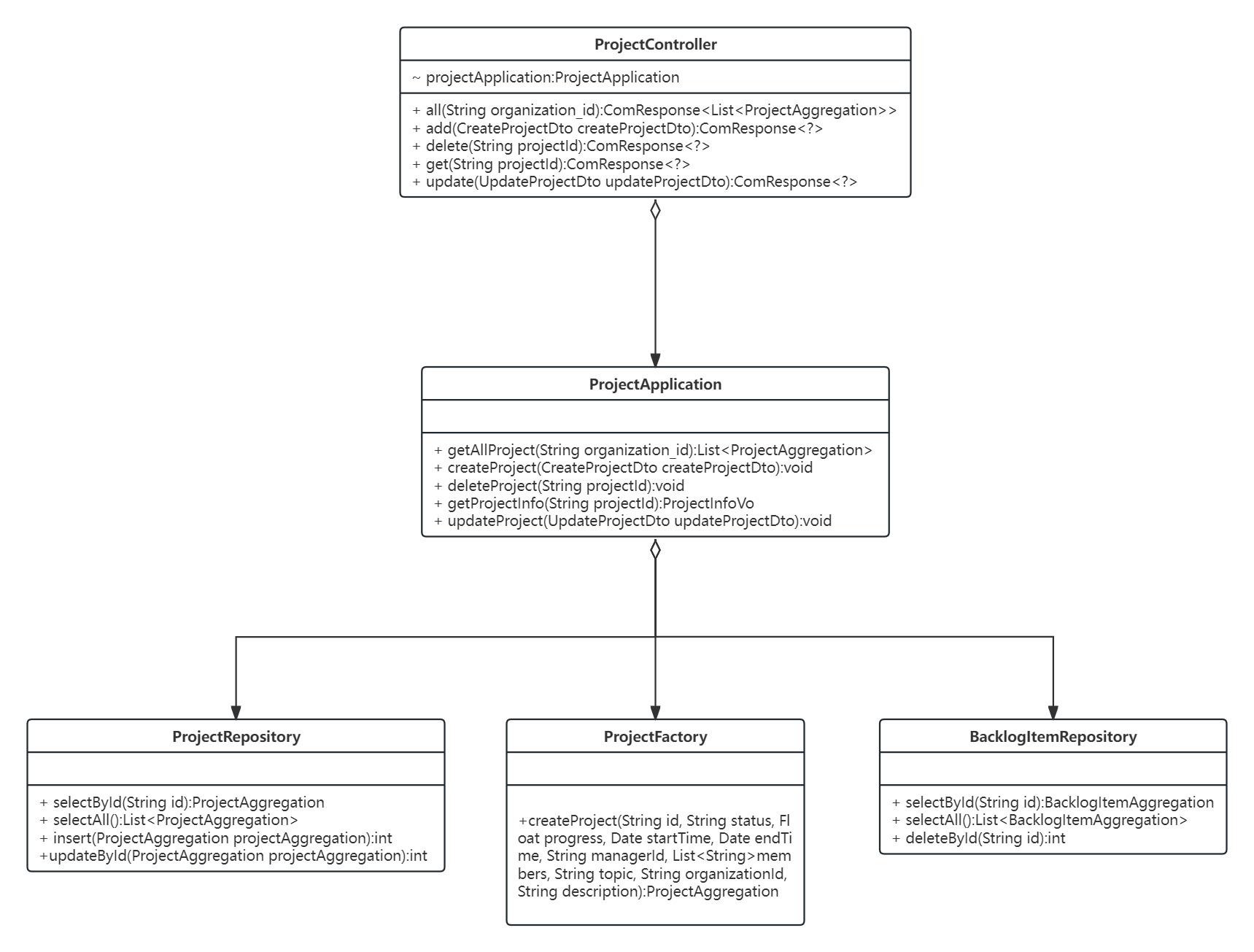
我们的项目的领域驱动设计中的应用服务层（Application Service）使用了外观模式。应用服务层充当领域模型和应用层之间的协调者。应用服务层负责接收用户的请求，协调领域模型的操作，封装复杂的业务流程，隐藏底层领域层的各种业务调用，包括领域服务、工厂、仓储等。把复杂的领域层业务调用封装成应用接口提供给适配器提供更简单的使用。

下面，以本项目中的【账号管理-微服务】中的应用层AccountApplication为例，AccountsApplication类扮演了外观类的角色。它封装了注册、登录、获取账号信息、更新账号信息和注销账号等功能，隐藏了底层的复杂业务流程。



以下是客户端使用外观模式对应用层的调用示例，主要就是AccountController类中调用应用层中的接口，完成对各个子系统（聚合）的复杂业务处理：：

### 3.8.2 UML建模



###### 3.8.3 代价分析

外观模式在提供简化的接口和子系统解耦方面虽然极为有效，但引入它也可能导致某些代价。设计和实现外观类需要时间和资源，且随着底层系统的演化，外观接口也可能需要更新，从而增加维护成本。虽然外观可能在某些情况下引入性能开销，例如增加方法调用层，但这通常是微不足道的，特别是当它在提高代码可读性和易用性方面带来的收益较大时。此外，过度地简化接口可能限制了高级用户使用子系统完整功能的能力。因此，在引入外观模式时，需要在易用性和灵活性、性能以及长期维护之间进行权衡。

**优点：**

1. 简化接口：外观模式通过提供一个统一的接口来简化客户端与多个子系统之间的交互。客户端只需要与外观类交互，而不需要与每个子系统直接交互。

2. 提高封装性：外观模式通过封装子系统，提高了系统的封装性。客户端不需要了解系统内部的具体实现，只需通过外观类来访问服务。

3. 降低耦合度：外观模式降低了客户端与子系统之间的耦合度，使得系统更易于维护和扩展。客户端与外观类之间的依赖关系相对简单。

4. 解耦：外观模式使得客户端与子系统之间解耦，客户端不需要依赖于具体的子系统实现，只需要关注外观类提供的接口。

5. 灵活性：外观模式提供了灵活的方式来管理子系统，可以很容易地在运行时添加新的子系统或修改现有子系统。

**缺点：**

1. 性能开销：外观模式可能会引入额外的性能开销，因为所有的请求都需要通过外观类来进行转发。这可能会导致系统的响应时间变长。

2. 复杂性：外观模式的实现相对复杂，特别是在需要管理多个子系统的外观时。外观类需要处理与各个子系统的交互逻辑，这可能会增加代码的复杂性。

3. 违反开闭原则：在某些外观模式的实现中，为了添加新的子系统，可能需要修改现有代码，这违反了开闭原则中对扩展开放但对修改封闭的要求。

4. 过度使用：如果在不必要的场景下使用外观模式，可能会增加系统的复杂性，导致过度设计。外观模式适用于复杂的系统，对于简单的系统可能不必要使用。

5. 外观泄漏：如果外观类没有正确管理，可能会导致内存泄漏，特别是在动态创建外观类的情况下。

### 3.9 装饰器模式

#### 3.9.1 实现描述

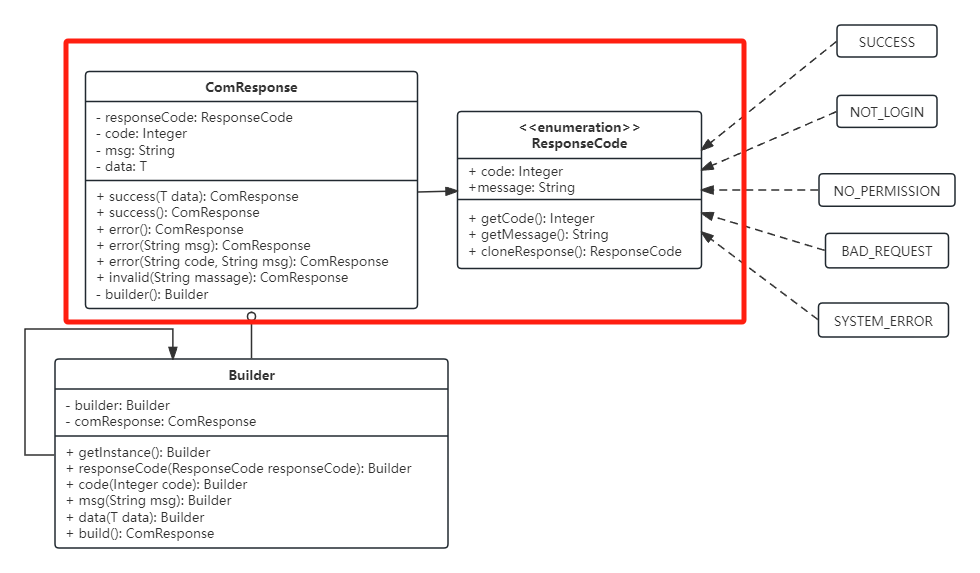
装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

具体实现：

在 ComResponse 类中定义了ResponseCode对象。ResponseCode 是用于处理HTTP请求相关信息的类，在此处相当于对 ResponseCode 进行包装，添加了data相关的处理。



#### 3.9.2 UML建模



#### 3.9.3 代价分析

ResponseCode在http请求的返回中负责错误码、错误信息的处理，也就是http请求相关信息，而Comresponse需要为Response附加处理返回数据的功能，通过装饰器模式封装ResponseCode实现该需求。

**好处：**

装饰器模式可以带来比继承更加灵活性的扩展功能，使用更加方便，可以通过组合不同的装饰者对象来获取具有不同行为状态的多样化的结果。装饰者模式比继承更具良好的扩展性，完美的遵循开闭原则，继承是静态的附加责任，装饰者则是动态的附加责任。装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

### 3.10 策略模式

#### 3.10.1 实现描述

策略模式是一种行为型设计模式，它定义了一组算法，并将它们封装起来，使它们之间可以相互替换，这种模式使得算法的变化可以独立于使用算法的客户，提高了代码的可维护性和可扩展性。

具体而言，策略模式定义了一般化的算法族，将不同类型的算法（称为策略）封装起来，并让它们可以互相替换。通常，策略模式会有一个上下文类（Context）来使用这些策略，上下文类负责管理对策略对象的引用，并可以定义一个接口，让策略能够访问其数据。在运行时，根据实际情况选择合适的策略执行，而无需修改客户端代码。

策略模式便于管理相关算法族，实现不同算法或行为的变化，提高系统的灵活性和可扩展性。同时，策略模式也有一定的局限性，例如客户端必须了解支持的所有策略，而且如果需要新增策略，则需要修改工厂类。

策略模式的主要角色如下：

**抽象策略（Abstract Strategy）类：**这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

**具体策略（Concrete Strategy）类：**实现了抽象策略定义的接口，提供具体的算法实现或行为。

**环境（Context）类：**持有一个策略类的引用，最终给客户端调用。

在本项目中，存在众多“关联”，如Backlog和Backlog的关联、Backlog和Release的关联，为了能够根据需要动态地添加、修改或替换关联策略类，而不需要修改具体代码，在【产品开发-微服务】的领域服务中使用策略模式，具体体现如下：

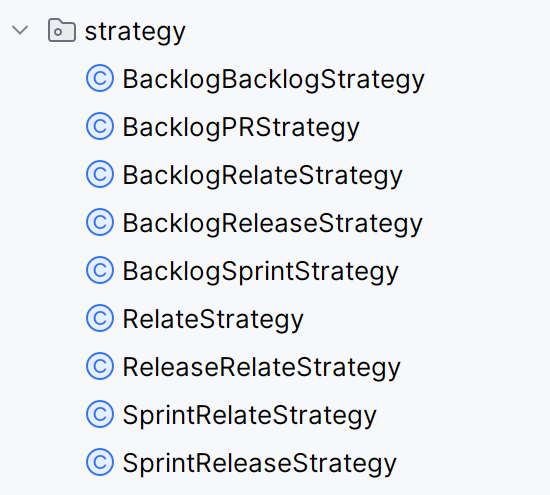
1. **抽象策略类**

抽象策略为RelateStrategy，给出所有具体策略类需要的接口，包括检测testType方法用于测试两个RelateBo对象类型是否匹配、checkSuccessor方法用于检查下一个关联策略类型、getSuccessor方法用于获取下一个相关策略、setSuccessor用于设置下一个关联策略、relate则是用于执行关联操作，RelateStrategy抽象策略如下：

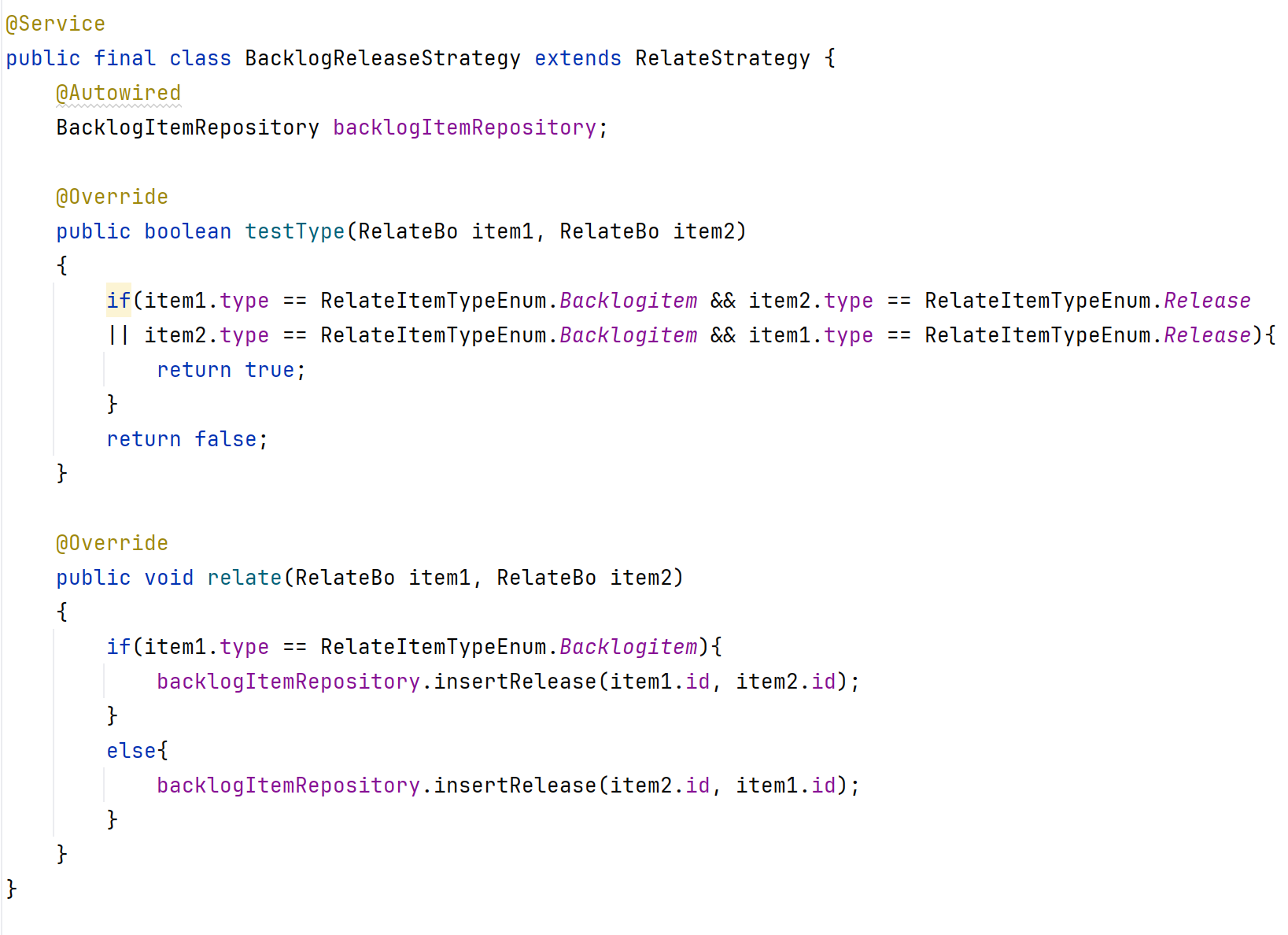


1. **具体策略类**

具体策略实现了抽象策略定义的接口，并提供具体的行为，在本项目中抽象策略RelateStrategy的具体策略类实现包括：BacklogBacklogStrategy、BacklogPRStrategy、BacklogReleaseStrategy、BacklogSprintStrategy、SprintReleaseStrategy等类，具体目录结构如下：

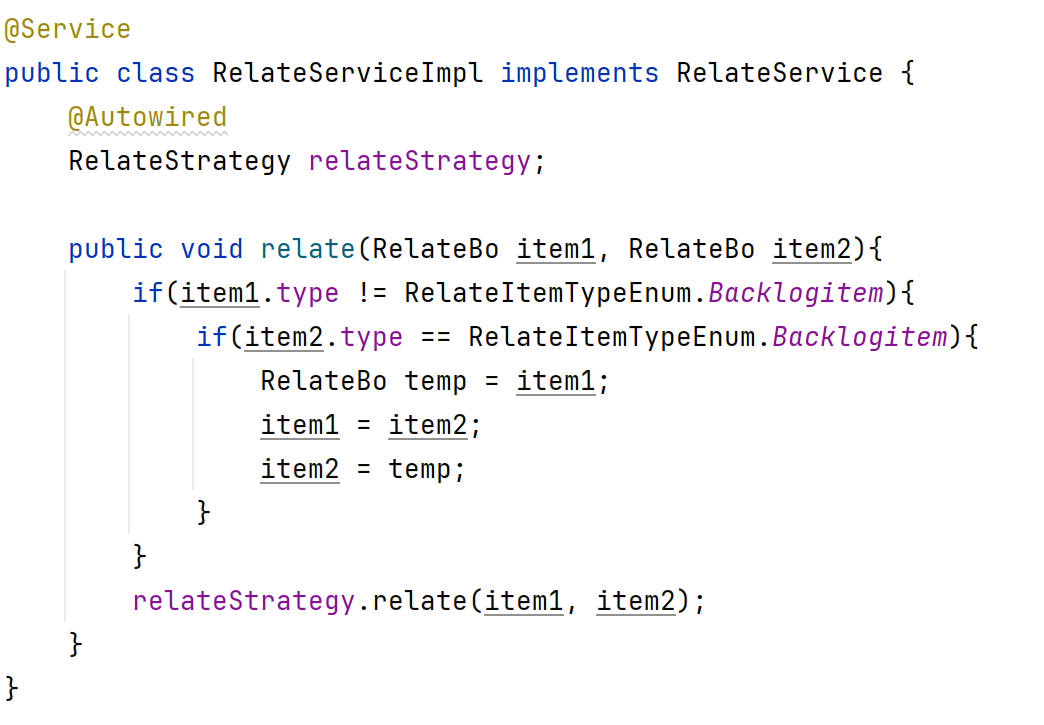


以BacklogReleaseStrategy类的实现为例，它继承自RelateStrategy抽象策略类，用于处理与Backlog和Release关联的策略，对testType和relate接口进行方法重写，如下所示：

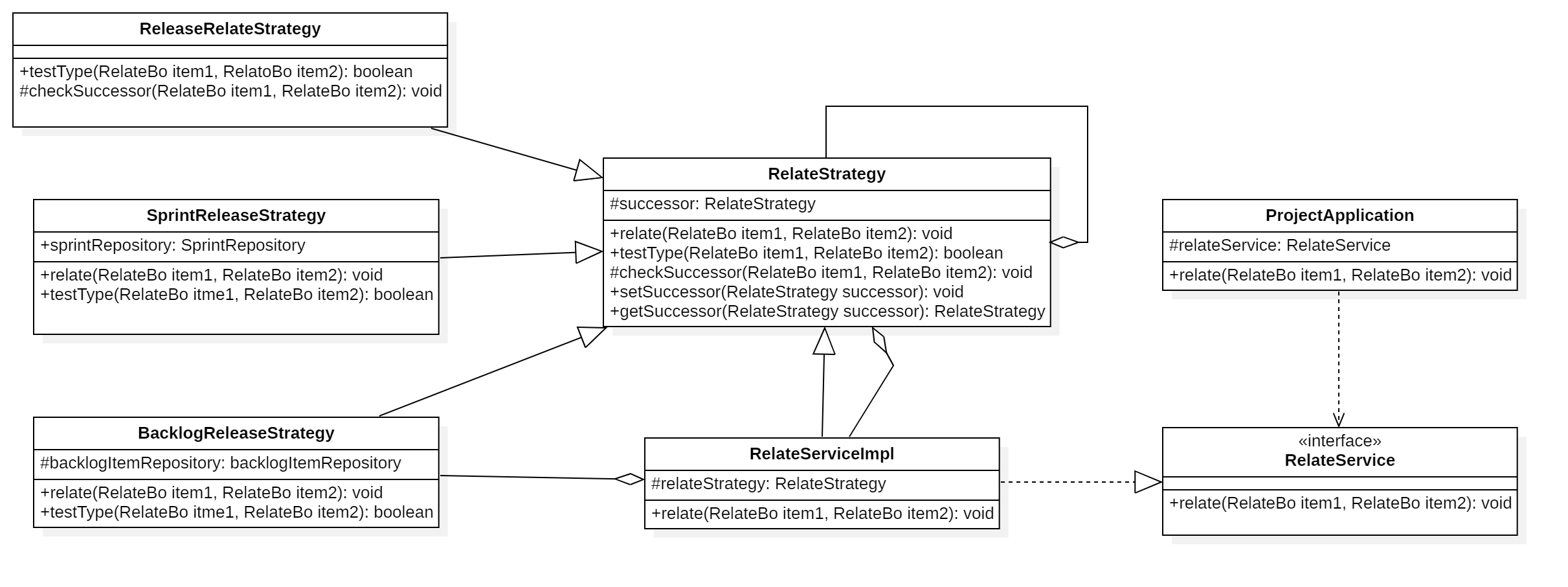


1. **环境类**

环境类持有一个策略类的引用，最终给客户端调用。在本项目中表现为关联服务的实现RelateServiceImpl，在relate方法中，调用抽象策略类的对象relateStrategy.relate(item1, item2)方法，将其委托给具体的相关性策略对象来执行相关性操作，如下图所示：



#### 3.10.2 UML建模



#### 3.10.3 代价分析

策略模式在软件设计中允许在运行时选择算法的行为，通过定义一系列可互换的策略，并在客户端与策略之间进行解耦，能够优化代码结构和复用性。使用策略模式可以增加应用程序的灵活性和维护性，因为它简化了单元测试，并且可以在不影响其他策略的情况下添加或修改策略。然而，策略模式的使用也随之带来其代价和考虑因素。其中包括增加了一定的复杂性，因为你需要定义策略接口和一系列实现该接口的具体策略。在策略数量较多的情况下，管理这些策略可能变得繁琐。同时，客户端需要适当地选择适合的策略，这可能需要额外的逻辑或配置。随着时间的推移，如果某些策略不再需要，那么这些策略的代码将需要被维护或移除，否则有可能导致代码的死重。

**优点：**

1. 算法的可互换性：策略模式允许在运行时根据需要切换算法，而不需要修改客户端代码。

2. 开闭原则：策略模式符合开闭原则，即对扩展开放，对修改关闭。新的策略可以很容易地加入到系统中，而无需修改现有的代码。

3. 避免多重条件转移：策略模式可以避免使用多重if-else或switch语句，从而减少代码的复杂性和维护难度。

4. 算法的封装：策略模式将算法封装在独立的策略类中，提高了代码的模块化，使得算法的实现和客户端的代码解耦。

5. 提高代码的可维护性：由于算法的封装和可互换性，策略模式使得代码更加易于理解和维护。

**缺点：**

1. 客户端需要知道所有策略：客户端必须了解所有可用的策略，并能够根据情况选择合适的策略。这可能会增加客户端的复杂性。

2. 策略过多可能导致性能问题：如果策略数量过多，每次切换策略都可能会对性能产生一定的影响。

3. 策略类数量增加：每个策略都需要一个单独的类来实现，这可能会导致类的数量急剧增加，从而增加了系统的复杂性。

4. 策略之间的共享状态：如果策略之间需要共享状态，可能会导致状态管理变得复杂，因为策略是独立切换的。

5. 违反单一职责原则：策略类通常负责算法的实现和相关的状态管理，这可能会违反单一职责原则，因为策略类可能同时负责多个方面的功能。

### 3.11脏标记模式

#### 3.11.1 实现描述

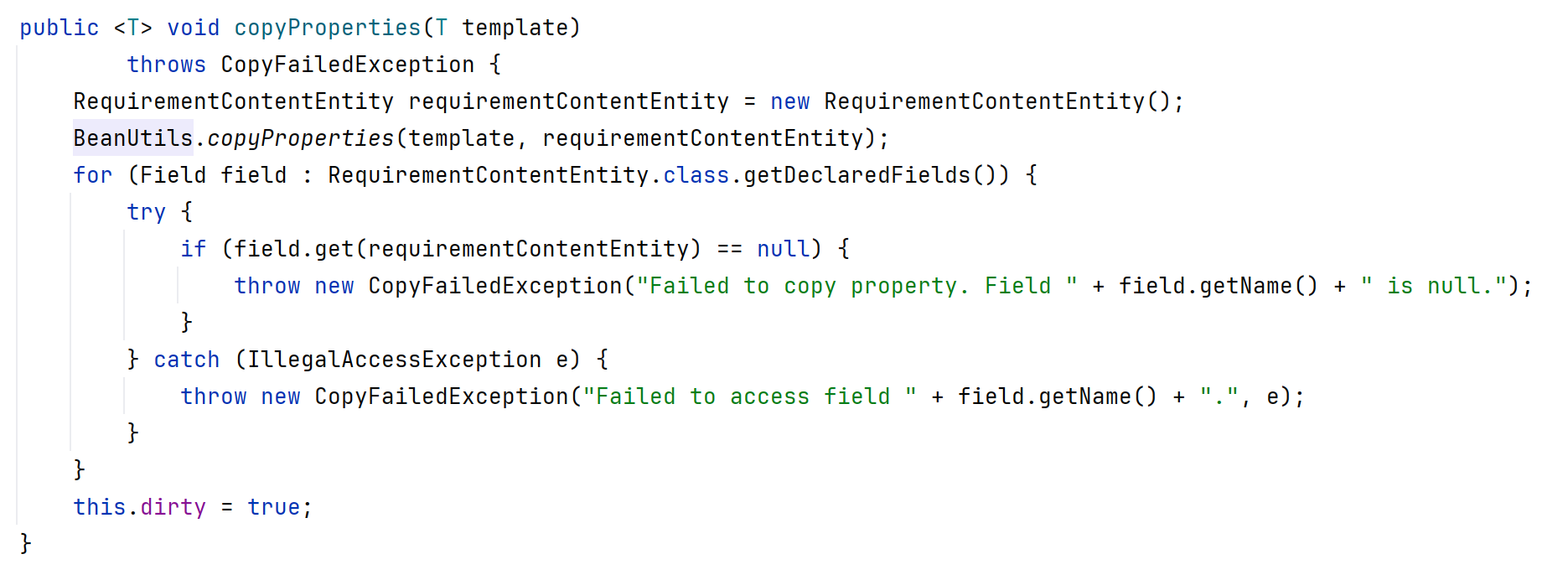
脏标记模式（Dirty Flag pattern）是一种软件设计模式，用于延迟处理或避免不必要的计算操作。它基于一个标志位，通常是一个布尔值，用于标记对象的状态是否已经发生变化。当对象的状态变化时，标志位被设置为"脏"（dirty），表示需要进行相应的处理。如果对象的状态没有发生变化，就可以直接使用缓存数据，避免重新计算。

脏标记模式通常在以下情况下使用：

1. **提高性能：**当某个操作需要进行复杂的计算或涉及到耗时的操作时，可以使用脏标记模式来避免重复计算。通过检查标志位，只有在必要时才执行计算操作，否则直接使用缓存数据。
2. **保护对象的封装性：**脏标记模式可以用于确保对象的状态在创建后不会被外部直接修改。通过将对象状态的修改限制在特定的方法或操作中，可以提高对象的封装性，并确保对象的内部状态不会被意外改变。
3. **事务性操作：**在事务性操作中，你可能希望在事务开始后禁止对某些对象属性的修改，直到事务完成。脏标记模式可以帮助实现这一点。在事务开始时，将标志位设置为"脏"，并在事务结束时进行处理。如果标志位没有被设置，就可以跳过相应的处理逻辑，提高事务的效率。

在本项目中，存在集成多个相关领域对象，跨越多个数据库表的"大聚合"，由一个聚合根和其它实体和值对象组成，聚合根是大聚合的入口点，外部对象只能通过聚合根对大聚合内部对象访问。比如在产品管理微服务中，RequirementAggregation就是一个大聚合，包括了requirementEntity、versionsEntity、requirementContentEntity、clientsEntity、backlogItemsEntity等多个实体，要对聚合内某个实体修改时只能通过聚合根RequirementAggregation，为了避免对数据库不必要的更新操作，这里每一个实体都属于一张数据库表的内容，为每个实体设置了对应的脏标记，在聚合对应的仓储实现对数据库的更新操作时先查看每个实体对应脏标记是否为true（即是否在业务层面有改动），若是才进行更新，否则不对这个实体做数据库更新写入操作，避免对数据库频繁以及不必要的写操作。

每个实体维护管理自己的脏标记，并且在涉及到更新自己属性的方法中置为“脏”：



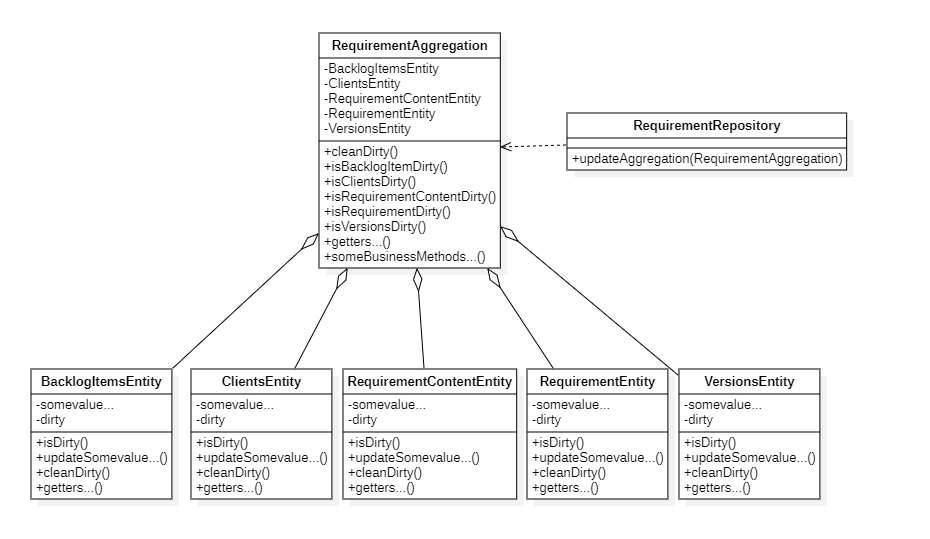
大聚合根统一查看清除子实体脏标记，大聚合更改自己子实体状态时要调用子实体的更新方法：



优化对RequirementAggregation的update操作 （更新大聚合根的时候检查其中每个实体的脏标记决定是否对对应数据表进行更新）：



#### 3.11.2 UML建模



#### 3.11.3 代价分析

脏标记模式通过追踪对象状态的变化来减少不必要的操作。它尤其适用于更新操作开销大且不频繁真正需要同步的场景。然而，代价分析表明，这种模式增加了系统的状态管理复杂性，引入新的错误源，并可能导致数据不一致性，若脏标记未能正确地置位或清除。实施这种模式还可能导致维护成本上升，因为开发人员必须理解哪些操作会导致对象变为“脏”的，并确保适时地处理。因此，虽然选择使用脏标记模式可优化性能，但在对必要的同步和数据完整性的影响方面需要仔细权衡。

**优点：**

1. 封装性：脏标记模式通过引入一个标记来保护对象的内部状态，确保对象的封装性，防止外部直接修改对象属性。

2. 事务性：在事务性操作中，脏标记模式可以确保在事务开始后禁止对某些对象属性的修改，直到事务完成，有助于维护事务的完整性。

3. 状态冻结：在某些场景下，如对象序列化前，脏标记模式可以冻结对象的状态，确保序列化过程中的状态一致性。

4. 简单易实现：脏标记模式的实现相对简单，只需引入一个布尔值作为标记，并在对象的属性修改时更新该标记。

5. 提高性能：只有当对象的状态发生变化时，才需要进行更新操作。这可以避免不必要的更新操作，从而提高性能。

**缺点：**

1. 性能管理开销：每次修改对象属性时，都需要检查脏标记，这可能会导致性能开销，尤其是在频繁修改对象属性的场景下。

2. 增加内部管理复杂性：脏标记模式可能会增加代码的复杂性，特别是在需要管理多个对象的脏标记时。此外，如果标记管理不当，可能会导致逻辑错误。

3. 过度保护：在某些情况下，脏标记模式可能会过度保护对象状态，导致无法灵活地修改对象属性。

4. 延迟更新：由于更新是在状态改变时进行的，所以可能会有一定的延迟。这可能会导致数据的不一致性，特别是在多线程环境中。

#### 3.11.4 出处

[1] Robert Nystrom.Game Programming Patterns.America：Genever Benning，2014. 1～354

### 3.12代理模式

#### 3.12.1 实现描述

代理模式是一种结构型设计模式，它允许通过代理对象控制对另一个对象的访问。代理模式通过引入一个代理对象来间接访问真实对象，从而可以在访问真实对象之前或之后添加一些额外的逻辑或功能。

在代理模式中，代理对象和真实对象通常实现相同的接口或继承相同的父类，以保持对真实对象的透明访问。代理对象负责接收客户端的请求，并在必要时将请求传递给真实对象，同时可以在传递请求前后执行一些附加操作。

代理模式的主要目的是提供一种间接访问和控制真实对象的方式，以便在不修改真实对象的情况下增加额外的功能、限制访问、提供安全性、实现性能优化等。

代理（Proxy）模式分为三种角色：

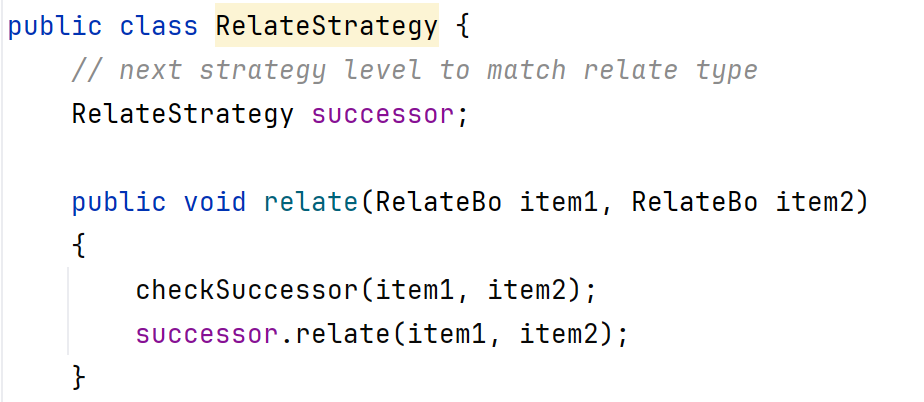
**抽象主题（Subject）类：** 通过接口或抽象类声明真实主题和代理对象实现的业务方法。

**真实主题（Real Subject）类：** 实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。

**代理（Proxy）类 ：** 提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。

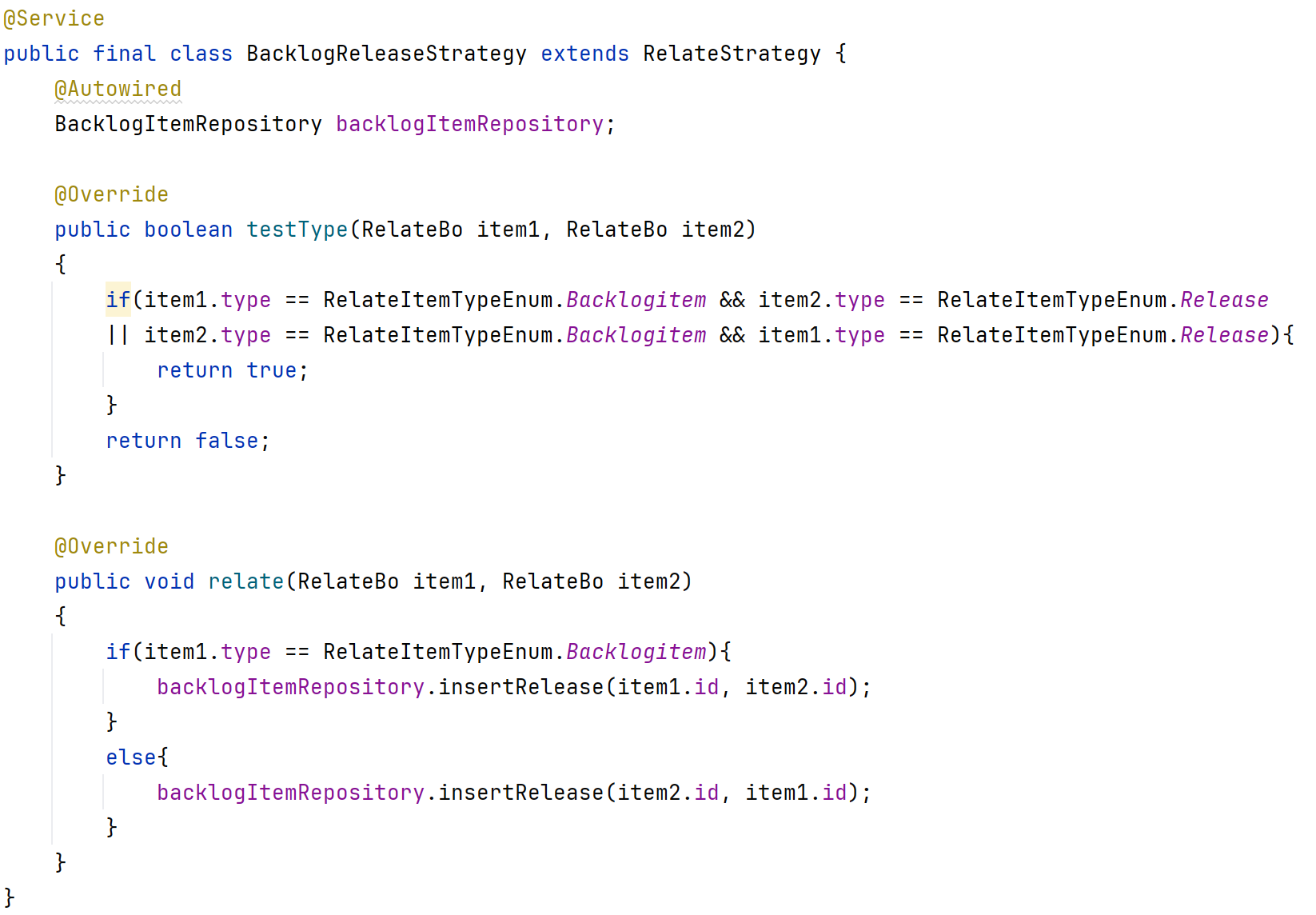
1. **抽象主题类：**

本项目中抽象主题类表现为RelateStrategy类，它通过抽象类声明真实主题和代理对象实现的业务方法relate，如下图所示：



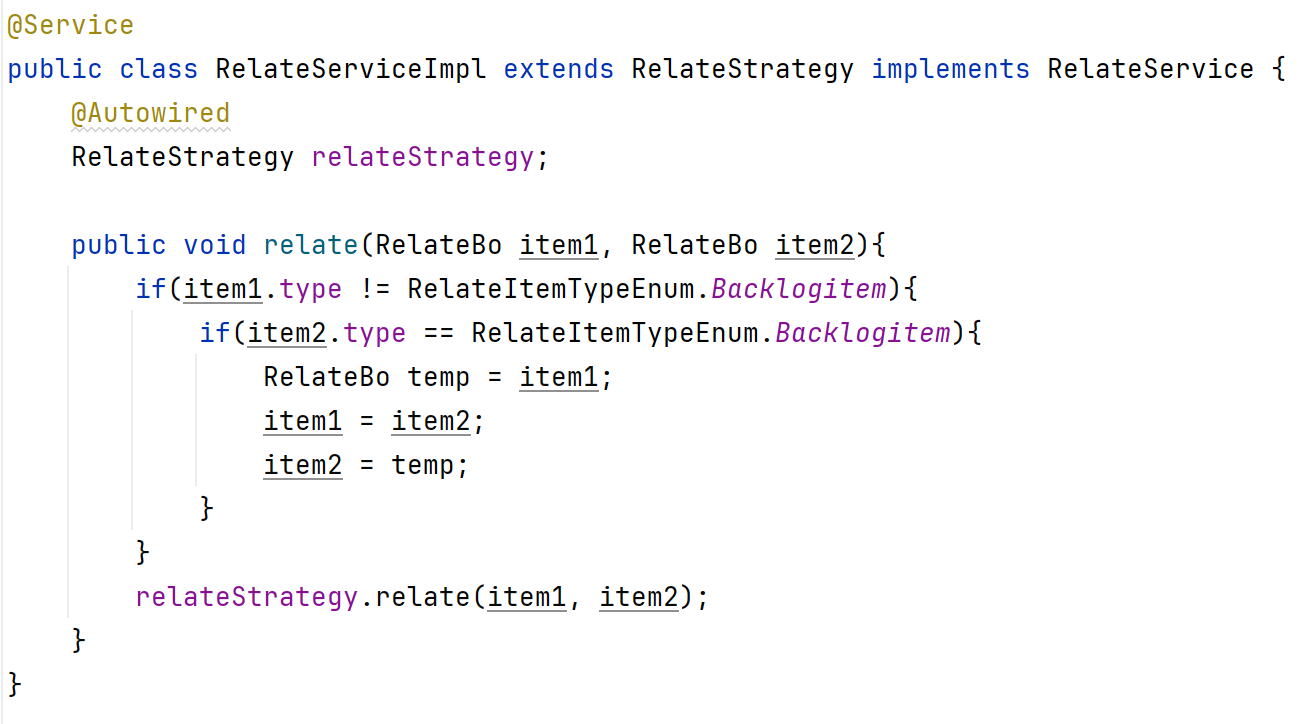
1. **真实主题类：**

本项目中真实主题类表现为BacklogBacklogStrategy、BacklogPRStrategy、BacklogReleaseStrategy、BacklogSprintStrategy、SprintReleaseStrategy等类，它们实现了抽象主题类RelateStrategy中的具体业务关联relate，如BacklogReleaseStrategy用来处理Backlog和Release的具体关联，如下图所示：

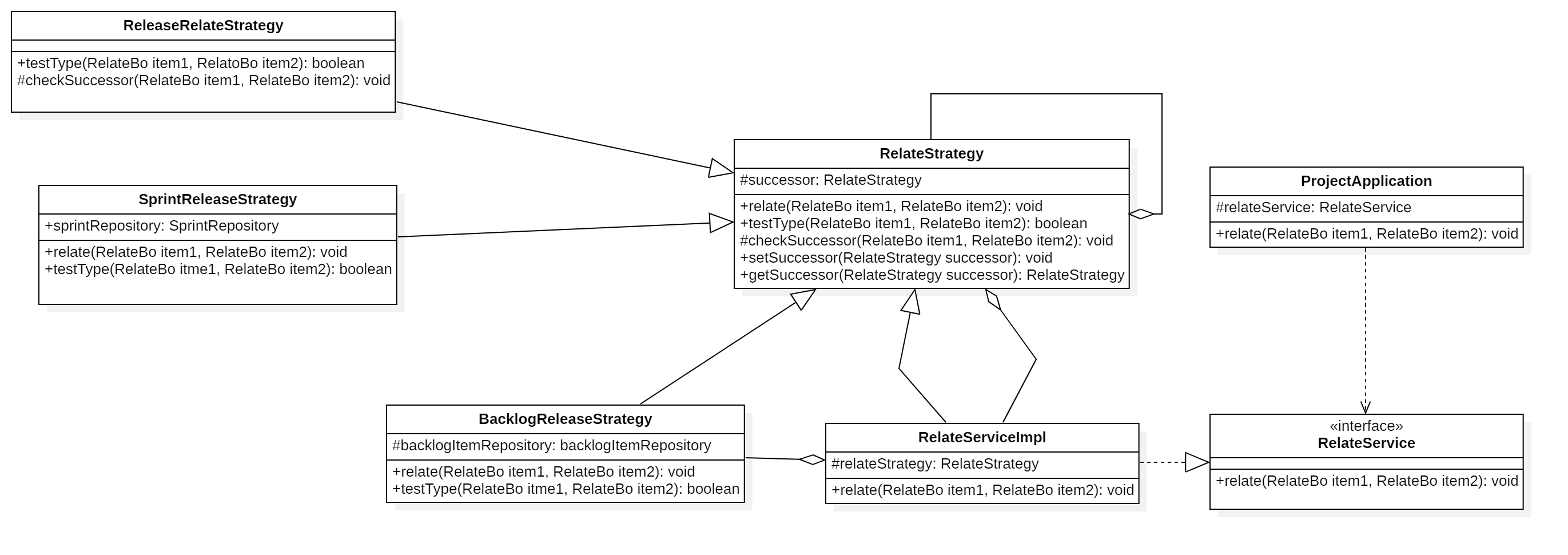


1. **代理类：**

本项目中代理类是RelateServiceImpl，它实现了对RelateStrategy的代理，它在访问真实类relateStrategy之前执行了一些额外的逻辑，在处理业务的方法relate对待关联对象进行判断，做了对象顺序交换，若待关联对象item1和item2中存在Backlogitem类型的对象，则将Backlogitem换到item1的位置，接着猜调用relateStrategy的具体实现类的relate方法，将item1和item2作为参数传递给该方法。



#### 3.12.2 UML建模



#### 3.12.3 代价分析

在关联业务逻辑的实现当中，领域服务RelateService代理了对RelateStrategy的访问，一方面将设计模式的使用封装成领域服务，更加规范的提供给应用层使用接口；另一方面在RelateService当中可以对关联对象预先进行一些操作，例如将bo从聚合中转换，或预处理bo使其符合策略的输入格式。

**优点：**

1. 封装性：代理模式通过引入代理类来控制对真实对象的访问，增强了系统的封装性。代理类作为真实对象和客户端之间的中介，可以隔离客户端直接访问真实对象。

2. 扩展性：代理模式允许在不修改原始类代码的情况下，通过扩展代理类来增加新的功能，如日志记录、权限校验等。这有助于实现开闭原则。

3. 可维护性：通过代理模式，可以将与业务逻辑无关的功能（如缓存、事务管理等）从业务逻辑中分离出来，使得业务逻辑更加清晰，系统更易于维护。

4. 解耦：代理模式降低了客户端和真实对象之间的耦合度，使得系统各部分之间的依赖关系更加灵活。

5. 性能优化：在某些情况下，代理模式可以优化系统性能，例如通过缓存机制减少对后端系统的重复查询。

**缺点：**

1. 性能开销：代理模式引入了额外的代理类，每次对真实对象的访问都需要通过代理，这可能会导致性能开销，尤其是在频繁访问的场景下。

2. 复杂性：代理模式的实现相对复杂，特别是在需要管理多个对象的代理时。此外，如果代理管理不当，可能会导致逻辑错误。

3. 违反开闭原则：在某些代理模式的实现中，为了控制对真实对象的访问，可能会违反开闭原则，即对扩展开放但对修改封闭。

4. 过度使用：如果在不必要的场景下使用代理模式，可能会增加系统的复杂性，导致过度设计。

5. 代理泄漏：如果代理类没有被正确管理，可能会导致内存泄漏，尤其是在动态代理的情况下。

## 附录：项目分工说明与成员贡献

|  |  |  |  |
| --- | --- | --- | --- |
| 学号 | 姓名 | 项目分工 | 成员贡献 |
| 2150266 | 时天逸 | 组长，架构设计，后端开发 | 10% |
| 2152590 | 王琳 | 架构设计，前端开发 | 10% |
| 2151396 | 张靖凯 | 前端开发，后端开发 | 10% |
| 2152034 | 吴杭 | 架构设计，后端开发 | 10% |
| 2150276 | 沈卓成 | 后端开发 | 10% |
| 2154298 | 王颖 | 架构设计，后端开发 | 10% |
| 2154059 | 樊佳怡 | 后端开发 | 10% |
| 2152216 | 蒋钰萱 | 前端开发 | 10% |
| 2151641 | 王佳垚 | 架构设计，后端开发 | 10% |
| 2154016 | 沙木哈尔·阿吾森 | 文档 | 10% |