目录

[第一章 项目简介 8](#_Toc57482245)

[第二章 设计模式汇总表 9](#_Toc57482246)

[第三章 设计模式详述 11](#_Toc57482247)

[3.1空对象模式 11](#_Toc57482248)

[3.1.1 API实现 11](#_Toc57482249)

[3.1.2类图与时序图 11](#_Toc57482250)

[3.1.3补充说明 12](#_Toc57482251)

[3.2工厂模式 12](#_Toc57482252)

[3.2.1API实现 12](#_Toc57482253)

[3.2.2类图与时序图 13](#_Toc57482254)

[3.3.3补充说明 15](#_Toc57482255)

[3.3单例模式 15](#_Toc57482256)

[3.3.1API实现 15](#_Toc57482257)

[3.3.2类图与时序图 15](#_Toc57482258)

[3.3.3补充说明 16](#_Toc57482259)

[3.4抽象工厂模式 16](#_Toc57482260)

[3.4.1API实现 16](#_Toc57482261)

[3.4.2类图与时序图 17](#_Toc57482262)

[3.4.3补充说明 19](#_Toc57482263)

[3.5 MVC模式 19](#_Toc57482264)

[3.5.1API实现 19](#_Toc57482265)

[3.5.2类图与时序图 19](#_Toc57482266)

[3.5.3补充说明 21](#_Toc57482267)

[3.5.4参考文献 22](#_Toc57482268)

[3.6 拦截过滤器模式 22](#_Toc57482269)

[3.6.1API实现 22](#_Toc57482270)

[3.6.2类图与时序图 22](#_Toc57482271)

[3.6.3补充说明 23](#_Toc57482272)

[3.7备忘录模式 24](#_Toc57482273)

[3.7.1API实现 24](#_Toc57482274)

[3.7.2类图与时序图 24](#_Toc57482275)

[3.7.3补充说明 25](#_Toc57482276)

[3.8过滤器模式 26](#_Toc57482277)

[3.8.1API实现 26](#_Toc57482278)

[3.8.2类图与时序图 26](#_Toc57482279)

[3.8.3补充说明 27](#_Toc57482280)

[3.9状态模式 28](#_Toc57482281)

[3.9.1API实现 28](#_Toc57482282)

[3.9.2类图与时序图 28](#_Toc57482283)

[3.9.3补充说明 28](#_Toc57482284)

[3.10访问者模式 29](#_Toc57482285)

[3.10.1API实现 29](#_Toc57482286)

[3.10.2类图与时序图 29](#_Toc57482287)

[3.10.3补充说明 32](#_Toc57482288)

[3.11犹豫模式 32](#_Toc57482289)

[3.11.1API实现 32](#_Toc57482290)

[3.11.2类图与时序图 32](#_Toc57482291)

[3.11.3补充说明 34](#_Toc57482292)

[3.11.4参考文献 34](#_Toc57482293)

[3.12流式接口模式 34](#_Toc57482294)

[3.12.1API实现 34](#_Toc57482295)

[3.12.2类图与时序图 34](#_Toc57482296)

[3.12.3补充说明 37](#_Toc57482297)

[3.12.4 参考文献 37](#_Toc57482298)

[3.13转换器模式 37](#_Toc57482299)

[3.13.1API实现 37](#_Toc57482300)

[3.13.2类图与时序图 37](#_Toc57482301)

[3.13.3补充说明 39](#_Toc57482302)

[3.13.4 参考文献 40](#_Toc57482303)

[3.14迭代子模式 40](#_Toc57482304)

[3.14.1API实现 40](#_Toc57482305)

[3.14.2类图与时序图 40](#_Toc57482306)

[3.15命令模式 42](#_Toc57482307)

[3.15.1API实现 42](#_Toc57482308)

[3.15.2类图与时序图 43](#_Toc57482309)

[3.16责任链模式 44](#_Toc57482310)

[3.16.1API实现 44](#_Toc57482311)

[3.16.2类图与时序图 44](#_Toc57482312)

[3.17回调模式 46](#_Toc57482313)

[3.17.1API实现 46](#_Toc57482314)

[3.17.2类图与时序图 47](#_Toc57482315)

[3.17.3补充说明 48](#_Toc57482316)

[3.18装饰器模式 48](#_Toc57482317)

[3.18.1API实现 48](#_Toc57482318)

[3.18.2基本类图/实例类图/时序图 49](#_Toc57482319)

[3.18.3补充说明 51](#_Toc57482320)

[3.19外观模式 51](#_Toc57482321)

[3.19.1API实现 52](#_Toc57482322)

[3.19.2基本类图/实例类图/时序图 52](#_Toc57482323)

[3.19.3补充说明 54](#_Toc57482324)

[3.20 代理模式 55](#_Toc57482325)

[3.20.1API实现 55](#_Toc57482326)

[3.20.2类图/实例类图/时序图 55](#_Toc57482327)

[3.20.3补充说明 58](#_Toc57482328)

[3.21数据访问对象模式 58](#_Toc57482329)

[3.21.1API实现 58](#_Toc57482330)

[3.21.2实例类图/时序图 59](#_Toc57482331)

[3.21.3补充说明 61](#_Toc57482332)

[3.21.4 参考文献 62](#_Toc57482333)

[3.22雇工模式 62](#_Toc57482334)

[3.22.1API实现 62](#_Toc57482335)

[3.22.2类图/实例类图/时序图 62](#_Toc57482336)

[3.22.3补充说明 63](#_Toc57482337)

[3.22.4参考文献 64](#_Toc57482338)

[3.23类型对象模式模式 64](#_Toc57482339)

[3.23.1API实现 64](#_Toc57482340)

[3.23.2类图/实例类图/时序图 64](#_Toc57482341)

[3.23.3补充说明 66](#_Toc57482342)

[3.23.4类型对象模式 67](#_Toc57482343)

[3.24工厂模式 67](#_Toc57482344)

[3.24.1API实现 67](#_Toc57482345)

[3.24.2类图与时序图 67](#_Toc57482346)

[3.24.3补充说明 68](#_Toc57482347)

[3.25建造者模式 68](#_Toc57482348)

[3.25.1API实现 68](#_Toc57482349)

[3.25.2类图与时序图 69](#_Toc57482350)

[3.25.3补充说明 70](#_Toc57482351)

[3.26适配器模式 70](#_Toc57482352)

[3.26.1API实现 70](#_Toc57482353)

[3.26.2类图与时序图 71](#_Toc57482354)

[3.26.3补充说明 71](#_Toc57482355)

[3.27 传输对象模式 72](#_Toc57482356)

[3.27.1API实现 72](#_Toc57482357)

[3.27.2类图与时序图 72](#_Toc57482358)

[3.27.3补充说明 72](#_Toc57482359)

[3.28委托模式 72](#_Toc57482360)

[3.28.1API实现 73](#_Toc57482361)

[3.28.2类图与时序图 73](#_Toc57482362)

[3.28.3补充说明 73](#_Toc57482363)

[3.29模版模式 74](#_Toc57482364)

[3.29.1API实现 74](#_Toc57482365)

[3.29.2类图与时序图 74](#_Toc57482366)

[3.29.3补充说明 74](#_Toc57482367)

[3.30 解释器模式 75](#_Toc57482368)

[3.30.1API实现 75](#_Toc57482369)

[3.30.2类图与时序图 75](#_Toc57482370)

[3.31 组合实体模式 76](#_Toc57482371)

[3.31.1API实现 76](#_Toc57482372)

[3.31.2类图与时序图 76](#_Toc57482373)

[3.32 中介者模式 78](#_Toc57482374)

[3.32.1API实现 78](#_Toc57482375)

[3.32.2类图与时序图 78](#_Toc57482376)

[3.33 GameLoop模式 80](#_Toc57482377)

[3.33.1 API实现 80](#_Toc57482378)

[3.33.2类图与时序图 80](#_Toc57482379)

[3.33.3补充说明 80](#_Toc57482380)

[3.33.4 参考文献 80](#_Toc57482381)

[3.34 Monostate模式 81](#_Toc57482382)

[3.34.1 API实现 81](#_Toc57482383)

[3.34.2类图与时序图 81](#_Toc57482384)

[3.34.3补充说明 81](#_Toc57482385)

[3.34.4 参考文献 81](#_Toc57482386)

[3.35 策略模式 81](#_Toc57482387)

[3.35.1API实现 82](#_Toc57482388)

[3.35.2类图/实例类图/时序图 82](#_Toc57482389)

[3.35.2 补充说明 83](#_Toc57482390)

[3.36 模版方法模式 83](#_Toc57482391)

[3.36.1API实现 83](#_Toc57482392)

[3.36.2类图与时序图 83](#_Toc57482393)

[3.36.3参考文献 86](#_Toc57482394)

[3.37 观察者模式 86](#_Toc57482395)

[3.37.1API实现 86](#_Toc57482396)

[3.37.2类图与时序图 86](#_Toc57482397)

[3.38 数据总线模式 87](#_Toc57482398)

[3.38.1API实现 87](#_Toc57482399)

[3.38.2类图与时序图 87](#_Toc57482400)

[3.38.3简要介绍 89](#_Toc57482401)

[3.38.4参考文献 90](#_Toc57482402)

[3.39 享元模式 90](#_Toc57482403)

[3.39.1API实现 90](#_Toc57482404)

[3.39.2类图与时序图 90](#_Toc57482405)

[3.39.3补充说明 93](#_Toc57482406)

[3.40 桥接模式 93](#_Toc57482407)

[3.40.1API实现 93](#_Toc57482408)

[3.40.2类图与时序图 93](#_Toc57482409)

[3.40.3补充说明 94](#_Toc57482410)

[3.41 组合模式 94](#_Toc57482411)

[3.41.1API实现 94](#_Toc57482412)

[3.41.2类图与时序图 94](#_Toc57482413)

[3.41.3补充说明 96](#_Toc57482414)

[3.42多例模式 96](#_Toc57482415)

[3.42.1API实现 96](#_Toc57482416)

[3.42.2类图与时序图 96](#_Toc57482417)

[3.42.3补充说明 97](#_Toc57482418)

[3.43 工厂模式 97](#_Toc57482419)

[3.43.1API实现 97](#_Toc57482420)

[3.43.2类图与时序图 98](#_Toc57482421)

[第四章 项目搭建及错误处理方法 99](#_Toc57482422)

[4.1 推荐环境及配置 99](#_Toc57482423)

[4.2 项目下载 99](#_Toc57482424)

[4.3 错误处理方法 100](#_Toc57482425)

[4.2.1 找不到模块 100](#_Toc57482426)

[4.2.2 未依赖模块 102](#_Toc57482427)

1. 项目简介

海底世界嘉年华聚焦于海洋世界，秉承打造“海洋文化”为核心内涵的理念，拉近人们与海洋生物之间的距离。走近自然才能保护自然，工业排放，海洋资源开采，旅游业发展导致的海洋污染，许多是人们不重视海洋环境保护，不了解海洋生态导致的，海底世界嘉年华提供了这样一个场合，人们可以与海洋生物面对面互动，了解“海洋文化”，踏入海底世界进行一场动人心弦的探险，从中获得对于海洋的感悟，稳固“环境友好”的理念。

本项目涉及了海底世界嘉年华的方方面面，用不同的设计模式体现出人们可以在海底世界嘉年华中参与的许多活动，以及海洋世界场馆的风格构造，从嘉年华中的场馆内饰到场馆风格的改变，从票务系统到餐饮服务，本项目力图在设计上借助于现有的设计模式达到最优。某些重要的设计理念体现在很多活动当中，而某些巧妙的设计模式又依附于特定场景，凸显其精妙之处。

置身于海底世界嘉年华，体会人类设计的巧夺天工和自然景观的鬼斧神工。

1. 设计模式汇总表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 设计模式名称 | 实现个数 | 何处体现 | 备注说明 |
| 1 | Null Object Pattern | 1 | ccd.nullobjectpattern | 迟承道 |
| 2 | Factory Pattern | 1 | ccd. factorypattern | 迟承道 |
| 3 | Singleton Pattern | 1 | ccd. singleton tpattern | 迟承道 |
| 4 | Abstract Factory Pattern | 1 | ccd.abstractfactorypattern | 迟承道 |
| 5 | MVC Pattern | 1 | ccd.mvctpattern | 迟承道 |
| 6 | Interceptingfilter Pattern | 1 | ccd.interceptingfilterpattern | 迟承道 |
| 7 | Filter Pattern | 1 | gjh.filterpattern | 关敬徽 |
| 8 | State Pattern | 1 | gjh.statepattern | 关敬徽 |
| 9 | Visitor Pattern | 1 | gjh.visitorpattern | 关敬徽 |
| 10 | Memento Pattern | 1 | gjh.mementopattern | 关敬徽 |
| 11 | Balking Pattern | 1 | gjh.balkingpattern | 关敬徽 |
| 12 | FluentInterface Pattern | 1 | wwz.fluentinterfacepattern | 王文政 |
| 13 | Converter Pattern | 1 | wwz.converterpattern | 王文政 |
| 14 | Iterator Pattern | 1 | wwz.iteratorpttern | 王文政 |
| 15 | Command Pattern | 1 | wwz.commandpattern | 王文政 |
| 16 | Chain of Responsibility Pattern | 1 | wwz.chainofresponsibility-  pattern | 王文政 |
| 17 | Callback Pattern | 1 | wwz.callbackpattern | 王文政 |
| 18 | Decorator Pattern | 1 | sfz.decoratorpattern | 沈放之 |
| 19 | Facade Pattern | 1 | sfz.facadepattern | 沈放之 |
| 20 | Proxy Pattern | 1 | sfz.proxypattern | 沈放之 |
| 21 | DAO Pattern | 1 | sfz.dataaccessobjectpattern | 沈放之 |
| 22 | Servant Pattern | 1 | sfz.servantpattern | 沈放之 |
| 23 | Type Object Pattern | 1 | sfz.typeobjectpattern | 沈放之 |
| 24 | Prototype Pattern | 1 | nyx.prototypepattern | 聂义鑫 |
| 25 | Buidler Pattern | 1 | nyx.builderpattern | 聂义鑫 |
| 26 | Adapter Pattern | 1 | nyx.adapterpattern | 聂义鑫 |
| 27 | Transfer Object Pattern | 1 | nyx.transferobjectpattern | 聂义鑫 |
| 28 | Delegation pattern | 1 | nyx.delegation | 聂义鑫 |
| 29 | Module Pattern | 1 | nyx.modulepattern | 聂义鑫 |
| 30 | Interpreter Pattern | 1 | zlf.interpreterpattern | 周力凡 |
| 31 | Composite Entity Pattern | 1 | zlf.compositeentitypattern | 周力凡 |
| (重复) | DAO Pattern | 1 | zlf.daopattern | 周力凡 |
| 32 | Mediator Pattern | 1 | zlf.mediatorpattern | 周力凡 |
| 33 | Gameloop Pattern | 1 | zlf.gamelooppattern | 周力凡 |
| 34 | Mono State Pattern | 1 | zlf.monostatepattern | 周力凡 |
| 35 | Strategy Pattern | 1 | sfz.strategypattern | 沈放之 |
| 36 | Template Method Pattern | 1 | wwz.templatemethodpattern | 王文政 |
| 37 | Observer Pattern | 1 | wwz.observerpattern | 王文政 |
| 38 | Data Bus Pattern | 1 | wwz. databuspattern | 王文政 |
| 39 | Flyweight Pattern | 1 | sjg.flyweightpattern | 宋俊钢 |
| 40 | Bridge Pattern | 1 | sjg.bridgepattern | 宋俊钢 |
| 41 | Composite Pattern | 1 | sjg.compositepattern | 宋俊钢 |
| 42 | Multiton Pattern | 1 | sjg.multitionpattern | 宋俊钢 |
| 43 | Double Check Locking Pattern | 1 | sjg.doublechecklocking-  pattern | 宋俊钢 |

1. 设计模式详述

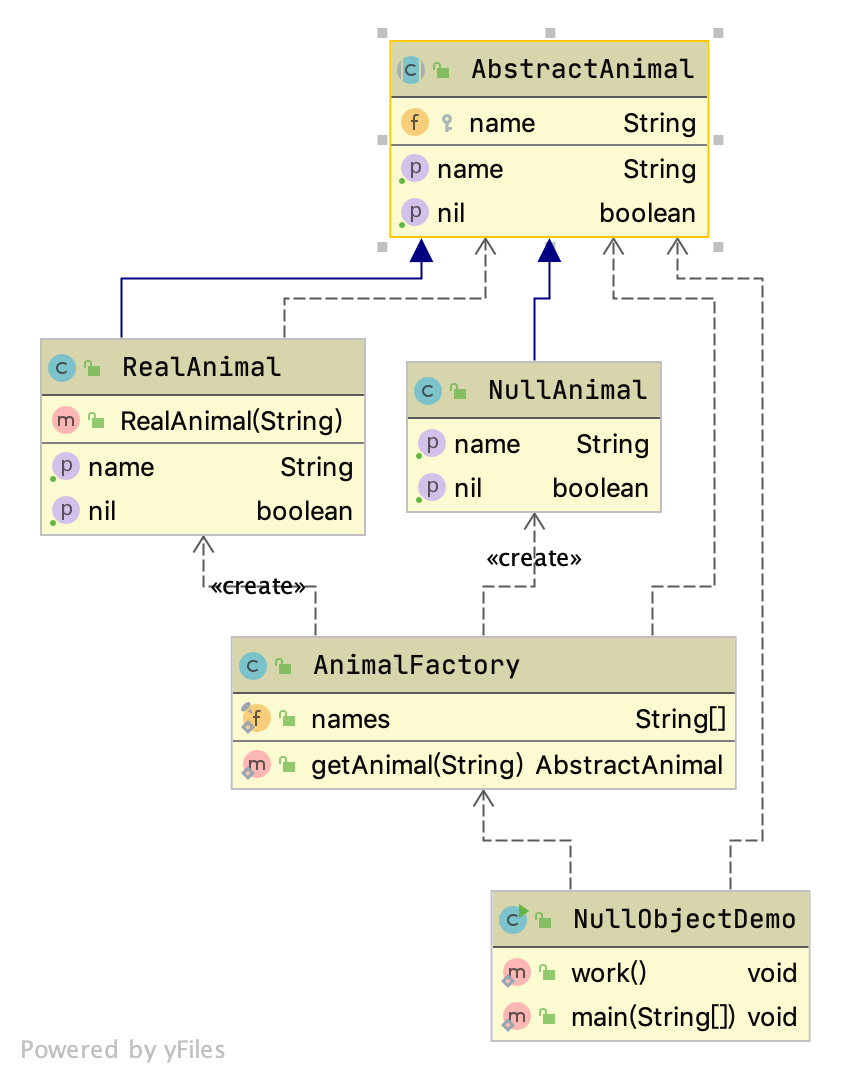
3.1空对象模式

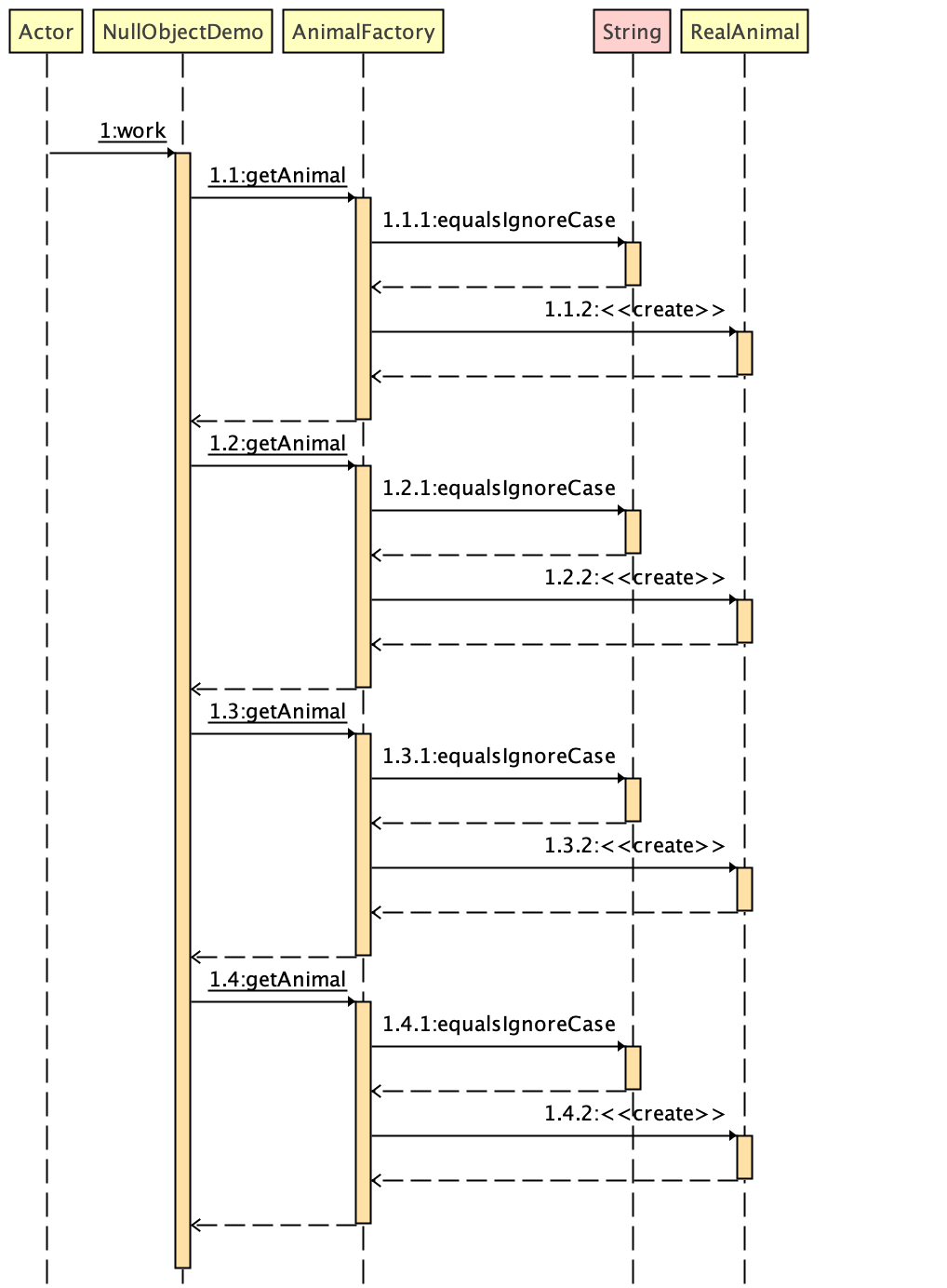
3.1.1 API实现

我们将创建一个定义操作（在这里，是动物的名称）的 AbstractAnimal 抽象类，和扩展了 AbstractAnimal 类的实体类。工厂类 AnimalFactory 基于客户传递的名字来返回 RealAnimal或 NullAnimal 对象。

NullPatternDemo，我们的演示类使用 AnimalFactory 来演示空对象模式的用法。

3.1.2类图与时序图





3.1.3补充说明

在空对象模式（Null Object Pattern）中，一个空对象取代 NULL 对象实例的检查。Null 对象不是检查空值，而是反应一个不做任何动作的关系。这样的 Null 对象也可以在数据不可用的时候提供默认的行为。

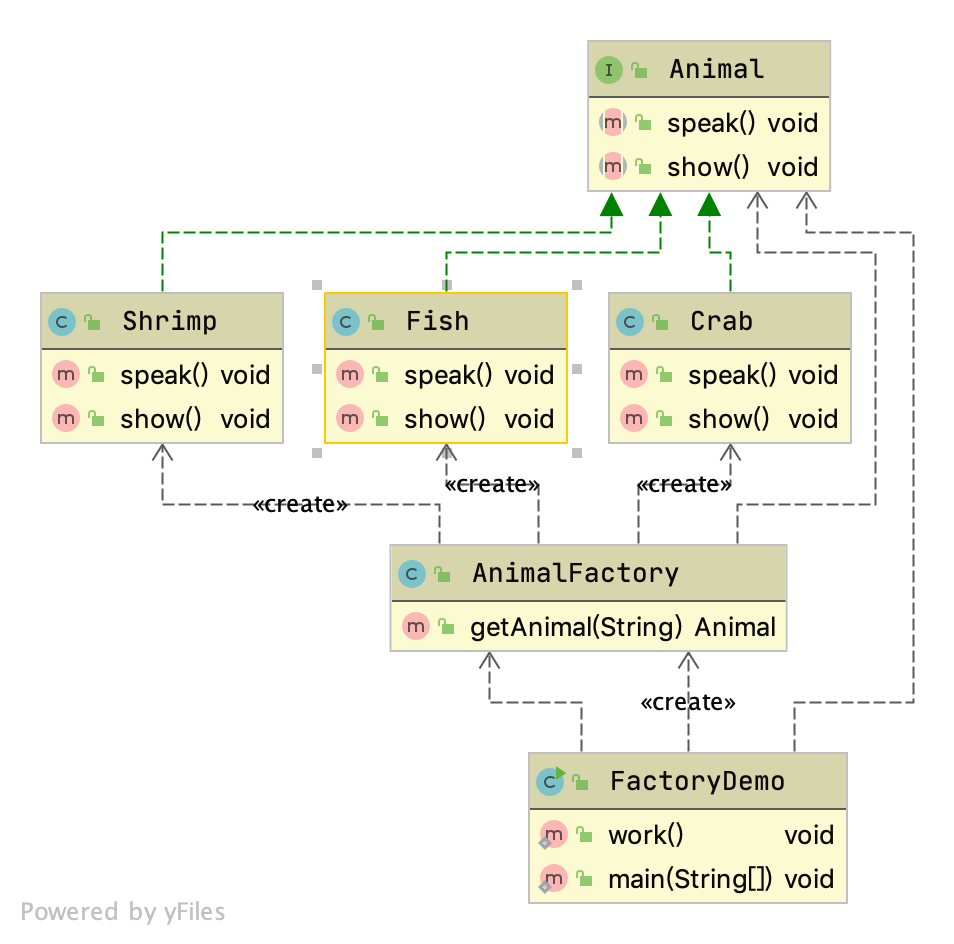
在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。

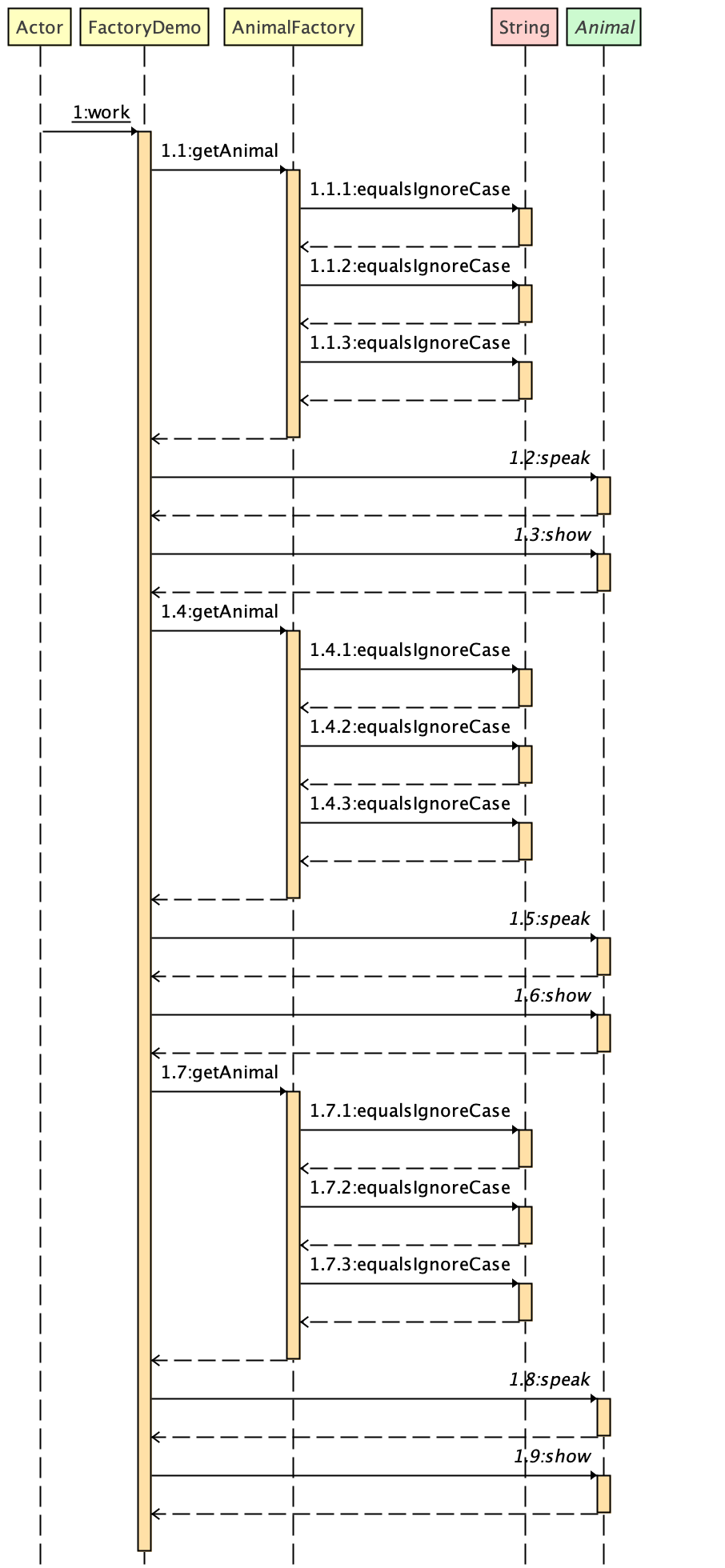
3.2工厂模式

3.2.1API实现

我们将创建一个 Animal 接口和实现 Animal 接口的实体类。下一步是定义工厂类 AnimalFactory。FactoryDemo 类使用 AnimalFactory 来获取 Animal 对象。它将向 Anima-lFactory 传递信息（Fish / Crab / Shrimp），以便获取它所需对象的类型。

3.2.2类图与时序图





3.3.3补充说明

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。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。

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

优点： 1、一个调用者想创建一个对象，只要知道其名称就可以了。

2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。

3、屏蔽产品的具体实现，调用者只关心产品的接口。

缺点：每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

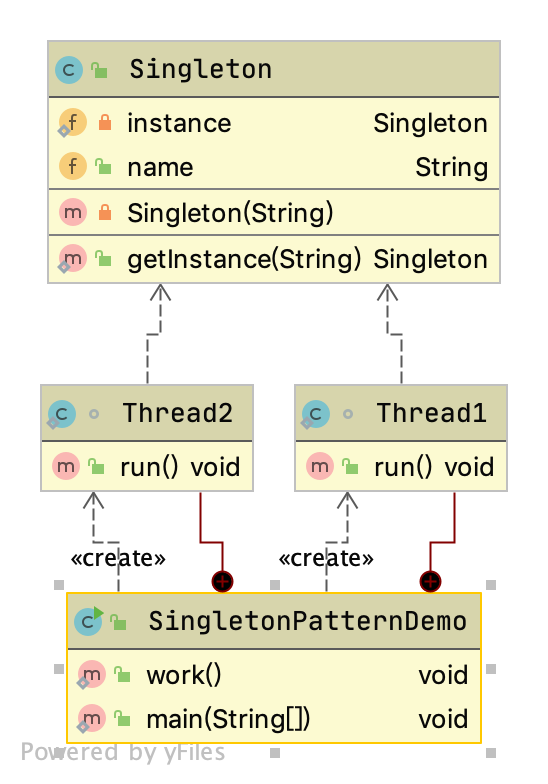
3.3单例模式

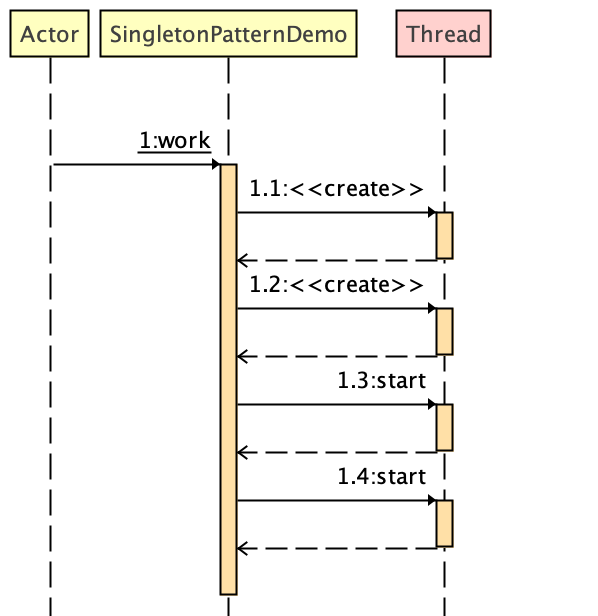
3.3.1API实现

我们将创建一个 Singleton类。Singleton 类有它的私有构造函数和本身的一个静态实例。

Singleton 类提供了一个静态方法，供外界获取它的静态实例。SingletonDemo 类使用 Singleton 类来获取 Singleton 对象。

3.3.2类图与时序图





3.3.3补充说明

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。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。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。需要满足：单例类只能有一个实例，单例类必须自己创建自己的唯一实例，单例类必须给所有其他对象提供这一实例。

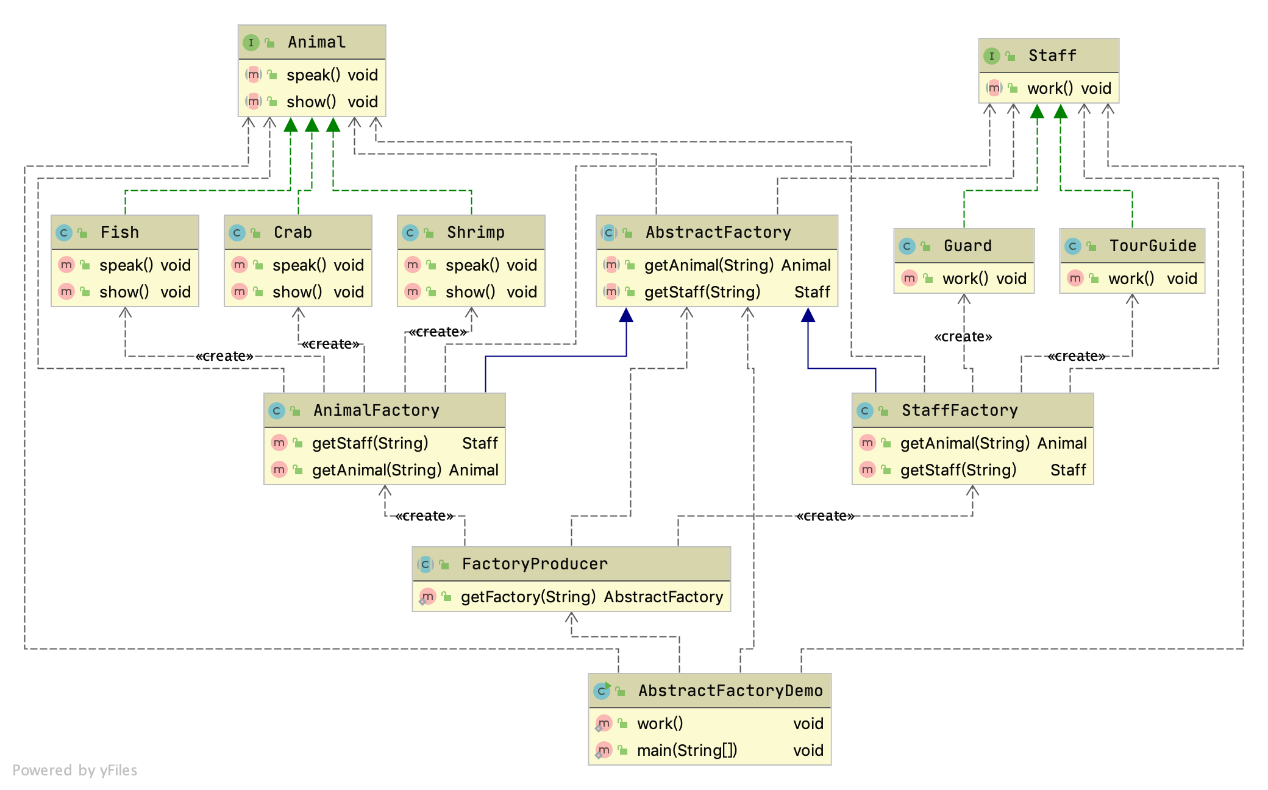
3.4抽象工厂模式

3.4.1API实现

我们将创建 Animal 和 Staff 接口和实现这些接口的实体类。下一步是创建抽象工厂类 AbstractFactory。接着定义工厂类 AnimalFactory 和 StaffFactory，这两个工厂类都是扩展了 AbstractFactory。然后创建一个工厂创造器/生成器类 FactoryProducer。

AbstractFactoryPatternDemo 类使用 FactoryProducer 来获取 AbstractFactory 对象。它将向 AbstractFactory 传递形状信息 Animal（Fish / Crab / Shrimp），以便获取它所需对象的类型。同时它还向 AbstractFactory 传递颜色信息 Staff（TourGuide / Guard ），以便获取它所需对象的类型。

3.4.2类图与时序图





3.4.3补充说明

抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。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。

在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。

优点：当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

缺点：产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

3.5 MVC模式

3.5.1API实现

实现了一个动物的MVC模型，view提供视图窗口的api，controller提供更改model的api，model实现获取数据和修改数据的api，最后实现展示MVC模式的api。

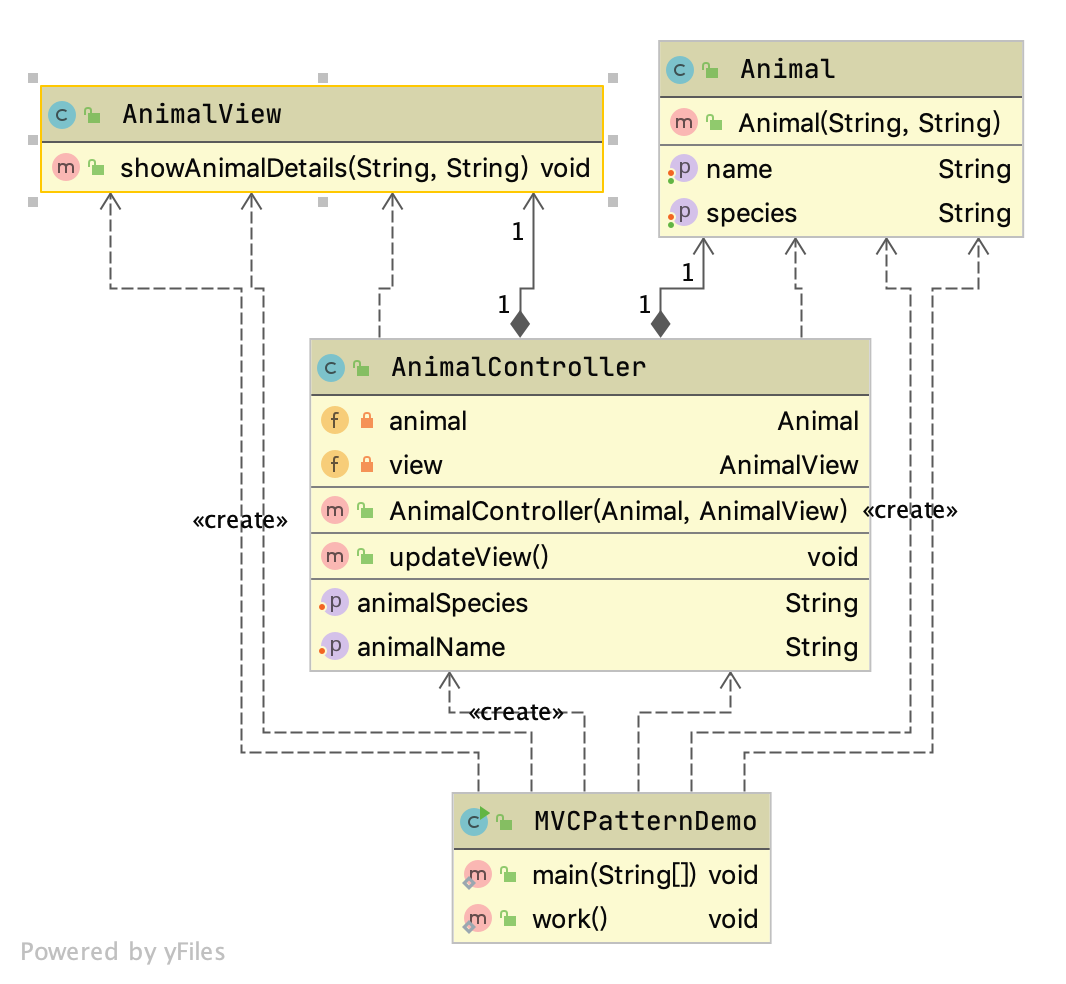
MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

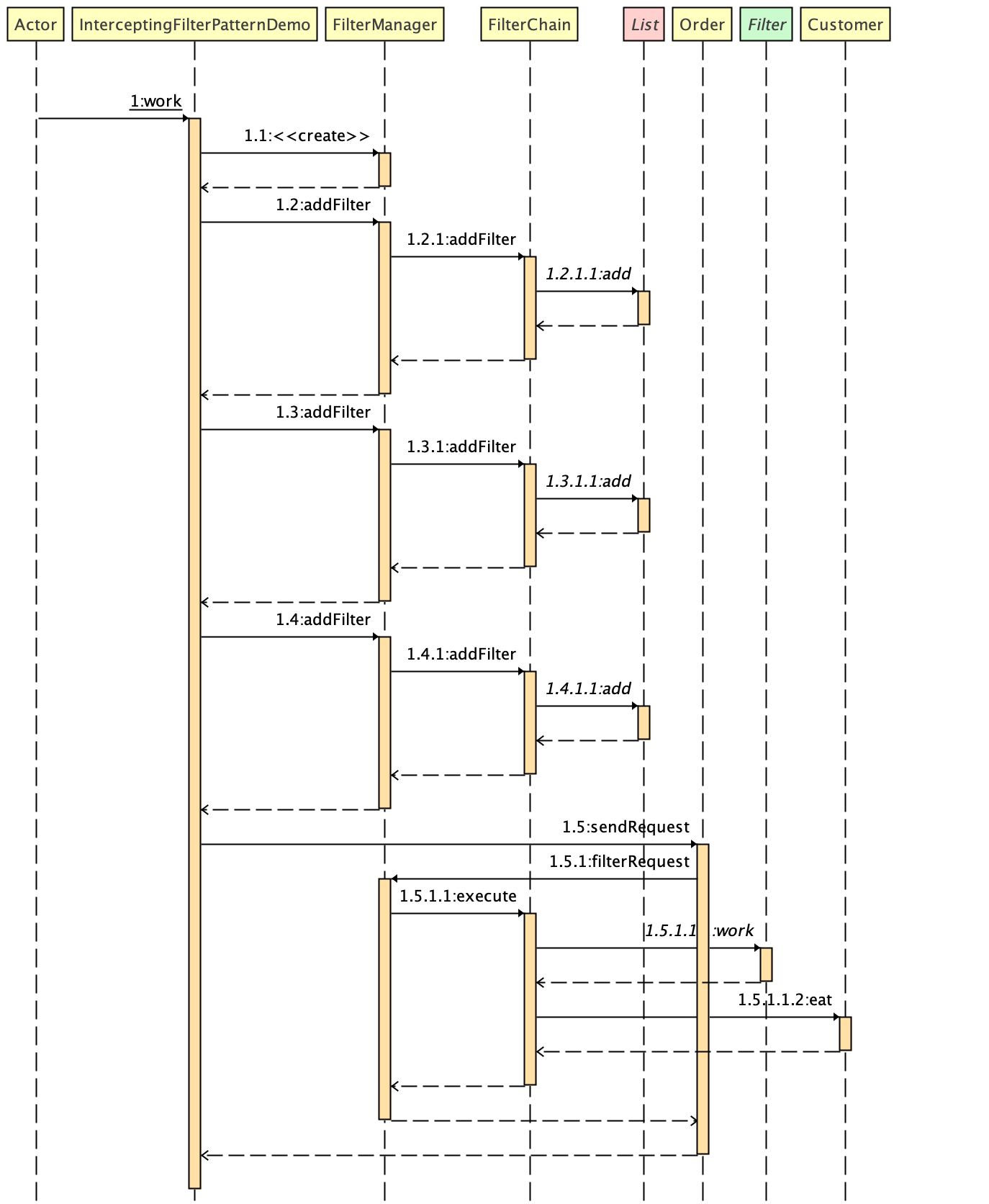
Model（模型） - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。

View（视图） - 视图代表模型包含的数据的可视化。

Controller（控制器） - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

3.5.2类图与时序图





3.5.3补充说明

我们将创建一个作为模型的 *Animal* 对象。*AnimalView* 是一个把动物详细信息输出到控制台的视图类，*AnimalController* 是负责存储数据到 *Animal* 对象中的控制器类，并相应地更新视图 *AnimalView*。

*MVCPatternDemo*，我们的演示类使用 *AnimalController* 来演示 MVC 模式的用法。

3.5.4参考文献

参考：[*https://www.runoob.com/design-pattern/mvc-pattern.html*](https://www.runoob.com/design-pattern/mvc-pattern.html)

3.6 拦截过滤器模式

3.6.1API实现

实现了3个filter（cook，cashier，waiter）的api，以及一个customer类的api，用于生成相应的log。一个order类的api用于实现控制点菜过程的所有log的生成。实现FilterChain的api，控制生成过滤器链、FilterManager用于管理filter的api。以及最终调用order进行展示的api。

拦截过滤器模式（Intercepting Filter Pattern）用于对应用程序的请求或响应做一些预处理/后处理。定义过滤器，并在把请求传给实际目标应用程序之前应用在请求上。过滤器可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

过滤器（Filter） - 过滤器在请求处理程序执行请求之前或之后，执行某些任务。

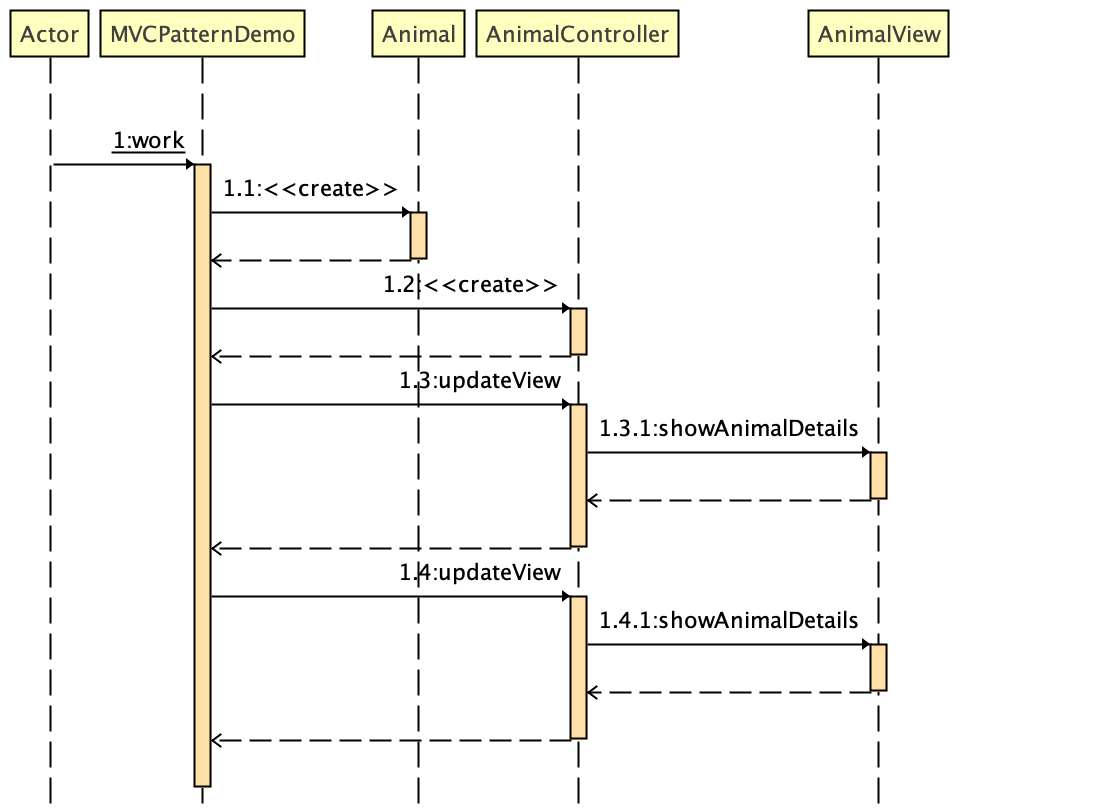
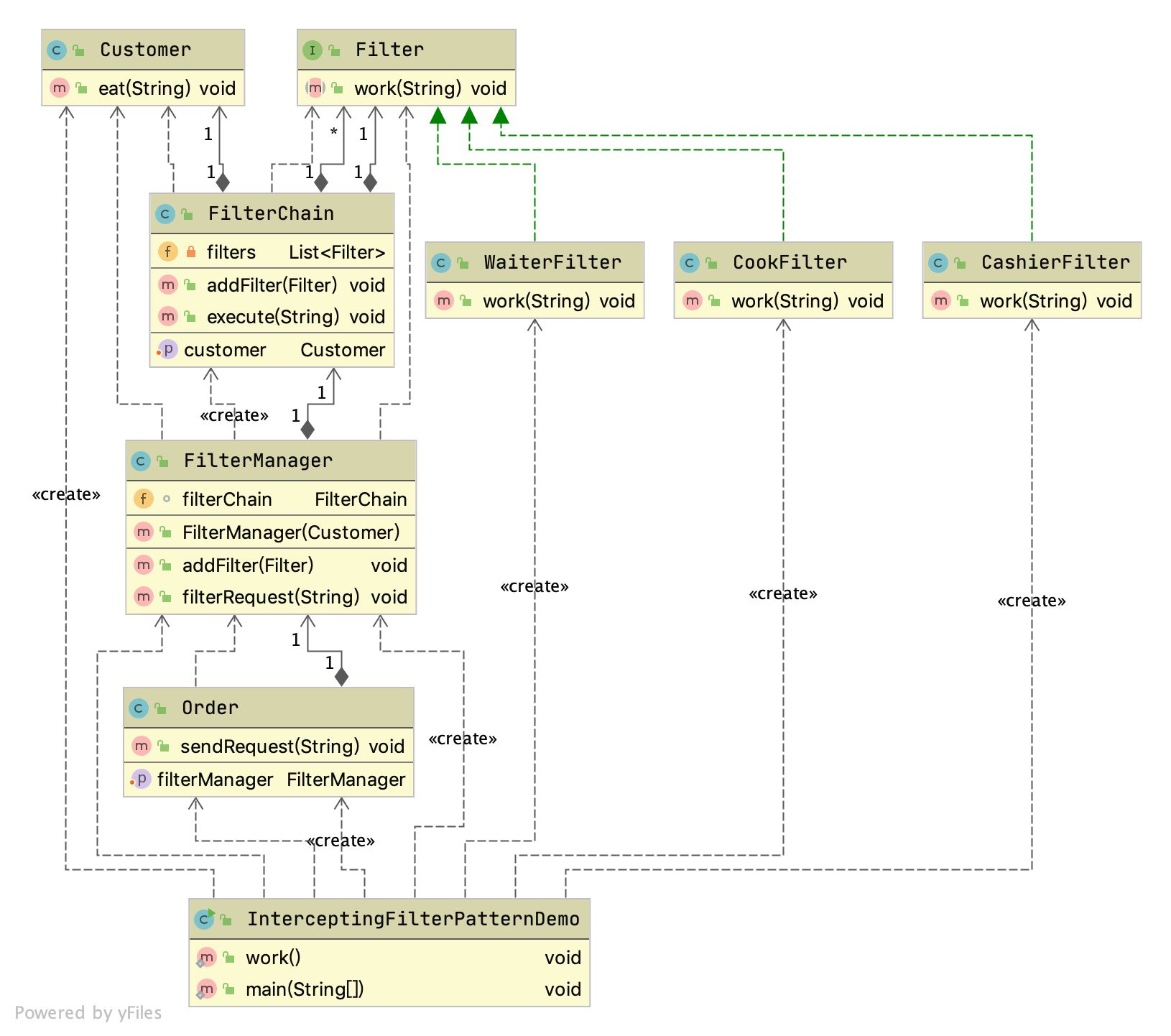
过滤器链（Filter Chain） - 过滤器链带有多个过滤器，并在 Target 上按照定义的顺序执行这些过滤器。

Target - Target 对象是请求处理程序。

过滤管理器（Filter Manager） - 过滤管理器管理过滤器和过滤器链。

客户端（Client） - Client 是向 Target 对象发送请求的对象。

3.6.2类图与时序图



3.6.3补充说明

我们将创建 *FilterChain*、*FilterManager*、*customer*、*order* 作为表示实体的各种对象。*Waiter*Filter，CookFilter 和 *Cash*ierFilter 表示实体过滤器用于实现log记录。

*InterceptingFilterDemo* 类使用 *order* 来演示拦截过滤器设计模式。

https://www.runoob.com/design-pattern/intercepting-filter-pattern.html

3.7备忘录模式

3.7.1API实现

在本例中，带入“海底嘉年华”场景，设计了一个为园区灯光进行调节的方法，具体而言，存储不同的灯光配色方法，并存储在备忘录中，可以对其进行增删，并根据需要回溯状态。

在本例中，构建了3个character，分别为Caretaker、Memento和originator，并利用一个Client来启动，其中根据Originator()方法创建了一个originator，再根据这个originator去创建新的Caretaker来实现备忘录模式。提供了三种方法，新建一个状态，回溯到某一状态，以及打印所有状态。

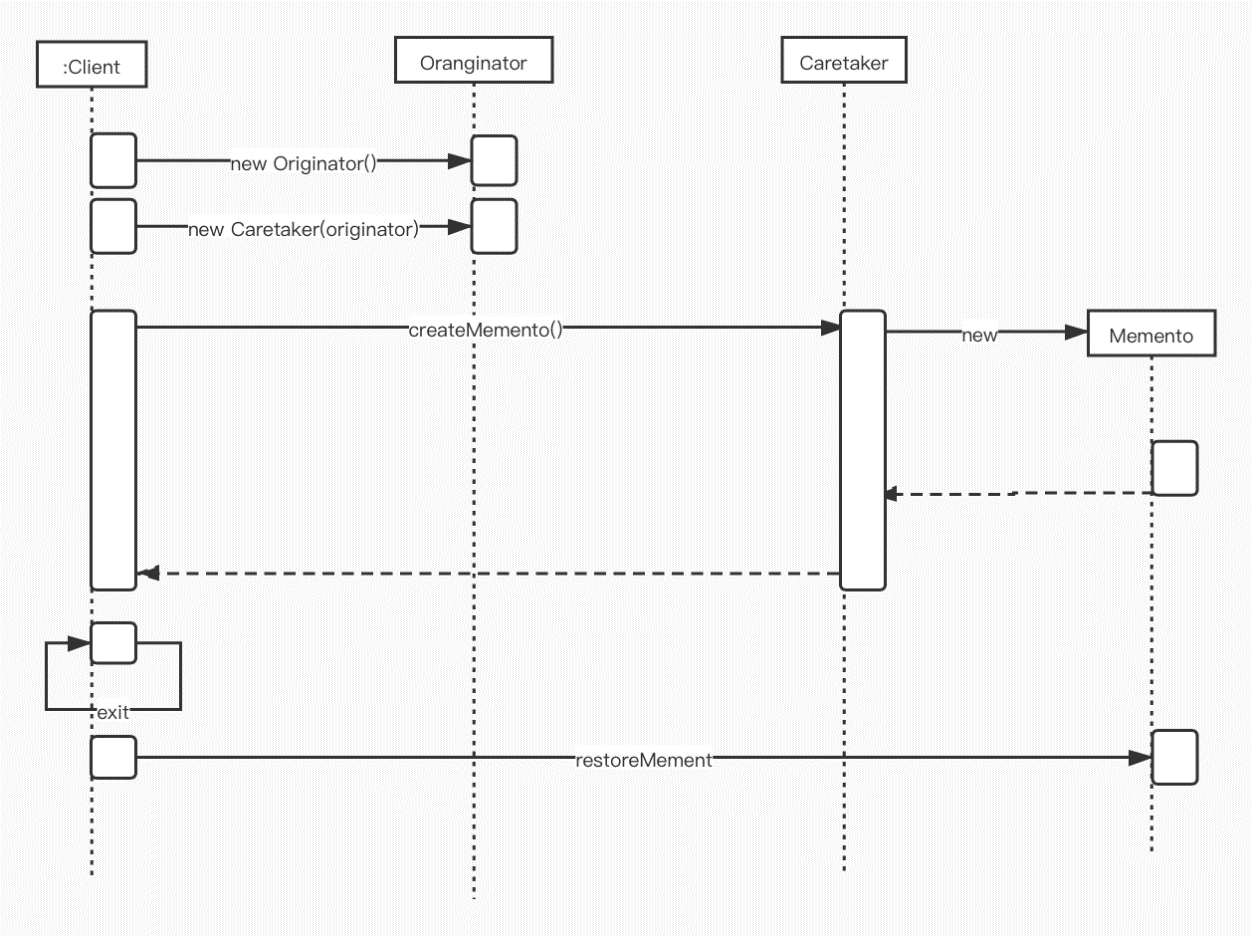
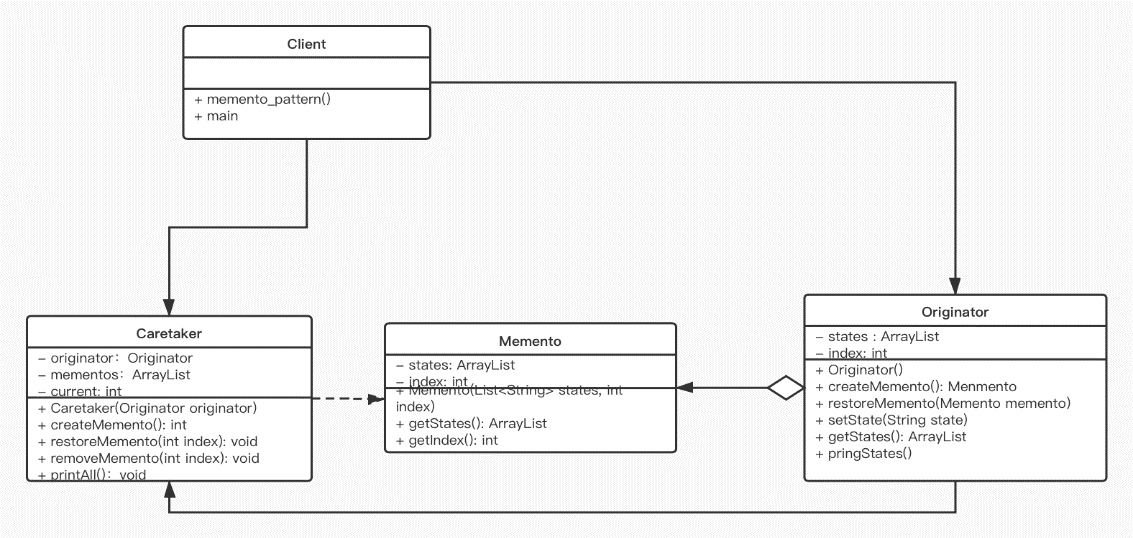
Originator原发器，生成自身状态的快照，并在需要时恢复某种状态。在本例中，实现了创建新的备忘录对象/恢复到备忘录记录对象的状态上，状态赋予实际参数的方法以及一个辅助的打印所有状态方法。

Memento备忘录，是原发器状态快照的值对象。在本例中通过构造函数，拷贝状态集合到新的集合中，保证原有集合变化不会影响到我们记录的值

Caretaker负责人，通过保存备忘录栈来记录原发器的历史状态。 当原发器需要回溯历史状态时， 负责人将备忘录传递给原发器的恢复 （restoreMemento） 方法。在本例中实现了创建新的备忘录，将状态回溯到某一备忘录节点，以及删除备忘录的方法。

最后在一个java类Client中创建了4个实例节点，并不断接受用户输入来测试。

3.7.2类图与时序图



3.7.3补充说明

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存着这个状态。这样以后就可将该对象恢复到原先保存的状态。很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，如在游戏设计相关的情境中，我们可以设置不同的节点来保存玩家的属性值，当玩家角色死亡重生后，可以回溯到他所需的节点，类似于存档的存在；除此之外，还有例子如windows系统下的Crtl+Z命令。

通过备忘录模式，我们可以给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态，以及实现了信息的封装，使得用户不需要关心状态的保存细节。但是备忘录模式也有其缺点，存储历史数据会占用较多的资源，如果成员变量过多，资源开销会很大。

在实现时，为了符合封闭开放原则，还要增加一个管理备忘录的类

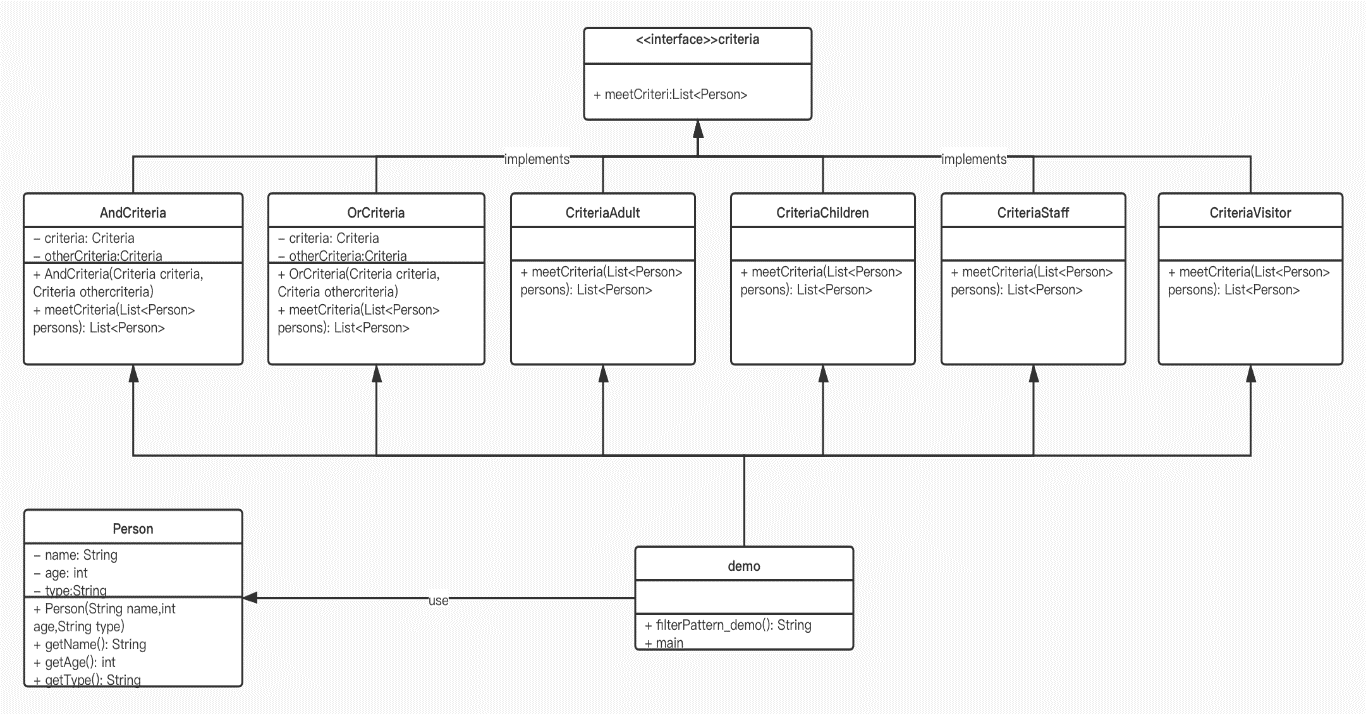
3.8过滤器模式

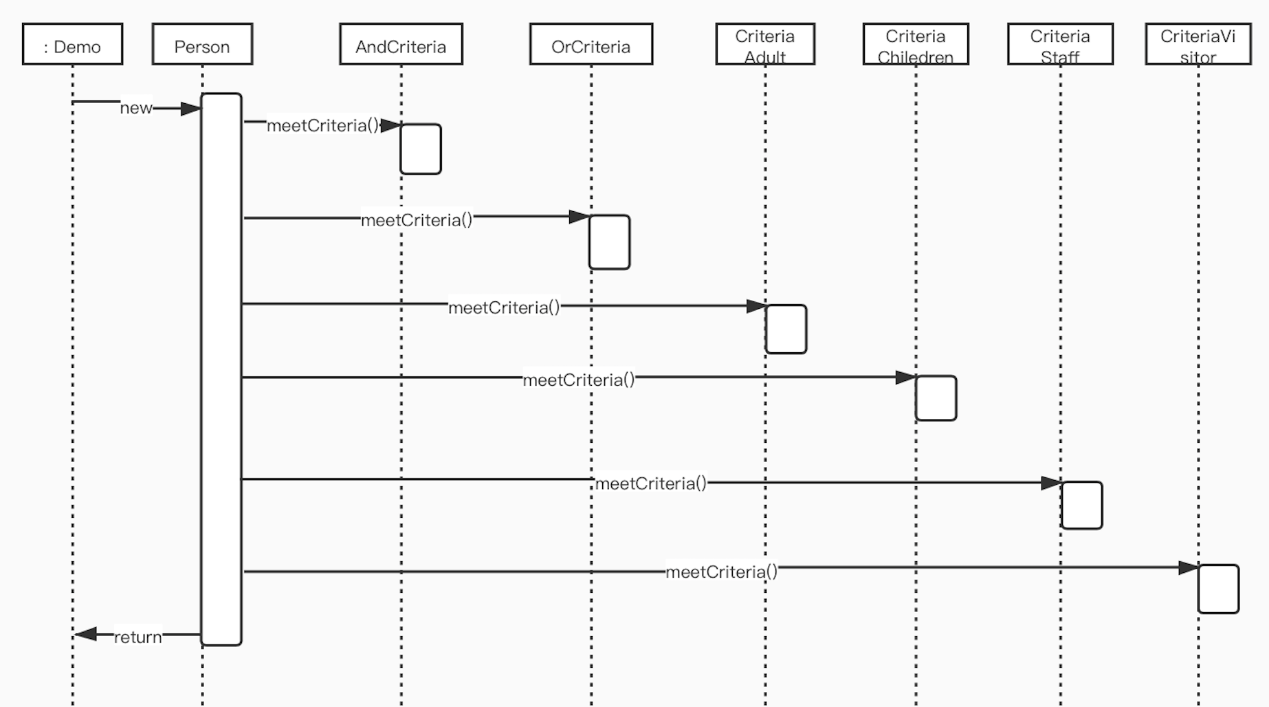
3.8.1API实现

在本例中，带入“海底嘉年华”场景，设计了一个对园区内所有人（模拟）进行统计分类的方法，可以统计所有的成年人、儿童、员工、游客，也可以在不修改源代码的情话下，增加统计类别，如增加统计“所有成年游客”，“所有未成年人或员工”（当然，此项应为空查询，不会返回任何结果，毕竟雇佣童工犯法）等，符合开放封闭原则。

在本例中，person有三个属性name，age，type分别代表人的姓名，年龄和人员分类（游客或员工），以及对应的get方法。在Criteria接口中，又一个meetCriteria方法，供后来实现的各类Criteria方法调用，如在CriteriaAdult中，实现了过滤出所有成年人的方法，AndCriteria则实现了过滤出某两种身份的方法，这个方法可以根据输入的不同Criteria来创建两个不同Criteria的and方法。

3.8.2类图与时序图





3.8.3补充说明

这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。这种类型的设计模式属于结构型模式，它结合多个标准来获得单一标准。

要实现此模式，首先需要一个Person对象，用于存放被过滤者的各类信息。以及Criteria接口，用于实现各种过滤模式，最后利用一个java类demo来实现功能验证。

3.9状态模式

3.9.1API实现

在本例中，带入“海底嘉年华”场景，设计了一个广场上的音乐喷泉，根据需求，前后切换音乐来改变喷泉表演方式，或者暂停、继续、开启关闭喷泉。

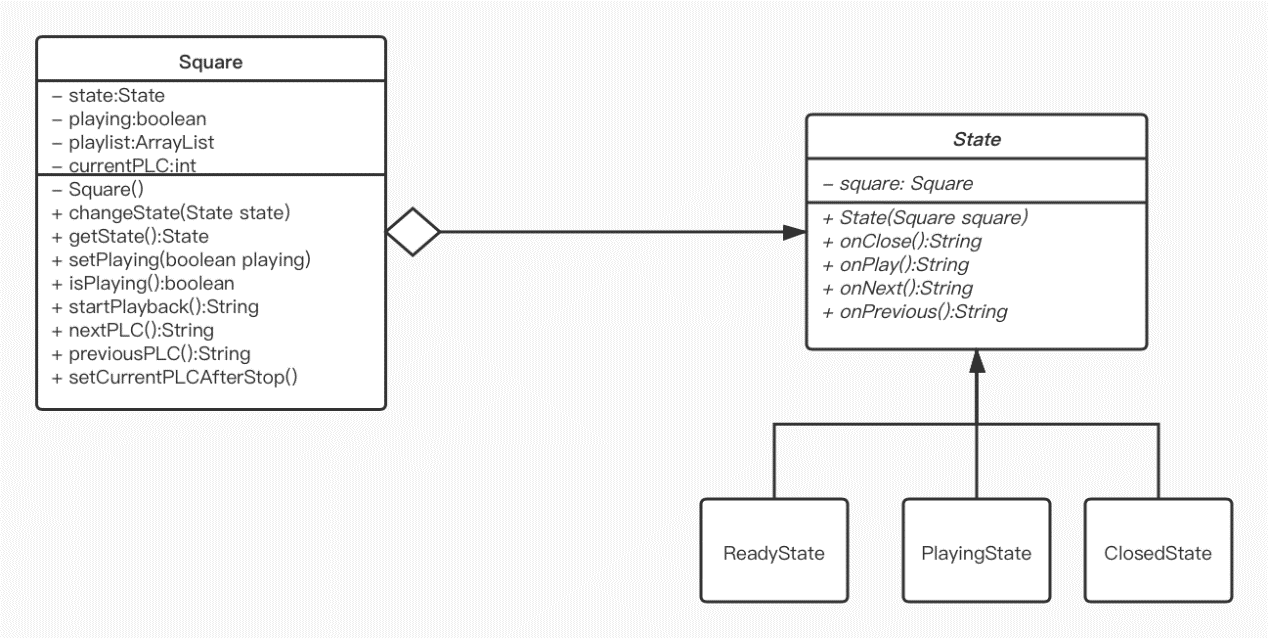
创建一个抽象类state，并实现不同的state继承这个抽象类，实现不同状态下的具体方法，在本例中实现了喷泉的三种状态切换。并创建一个实体对象，上下文Context，本例中创建了Square，来承载喷泉状态的不同方法。通过不同状态下调用上下文的方法，实现不同状态间的切换。

在State中，有四个抽象方法，onClose、onPlaying、onNext、onPrevious分别表示使喷泉停止，开始表演、下一种风格、上一种风格。

有三个不同子类继承他，分别是表演状态、暂停状态和关闭状态，对应状态机中三种不同的状态。通过继承父类State的方法来改变其状态。

在Square中实现了上下文的功能，为外界提供对这个“喷泉”进行操作的方法。

3.9.2类图与时序图



3.9.3补充说明

状态模式是一种行为设计模式， 让你能在一个对象的内部状态变化时改变其行为， 使其看上去就像改变了自身所属的类一样。可以将状态模式类比于有限自动机，其主要思想是程序在任意时刻仅可处于几种有限的状态中。 在任何一个特定状态中， 程序的行为都不相同， 且可瞬间从一个状态切换到另一个状态。 不过， 根据当前状态， 程序可能会切换到另外一种状态， 也可能会保持当前状态不变。 这些数量有限且预先定义的状态切换规则被称为转移。

状态模式建议为对象的所有可能状态新建一个类， 然后将所有状态的对应行为抽取到这些类中。原始对象被称为上下文， 它并不会自行实现所有行为， 而是会保存一个指向表示当前状态的状态对象的引用， 且将所有与状态相关的工作委派给该对象。如需将上下文转换为另外一种状态， 则需将当前活动的状态对象替换为另外一个代表新状态的对象。

3.10访问者模式

3.10.1API实现

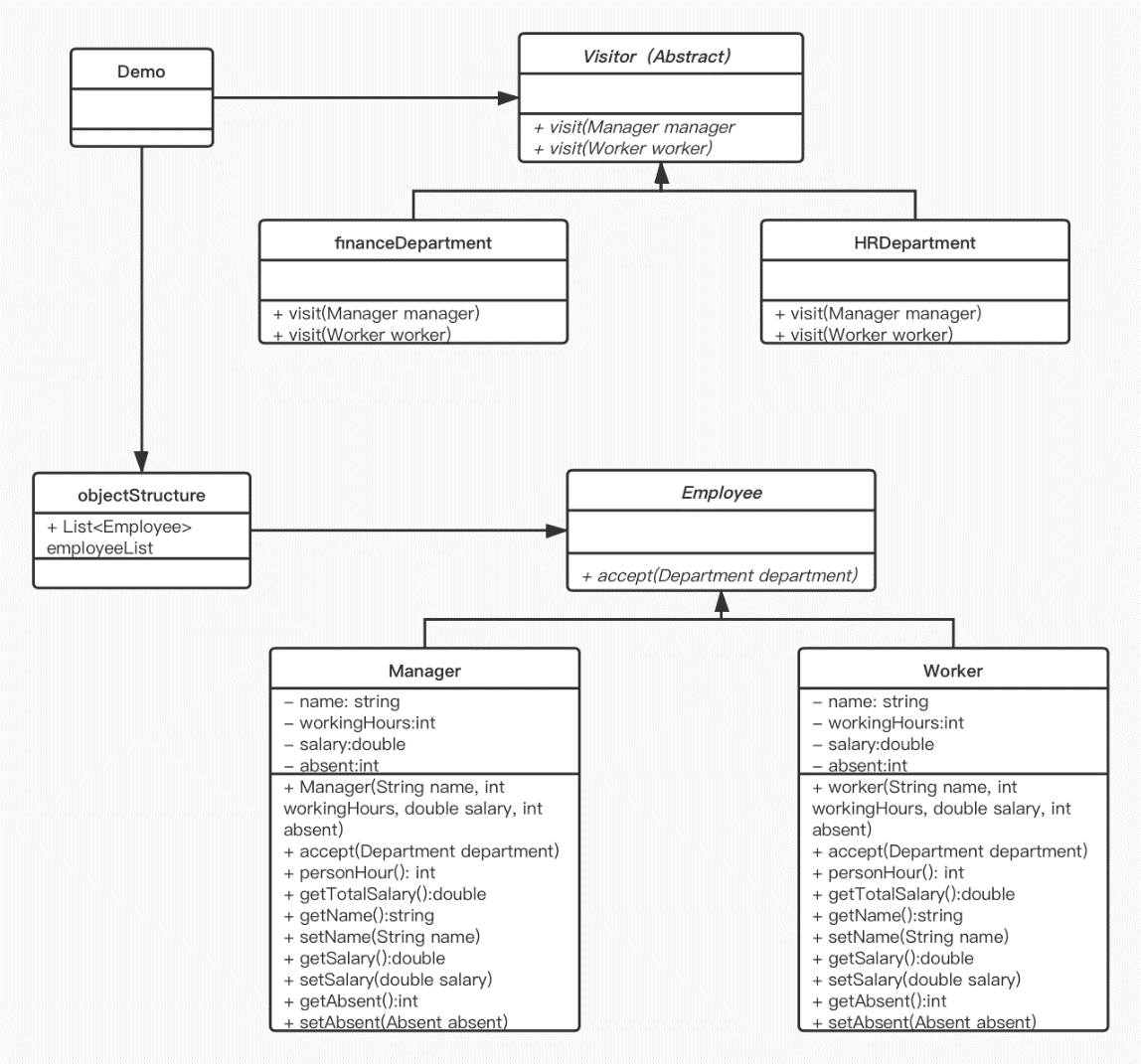
在本例中，带入“海底嘉年华”场景，设计了一个公司不同部门对不同类别员工查询报表的方式，模拟了人事部和财务部查询报表，员工类型设置了经理和普通职员两类，意图在不同部门查询员工时，返回的报表应不相同，且不同员工的实现方法不同。

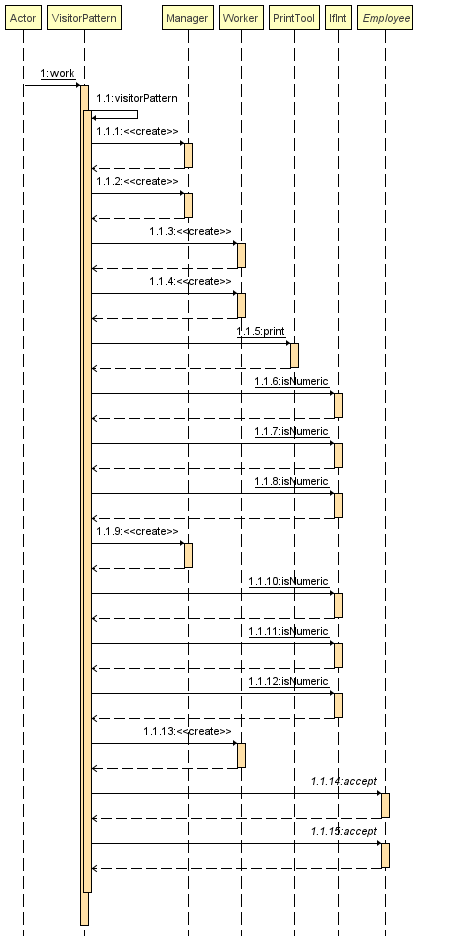
首先是Visitor，在本例中对应不同的部门。创建一个Visitor接口“Department”，使得Hr部门和财务部门可以继承它的方法。不同的部visitor（即模拟查询报表）的方式不同，且对不同员工的visitor方法大体上相同，只是根据员工分类，查询对象不同。

其次是Element，在本例中对应不同类型的员工。创建了Employee接口，使得不同的员工能继承其方法，来实现对不同员工的各项属性的操作。在本例中有name、workingHours、salary以及absent这四个属性，分别代表员工姓名、每个月的工作时长、薪水（按小时算）以及缺勤时长（按小时算）。

最后在Demo中创建员工对象，对不同对象进行不同报表的查询。用户也可以自定义员工。

3.10.2类图与时序图





3.10.3补充说明

使用一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。这种类型的设计模式属于行为型模式。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

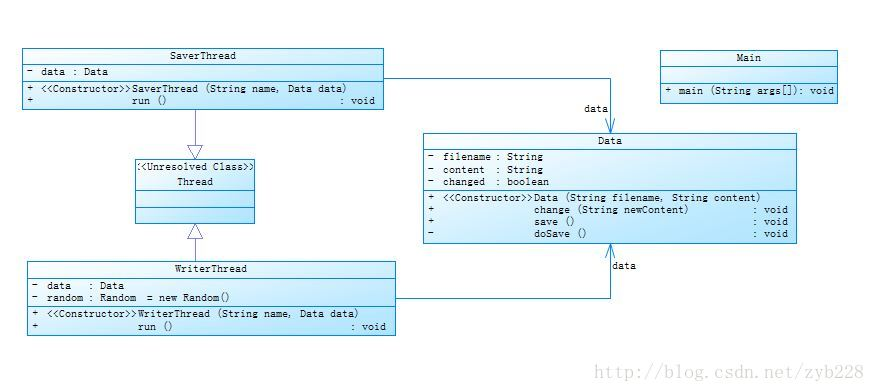
利用访问者模式的主要意图是，将数据结构与数据操作分离。需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作”污染”这些对象的类，使用访问者模式将这些封装到类中。如在公司的不同部门，检查其员工月报表时，看到的应该是不同的报表。

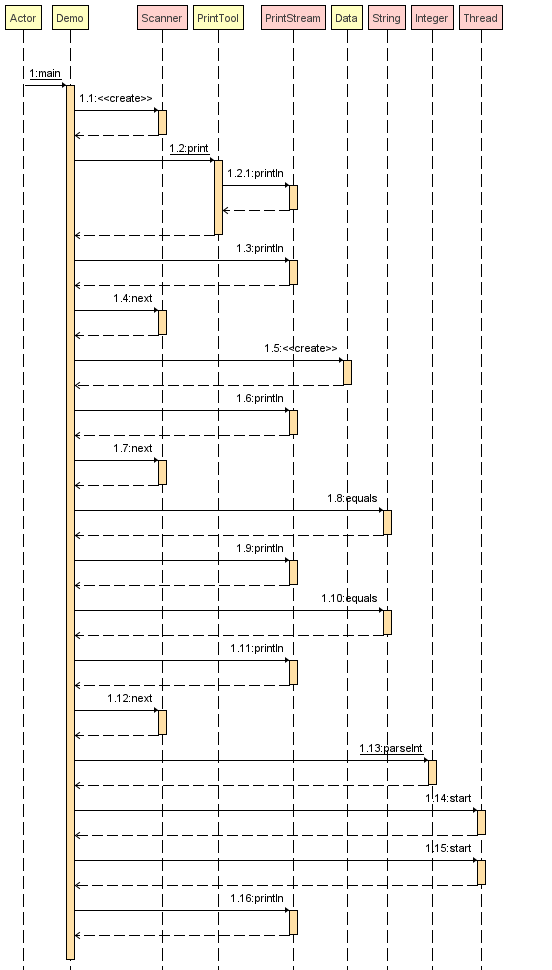
3.11犹豫模式

3.11.1API实现

在这个模式中，实现了对于文件的修改保存。通过save与change类来改变文件的状态。根据用户的输入来创建或者修改以及保存。

3.11.2类图与时序图





3.11.3补充说明

如果现在不合适执行这个操作，或者没必要执行这个操作，就停止处理，直接返回。在Balking模式中，如果守护条件不成立，就立即中断处理。

定期将当前数据内容写入文件中，比如文本工具的自动保存功能，定期的将数据保存到文件中。当数据内容被写入时，会完全覆盖上次写入的内容，只有最新的内容才会被保存；当写入的内容和上次的内容完全相同时，再向文件写入就多余了，所以就不再执行写入操作。所以这个程序就是以 数据内容不同 作为守护条件，如果数据内容相同，就不执行写入操作，直接返回(balk)。

3.11.4参考文献

https://blog.csdn.net/qq\_35211818/article/details/104186567

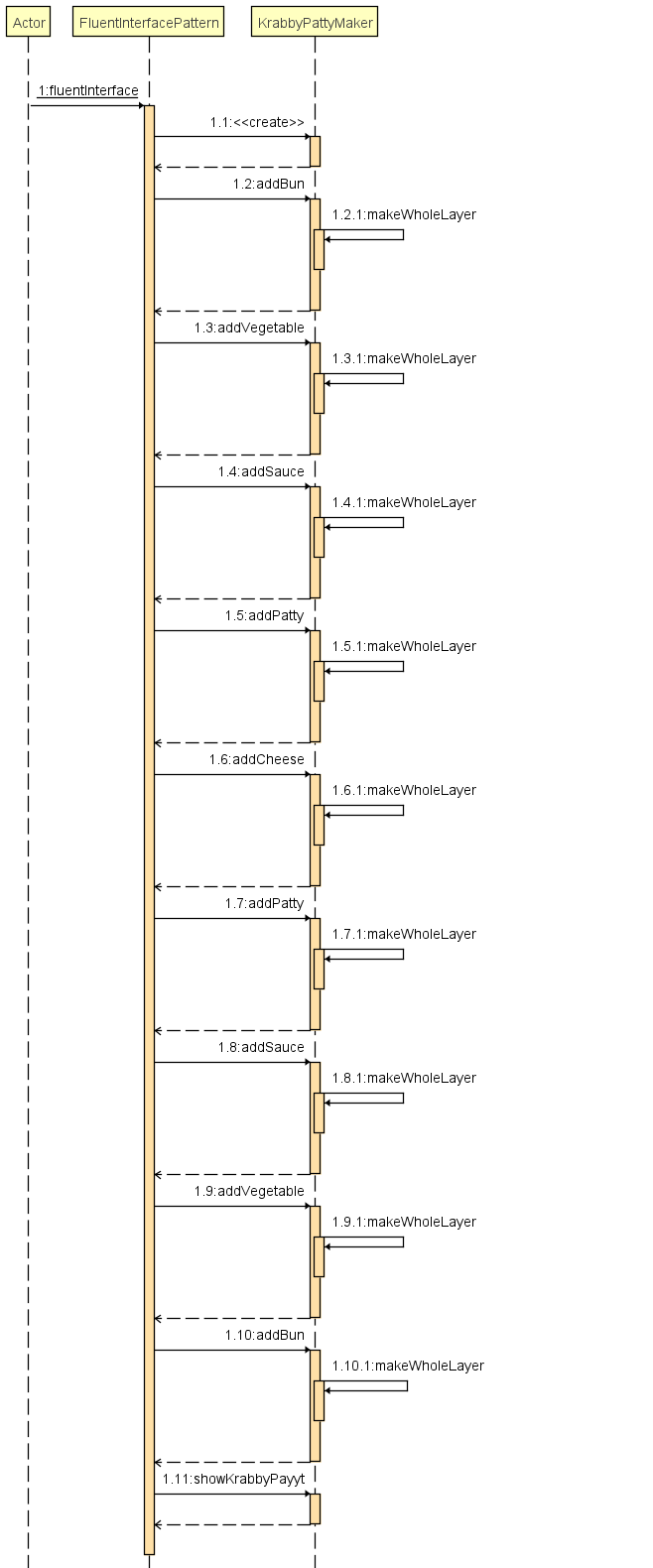
3.12流式接口模式

3.12.1API实现

海绵宝宝制作蟹黄堡，一个蟹黄堡由许多“层”组成，在海绵宝宝每给蟹黄堡增加一层时，方法会返回这个蟹黄堡对象本身，从而可以实现流式地调用方法。

3.12.2类图与时序图





3.12.3补充说明

类的方法返回类的本身，使得调用者能够继续调用类方法。常用于set函数等具有赋值功能的方法。

3.12.4 参考文献

<https://martinfowler.com/bliki/FluentInterface.html>

<https://www.cnblogs.com/zhengshiqiang47/p/5712959.html>

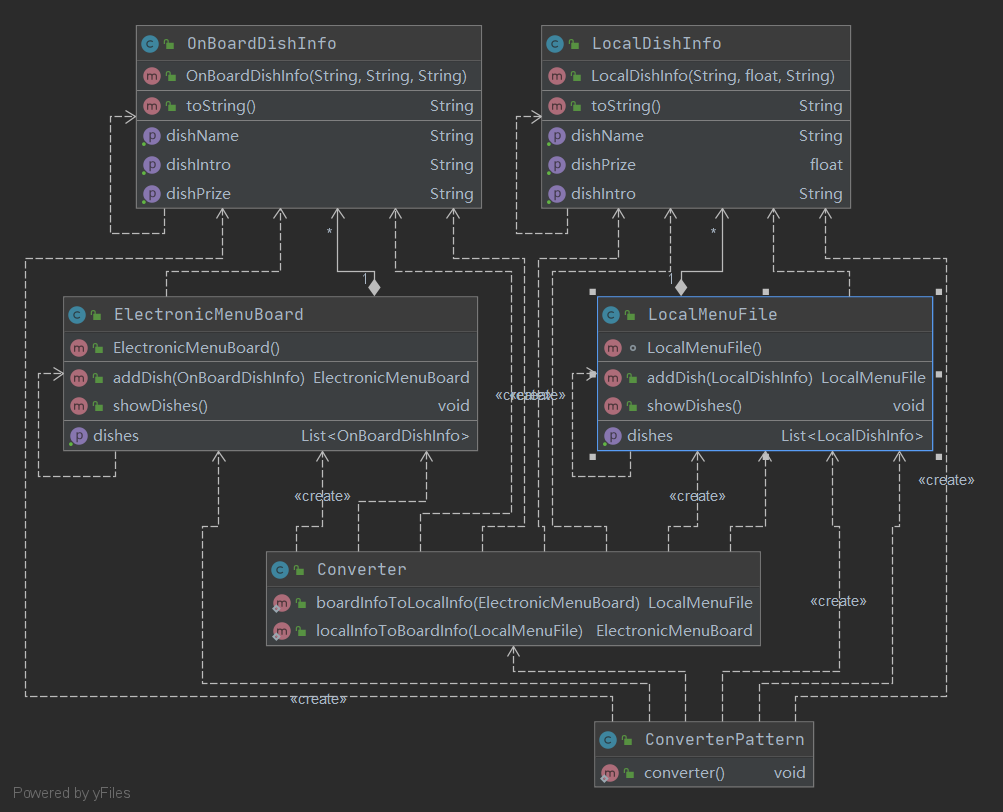
3.13转换器模式

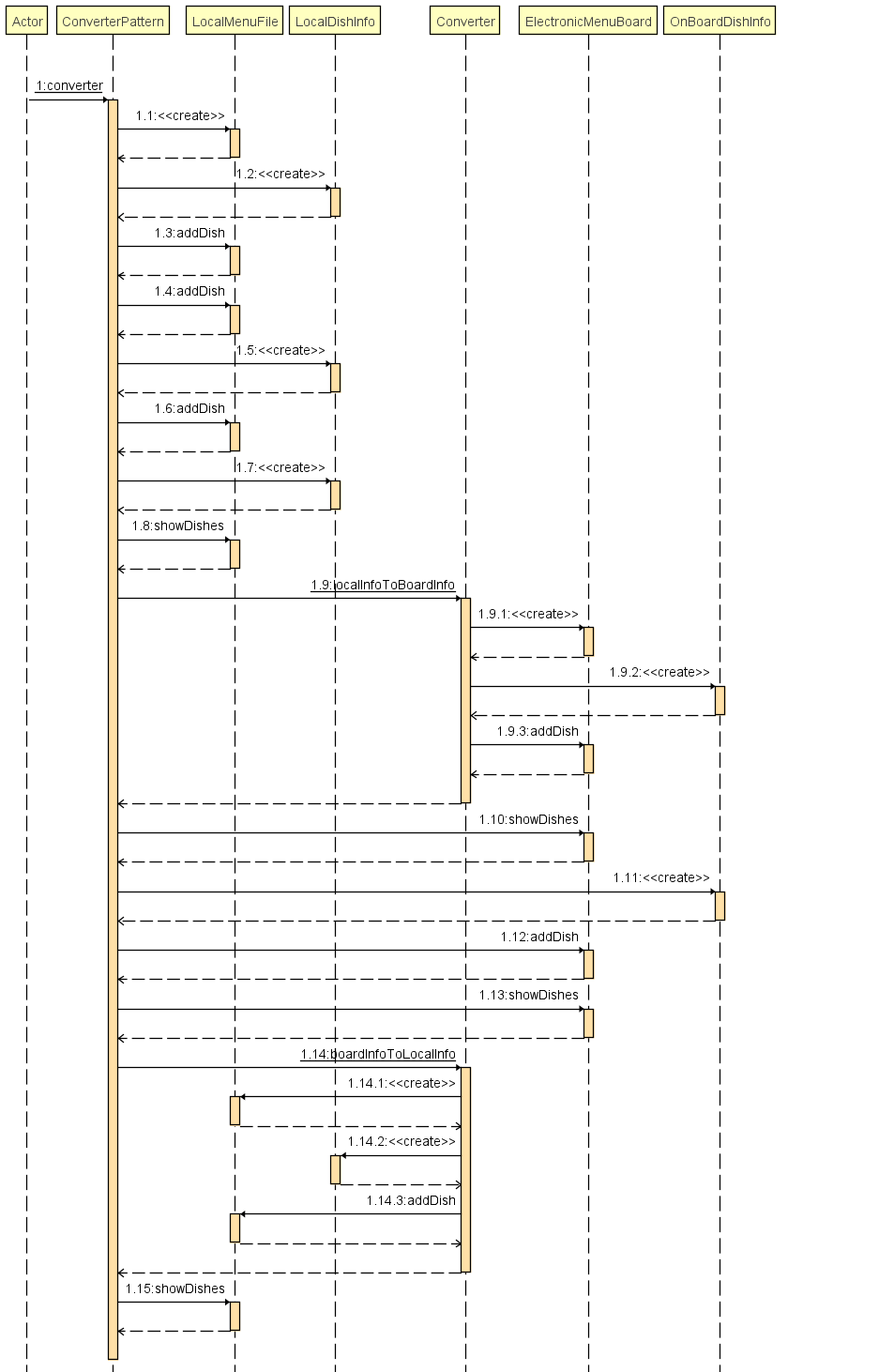
3.13.1API实现

在蟹老板的PC上保存着本地菜单文件，如果要将其上传到电子菜单板上就需要将其转换为电子菜单板专用的菜单信息格式。同时从电子菜单板上下载菜单信息到蟹老板PC上时需要再将其转换为PC能够解析的文件格式。蟹老板的PC和电子菜单板都不需要实现信息转换的过程，因为这个转换的步骤由Converter类来实现。

该API展示了一个蟹老板上传菜单后再下载的过程。

3.13.2类图与时序图





3.13.3补充说明

转换器设计模式的目的是为相应类型之间的双向转换提供一种通用的方式，允许类型无需彼此了解的简洁的实现。此外，转换器设计模式引入了双向收集映射，将样板代码减少到最小。

转换器设计模式是一种行为设计模式，允许在相应类型（如DTO和逻辑同构类型的域表示）之间进行双向转换。此外，该模式还引入了一种在类型之间转换对象集合的通用方法。

3.13.4 参考文献

<https://www.cnblogs.com/liaobudao/p/11266369.html>

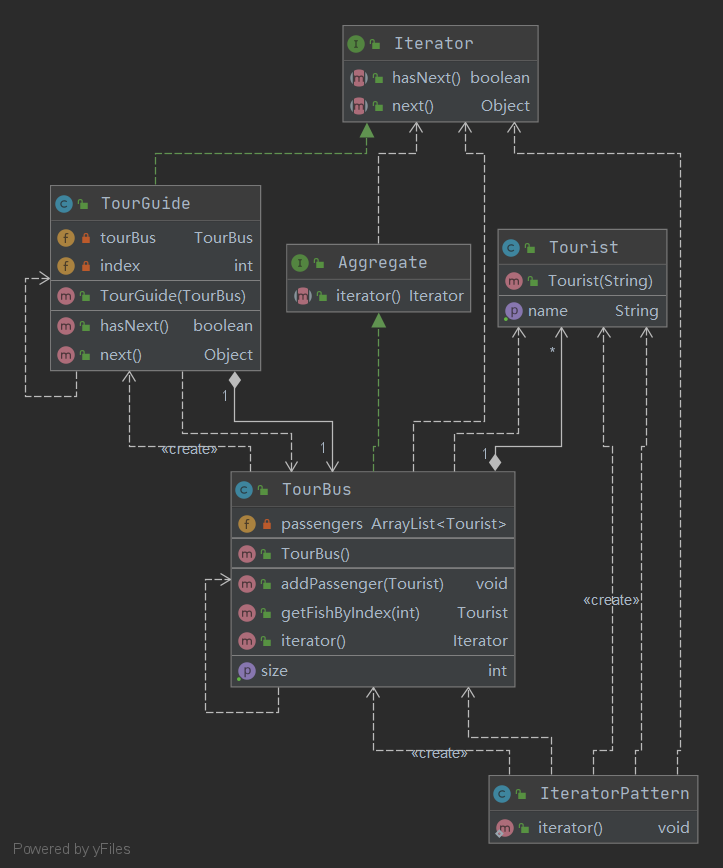
<https://www.jdon.com/52040>

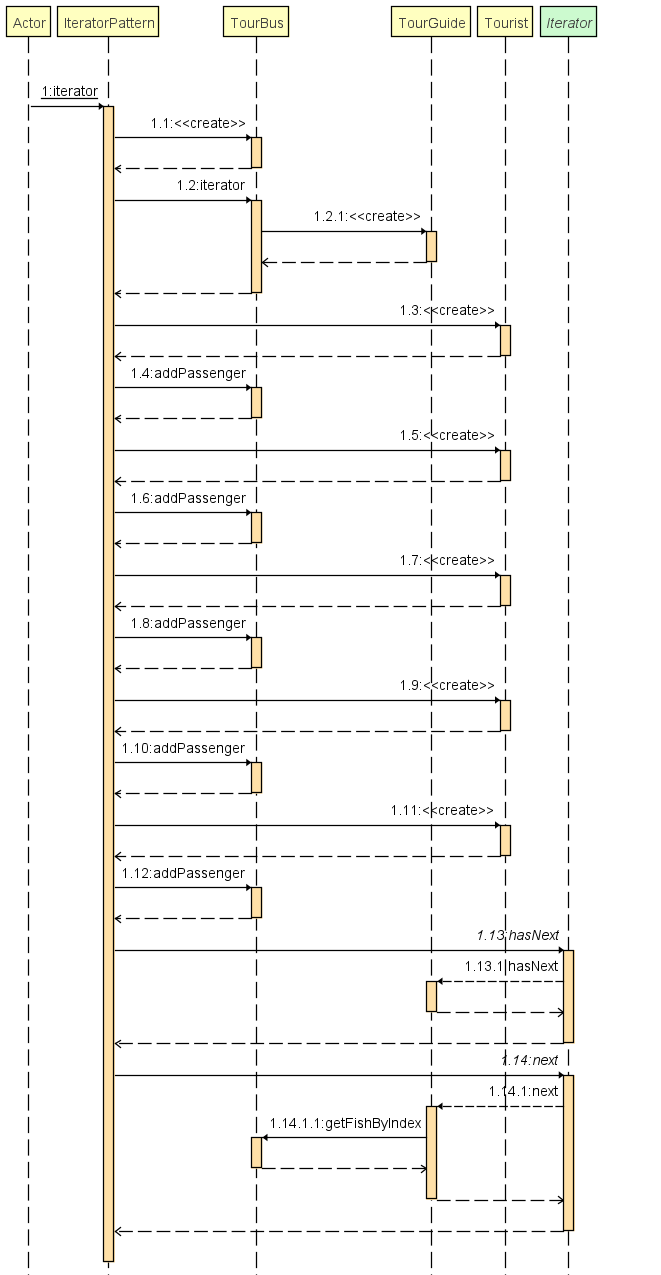
3.14迭代子模式

3.14.1API实现

旅游大巴车上有若干乘客和一个导游，在出发前，导游需要根据乘客名单点名，此处采用迭代器的方式依次遍历乘客名单，每一位乘客都在车上之后，大巴车驶向海底世界狂欢节活动现场。

3.14.2类图与时序图



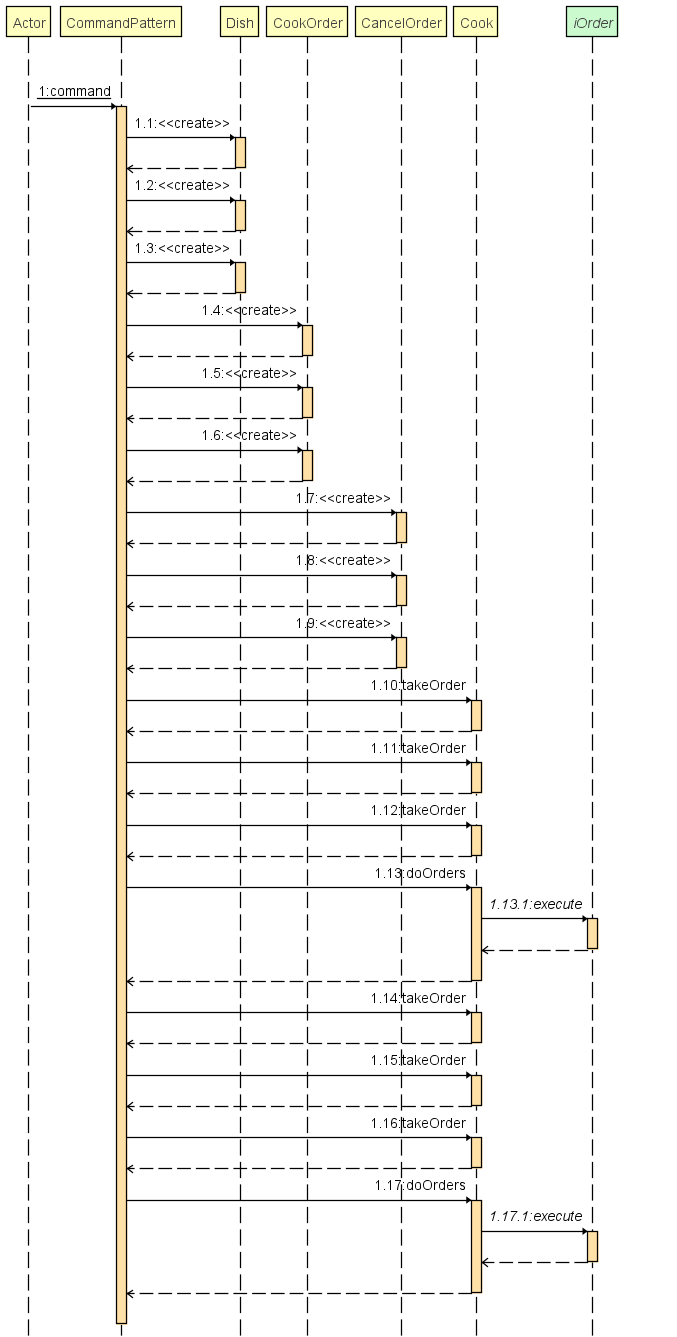
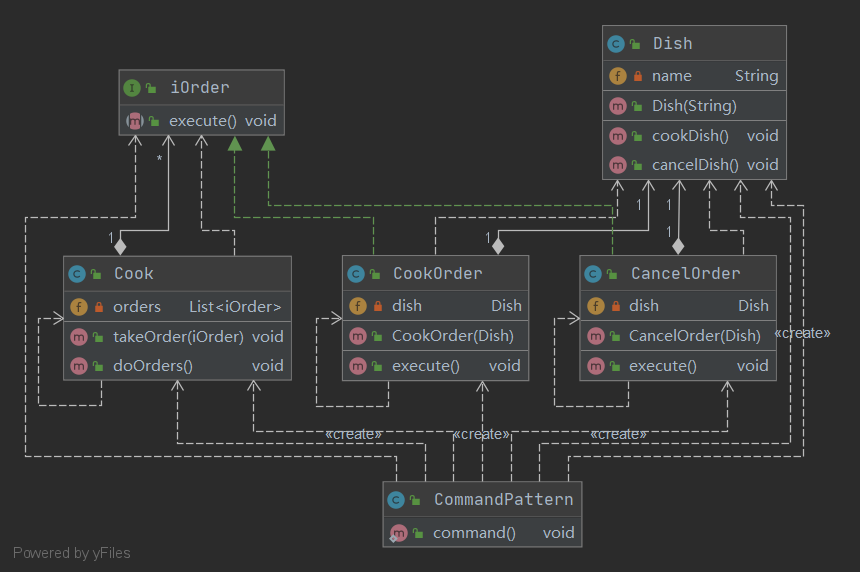


3.15命令模式

3.15.1API实现

在海底世界狂欢节的快餐店蟹堡王里，有各种各样的快餐食物，顾客可以点这些食物也可以取消之前的点单。当积累一些订单之后，海绵宝宝会根据订单生成顺序依次处理全部订单。

3.15.2类图与时序图

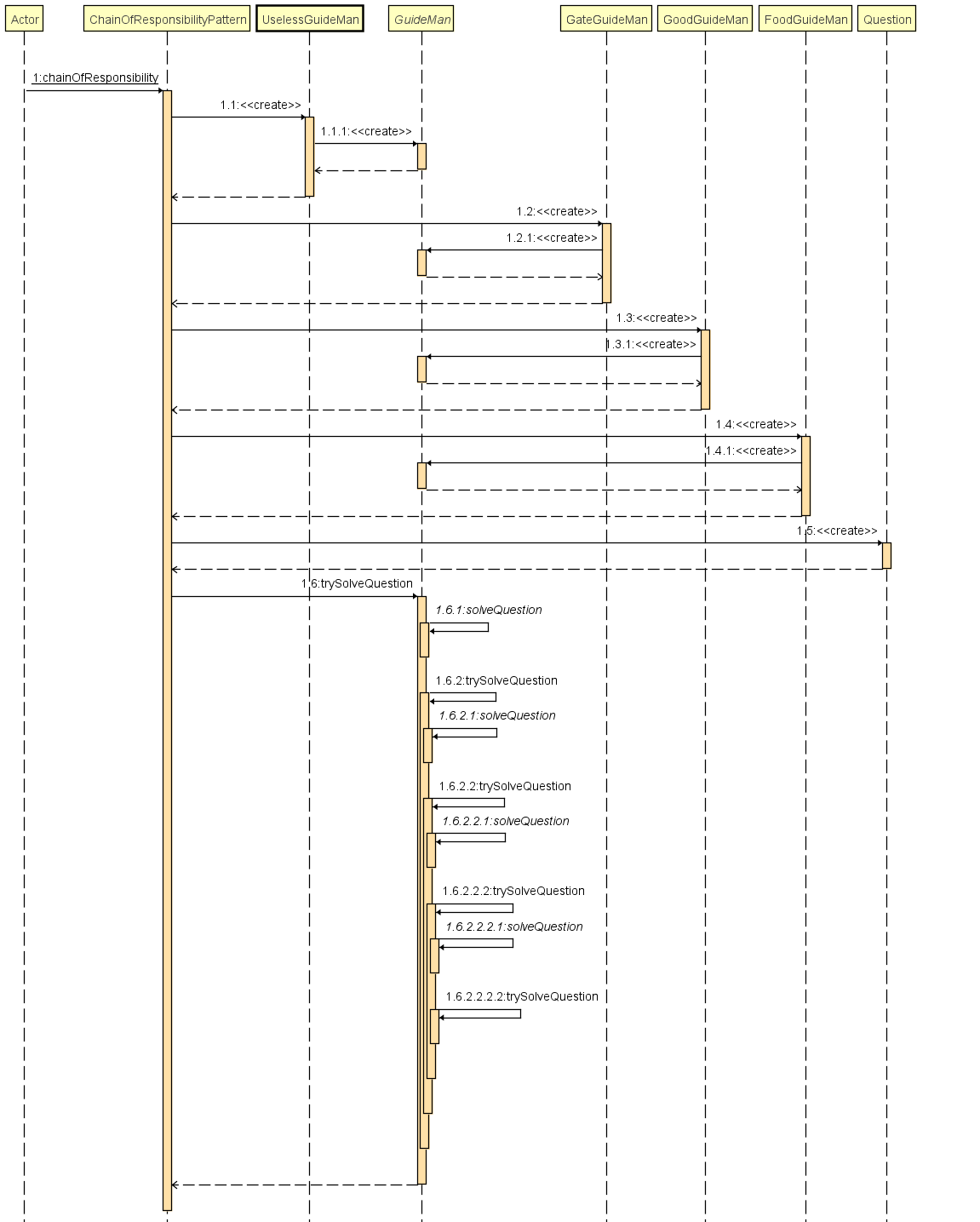
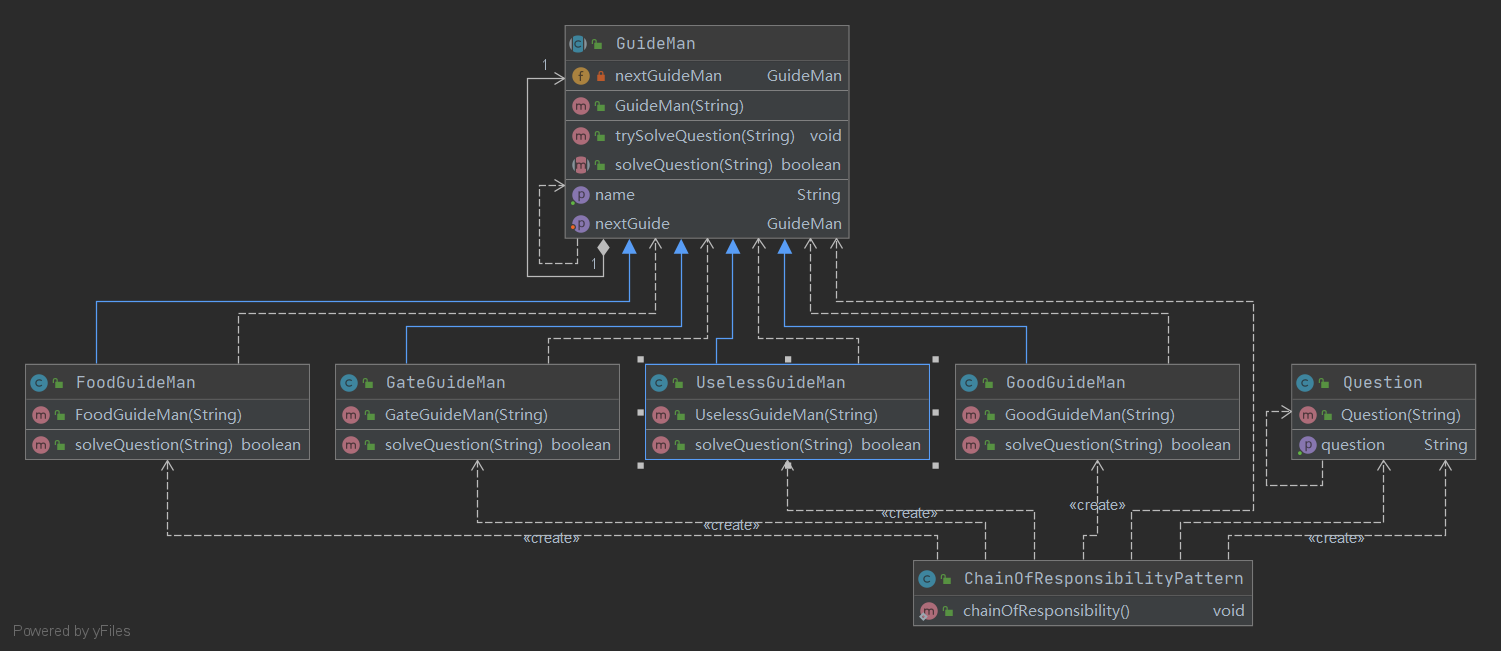


3.16责任链模式

3.16.1API实现

海底世界狂欢节里有一些志愿者，这些志愿者根据自身条件的不同可以回答不同的问题。游客可以找到这些志愿者并询问自己想问的问题。如果被询问的志愿者可以解决问题，那么这个问题将被回答，否则这个志愿者将推荐游客去问下一个志愿者，直到问题被解决或没有志愿者可以解决问题

3.16.2类图与时序图

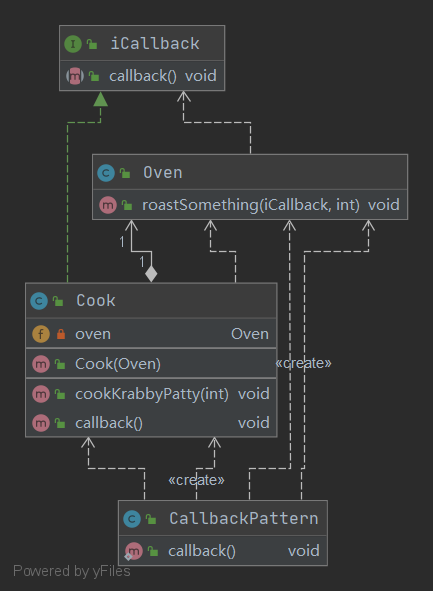


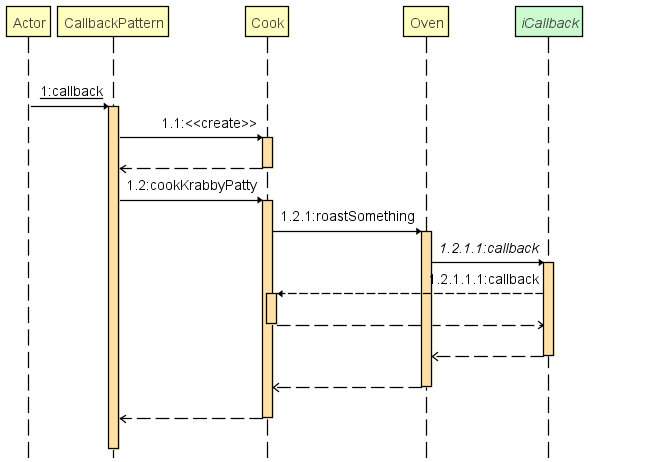
3.17回调模式

3.17.1API实现

蟹堡王的厨师海绵宝宝需要制作蟹黄堡，他首先需要用烤箱烘烤蟹黄堡，将蟹黄堡交给烤箱烘烤后海绵宝宝需要等待烤箱烘烤完毕。烤箱烘烤完毕后会提醒海绵宝宝，这时候海绵宝宝就会取回蟹黄堡并进行后续的步骤。

3.17.2类图与时序图





3.17.3补充说明

<https://blog.csdn.net/c275046758/article/details/51509218>

<https://www.jianshu.com/p/f1c334fb5afb>

3.18装饰器模式

3.18.1API实现

考虑到在主题为海底世界狂欢节，在主题乐园中显然会需要不停的添加许多新的或者重复的装饰品。因此考虑使用装饰器模式，设计一个能为特定场景/活动添加装饰物的类。

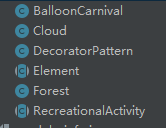


图1 装饰器模式案例结构

首先简单使用了template method的思想构造了代表某种娱乐活动的抽象类RecreationalActivity并给出了它的一个具体实现——气球嘉年华BalloonCarnival。同时设计其子类（抽象类）element代表装饰物，并给出两个具体的抽象类cloud和forest。

当工作人员想向气球嘉年华中添加某种装饰物时，只需要在以该类为参数构建新的cloud类或者forest类即可。同时可以进行多次操作，即可以为一个活动添加多个装饰物。

3.18.2基本类图/实例类图/时序图

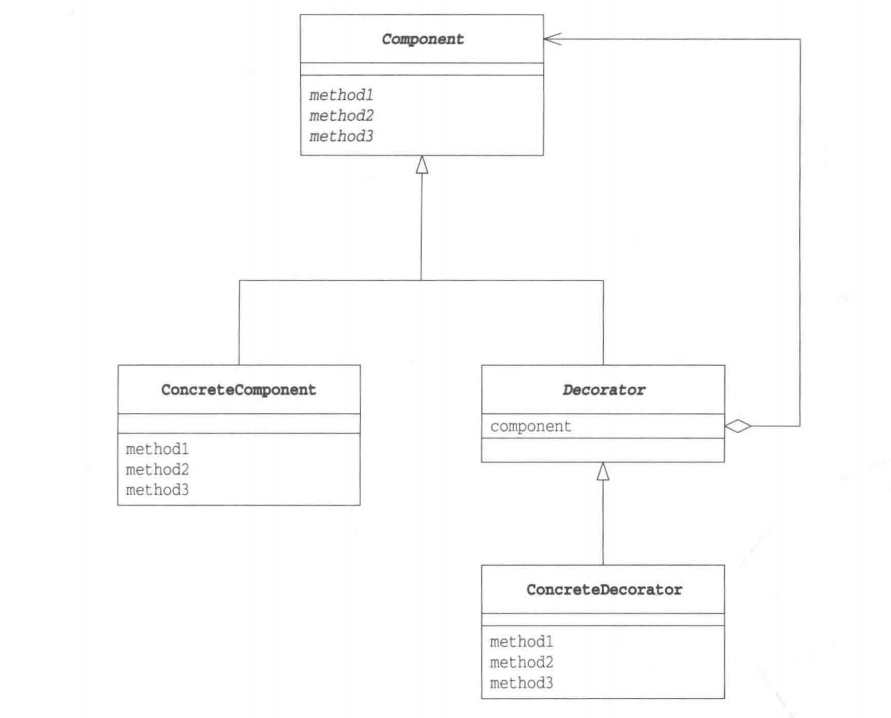


图 2 代理模式基本类图

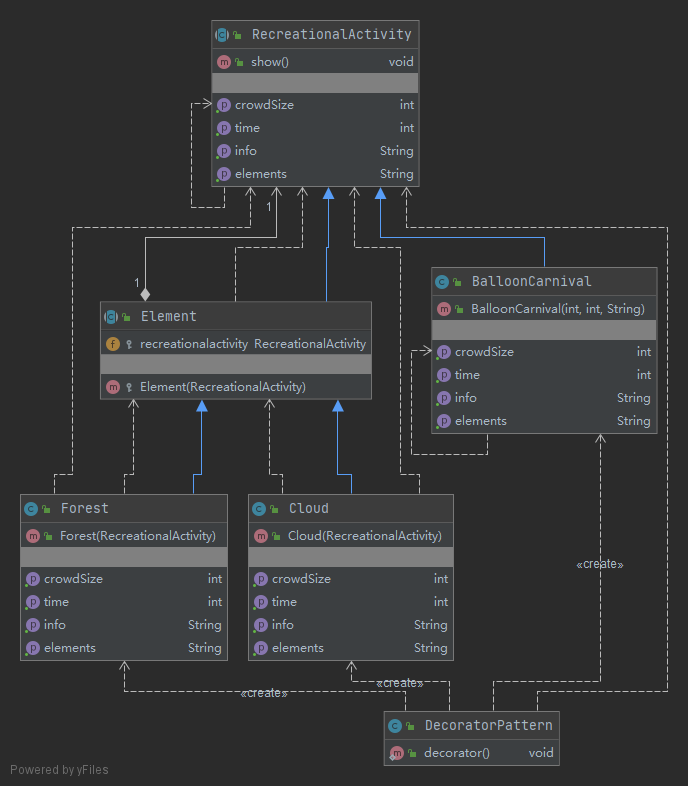
****

图 3 装饰器模式案例类图

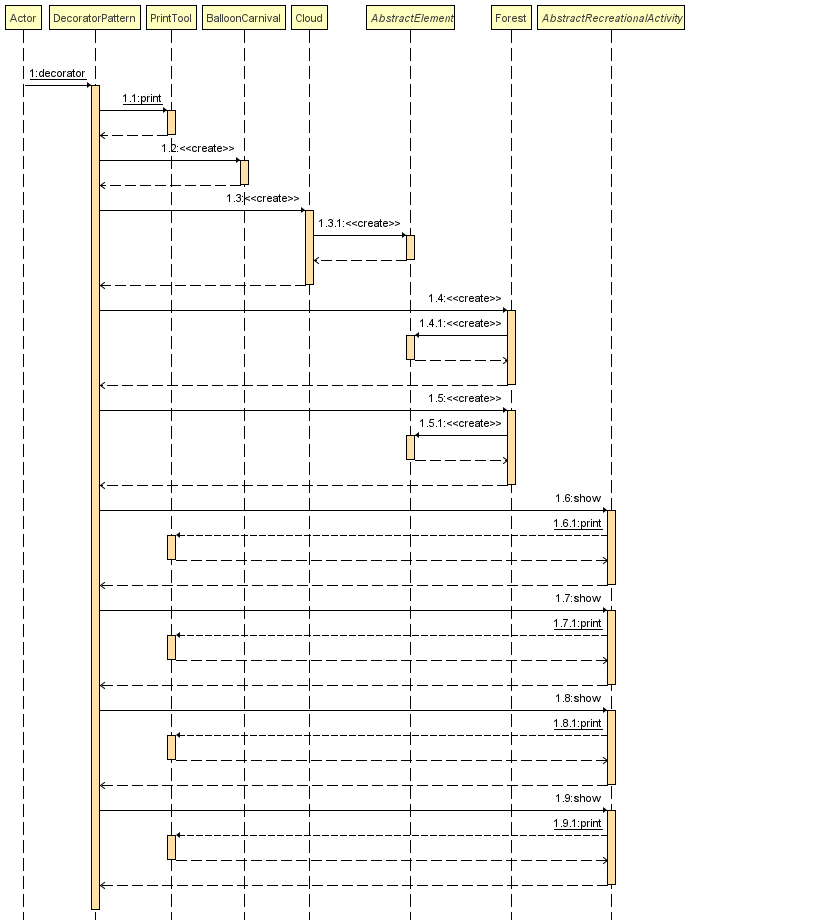


图 4 装饰器模式案例时序图

3.18.3补充说明

装饰器模式(Decorator Pattern)是用来向被装饰物不断添加装饰物的一个模式。程序就类似装饰器，可以通过这个模式向其中不断添加新的功能，使其使用目的更加明确。这样能做到在不改变原程序本身的基础上增加新的功能。

3.19外观模式

3.19.1API实现

考虑到主题海底世界狂欢节中，显然会有很多活动需要使用大型设备进行视频、音频或者其他多媒体形式的播放，且与其需求相匹配的播放设备（荧幕、音响、灯光等等）一般都规模较大，操作较复杂，我设计了一个用于简化多媒体播放流程的类。

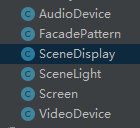


图 5 外观模式案例结构

我将所有所需设备分为了屏幕(Screen)、视频播放设备(VideoDevice)、音频播放设备(AudioDevice)、场景灯光(SceneLight)四个部分。每个类提供了相关设备的启动、运行、停止等功能；并设计了SceneDisplay类来管理所有设备，将相关设备的相应方法在指定阶段调用，从而将播放视频到停止的整个过程中对所有设备的所有操作分成了三个阶段：启动、运行、停止。

在某个场景需要进行视频播放时，后台人员只需要调用这三个阶段的统一接口即可，无需再对单一设备进行操作，避免了冗余操作。

3.19.2基本类图/实例类图/时序图

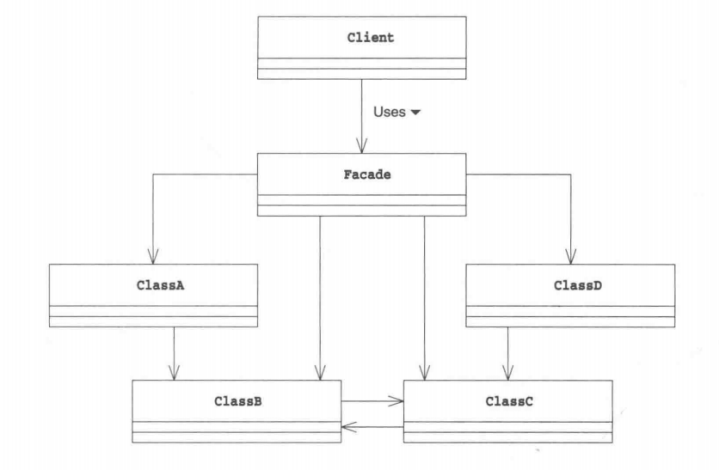


图 6 外观模式基本类图

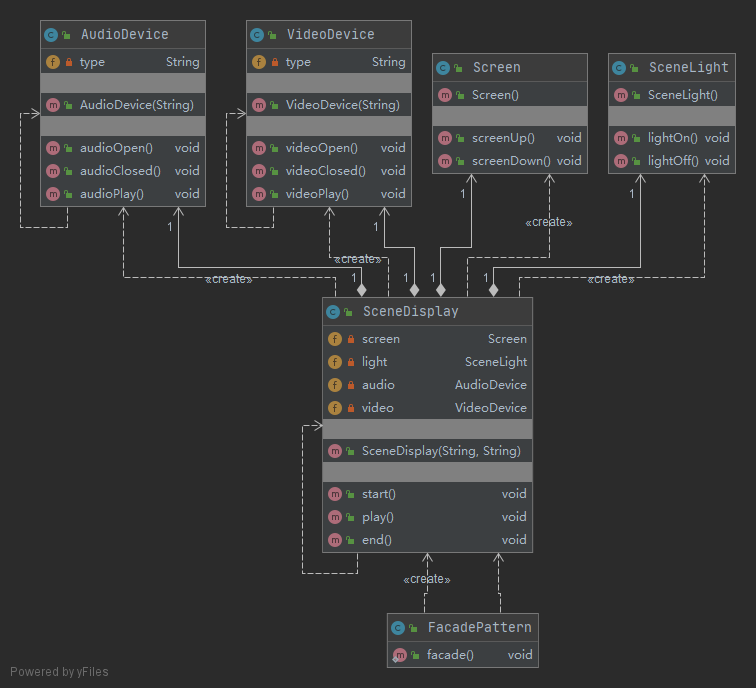


图 7 外观模式案例类图

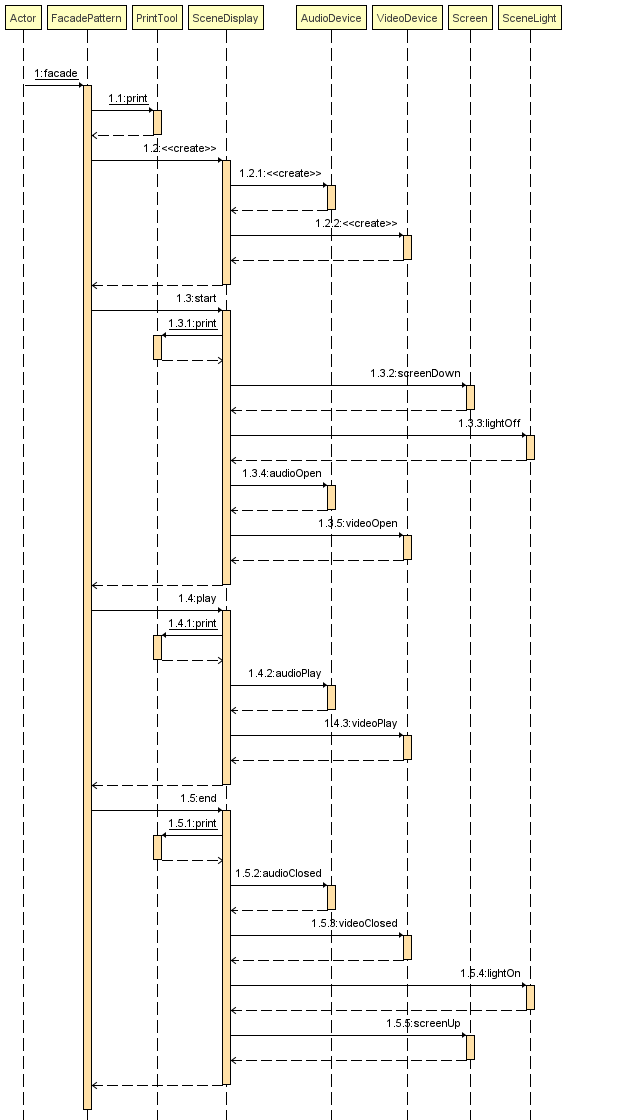


图 8 外观模式时序图

3.19.3补充说明

随着时间的推移，一个程序总是会越变越大，其中的类会越来越多，之间的相互关联会变得越来越繁琐与复杂。很典型的一个例子就是java本身的jar hell，过多的包之间复杂的依赖关系让所有使用者都感到头疼，这最终促使了模块化的产生。

在设计模式中，外观模式的作用是类似的，它最直观的作用就是显著的减少接口的数量，因此用户在使用时无需考虑内部功能的选择，只需调用提供的集成化的少数接口（高层接口）即可。这也就意味着程序本身与外部的关联关系弱化了，这样的包更容易被复用，调用。

3.20 代理模式

3.20.1API实现

考虑到主题为海底世界狂欢节，很可能会存在多个展馆或是参观、游玩的部分。而每一部分都需要购买相应的门票，同时存在总票，可以适配所有场馆。由此设计了售票处的相应类(TicketOffice)。

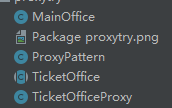


图 9 代理模式案例结构

当用户需要购买某个类型的票时，如食物票，观光票等等，则只需要在当前区域的售票代理处(TicketPfficeProxy)购买适用于当前区域的票即可；但若用户需要购买全票，则需要在主售票厅(MainOffice)进行购买

3.20.2类图/实例类图/时序图

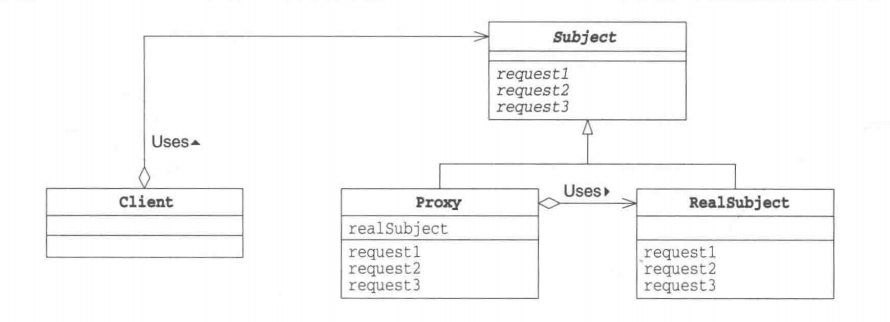


图 10 代理模式基本类图

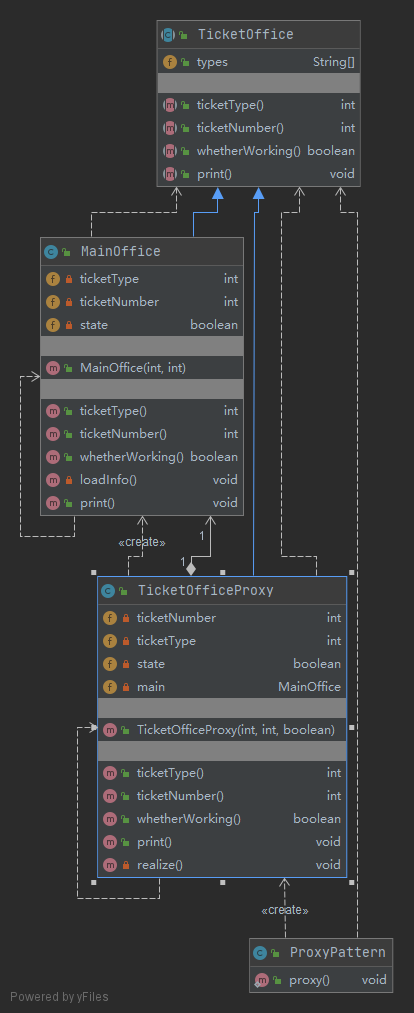


图 12 代理模式案例类图

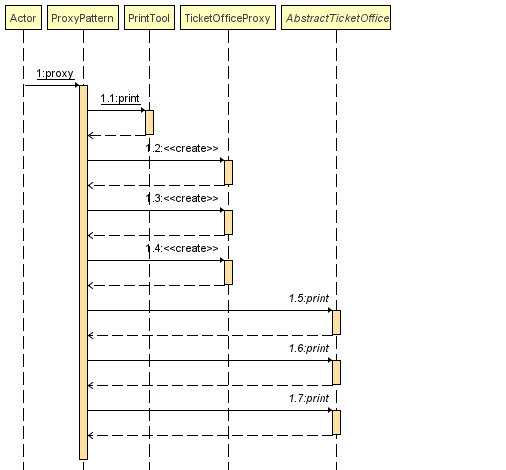


图 13 代理模式时序图

3.20.3补充说明

在某些情况下，一个大型的系统启动需要大量的时间和资源，而往往其中的某些功能并不需要使用，因此耗费的时间可能会相对较长。这将会引起用户的不满。若使用代理模式的话，代理人代表能完成基本工作的角色。只有当工作不能被代理人胜任时在生成本人的实例进行处理。这样能够显著的提升处理的速度。

3.21数据访问对象模式

3.21.1API实现

考虑到主题为海底世界狂欢节，显然会有许多游客游玩，因此需要一个相关系统对游客进行管理，为此设计了游客类(Passenger)和游客管理接口(PassengerInterface)。其中有关于增加游客信息，减少游客信息，查看所有游客信息，查看指定游客信息，修改指定游客信息等功能，并在具体实现类(PassengerInterfaceImplement)中进行了详细实现。

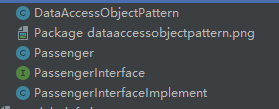


图 14 数据访问模式实例结构

理想情况下，每一位进入狂欢节的游客被记录在系统中（通过进入时购买门票、刷卡等方式被记录）；其信息会在该游客离开狂欢节时从系统中删除。管理人员可以在后台通过指定接口查看到所有游客或是指定游客的信息。

3.21.2实例类图/时序图

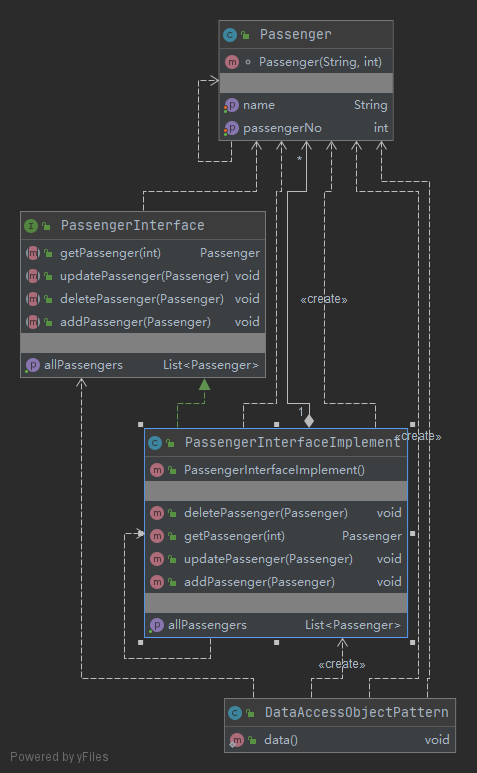


图 15 数据访问对象模式实例类图

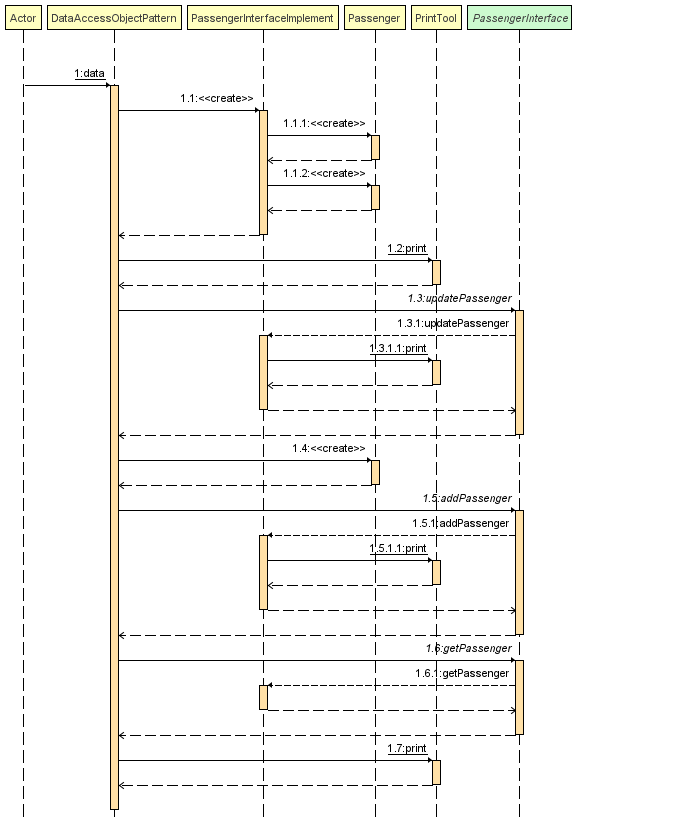


图 16 数据访问对象模式时序图

3.21.3补充说明

与数据源特性相关的数据访问代码,往往将数据访问逻辑和业务逻辑捆绑在一起.这使替换或更改程序的数据源变得十分困难,降低程序的扩展性和移植性,同时增加了维护成本。数据访问对象模式主要用于把低级的数据访问的接口或操作从高级的业务服务中分离出来。这样数据访问机制的代码便可根据数据源的变化单独更改,系统的扩展性获得极大提升,维护亦更加容易。

3.21.4 参考文献

https://www.baeldung.com/java-dao-pattern)

https://www.tutorialspoint.com/design\_pattern/data\_access\_object\_pattern.htm

3.22雇工模式

3.22.1API实现

考虑到主题为海底世界狂欢节，因此设计的场景为客户在狂欢节的某个餐厅中点单鱼类寿司(Sushi)的场景。不同的寿司(Tuna Sushi,Salmon Sushi...)有不同的制作方法，在Sushi的子类中分别模拟实现了其制作的过程。



图 17 雇工模式案例结构

而该场景中的雇工为厨师Cook，每一个顾客点的寿司的订单都会以Sushi类传给Cook，他会根据Sushi的种类调用其不同的制作方法，从而完成制作。

3.22.2类图/实例类图/时序图

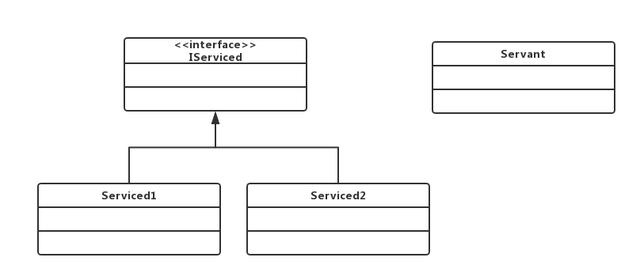


图 18 雇工模式基本类图

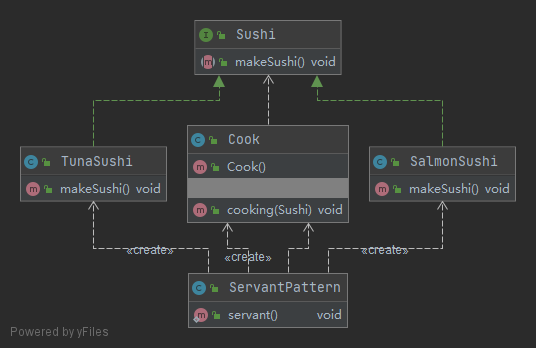


图 19 雇工模式实现案例类图

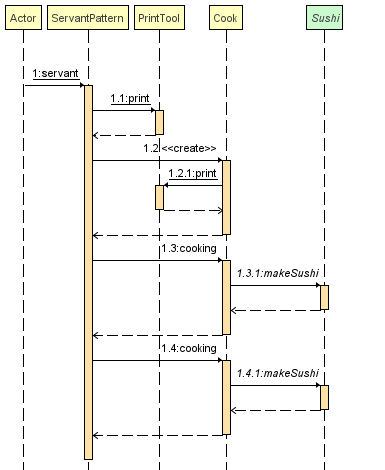


图 20 雇工模式实现时序图

3.22.3补充说明

雇工模式属于行为模式的一种，为一组类提供通用的功能，而类不需要去具体实现这些功能，是命令模式的一种拓展。

在雇工模式中，有一个各种服务的接口，根据不同的服务对象和所需的服务内容，调用接口进行具体实现；同时有一个雇工将这些功能聚集起来方便客户调用。

3.22.4参考文献

(https://java-design-patterns.com/patterns/servant/)

3.23类型对象模式模式

3.23.1API实现

考虑到主题为海底世界狂欢节，因此设计的场景为客户选择在海上冲浪游玩

的设施(SurfingFacility)，如海上摩托motor、快艇speedboat。将设施的所有关键属性都提取出来整合为一个类(SurfingFacilityAttributes)。

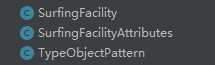


图 21 类型对象案例结构

这样在运行的方法中，可以直接创建属性类，再用它作为参数创建设施类，这样就实现了在运行时动态灵活的创建新的类。

3.23.2类图/实例类图/时序图

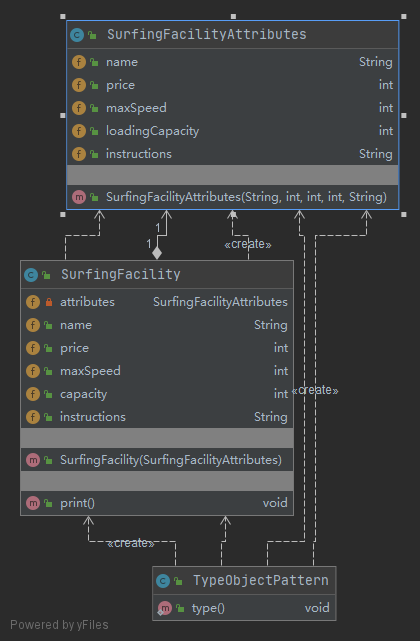


图 22 类型对象案例类图

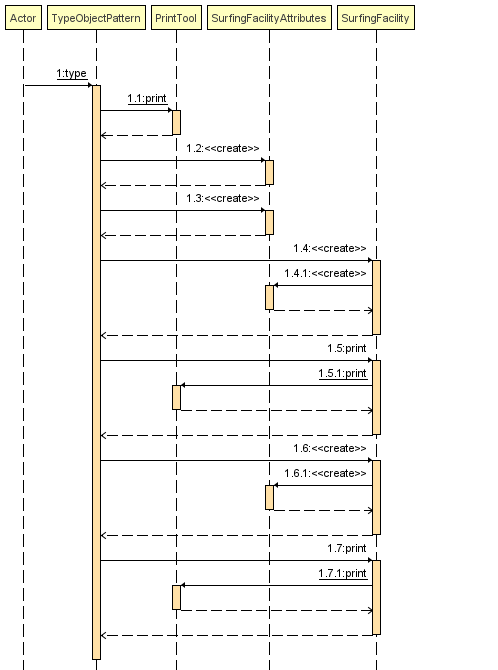


图 23 类型对象SHIXU 图

3.23.3补充说明

在设计某种相同类型的物体时，自然而然的会想到为它们设计一个父类抽象类，再用子类将它们一一实现。但当子类太多时会很难管理。对象模式的目的就时允许程序在运行中灵活的创建新的类，而不用修改代码。

3.23.4类型对象模式

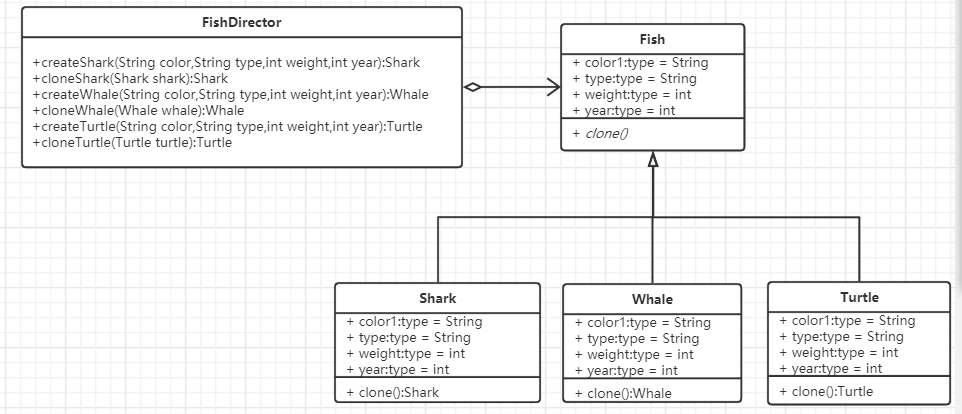
[设计模式之美：Type Object（类型对象） - sangmado - 博客园 (cnblogs.com)](https://www.cnblogs.com/gaochundong/p/design_pattern_type_object.html)

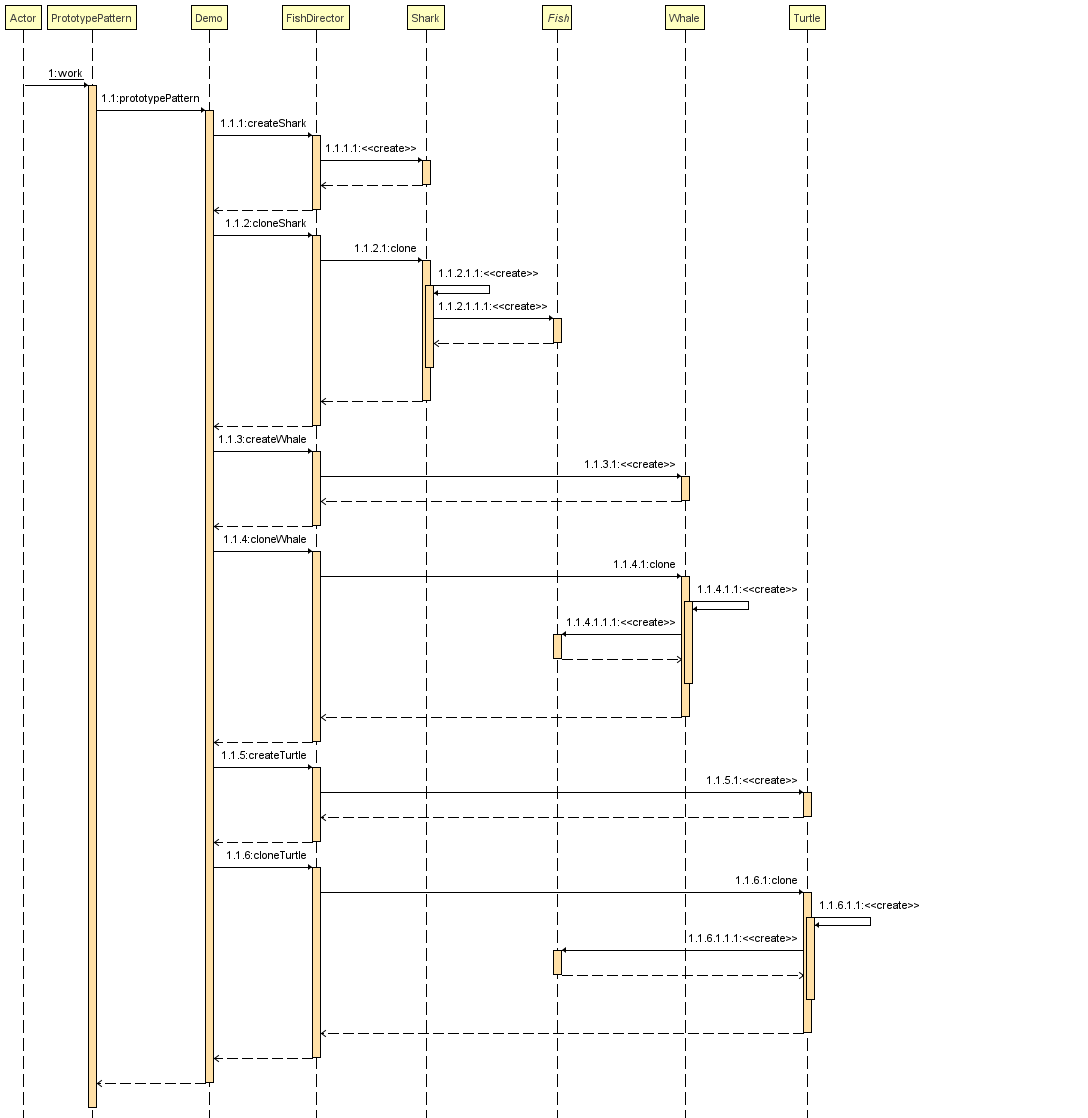
3.24工厂模式

3.24.1API实现

海底世界嘉年华中，海底生物不仅种类繁多，数量也十分庞大。在本例中，我们需要用到“克隆”方法来生成完全相同的鱼类副本。所有海底生物都遵循同一个提供克隆方法的接口，在复制自身成员变量值到结果对象前， 子类可调用其父类的克隆方法。

3.24.2类图与时序图





3.24.3补充说明

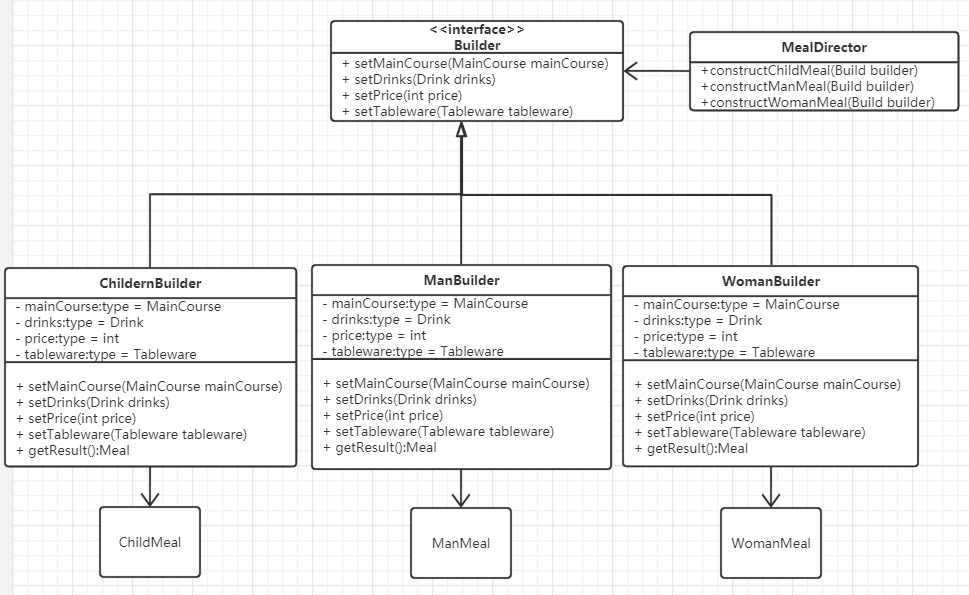
原型模式是一种创建型设计模式， 使你能够复制已有对象， 而又无需使代码依赖它们所属的类。

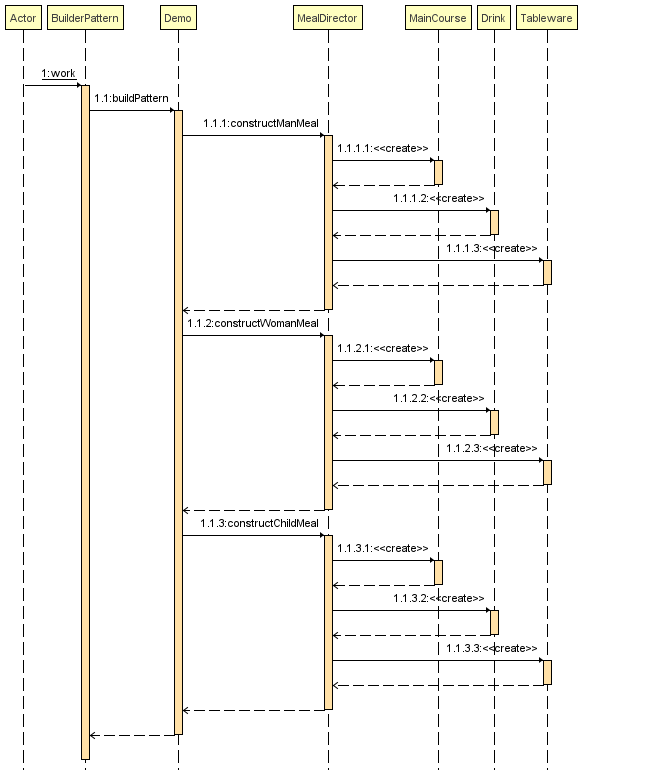
3.25建造者模式

3.25.1API实现

在海底世界嘉年华的餐厅里，提供了三种套餐：男性套餐、女性套餐和儿童套餐。通过建造者模式，分步骤创建三种不同的套餐。

3.25.2类图与时序图





3.25.3补充说明

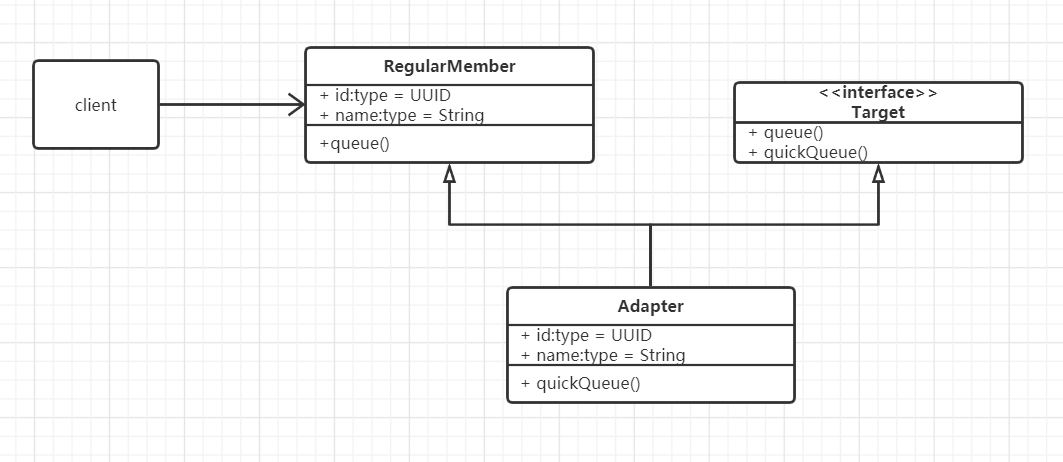
建造者模式是一种创建型设计模式， 使你能够分步骤创建复杂对象。 该模式允许你使用相同的创建代码生成不同类型和形式的对象。

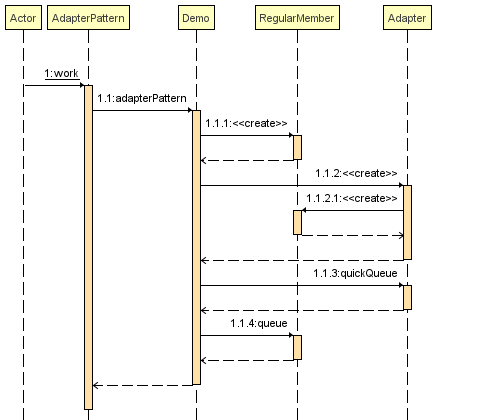
3.26适配器模式

3.26.1API实现

在海底世界嘉年华中，游客分为两种：普通会员和高级会员。当需要排队时，普通会员只能选择普通通道，而高级会员可以选择高速通道进行排队。如果普通会员想在高速通道排队，则需要适配器——本例中即为升级为高级会员。

3.26.2类图与时序图





3.26.3补充说明

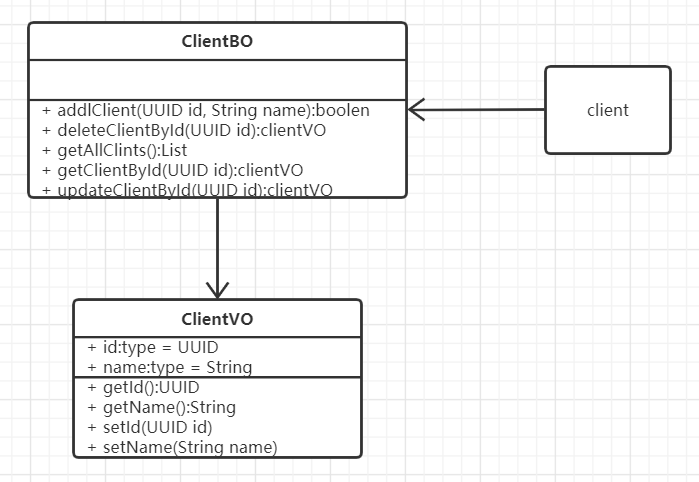
适配器模式是一种结构型设计模式， 它能使接口不兼容的对象能够相互合作。

3.27 传输对象模式

3.27.1API实现

在海底世界嘉年华中，当然需要进行客户数据的增删改查，采用传输对象模式，创建传输对象clientVO和业务对象clientBO，clientVO包含客户的id，name等信息，而clientBO则封装了对客户信息进行增删改查的操作。

3.27.2类图与时序图



3.27.3补充说明

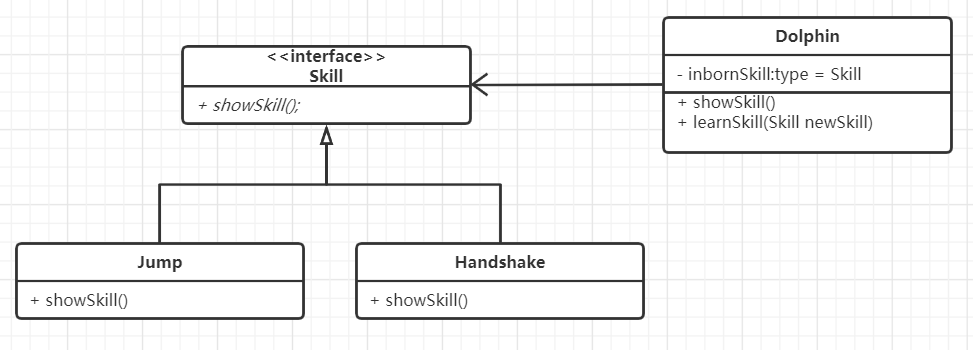
传输对象模式（Transfer Object Pattern）用于从客户端向服务器一次性传递带有多个属性的数据。

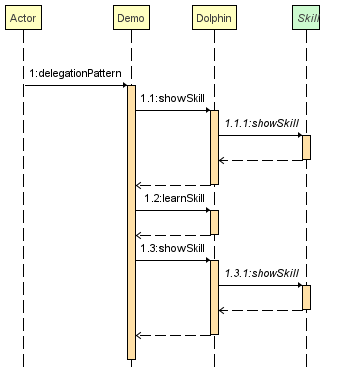
3.28委托模式

3.28.1API实现

在海底世界嘉年华中，水族馆是必不可少的游玩景点。在水族馆中，海豚经过训练可以进行一些技巧的表演。其中，海豚天生就会跳出水面，而经过后天的学习，可以和训练时握手。采用委托模式进行模拟，创建Skill接口，类Jump和Handshake则继承该接口。海豚天生就会跳，并且可以通过委托方式“学会”握手。

3.28.2类图与时序图





3.28.3补充说明

委托模式（delegation pattern）是一项基本技巧，在委托模式中，有两个对象参与处理同一个请求，接受请求的对象将请求委托给另一个对象来处理。委托模式使得我们可以用聚合来替代继承。

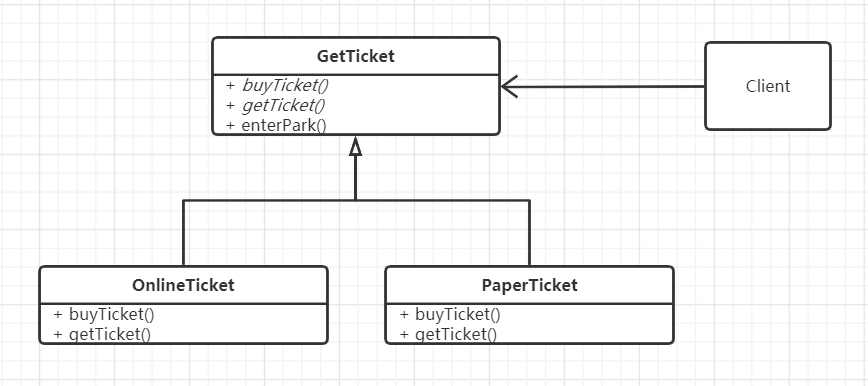
与代理模式（proxy pattern）的区别在于委托模式更注重结果，代理模式更注重过程。

3.29模版模式

3.29.1API实现

在海底世界嘉年华中，我们一般需要1.买票，2.取票才能进园。而买票的方式分为网上购票和现金购票。进园的方法是相同的，都是先购票再取票。而因为买票的方式不同，于是会有两种进园方式。采用模块方法，将进园的流程封装到一个final方法中，而且根据买票方式的不同，定制不同的进园方式。

3.29.2类图与时序图



3.29.3补充说明

在模板模式中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。使用场景有：

多个子类有公共方法，并且逻辑基本相同时。

重要的、复杂的方法，可以考虑作为模板方法。

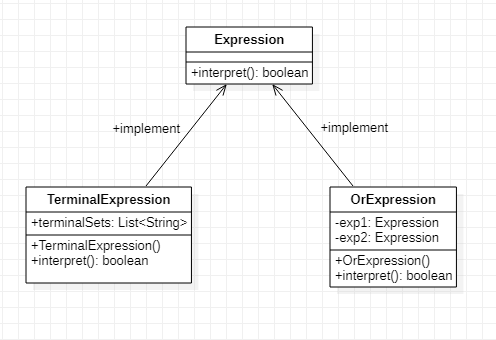
重构时，模板模式一般是最常用的，把相同的代码抽取到父类中，通过子类约束其行为。

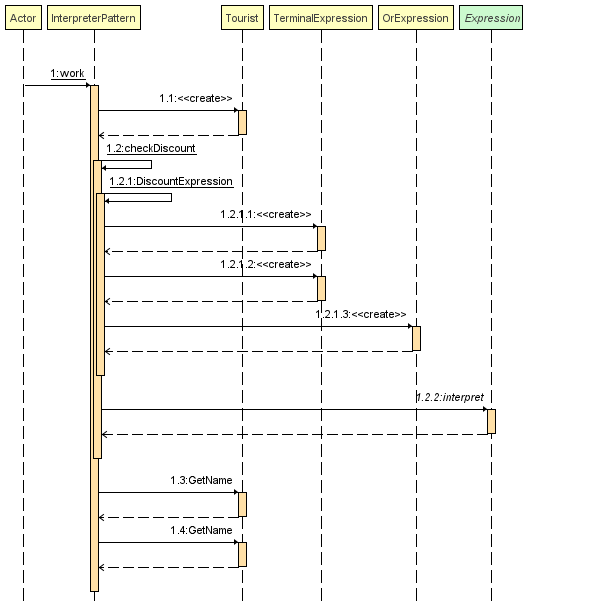
3.30 解释器模式

3.30.1API实现

解释器模式提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。本项目中解释器模式解决的问题是“游客是否能够享受折扣”这一判断，将用户信息汇集为一段文本进行处理。

3.30.2类图与时序图



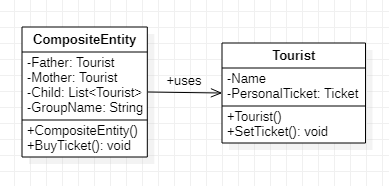


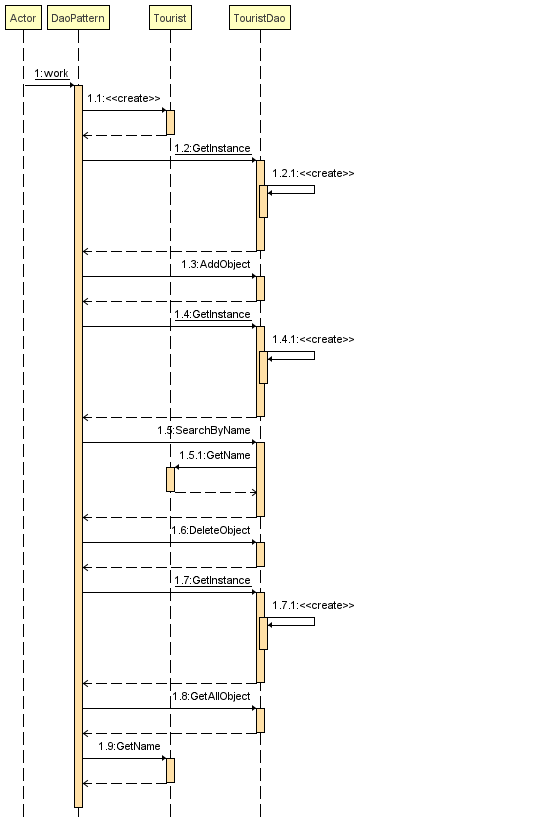
3.31 组合实体模式

3.31.1API实现

组合实体模式中包含粗粒度对象，粗粒度对象包含依赖对象，有自己的生命周期，也能管理依赖对象的生命周期。依赖对象是一个持续生命周期依赖于粗粒度对象的对象，在粗粒度对象中依赖对象的组合，粗粒度之间的组合可以采取一定策略，可以是树状的也可以是线性的。

3.31.2类图与时序图



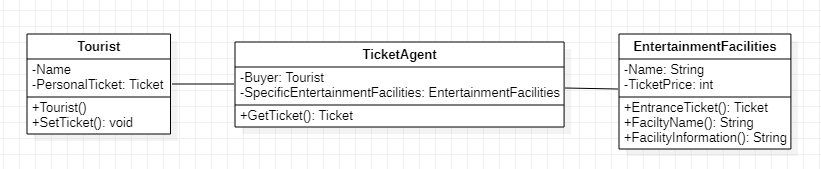


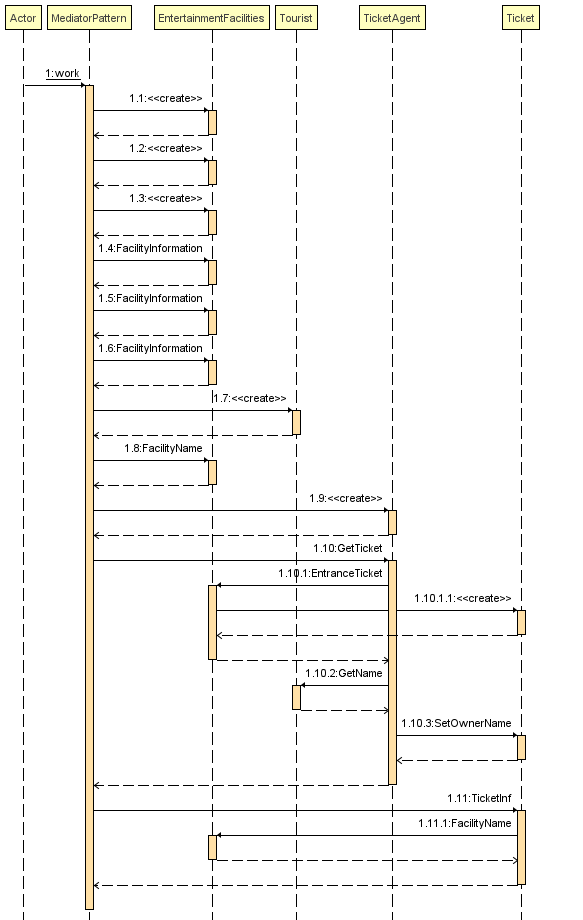
3.32 中介者模式

3.32.1API实现

中介者模式是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。中介者模式属于行为型模式。采取中介者模式可以将多对多的网状关系简化为多对一，其中一表示中介者，利用中介者可以实现多对多的通信。在本案例中游客通过中介者进行多游玩场景购票，简化了多到多的复杂场景。

3.32.2类图与时序图



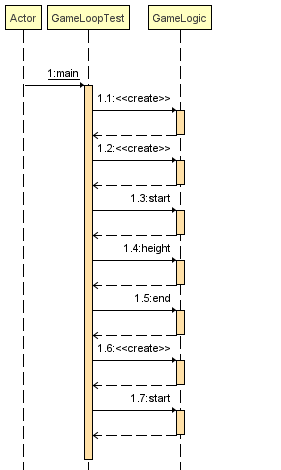
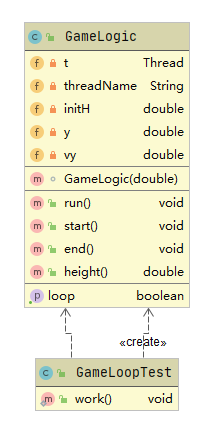


3.33 GameLoop模式

3.33.1 API实现

主函数中存在一个主循环，根据用户输入进行状态的更新。

3.33.2类图与时序图



3.33.3补充说明

Gameloop模式通常出现在游戏设计当中，许多结构复杂的游戏都有一个主循环，利用多线程机制进行鼠标和键盘等事件的监听工作，并根据用户操作在主循环中更新游戏状态等数据，控制游戏的进程。

3.33.4 参考文献

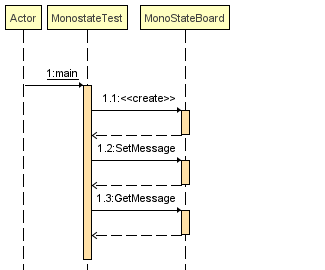
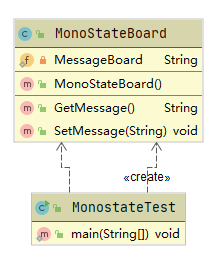
[Game Programming Patterns](http://gameprogrammingpatterns.com/)

3.34 Monostate模式

3.34.1 API实现

采用get和set方法对静态数据进行查询和修改。

3.34.2类图与时序图



3.34.3补充说明

Monostate模式是单例模式的另一种的实现，单例模式的构造函数为私有，用户访问利用getInstance方法，获取到的都是类的唯一实例。应用Monostate模式的类构造方法不是私有，但是数据为静态数据，采用get和set方法进行数据获取和数据的修改，所有访问monostate模式的用户获取到的都是同一份实例。

3.34.4 参考文献

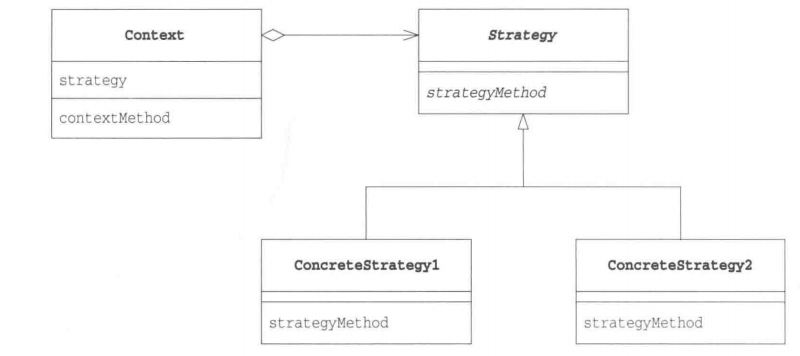
[【设计模式】--MONOSTATE模式\_qfzhangwei的专栏-CSDN博客\_monostate模式](https://blog.csdn.net/qfzhangwei/article/details/78448607)

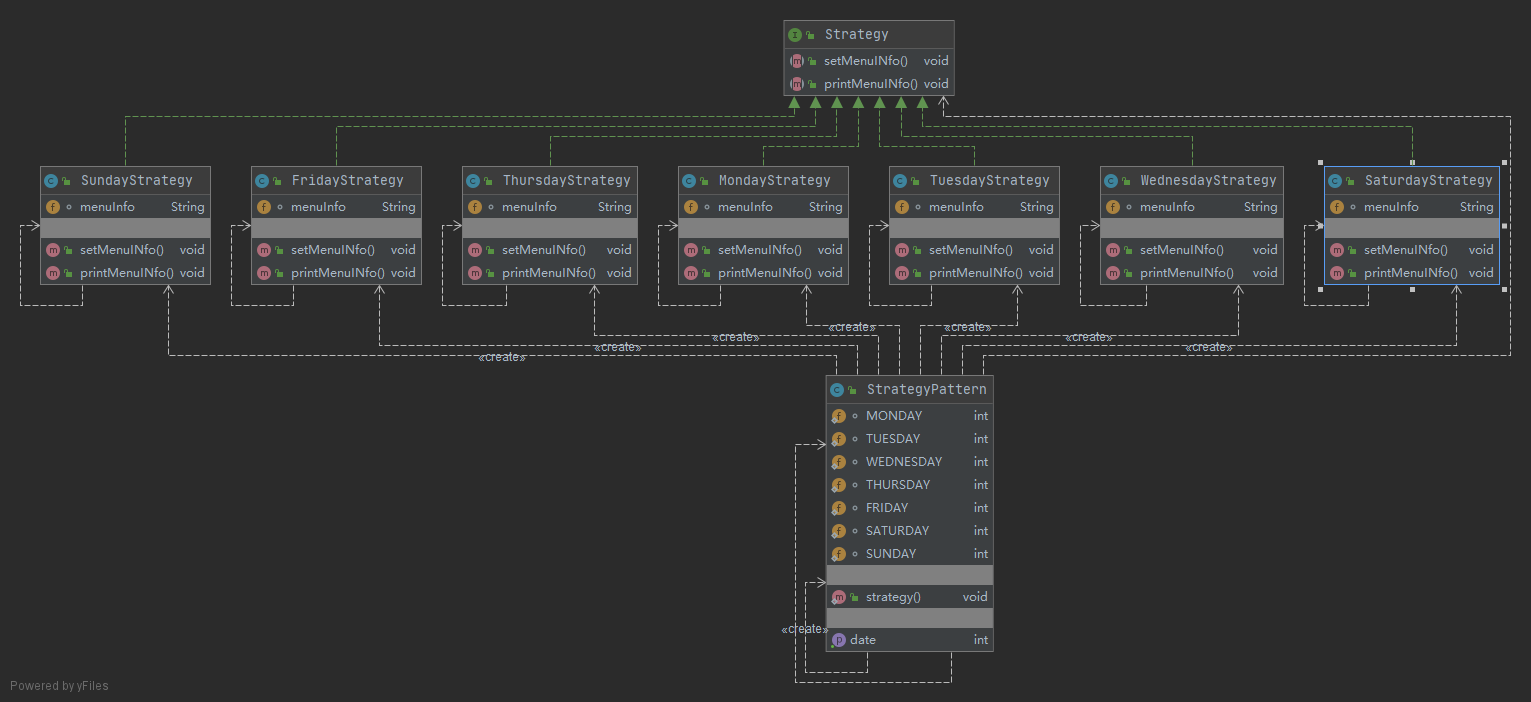
3.35 策略模式

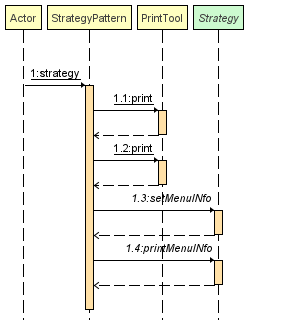
3.35.1API实现

考虑到主题为海底世界狂欢节，因此设计了切换主题公园中餐厅菜系的场景。设计了策略接口Strategy，并为一个星期中的每一天设计了菜单策略(Monday/Tuesday/Wednesday/Thursday/Friday/Saturday/Sunday)，餐厅会依据星期几切换菜单

3.35.2类图/实例类图/时序图







3.35.2 补充说明

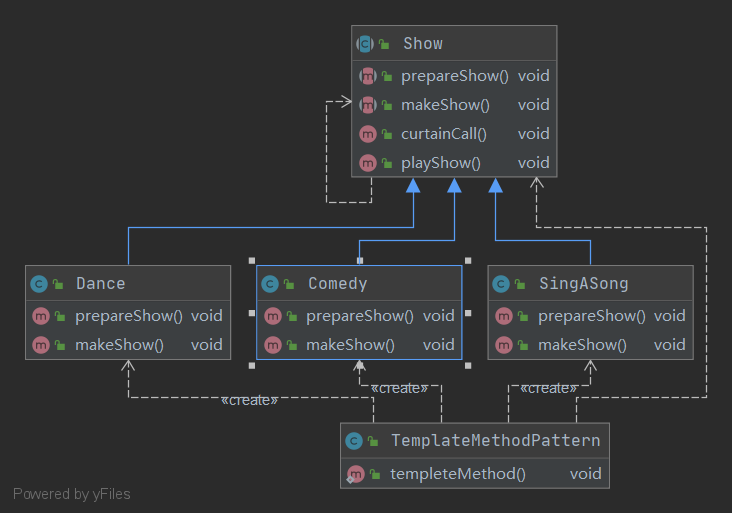
无论什么程序，其目的都是解决问题。而为了解决问题，我们又需要编写特定的算法。使用Strategy模式可以整体的替换算法的实现部分。能够整体的替换算法，能让我们轻松的以不同的算法去解决同一个问题，这种模式就是Strategy模式。

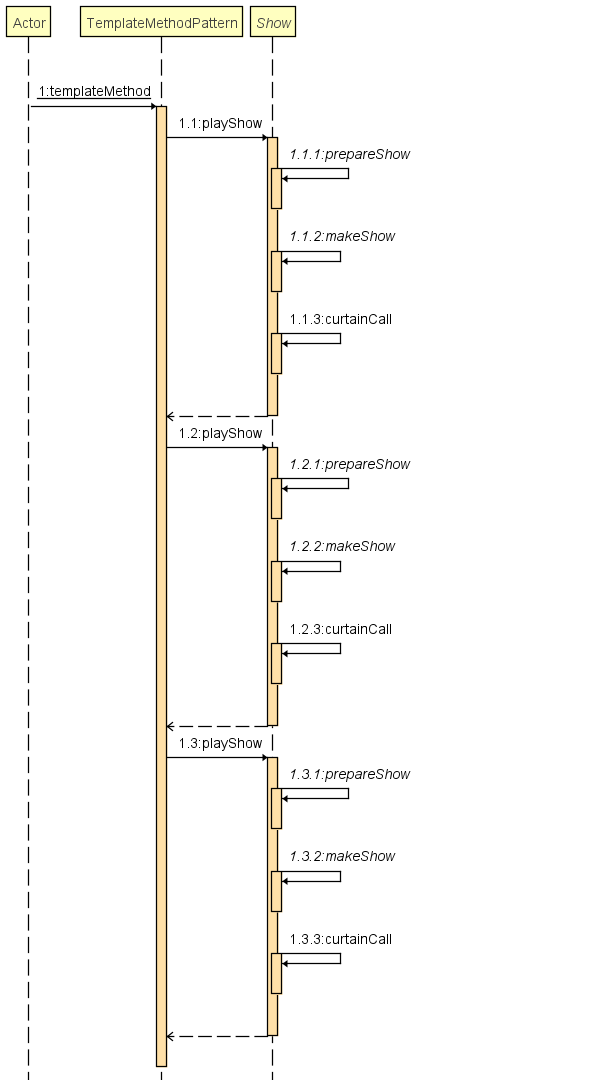
3.36 模版方法模式

3.36.1API实现

海底世界嘉年华有一场华丽的演出。其中有小品，舞蹈，歌曲三个节目，每种节目都有准备，表演，谢幕三个步骤，且每种节目的准备，表演步骤不同，但是谢幕阶段步骤相同。

3.36.2类图与时序图





3.36.3参考文献

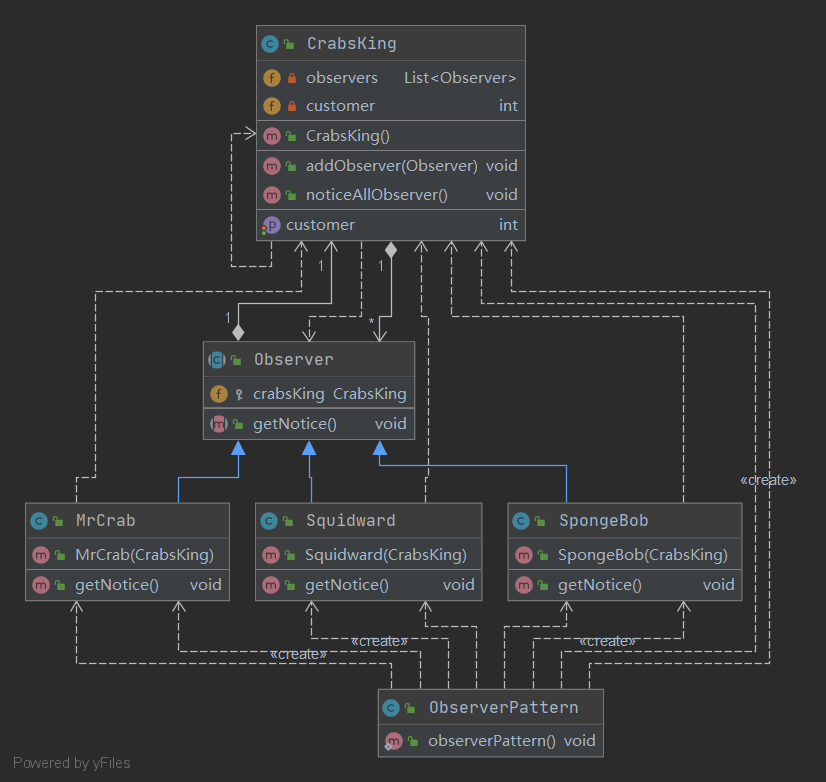
https://www.runoob.com/design-pattern/template-pattern.html

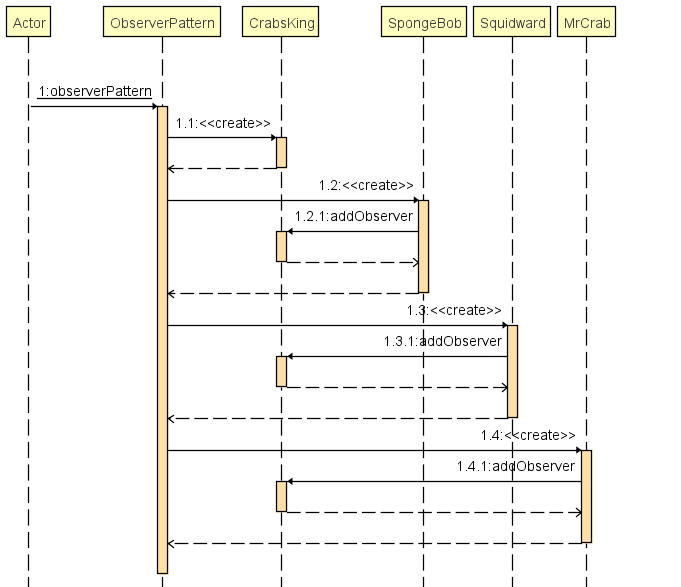
3.37 观察者模式

3.37.1API实现

海绵宝宝、章鱼哥、蟹老板时刻关注着蟹堡王内顾客的数量。Crabs King类时刻记录并更新蟹堡王内顾客的数量，并在顾客数量变更时及时通知海绵宝宝，章鱼哥，蟹老板。

3.37.2类图与时序图





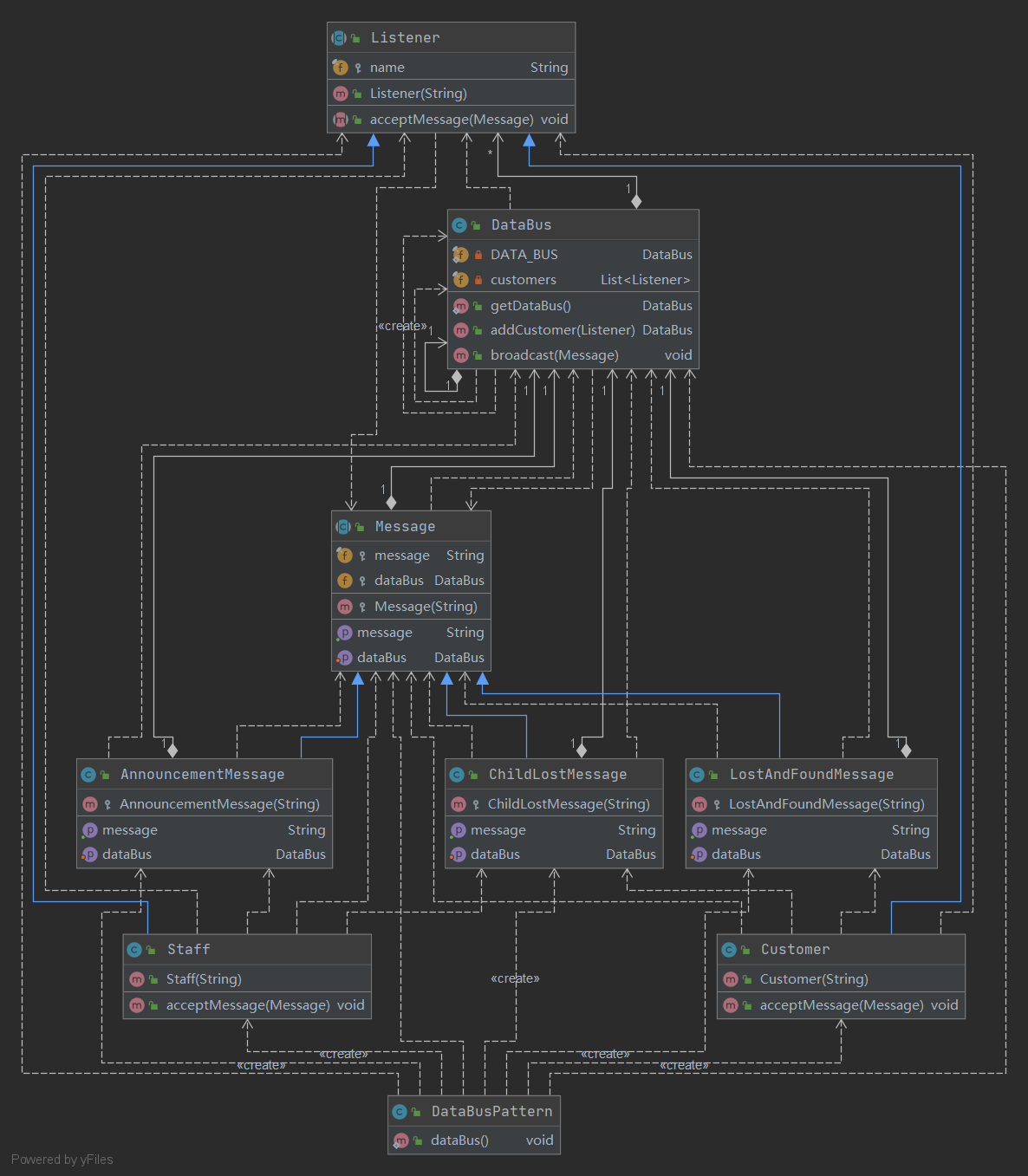
3.38 数据总线模式

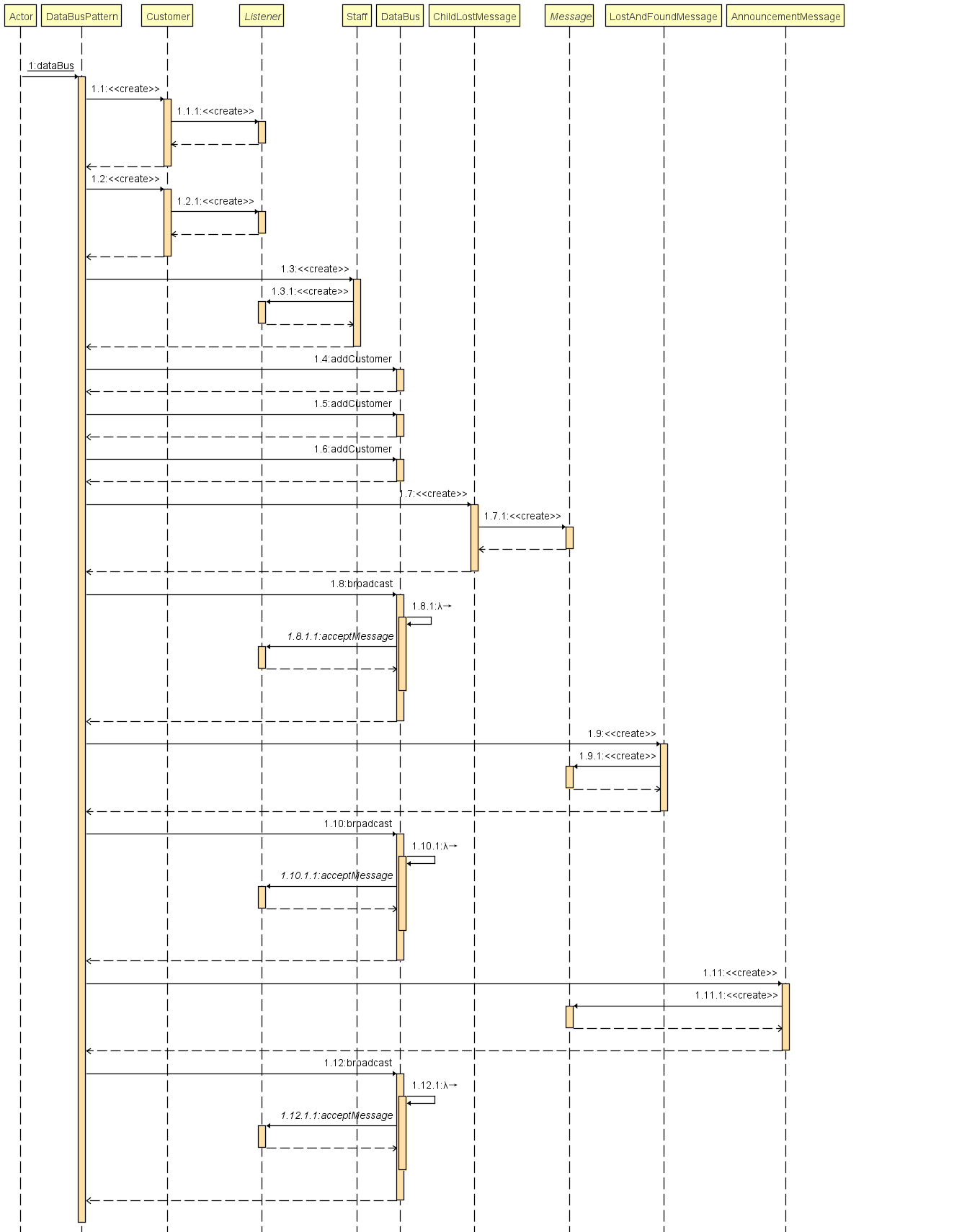
3.38.1API实现

在海底世界狂欢节中，有用于应对突发情况的广播，广播可以向所有游客和工作人员传递信息，并且信息分为通知信息，寻人信息，失物招领信息三类。其中游客只需要在意失物招领和寻人启事消息，工作人员只需要在意寻人启事和通知消息。

在展示中，接收到消息即代表着处理完成

3.38.2类图与时序图





3.38.3简要介绍

允许在应用程序的组件之间发送消息/事件，而不需要相互了解。他们只需要知道要发送的消息/事件的类型，就像数据总线一样。

3.38.4参考文献

<https://www.jianshu.com/p/f9807c420bfe>

<https://java-design-patterns.com/patterns/data-bus/>

3.39 享元模式

3.39.1API实现

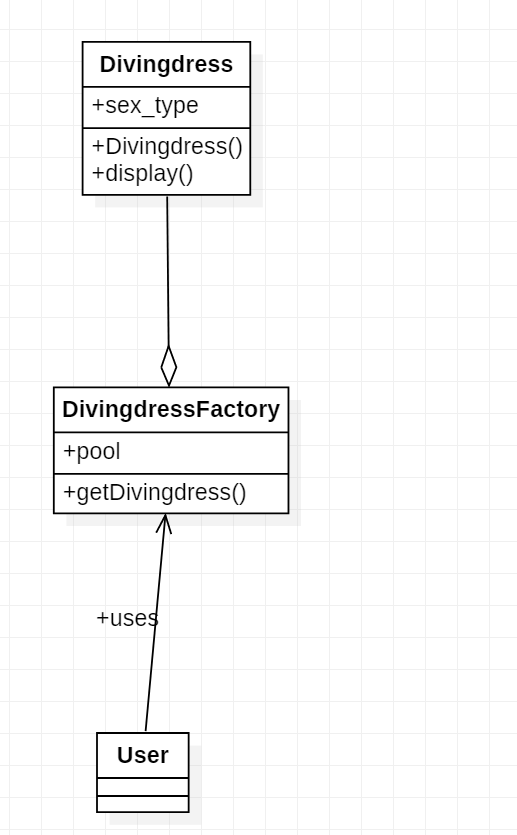
DivingdressFactory.getDivingdress()

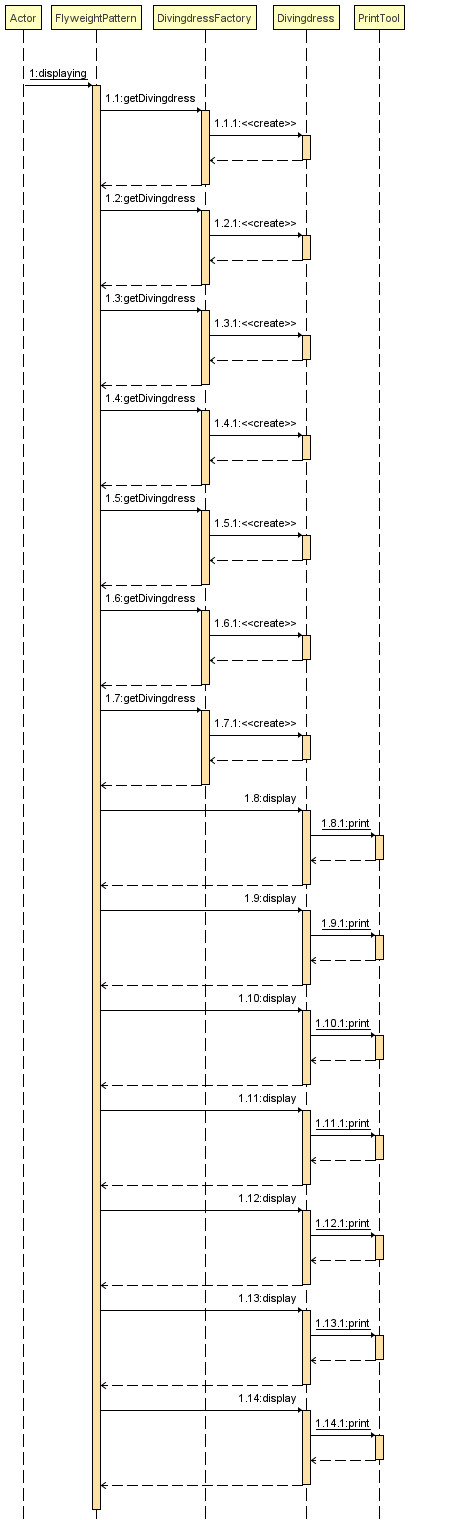
说明：获取潜水服实例

参数：sex\_type(性别，尺码）

返回：dg(潜水服实例）

3.39.2类图与时序图





3.39.3补充说明

享元模式使用共享物件，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似物件。本列中，考虑到向游客租借潜水服时，会出现很多相同的型号，因此使用享元模式避免创建多个实例浪费内存。

3.40 桥接模式

3.40.1API实现

Drink()

说明: 接口类，对应实现化角色。

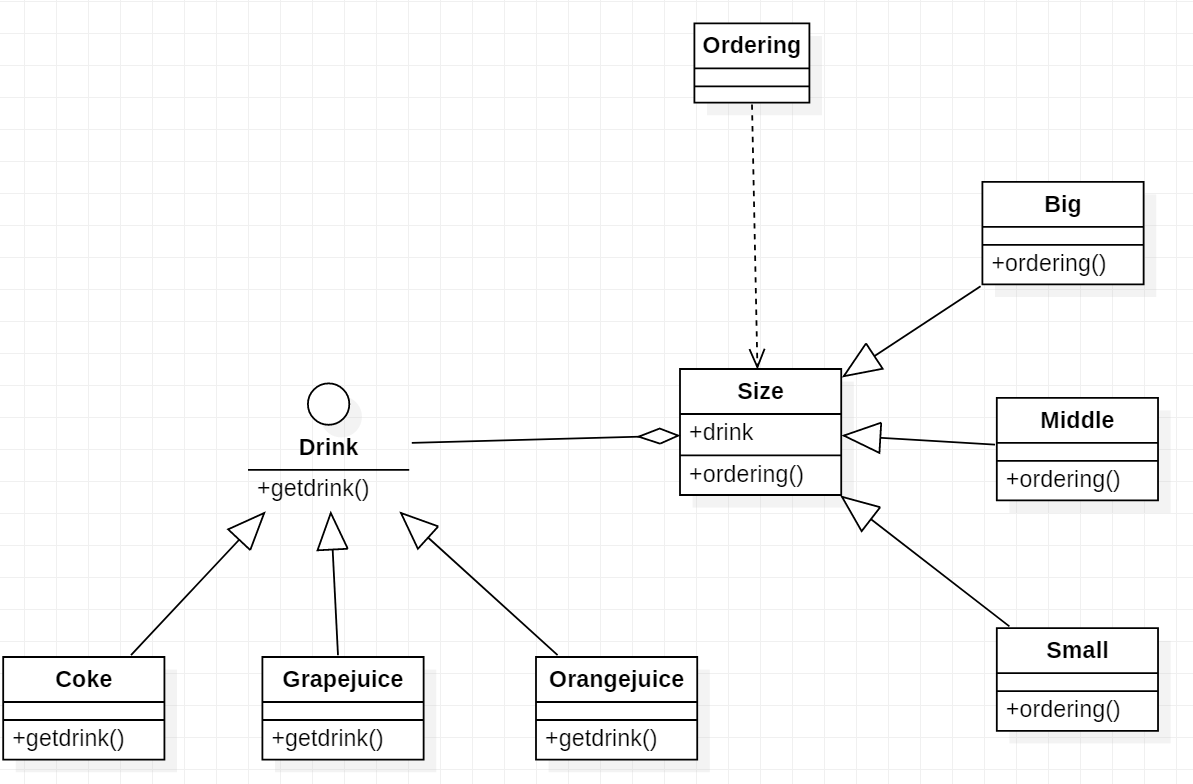
参数: 无

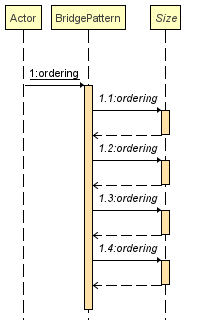
Size.ordering()

说明: 选择具体饮料

参数: 无

3.40.2类图与时序图





3.40.3补充说明

桥接模式是将抽象部分与它的实现部分分离，使它们都可以独立地变化。本例中，饮料的容量Size类为抽象化角色，他有三个扩展抽象化角色：Big, Middle, Small。品种Drink类为实现化角色，他有三个具体实现化角色：Orangejuice, Grapejuice, Coke。通过Ordering()方法可以选择不同容量的不同饮料。

3.41 组合模式

3.41.1API实现

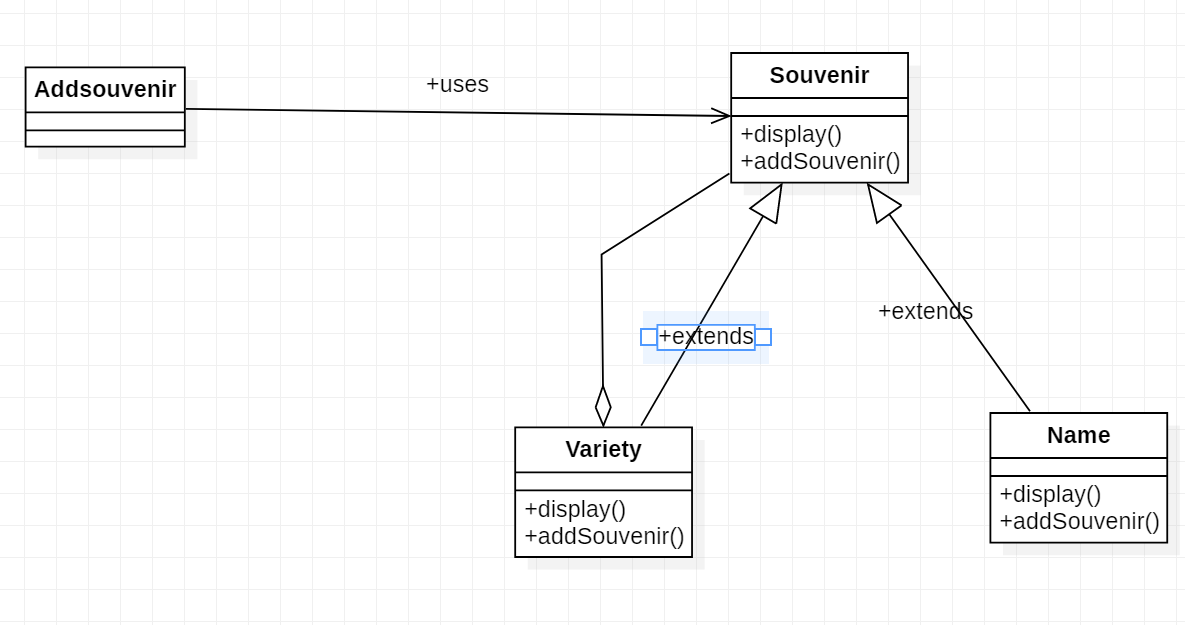
Souvenir.display()

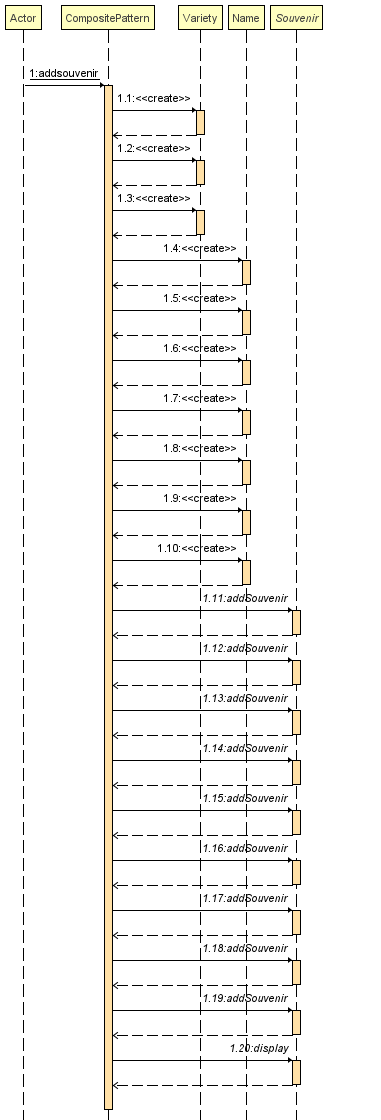
参数：String blank

说明：展示纪念品的层级结构

返回：无

3.41.2类图与时序图





3.41.3补充说明

组合模式将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。有时候又叫做部分-整体模式，它使我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素,从而使得客户程序与复杂元素的内部结构[解耦](https://baike.baidu.com/item/%25E8%25A7%25A3%25E8%2580%25A6)。本例中，游客可按照此模式自主选购纪念品，Souvenir为公共父类，Name，Variety都是它的子类，Variety中可存放Variety和Name。

3.42多例模式

3.42.1API实现

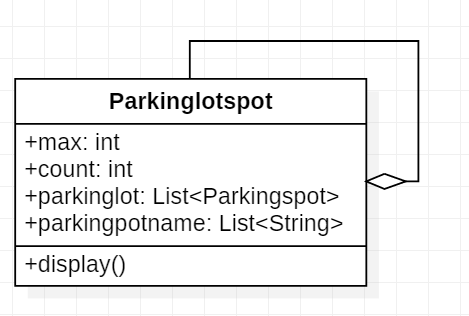
管理停车位实例：

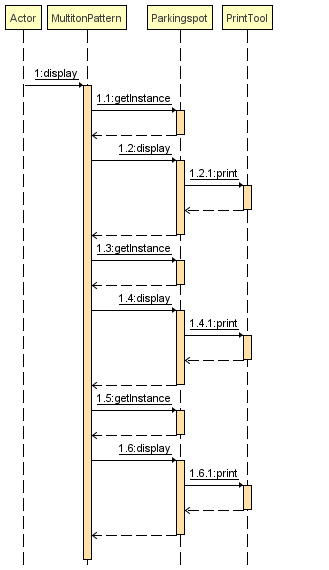
Parkingsopt.getInstance()

说明: 获取多例类中的实例

参数: index(停车位位置）

3.42.2类图与时序图





3.42.3补充说明

所谓多例实际上就是单例模式的自然推广，属于对象创建类型的模式，多例模式其实就是限制了对象的数量，并且有可能对对象进行重复使用。多例类可以有多个实例。多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。本例中，Parkingspot类就是就是多例类，创建并管理自己的实例。

3.43 工厂模式

3.43.1API实现

游客游玩时，需要获取地图，因为地图都是一样的，所以获取地图时，为了防止地图实例不同产生错误，采取双重检查锁模式。

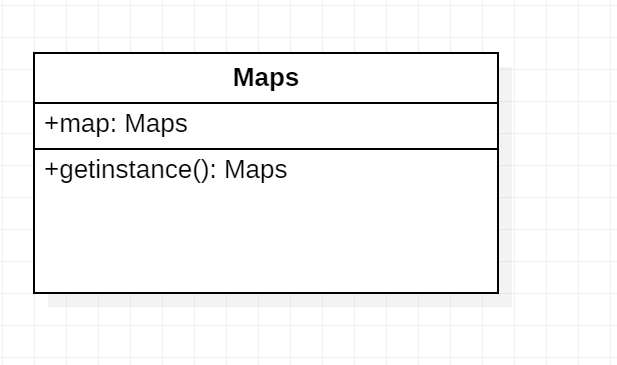
Maps.getinstance()

**说明**：多线程场景下正确获取地图实例

**参数**：无

**返回**：地图实例

3.43.2类图与时序图



第四章 项目搭建及错误处理方法

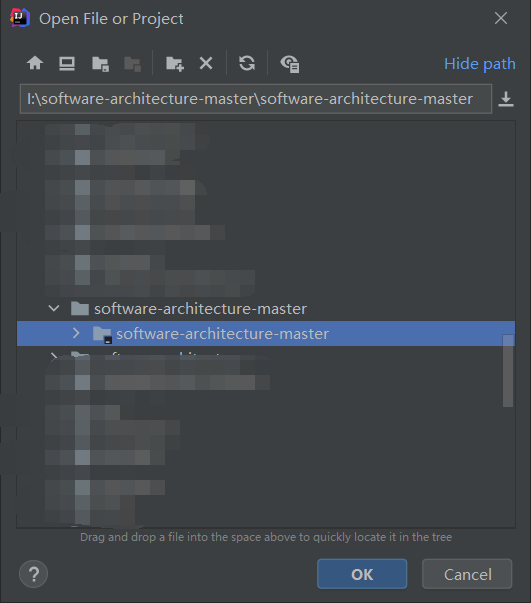
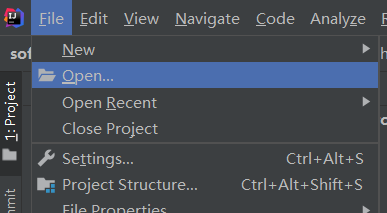
4.1 推荐环境及配置

java jdk15

intellij idea 旗舰版

4.2 项目下载

1.在<https://github.com/ccd233/software-architecture> 下载master分支下的全部文件。



2.使用intellij idea直接打开(file->open->项目根目录)。

3.编译运行

4.项目搭建过程种遇到的错误请参考4.2或联系组长。

4.3 错误处理方法

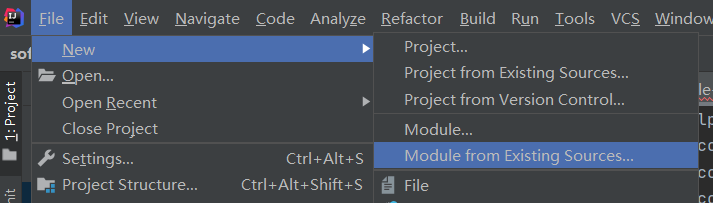
4.2.1 找不到模块

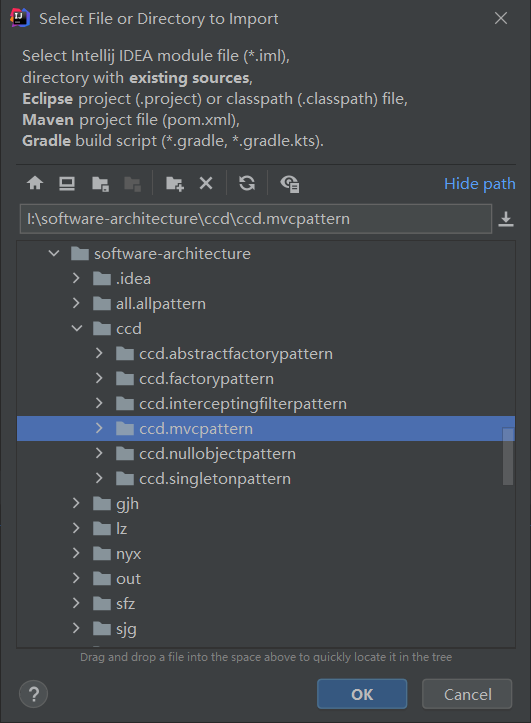
将相应文件夹构建为模块。

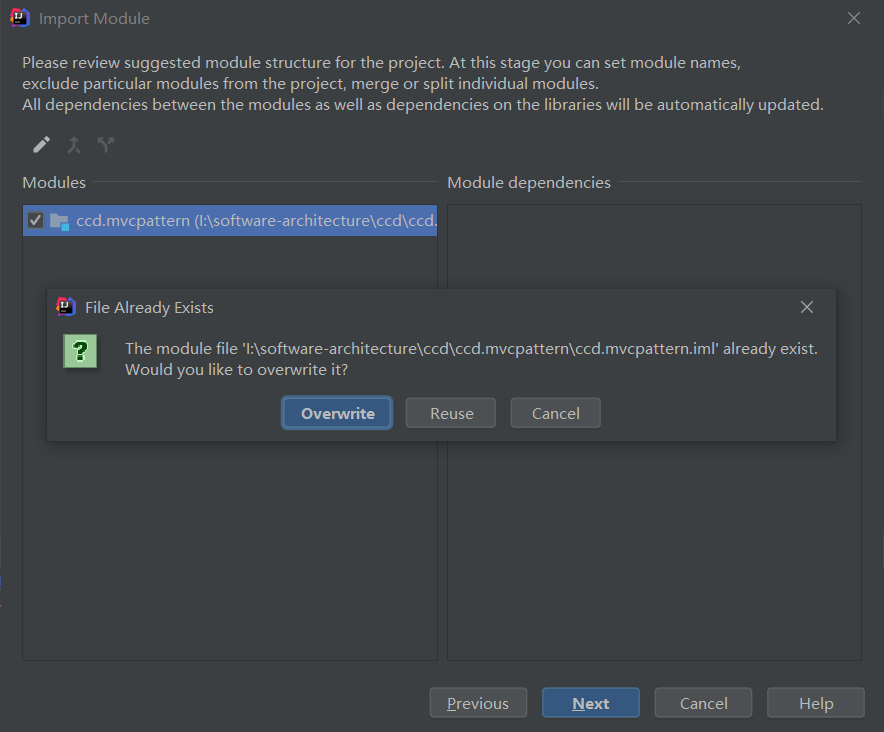
示例如下：



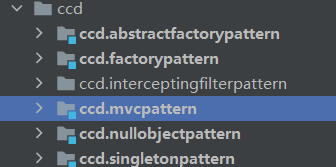
file->new->module from existing sources

 找到相应文件夹，点击确定





全部执行默认操作（选择next 或overwrite）



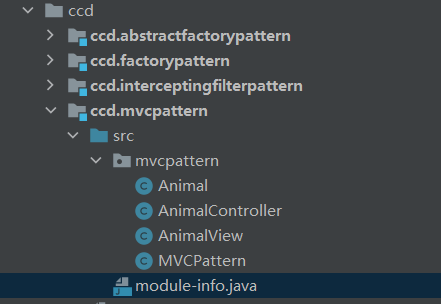
文件夹图标改变后说明处理成功

4.2.2 未依赖模块

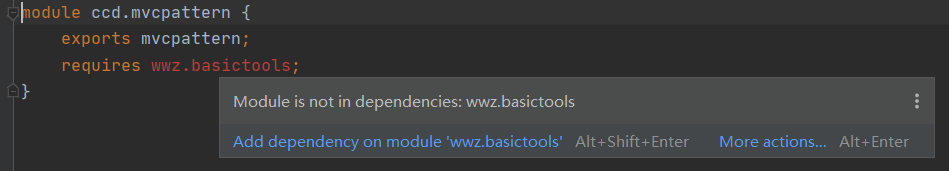
添加依赖即可。

示例如下：

打开未添加依赖的模块文件夹下的module-info.java文件



鼠标悬停在错误上



点击add dependency on “modulename”即可

