Федеральное государственное бюджетное образовательное

учреждение высшего образования

«Алтайский государственный технический университет

им. И.И. Ползунова»

Факультет (институт) Информационных технологий

Кафедра Прикладная математика

Отчет защищен с оценкой\_\_\_\_\_\_\_\_\_\_\_\_\_\_

С.М.Старолетов

(подпись преподавателя) (инициалы, фамилия)

“\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_ 2018 г.

Комплексный отчет

по лабораторным работам

по дисциплине Архитектурное проектирование

и паттерны программирования

(наименование дисциплины)

ЛР 09.03.04.25.001 ОТ

(обозначение документа)

Студент группы ПИ-42 И.С. Харахордин

(инициалы, фамилия)

Преподаватель к.ф.-м.н., доцент С.М. Старолетов

(должность, ученое звание) (инициалы, фамилия)

Барнаул 2018

**Задание**: проектирование и реализация системы доставки заказов с использованием самолетов, автомобилей, поездов.

**Расширения**:

- доставка с "пересадками"

- доставка при помощи кораблей

- различная обработка заказов (включая операторов)

Интерфейсы:

IOrder - интерфейс заказа

+ shipOrder() - отправить заказ

IPerson - интерфейс человека

+ confirmOrder() - подтвердить заказ

+ cancelOrder() - отменить заказ

Классы:

Order - информация о заказе

Address - информация об адресе

Dispatcher - оператор для подтверждения заказа

Item - информация об одном товаре

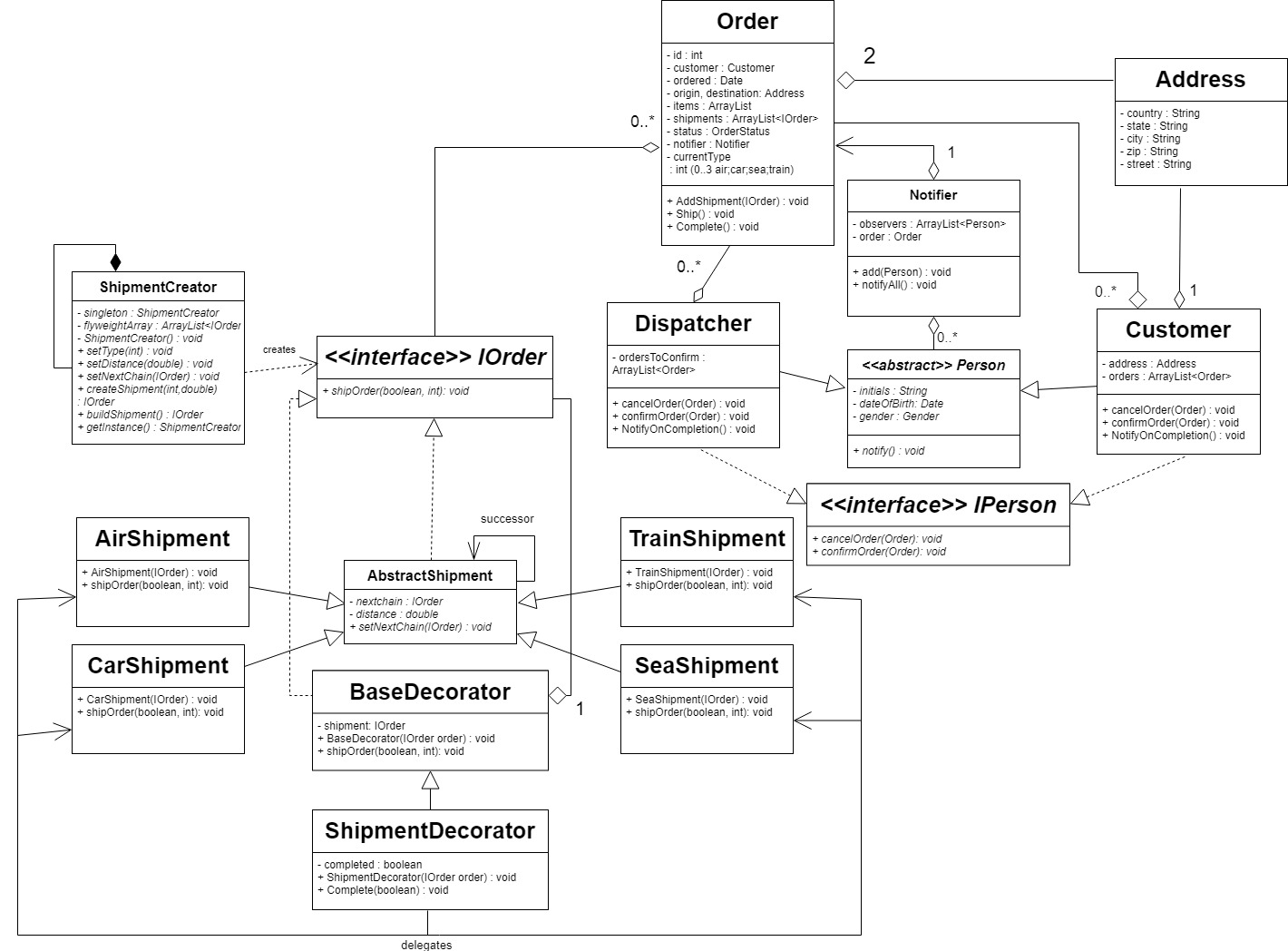
Customer - информация о заказчике

Person - информация о человеке

AirShipment, CarShipment, TrainShipment, SeaShipment - доставка при помощи самолета, автомобилей, поездов и кораблей

|  |  |  |  |
| --- | --- | --- | --- |
| Рисунок 2.1. Общая UML-диаграмма системы |  |  |  |

Используемые шаблоны проектирования:



**Делегирование** - это способ, которым объект внешне выражает некоторое поведение, но в реальности передаёт ответственность за выполнение этого поведения связанному объекту.

Цель: Возможность изменить поведение конкретного экземпляра объекта вместо создания нового класса путемнаследования.

Минусы: Этот шаблон обычно затрудняет оптимизацию по скорости в пользу улучшенной чистоты абстракции.

Реализация: Классы AirShipment, CarShipment, TrainShipment, SeaShipment, которые наследуются от ShipmentDecorator, через функцию shipOrder вызывают эту же функцию у других классов.

**Декоратор** — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Используется:

- Когда нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.

- Когда нельзя расширить обязанности объекта с помощью наследования.

Плюсы:

* Большая гибкость, чем у наследования.
* Позволяет добавлять обязанности на лету.
* Можно добавлять несколько новых обязанностей сразу.
* Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

Минусы:

* Трудно конфигурировать многократно обёрнутые объекты.
* Обилие крошечных классов.

Реализация: Класс ShipmentDecorator расширяет объекты от интерфейса IOrder, храня «оборачиваемый» объект в поле shipment.

- **Цепочка обязанностей** - это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Используется:

- Когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно какие конкретно запросы будут приходить, и какие обработчики для них понадобятся.

- Когда важно, чтобы обработчики выполнялись один за другим в строгом порядке.

- Когда набор объектов, способных обработать запрос, должен задаваться динамически.

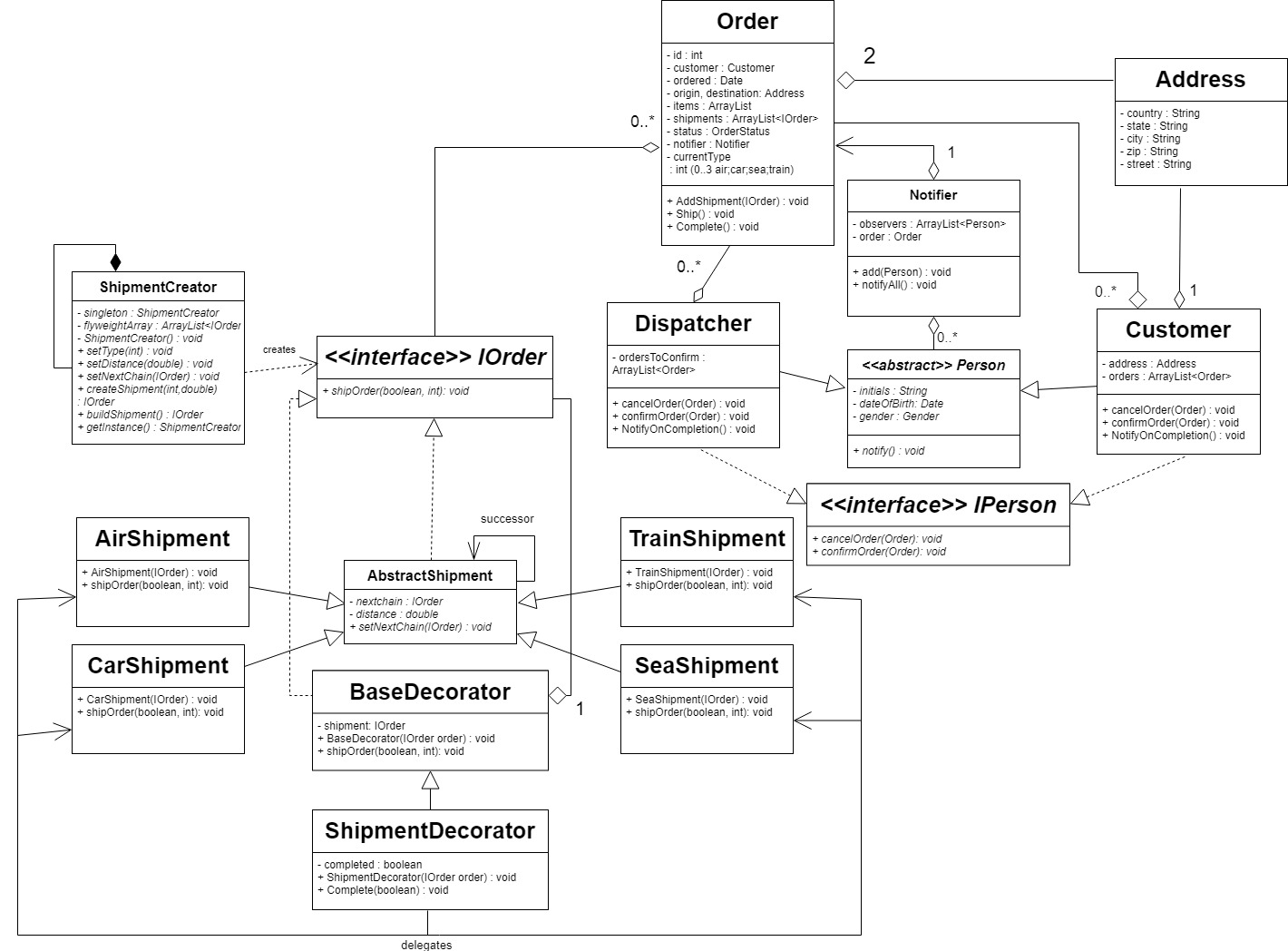
Плюсы:

* Уменьшает зависимость между клиентом и обработчиками.
* Реализует *принцип единственной обязанности*.
* Реализует *принцип открытости/закрытости*.

Минусы:

* Запрос может остаться никем не обработанным.

Реализация: Классы AirShipment, CarShipment, TrainShipment, SeaShipment имеют поля следующего звена в цепи, которому они будут передавать запрос.



**Фабричный метод** — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Используется:

- Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать код.

- Когда необходимо дать возможность пользователям расширять части фреймворка или библиотеки.

- Когда необходимо экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

Плюсы:

* Избавляет класс от привязки к конкретным классам продуктов.
* Выделяет код производства продуктов в одно место, упрощая поддержку кода.
* Упрощает добавление новых продуктов в программу.
* Реализует *принцип открытости/закрытости*.

Минусы:

* Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

Реализация:

Класс ShipmentCreator имеет метод createShipment, который, в зависимости от аргументов, создает и возвращает объект одного из классов AirShipment, CarShipment, TrainShipment, SeaShipment.

**Приспособленец** (англ. Flyweight) — структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым. Паттерн использует разделение для эффективной поддержки большого числа мелких объектов.

Плюсы:

* Экономит оперативную память.

Минусы:

* Расходует процессорное время на поиск/вычисление контекста.
* Усложняет код программы из-за введения множества дополнительных классов.

Реализация: В классе фабрики ShipmentCreator имеется массив созданных объектов flyweightarray. При создании нового объекта проверяется, не создан ли уже идентичный объект. Если такой имеется, то он возвращается функцией.

- **Строитель** — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Используется:

- Когда необходимо избавиться от «телескопического конструктора».

- Когда код должен создавать разные представления какого-то объекта. Например, деревянные и железобетонные дома.

- Когда нужно собирать сложные составные объекты.

Плюсы:

* Позволяет создавать продукты пошагово.
* Позволяет использовать один и тот же код для создания различных продуктов.
* Изолирует сложный код сборки продукта от его основной бизнес-логики.

Минусы:

* Усложняет код программы из-за введения дополнительных классов.
* Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя может не быть метода получения результата.

Реализация: Методы setDistance и setNextChain в классе ShipmentCreator устанавливают значения для создаваемого объекта.

- **Синглтон** - это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Используется:

- Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).

- Когда необходимо иметь больше контроля над глобальными переменными.

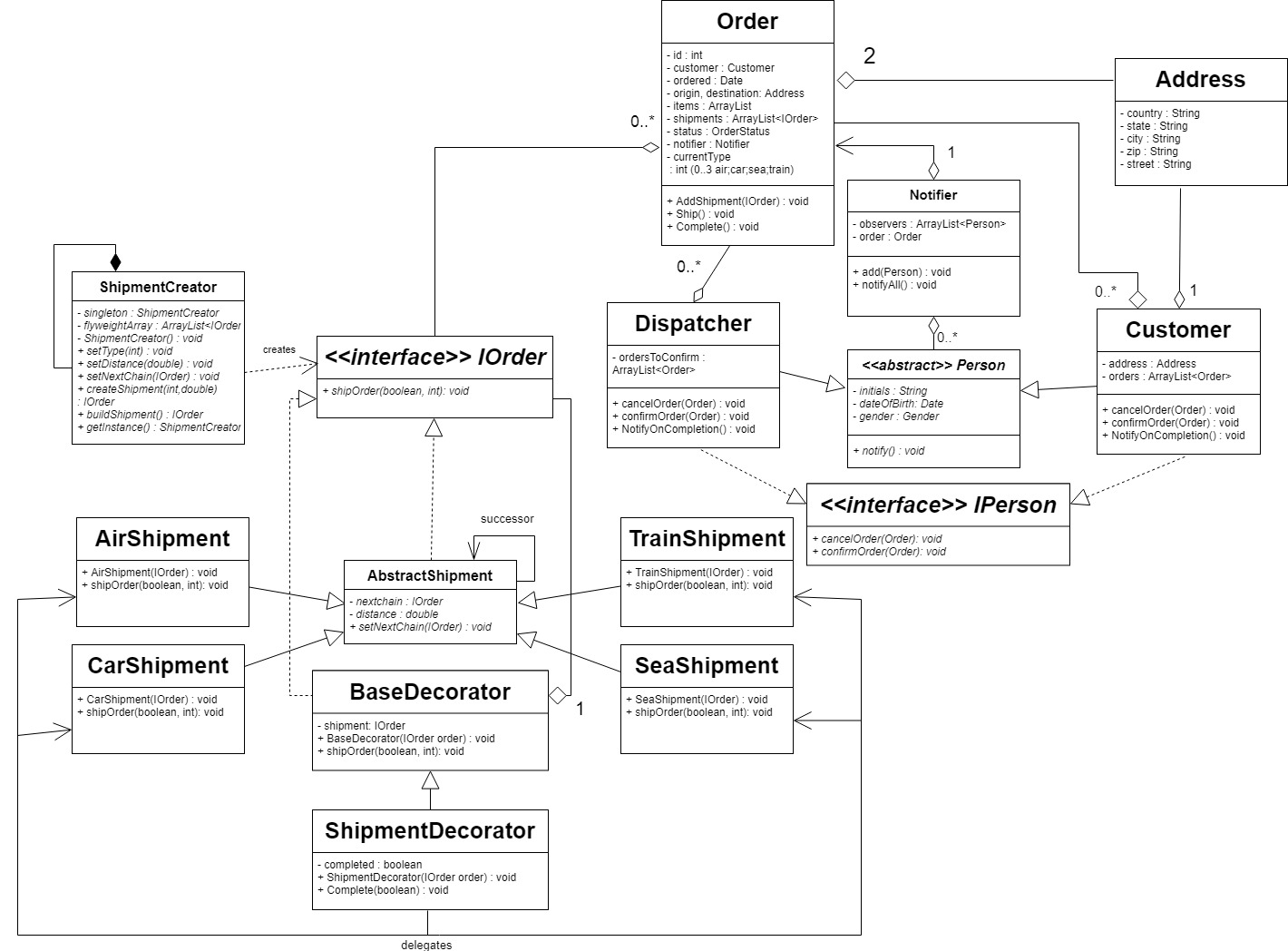
Плюсы:

* Гарантирует наличие единственного экземпляра класса.
* Предоставляет к нему глобальную точку доступа.
* Реализует отложенную инициализацию объекта-одиночки.

Минусы

* Нарушает *принцип единственной ответственности класса*.
* Маскирует плохой дизайн.
* Проблемы мультипоточности.
* Требует постоянного создания Mock-объектов при юнит-тестирования.

Реализация: Единственный экземпляр класса ShipmentCreator хранится в статическом поле singleton, и доступ к нему осуществляется через статическую функцию getInstance(), которая возвращает уже созданный экземпляр, либо создает его.



- **Наблюдатель** - это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Используется:

- Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд какие именно объекты должны отреагировать.

- Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

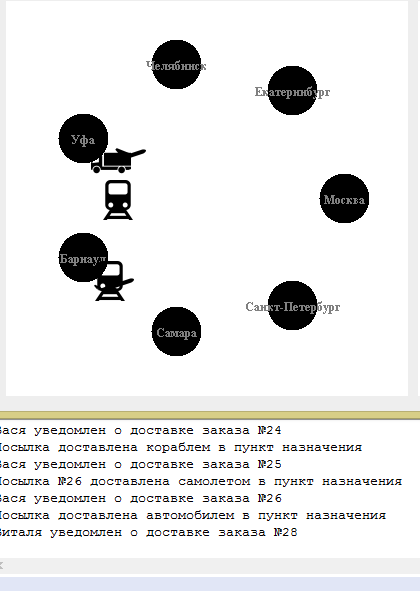
Плюсы:

* Издатели не зависят от конкретных классов подписчиков и наоборот.
* Вы можете подписывать и отписывать получателей на лету.
* Реализует *принцип открытости/закрытости*.

Минусы:

* Подписчики оповещаются в случайном порядке.

Реализация: В классе Notifier хранятся оповещаемые объекты класса Person. При вызове функции Completion класса Notifier все подписанные объекты уведомляются.



Исходный код

**interface** IPerson {

**public** **void** confirmOrder(Order o);

**public** **void** cancelOrder(Order o);

}

**interface** IOrder {

**public** **void** shipOrder(**boolean** destination, **int** type, Order order);

}

**abstract** **public** **class** Person {

**public** String initials;

**public** Date dateOfBirth;

**public** Gender gender;

Person(String \_i, Date \_d, Gender \_g) {

initials = \_i;

dateOfBirth = \_d;

gender = \_g;

}

**public** **void** notifyy(Order or) {

System.***out***.println(initials + " уведомлен о доставке заказа №" + or.id);

}

}

**public** **class** Dispatcher **extends** Person **implements** IPerson {

ArrayList<Order> ordersToConfirm;

Dispatcher(String \_i, Date \_d, Gender \_g) {

**super**(\_i, \_d, \_g);

ordersToConfirm = **new** ArrayList<Order>();

}

@Override

**public** **void** confirmOrder(Order o) {

System.***out***.println("Заказ №" + o.id + " подтвержден оператором " + **this**.initials);

o.status = OrderStatus.***CONFIRMED***;

removeOrderFromQueue(o);

}

@Override

**public** **void** cancelOrder(Order o) {

System.***out***.println("Заказ №" + o.id + " отменен оператором");

o.status = OrderStatus.***CANCELLED***;

removeOrderFromQueue(o);

}

**private** **void** removeOrderFromQueue(Order o) {

ordersToConfirm.remove(o);

}

}

**public** **class** Customer **extends** Person **implements** IPerson {

Address address;

ArrayList<Order> orders;

Customer(String \_i, Date \_d, Gender \_g, Address \_a) {

**super**(\_i, \_d, \_g);

address = \_a;

orders = **new** ArrayList<Order>();

}

@Override

**public** **void** confirmOrder(Order o) {}

@Override

**public** **void** cancelOrder(Order o) {

System.***out***.println("Заказ отменен покупателем");

o.status = OrderStatus.***CANCELLED***;

}

}

**public** **class** Address {

**public** String country;

**public** String state;

**public** String city;

**public** String zip;

**public** String street;

**public** Point2D.Double position;

Address(String \_country, String \_state, String \_city, String \_zip, String \_street) {

country = \_country;

state = \_state;

city = \_city;

zip = \_zip;

street = \_street;

position = **new** Point2D.Double();

}

}

**public** **class** ShipmentDecorator **implements** IOrder {

**protected** IOrder shipment;

**protected** IOrder nextChain;

**protected** **boolean** completed;

**public** ShipmentDecorator(IOrder order) {

shipment = order;

}

**public** **void** shipOrder(**boolean** destination, **int** type, Order order) {}

**public** **void** setNextChain(IOrder order) {

nextChain = order;

}

**public** **void** Complete(**boolean** comp) {

completed = comp;

}

}

**public** **class** TrainShipment **extends** ShipmentDecorator **implements** IOrder {

**public** TrainShipment(IOrder order) {

**super**(order);

}

@Override

**public** **void** shipOrder(**boolean** destination, **int** type, Order order) {

**double** fulldist = order.origin.position.distance(order.customer.address.position);

**double** dist = order.position.distance(order.customer.address.position);

**if** ((fulldist - dist)/fulldist > 1.\*(order.shipments.indexOf(**this**)+1)/order.shipments.size()-.03)

;

// System.out.printf("%g %g\n",(fulldist - dist)/fulldist, 1.\*(order.shipments.indexOf(this)+1)/order.shipments.size());

**else**

**return**;

**if** (type != 3) {

System.***out***.println("Посылка направлена в другой пункт");

order.current++;

**this**.nextChain.shipOrder(order.IsFinal(), type, order);

**return**;

}

System.***out***.print("Посылка доставлена поездом");

**if** (destination)

System.***out***.println(" в пункт назначения");

**else**

System.***out***.println(" в промежуточный пункт");

order.current++;

}

}

**public** **class** SeaShipment **extends** ShipmentDecorator **implements** IOrder {

**public** SeaShipment(IOrder order) {

**super**(order);

}

@Override

**public** **void** shipOrder(**boolean** destination, **int** type, Order order) {

**double** fulldist = order.origin.position.distance(order.customer.address.position);

**double** dist = order.position.distance(order.customer.address.position);

**if** ((fulldist - dist)/fulldist > 1.\*(order.shipments.indexOf(**this**)+1)/order.shipments.size()-.03)

;

// System.out.printf("%g %g\n",(fulldist - dist)/fulldist, 1.\*(order.shipments.indexOf(this)+1)/order.shipments.size());

**else**

**return**;

**if** (type != 2) {

System.***out***.println("Посылка направлена в другой пункт");

order.current++;

**this**.nextChain.shipOrder(order.IsFinal(), type, order);

**return**;

}

System.***out***.print("Посылка доставлена кораблем");

**if** (destination)

System.***out***.println(" в пункт назначения");

**else**

System.***out***.println(" в промежуточный пункт");

order.current++;

}

}

**public** **class** CarShipment **extends** ShipmentDecorator **implements** IOrder {

**public** CarShipment(IOrder order) {

**super**(order);

}

@Override

**public** **void** shipOrder(**boolean** destination, **int** type, Order order) {

**double** fulldist = order.origin.position.distance(order.customer.address.position);

**double** dist = order.position.distance(order.customer.address.position);

**if** ((fulldist - dist)/fulldist > 1.\*(order.shipments.indexOf(**this**)+1)/order.shipments.size()-.03)

;

// System.out.printf("%g %g\n",(fulldist - dist)/fulldist, 1.\*(order.shipments.indexOf(this)+1)/order.shipments.size());

**else**

**return**;

**if** (type != 1) {

System.***out***.println("Посылка направлена в другой пункт");

order.current++;

**this**.nextChain.shipOrder(order.IsFinal(), type, order);

**return**;

}

System.***out***.print("Посылка доставлена автомобилем");

**if** (destination)

System.***out***.println(" в пункт назначения");

**else**

System.***out***.println(" в промежуточный пункт");

order.current++;

}

}

**public** **class** AirShipment **extends** ShipmentDecorator **implements** IOrder {

**public** AirShipment(IOrder order) {

**super**(order);

}

@Override

**public** **void** shipOrder(**boolean** destination, **int** type, Order order) {

//**TODO**: shipments 1..n

//order.shipments.size() n

//order.shipments.indexOf(this) 1..n

**double** fulldist = order.origin.position.distance(order.customer.address.position);

**double** dist = order.position.distance(order.customer.address.position);

**if** ((fulldist - dist)/fulldist > 1.\*(order.shipments.indexOf(**this**)+1)/order.shipments.size()-.03)

;

// System.out.printf("%g %g\n",(fulldist - dist)/fulldist, 1.\*(order.shipments.indexOf(this)+1)/order.shipments.size());

**else**

**return**;

**if** (type != 0) {

System.***out***.println("Посылка №" + order.id + " направлена в другой пункт");

order.current++;

**this**.nextChain.shipOrder(order.IsFinal(), type, order);

**return**;

}

System.***out***.print("Посылка №" + order.id + " доставлена самолетом");

**if** (destination)

System.***out***.println(" в пункт назначения");

**else**

System.***out***.println(" в промежуточный пункт");

order.current++;

}

}

**public** **class** Item {

**public** String name;

**public** **int** quantity;

**public** **float** price;

Item(String \_n, **int** \_q, **int** \_p) {

name = \_n;

quantity = \_q;

price = \_p;

}

}

**public** **class** Notifier {

**private** ArrayList<Person> observers;

**public** Order order;

Notifier(Order o) {

observers = **new** ArrayList<Person>();

order = o;

}

**public** **void** Add(Person p) {

observers.add(p);

}

**public** **void** Completion() {

**for** (Person p : observers)

p.notifyy(order);

}

}

**public** **enum** OrderStatus {

***WAITING***, ***CONFIRMED***, ***SHIPPED***, ***CANCELLED***

}

**public** **class** Order {

**private** **static** **int** *nextId* = 0;

**private** Notifier notifier;

**public** **int** current = 0;

**public** **int** id;

**public** Customer customer;

**public** Date ordered;

**public** Address origin, destination;

**public** ArrayList<Item> items;

**public** ArrayList<IOrder> shipments;

**public** OrderStatus status;

**public** Point2D.Double position;

Order(Customer \_customer, Date \_ordered, Address \_origin) {

id = *nextId*++;

customer = \_customer;

ordered = \_ordered;

origin = \_origin;

destination = customer.address;

items = **new** ArrayList<Item>();

shipments = **new** ArrayList<IOrder>();

status = OrderStatus.***WAITING***;

notifier = **new** Notifier(**this**);

notifier.Add(customer);

position = **new** Point2D.Double();

// Main.disp.ordersToConfirm.add(this);

}

**public** **void** AddShipment(IOrder ship) {

shipments.add(ship);

}

**public** **void** Ship() {

// int type = GetCurrentType(i);

// for (int i = current; i < shipments.size(); i++)

**if** (current < shipments.size())

shipments.get(current).shipOrder(IsFinal(), GetCurrentType(current), **this**);

}

@Override

**public** String toString() {

**return** "\nЗаказ №" + id + " для клиента " + customer.initials + ":";

}

**public** **void** Complete() {

notifier.Completion();

}

**public** **boolean** IsFinal() {

**return** current == shipments.size()-1 ? **true** : **false**;

}

**public** **int** GetCurrentType(**int** index) {

**int** type = -1;

**switch** (shipments.get(index).getClass().getSimpleName()) {

**case** "AirShipment":

type = 0;

**break**;

**case** "CarShipment":

type = 1;

**break**;

**case** "SeaShipment":

type = 2;

**break**;

**case** "TrainShipment":

type = 3;

**break**;

}

**return** type;

}

}