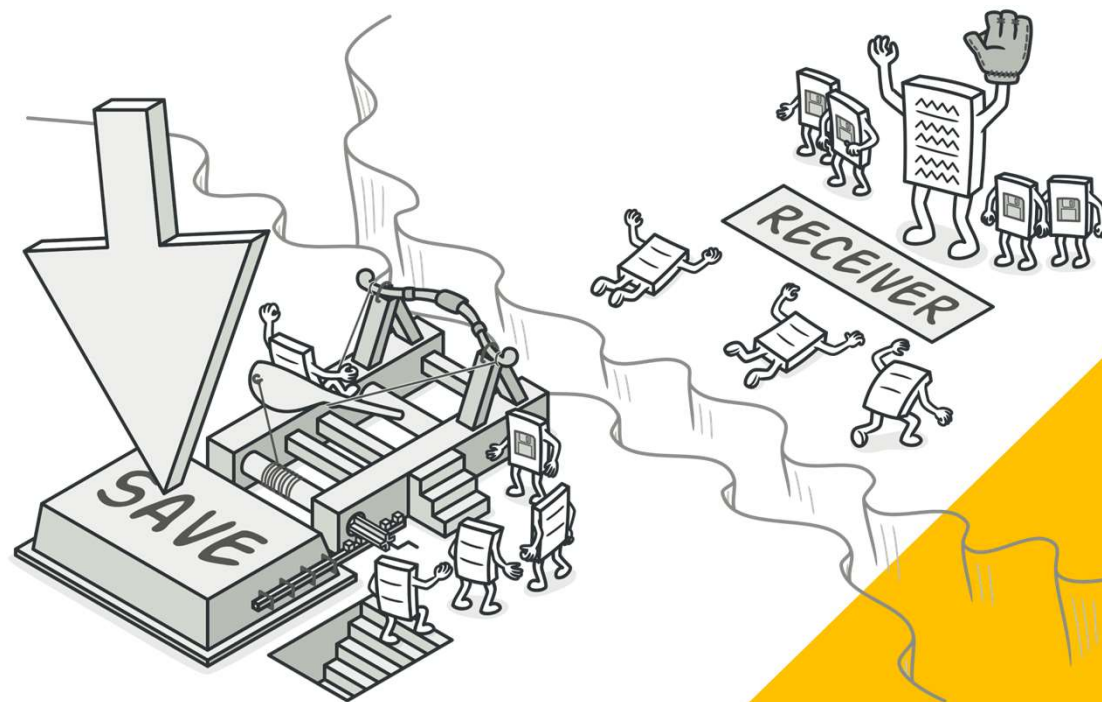


КОМАНДА



КОМАНДА

поведінковий патерн проектування, який перетворює запити на об'єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

УМОВА

- Припустимо, Ви пишете сайт інтернет магазину і у клієнта є декілька можливих операцій з замовленням: додати, видалити, оновити товари у ньому.
- Всі операції працюють з певним «станом» – списком товарів.
- Результат виконання операції – зміна «стану».
- Ми не хочемо змінювати кожний продукт окремо, а зробити декілька змін за раз і зберегти їх.

Виглядає трохи complicated, правда?

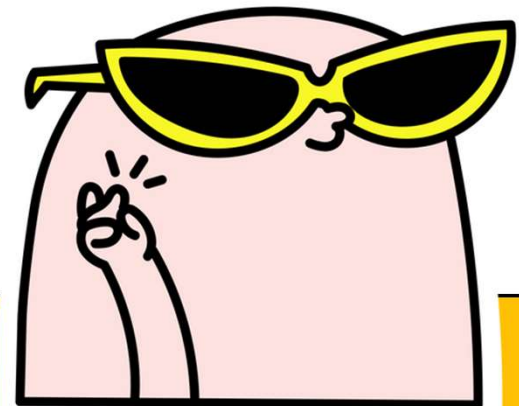


ЦІЛЬ ПАТТЕРНУ

Реалізація структури операцій таким чином, щоб відправнику не потрібно було «напружуватись» і розбиратись, що потрібно саме відправити, щоб обробник зрозумів задачу правильно.

РІШЕННЯ

Класи команд можна об'єднати під загальним інтерфейсом, що має єдиний метод запуску команди. Після цього одні й ті самі відправники зможуть працювати з різними командами, не прив'язуючись до їхніх класів. Навіть більше, команди можна буде взаємозамінити «на льоту», змінюючи підсумкову поведінку відправників.



А ЯК БУТИ З ДАНИМИ, ЩО ПЕРЕДАЮТЬСЯ З «КОМАНДОЮ»?

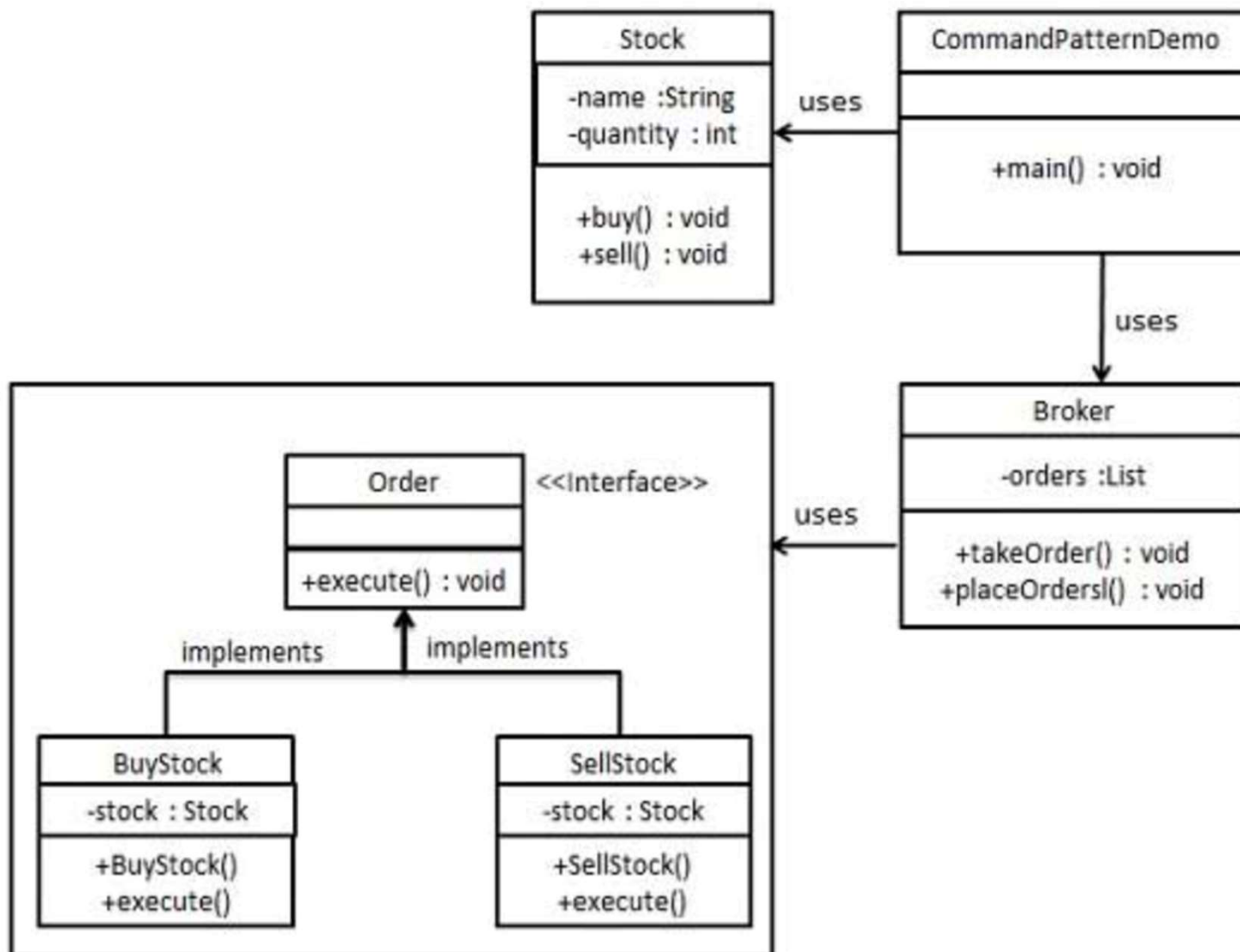
Параметри, з якими повинен бути викликаний метод об'єкта одержувача, можна заздалегідь зберегти в полях об'єкта-команди.

Завдяки цьому, об'єкти, які надсилають запити, можуть не турбуватися про те, щоб зібрати необхідні дані для одержувача. Навіть більше, вони тепер взагалі не знають, хто буде одержувачем запиту. Вся ця інформація прихована всередині команди.




ДІЙОВІ ОСОБИ

- Інтерфейс команди з методом її запуску.
- «Стан» з усіма потрібними командам даними.
- Реалізації команд від інтерфейсу з «станом» усередині і власне реалізацією того, що вони будуть робити



```
public interface Order {  
    void execute();  
}
```



```
public class BuyStock implements Order {
```

```
    private Stock abcStock;  
    public BuyStock(Stock abcStock){  
        this.abcStock = abcStock;  
    }  
    public void execute() {  
        abcStock.buy();  
    }  
}
```

```
public class SellStock implements Order {
```

```
    private Stock abcStock;  
    public SellStock (Stock abcStock){  
        this.abcStock = abcStock;  
    }  
    public void execute() {  
        abcStock.sell();  
    }  
}
```

```
public class Stock {  
    private String name = "ABC";  
  
    private int quantity = 10;  
  
    public void buy(){  
        System.out.println("Stock [ Name: "+name+", Quantity: " + quantity +" ] bought");  
    }  
  
    public void sell(){  
        System.out.println("Stock [ Name: "+name+", Quantity: " + quantity +" ] sold");  
    }  
}
```

```
public class Broker {  
  
    private List<Order> orderList = new ArrayList<Order>();  
  
    public void takeOrder(Order order){  
        orderList.add(order);  
    }  
  
    public void placeOrders(){  
        for (Order order : orderList) { order.execute(); }  
        orderList.clear();  
    }  
}
```

```
public class Demo {  
  
    public static void main(String[] args) {  
        Stock abcStock = new Stock();  
  
        BuyStock buyStockOrder = new BuyStock(abcStock);  
        SellStock sellStockOrder = new SellStock(abcStock);  
  
        Broker broker = new Broker();  
  
        broker.takeOrder(buyStockOrder);  
        broker.takeOrder(sellStockOrder);  
        broker.placeOrders();  
    }  
}
```

ПЕРЕЙДЕМО ДО КОДУ



ІНШІ ПРИКЛАДИ

- Кнопки і шорткати інтерфейсу редактора тексту (аналогічно до нашого прикладу про операції над замовленням)
- Передача замовлення від Вас до кухаря (через офіціанта, який буквально розводить «команди»-блюда по кухарям).

КОЛИ ТРЕБА «КОМАНДА»

- Якщо ви хочете параметризувати об'єкти виконуваною дією
- Якщо ви хочете поставити операції в чергу, виконувати їх за розкладом або передавати мережею.
- Якщо вам потрібна операція скасування

ПАТЕРН «КОМАНДА» ТА ЗБЕРІГАННЯ СТАНУ

Знімок — це поведінковий патерн проектування, що дає змогу зберігати та відновлювати минулий стан об'єктів, не розкриваючи подробиць їхньої реалізації.

Патерн пропонує тримати копію стану в спеціальному об'єкті-знімку з обмеженим інтерфейсом, що дозволяє, наприклад, дізнатися дату виготовлення або назву знімка. Проте, знімок повинен бути відкритим для свого *творця* і дозволяти прочитати та відновити його внутрішній стан.

ПЛЮСИ ТА МІНУСИ



- Прибирає пряму залежність між об'єктами, що викликають операції, та об'єктами, які їх безпосередньо виконують.
- Дозволяє реалізувати просте скасування і повтор операцій.
- Дозволяє реалізувати відкладений запуск операцій.
- Дозволяє збирати складні команди з простих.
- Реалізує *принцип відкритості/закритості*.



Ускладнює код програми внаслідок введення великої кількості додаткових класів.

З ІНШИМИ ПАТТЕРНАМИ

Обробники в Ланцюжкові обов'язків можуть бути виконані у вигляді Команд.

В цьому випадку роль запиту відіграє контекст команд, який послідовно подається до кожної команди у ланцюгу.

Але є й інший підхід, в якому сам запит є *Командою*, надісланою ланцюжком об'єктів.

У цьому випадку одна і та сама операція може бути застосована до багатьох різних контекстів, представлених у вигляді ланцюжка.

З ІНШИМИ ПАТТЕРНАМИ

Команда та **Стратегія** схожі за принципом, але відрізняються масштабом та застосуванням.

Команду використовують для перетворення будь-яких різнорідних дій на об'єкти. Параметри операції перетворюються на поля об'єкта. Цей об'єкт тепер можна логувати, зберігати в історії для скасування, передавати у зовнішні сервіси тощо.

З іншого боку, *Стратегія* описує різні способи того, як зробити одну і ту саму дію, дозволяючи замінювати ці способи в якомусь об'єкті контексту прямо під час виконання програми