Inventory Class

이 Class는 HolubSQL을 통해 사용자에게 Product와 Stock에 대한 정보를 받아 그 실제 데이터를 clone하여 저장하고, 값의 변화를 인식하고 다른 Class에 해당 내용을 전달하는 역할을 한다. 이는 Mediator Pattern의 Mediator역할이며, Product, Stock, Shop 등등 우리가 구현한 대부분의 class의 중심이 되어 상호 커뮤니케이션을 돕는다.

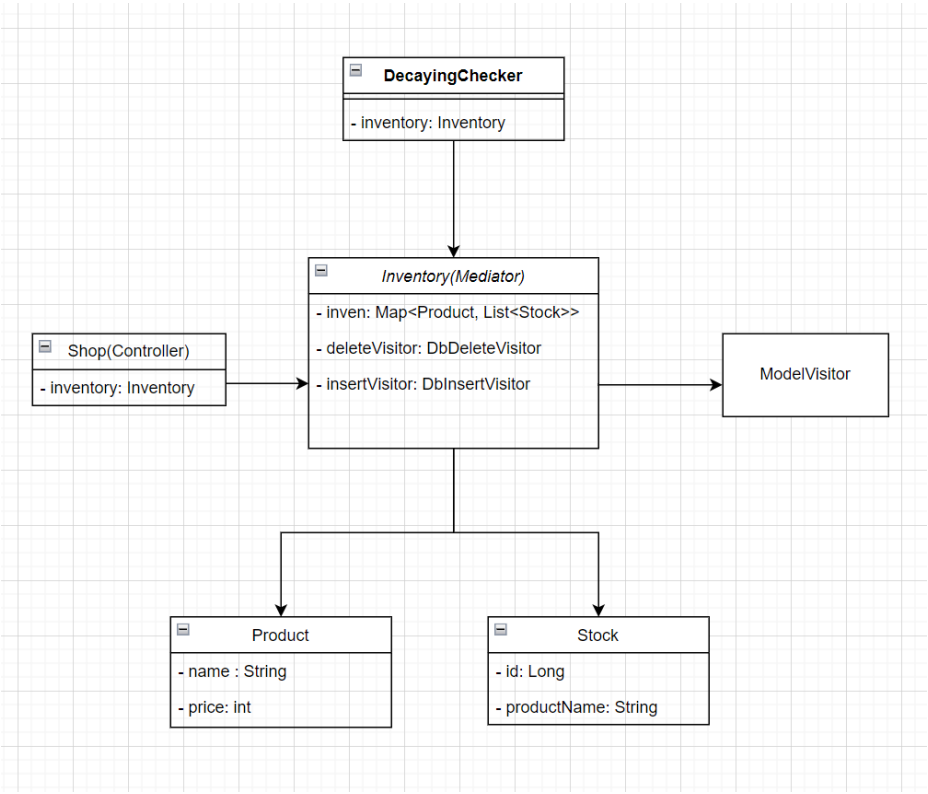
Inventory에는 모든 Product를 키 값으로, Stock 리스트가 HashMap으로 연결되어 다른 클래스에서 특정 Product의 이름으로 Stock 리스트에 대한 제어를 요청할 수 있다.

해당 HashMap은 인스턴스 생성시 DB 팩토리 모델에서 기존에 저장되어 있던 DB정보를 받아 저장하게 된다.

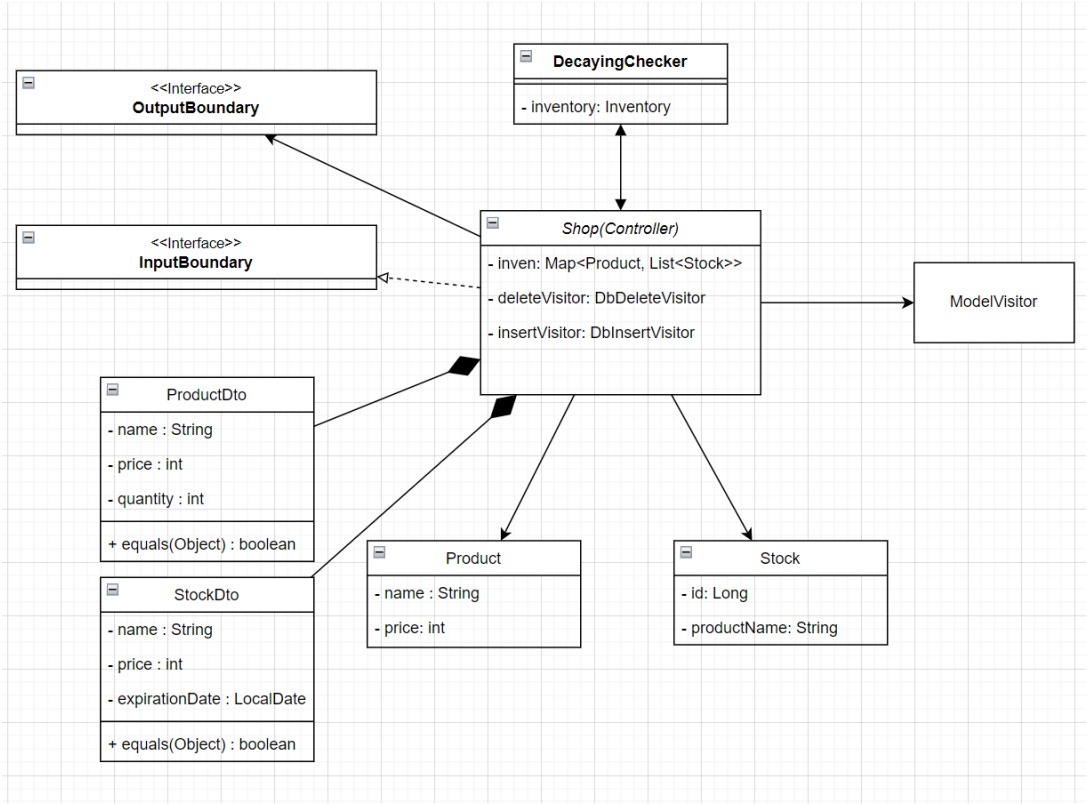
이후 Client로부터 품목 추가, 재입고 등 실시간으로 값의 변화를 받고, 현재 다루고 있는 HashMap뿐만 아니라 DB까지 연동시키기 위해 Visitor Model을 적용시킨 DB입력부와 연결되어 있다.

유통기한 검사 기능을 하는 Decaying Checker역시 inventory를 멤버변수로 받아 Stock리스트에서 유통기한이 지난 항목에 대한 처리를 요청하게 된다.

아래는 대략적인 Inventory와 그 주변 Colleague들의 관계를 나타낸 다이어그램이다



만약 Inventory같은 중개자가 없어진다면



위 다이어그램과 같이 Shop(Controller) 혼자서 IO부분과 시스템 내부 관계 모두 다루게 된다. 이 상태로 기능이 확장되어 Input, output에서 요청하는 데이터의 종류가 많아질수록 Shop에 대한 의존도는 계속 높아질 것이다.

코드 설명

실제 데이터로 저장할 공간 inven과 각각 Visitor Pattern이 적용된 delete, insert 변수를 정의하고, inventory생성자로 DB에 있는 모든 데이터를 가져온다.

```
private final Map<Product, List<Stock>> inven = new ConcurrentHashMap<>();
3개 사용 위치
private final DbDeleteVisitor deleteVisitor = new DbDeleteVisitor();
3개 사용 위치
private final DbInsertVisitor insertVisitor = new DbInsertVisitor();

2개 사용 위치  박동윤
public InventoryImpl() {    박동윤, Today • Extract Inventory interface
    for (Product product : DbModelFactory.getAllProducts()) {
        List<Stock> stocks = DbModelFactory.getAllStocksByProduct(product.getName());
        inven.put(product, stocks);
    }
}
```

expDate를 따로 parameter로 받지 않으므로 count만큼 UndecayingStock을 생성하여 해당 Product를 찾아 리스트에 추가해준다. 또한 Visitor를 사용해 DB에 바로 업데이트.

```
public List<Stock> addStock(String productName, int count){
    Product keyProduct = findProduct(productName);
    List<Stock> stocks = inven.get(keyProduct);
    if (stocks == null) {
        throw new NoSuchProductException();
    }
    for (int i = 0; i < count; i++) {
        Stock newStock = new UndecayingStock(productName);
        inven.get(keyProduct).add(newStock);
        newStock.accept(insertVisitor);
    }
    return stocks;
}
```

Parameter에 localDate가 추가된 경우 count에 맞춰 DecayingStock을 생성하고, 기존에 있던 리스트와 expDate를 비교하여 오름차순으로 정렬하는 과정을 거친다.

```
@Override
public void addStock(String productName, int count, LocalDate expDate){
    Product keyProduct = findProduct(productName);
    List<Stock> stocks = inven.get(keyProduct);
    if (stocks == null) {    박동윤, Today • Extract Inventory interface
        throw new NoSuchProductException();
    }
    for (int i = 0; i < count; i++) {
        Stock newStock = new DecayingStock(productName, expDate);
        inven.get(keyProduct).add(newStock);
        박동윤
        inven.get(keyProduct).sort(new Comparator<Stock>(){
            박동윤
            public int compare(Stock o1, Stock o2){
                return o1.getExpirationDate().compareTo(o2.getExpirationDate());
            }
        });
        newStock.accept(insertVisitor);
    }
}
```

특정 Stock을 찾아 리스트에서 지우는 과정이다.

```
public void deleteStock(Stock stock) {
    List<Stock> stocks = inven.get(new Product(stock.getProductName(), price: 0));
    stocks.remove(stock);    박동윤, Today • Extract Inventory interface
    stock.accept(deleteVisitor);
}
```

DecayingChecker에서 활용할 모든 Stock데이터를 반환하는 메소드

```
@Override
public List<Stock> getAllStocks() {
    //generate flatten all list of stock into one list
    return inven.values().stream().flatMap(List::stream).toList();
}
```

새로운 Product를 추가하는 메소드. 초기에는 비어 있는 리스트를 추가한다.

```
@Override
public synchronized void addProduct(String productName, int price){
    Product newProduct = new Product(productName, price);
    List<Stock> stockList = new ArrayList<>();
    inven.put(newProduct, stockList);
    newProduct.accept(insertVisitor);
}
```

해당 Product가 몇 개의 Stock을 갖고 있는지 확인하는 메소드

```
@Override
public int getProductsQuantity(String productName) {
    Product keyProduct = new Product(productName, price: 0);
    if (inven.get(keyProduct) != null) {
        return inven.get(keyProduct).size();
    } else {
        return 0;
    }    박동윤, Today • Extract Inventory interface
}
```

Product삭제 메소드

```java
@Override
public synchronized void deleteProduct(String productName){
    Product keyProduct = findProduct(productName);

    inven.remove(keyProduct);
    keyProduct.accept(deleteVisitor);
}
```

Product 이름으로 Product를 찾아서 반환하는 메소드. 찾지 못하면 NoSuchProductException이 발생한다.

```java
@Override
public Product findProduct(String name){
    Product comp = new Product(name,  price: 0);

    return inven.keySet().stream() Stream<Product>
            .filter(stocks -> stocks.equals(comp))
            .findFirst() Optional<Product>
            .orElseThrow(NoSuchProductException::new);
}
```

정해진 숫자 만큼 리스트에서 Stock을 지우는 메소드. MultiThreading문제로 반복문과 제거하는 메소드를 분리하였고, 처음 keyProduct에서 findProduct로 NoSuchProductException 검사 후 제거하는 메소드에서 OutOfStockException을 검사하게 된다.

```java
@Override
public synchronized void sell(String productName, int count) {
    Product keyProduct = findProduct(productName);
    for (int i = 0; i < count ; i++) {
        sell(keyProduct);
    }

}

1개 사용 위치  박동윤
private synchronized void sell(Product target) {
    synchronized (inven) {
        List<Stock> productStocks = inven.getOrDefault(target, new ArrayList<>());
        Stock stockToSell = productStocks
            .stream().findFirst()
            .orElseThrow(OutOfStockException::new);
        productStocks.remove(stockToSell);
        stockToSell.accept(deleteVisitor);    박동윤, Today • Extract Inventory inter
    }
}
```

*JUnit으로 테스트 진행

Inventory 인스턴스와 이를 활용할 임시 productName 정의한다.

```
Inventory inventory = new InventoryImpl();
15개 사용 위치
final String testProductName = "test product name";
```

Inventory의 addProduct 메소드와 findProduct 메소드가 정상적으로 작동하는지 확인한다.

```
@Test
@Order(100)
void insertProduct() {
    // given
    inventory.addProduct(testProductName,  price: 12345);

    // when
    Product product = inventory.findProduct(testProductName);

    // then
    assertThat(product.getName()).isEqualTo(testProductName);
    assertThat(product.getPrice()).isEqualTo( expected: 12345);
}
```

이어서 추가했던 Product에 Stock 10개 추가하여 size가 맞는지 확인한다.

```
@Test
@Order(110)
void insertStock() {
    // given
    inventory.addStock(testProductName,  count: 10);

    // when
    int productsQuantity = inventory.getProductsQuantity(testProductName);

    // then
    assertThat(productsQuantity).isEqualTo( expected: 10);
}
```

Stock을 한 개 팔아서 9개가 정상적으로 되었는지 확인한다.

```
@Test   박동윤, Today • Refactor name (remove → delete) for naming consist
@Order(120)
void deleteStock() {
    // given
    int productsQuantity = inventory.getProductsQuantity(testProductName);
    assertThat(productsQuantity).isEqualTo( expected: 10);

    // when
    inventory.sell(testProductName,  count: 1);

    // then
    productsQuantity = inventory.getProductsQuantity(testProductName);
    assertThat(productsQuantity).isEqualTo( expected: 9);
}
```

생성되었던 Product를 찾아 삭제하는데, 삭제 후 findProduct를 돌렸을 때 NoSuchProductException이 발생하는지 확인한다.

```
@Test
@Order(130)
void deleteProduct() {
    // given
    inventory.findProduct(testProductName);

    // when
    inventory.deleteProduct(testProductName);

    // then
    assertThatThrownBy(() -> inventory.findProduct(testProductName))
        .isInstanceOf(NoSuchProductException.class);

}
```

랜덤으로 LocalDate를 생성해주는 메소드를 정의하고,

```
private LocalDate getRandomDate(long minDay, long maxDay){
    long randomDay = ThreadLocalRandom.current().nextLong(minDay, maxDay);
    LocalDate randomDate = LocalDate.ofEpochDay(randomDay);
    return randomDate;
}
```

두개의 LocalDate를 생성, 날짜가 느린 순서부터 Stock에 추가해준다. 정상적으로 정렬이 되었다면 삭제되는 Stock 인스턴스는 날짜가 더 빠른 Stock일 것이고, 이를 확인하는 작업이다.

```java
@Test
@Order(140)
void addDecayingStock(){
    inventory.addProduct(testProductName,  price: 10000);
    LocalDate latterDate = getRandomDate(LocalDate.of( year: 1999,  month: 12,  dayOfMonth: 31).toEpochDay(),
            LocalDate.of( year: 2023,  month: 11,  dayOfMonth: 30).toEpochDay());
    LocalDate formerDate = getRandomDate(LocalDate.of( year: 1999,  month: 12,  dayOfMonth: 31).toEpochDay(),
            LocalDate.of( year: 2023,  month: 11,  dayOfMonth: 30).toEpochDay());


    if(latterDate.isBefore(formerDate)){
        LocalDate temp = latterDate;
        latterDate = formerDate;
        formerDate = temp;
    }

    inventory.addStock(testProductName,  count: 1, latterDate);
    inventory.addStock(testProductName,  count: 1, formerDate);


    inventory.sell(testProductName,  count: 1);

    assertThat(inventory.getAllStocks().get(0).getExpirationDate().equals(latterDate))
            .isTrue();

}
```