Esame di Programmazione II, 28 luglio 2017

Si consideri un'implementazione di un negozio, al quale è possibile aggiungere dei prodotti in vendita e fare degli ordini di acquisto. Per esempio:

```
public class Main {
  public static void main(String[] args) throws MissingProductException {
    Product bike = new Product("bike", 300.0, 7); // una bicicletta costa 300 euro ed e' disponibile per la spedizione in 7 giorni
    Product phone = new Product("phone", 129.9, 1); // un telefono costa 129.9 euro ed e' disponibile per la spedizione in un giorno
    Shop amazing = new Shop(); amazing.add(bike, 3); amazing.add(phone, 4); // il negozio amazing ha disponibili 3 bici e 4 telefoni
    // creiamo tre ordini, due semplici e uno dividendo i prodotti per attesa di spedizione
    Order order1 = new SimpleOrder(amazing, bike, phone, phone); // una bici e due telefoni
    Order order2 = new SplitOrder(amazing, phone, bike, phone); // una bici e due telefoni
    Order order3 = new SimpleOrder(amazing, bike, phone); // una bici e un telefono
    // effettuiamo i tre ordini e stampiamo le spedizioni che ne risultano
    printShipping("FIRST ORDER:", order1.ship());
    printShipping("SECOND ORDER:", order2.ship());
    printShipping("THIRD ORDER:", order3.ship());
  }

  private static void printShipping(String title, Iterable<Shipping> shippings) {
    System.out.println(title + '\n');
    int counter = 1;
    for (Shipping shipping: shippings)
      System.out.println("shipping #" + counter++ + '\n' + shipping);
    System.out.println();
  }
}
```

stamperà:

```
FIRST ORDER:

shipping #1
bike: 300.00 euros, available in 7 days
phone: 129.90 euros, available in 1 days
phone: 129.90 euros, available in 1 days

SECOND ORDER:

shipping #1
phone: 129.90 euros, available in 1 days
phone: 129.90 euros, available in 1 days

shipping #2
bike: 300.00 euros, available in 7 days

Exception in thread "main" it.univr.ecommerce.MissingProductException
...
```

terminando con un'eccezione. Si noti che un `SimpleOrder` viene spedito mettendo tutti i prodotti in un'unica spedizione, mentre uno `SplitOrder` fa tante spedizioni, dividendo i prodotti per giorni di attesa prima della loro disponibilità alla spedizione. La spedizione dell'ultimo `SimpleOrder` va in eccezione poiché non ci sono più telefoni disponibile nel negozio, essendo stati già tutti spediti con i primi due ordini.

Esercizio 1 [4 punti] Si completi la seguente classe `Product`:

```
public class Product {
  private final String name;
  private final double price;
  private final int daysBeforeShipping;

  public Product(String name, double price, int daysBeforeShipping) { ... }

  @Override public String toString() {
    return String.format("%s: %.2f euros, available in %d days", name, price, daysBeforeShipping);
  }

  @Override public boolean equals(Object other) { ...confronta tutti e tre i campi }
  @Override public int hashCode() { ...consistente con equals() e non banale }

  public int getDaysBeforeShipping() {
    return daysBeforeShipping;
  }
}
```

Esercizio 2 [**9 punti**] Si completi la seguente classe che rappresenta un negozio a cui è possibile aggiungere prodotti in vendita e da cui è possibile comprare prodotti (se disponibili):

```
public class Shop {
  ...
  public void add(Product product, int howMany) {
    ...aggiunge howMany volte il prodotto indicato, che poteva gia' essere presente in negozio
  }

  void buy(Product[] productsToBuy) throws MissingProductException {
    ...rimuove i prodotti indicati da quelli disponibili in questo negozio;
    se non fossero disponibili tutti i prodotti, lancia una MissingProductException;
    in tal caso, il negozio dovra' restare immutato e nessun prodotto dovra' venire tolto
  }
}
```

Esercizio 3 [**1 punti**] Si implementi l'eccezione controllata `MissingProductException`.

Esercizio 4 [**3 punti**] Si completi la seguente classe, che rappresenta una spedizione di alcuni prodotti:

```
public class Shipping {
  ...
  Shipping(Iterable<Product> products) { ...crea una spedizione dei prodotti indicati }
  @Override public String toString() { ...ritorna la concatenzione del toString() dei prodotti spediti }
}
```

Esercizio 5 [**5 punti**] Si consideri la classe astratta che rappresenta un ordine da passare a un negozio per acquistare certi prodotti:

```
public abstract class Order {
  private final Shop shop; // il negozio a cui si passa l'ordine
  private final Product[] products; // i prodotti che si vuole acquistare in tale negozio

  protected Order(Shop shop, Product... products) {
    this.shop = shop; this.products = products;
  }

  protected final Product[] getProducts() { return products; }

  // acquista i prodotti di questo ordine, andando in eccezione se non sono tutti disponibili in negozio
  protected final void buy() throws MissingProductException { shop.buy(products); }

  // acquista i prodotti di questo ordine, andando in eccezione se non sono tutti disponibili in negozio,
  // e ritorna le spedizioni da fare per inviarli
  public abstract Iterable<Shipping> ship() throws MissingProductException;
}
```

Se ne completi la sottoclasse che spedisce tutti i prodotti in un'unica spedizione:

```
public class SimpleOrder extends Order {
  public SimpleOrder(Shop shop, Product... products) { ... }
  @Override public Iterable<Shipping> ship() throws MissingProductException {
    ...compra i prodotti e ritorna un iterabile con un'unica spedizione
  }
}
```

Esercizio 6 [**9 punti**] Si completi l'altra sottoclasse di `Order`, che spedice i prodotti da acquistare in tante spedizioni, dividendoli per giorni di attesa prima della loro disponibilità all'invio:

```
public class SplitOrder extends Order {
  public SplitOrder(Shop shop, Product... products) { ... }
  @Override public Iterable<Shipping> ship() throws MissingProductException {
    ...compra i prodotti e ritorna una o piu' spedizioni
  }
}
```

È possibile definire campi, metodi, costruttori e classi aggiuntive, ma solo `private`.