Exercises for lecture “Software Engineering 2“ – *Design Patterns*

1 Refactoring an Application with the help of Design Patterns

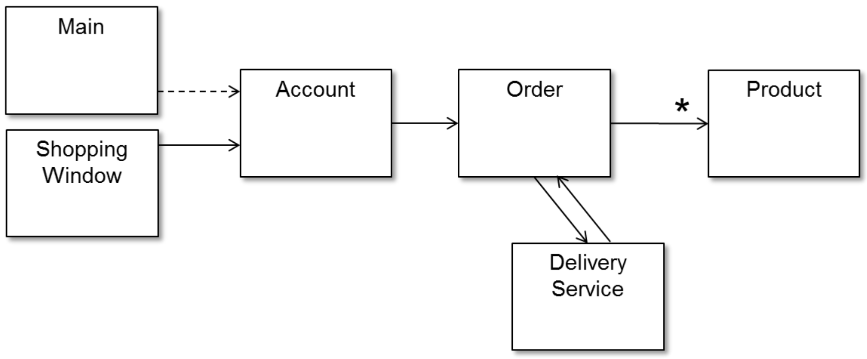
For the exercises you find in Moodle the file `SE2 - Exercise (Patterns) - Initial.zip.` The file contains the program, which you should refactor with the help of patterns during the exercise. Also included are two jar-files: a graphical user interface and a class to be adapted. As part of the pre-exam you should submit a document with the final UML and answering the questions of chapter 0.

1.0 Preparation / Initial Version

Download and unpack the zip-file. Create a Java-Project and drag the folder `shop` (with its subfolders) into the source-folder. Create a general project called `libs` and drag the jar-files into this folder:

- The `shop`-folder contains the (fully functional) system you should refactor. You can run and test the functionality by running the class `Main` (in the package `shop.model.main`).
- `shop-io.jar` contains a graphical user interface on the above application and some sample data. If you add the jar-file to your build path you can run the GUI by calling the class `ShoppingWindow` (see below).[1]
- `articles.jar` contains a class (and some sample data for the GUI) which should be adapted in the first step of the exercise.
- The reworked classes `Observer` and `Observable` which may be used for step 4 instead of the deprecated classes

Now you are the maintenance manager of an on-line-shop with products such as CDs or DVDs to sell. Customers create an account and add products to an order. Additionally, the option (with extra charge) exists for express-delivery and/or tracking the delivery. Customers may pay with direct debit, credit card or transfer pays. The following UML diagram shows the basic structure.



[1] To my best knowledge the graphical user interface should grow with your application if you stick to the given names and follow the exercise precisely – if there are any issues, you can always unpack the jar-file and use/change the contained java-files.

The following sample program simulates the creation of an account and completion of an order. For the sample the products are created on the fly – the GUI loads them from a list. Products are constructed with a *type* (as String converted to a Type), a *name*, a *price* (given as double/€, but converted to int/¢ during construction for precise calculation, and the *days to deliver* (with 0 for downloads, which are immediately available).

```java
public class Main {

    public static void main(String[] args) throws InterruptedException{

        // Create the account
        Account account = new Account("John", "Doe", "555 555 123",
                                                  "john.doe", "secret");
        account.login("john.doe", "secret");

        account.setDirectDebit("500600", "12345678");

        // Create and add some products
        account.add(new Product("download", "Java eBook", 9.99, 0));
        account.add(new Product("CD", "Some Music", 4.99, 1));
        account.add(new Product("BluRay", "A Movie", 8.99, 2));
        account.add(new Product("BluRay", "A Second Movie", 8.99, 2));
        account.add(new Product("Book", "Java Book", 6.99, 3));
        account.add(new Product("DVD", "Another Movie", 8.99, 4));
        account.add(new Product("CD", "Some Rare Music", 10.99, 5));
        account.add(new Product("Book", "Some Rare Book", 10.99, 6));

        // Add options
        account.addExpress();
        account.addTracking();

        // Complete the order
        account.complete();

        // logout after some time
        Thread.sleep(10000);
        account.logout();
        System.out.println("logged out");
    }
}
```

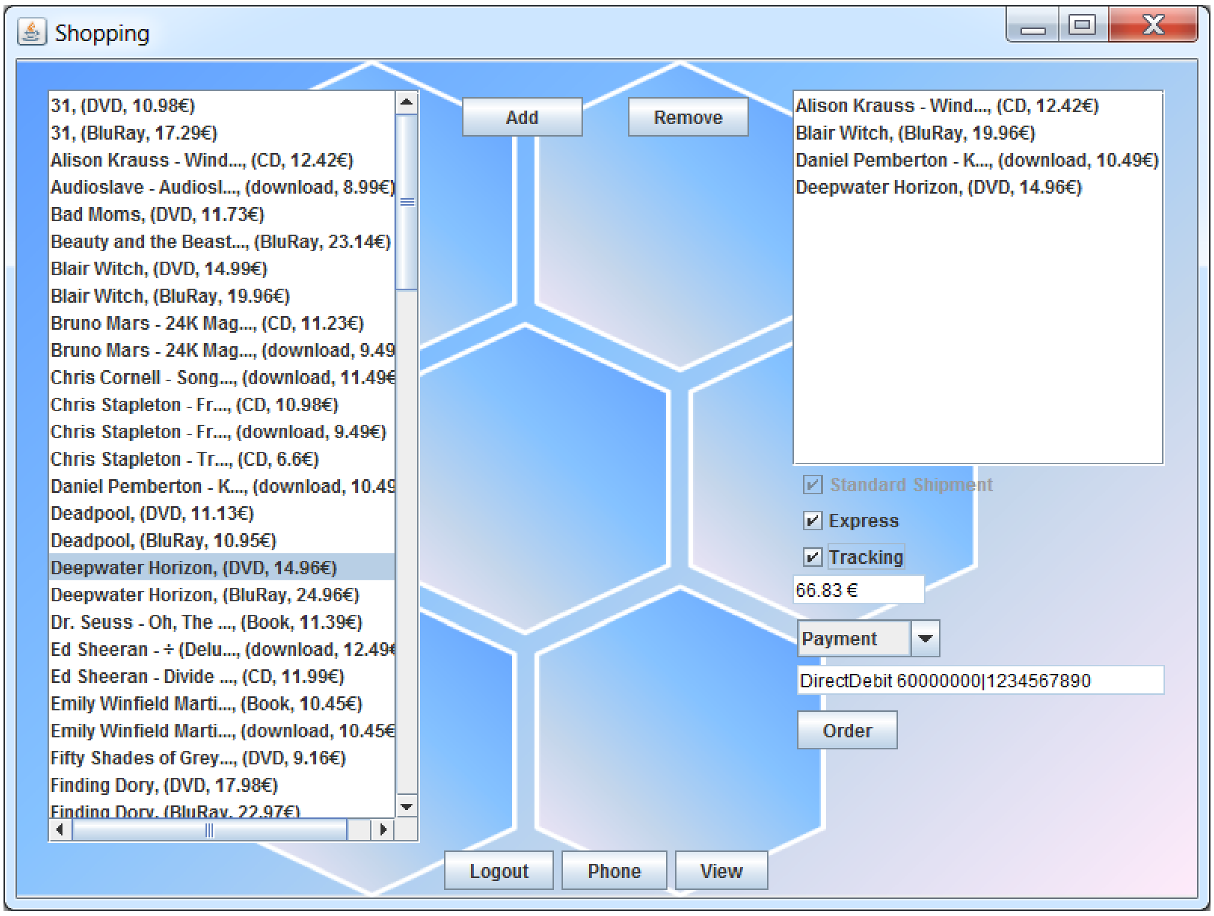
This example gives the following results (because of the tracking).

```
Order of Thu Jun 08 11:27:50 CEST
2017: 79.92€ will be transfered from
your account.

Sent out:
- Java eBook, (download, 9.99€)
- Some Music, (CD, 4.99€)
- A Movie, (BluRay, 8.99€)
- A Second Movie, (BluRay, 8.99€)
- Java Book, (Book, 6.99€)
- Another Movie, (DVD, 8.99€)
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)
```

```
Tracking started

Day 0, delivered:
* Java eBook, (download, 9.99€)

Day 0, still on its way:
- Some Music, (CD, 4.99€)
- A Movie, (BluRay, 8.99€)
- A Second Movie, (BluRay, 8.99€)
- Java Book, (Book, 6.99€)
- Another Movie, (DVD, 8.99€)
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)
```

```
Day 1, delivered:
* Some Music, (CD, 4.99€)
* A Movie, (BluRay, 8.99€)
* A Second Movie, (BluRay, 8.99€)
* Java Book, (Book, 6.99€)

Day 1, still on its way:
- Another Movie, (DVD, 8.99€)
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)

Day 2, delivered:
* Another Movie, (DVD, 8.99€)

Day 2, still on its way:
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)

Day 3, delivered:
* Some Rare Music, (CD, 10.99€)

Day 3, still on its way:
- Some Rare Book, (Book, 10.99€)

logged out
Day 4, delivered:
* Some Rare Book, (Book, 10.99€)

Tracking finished

Everything delivered after 4 days
```

Alternatively (after adding the `shop-io.jar`) you can also call the graphical user interface. You may use the existing accounts in in `shop.data` (e.g. `john.doe` | with password “secret”) but you can also create your own by pressing “Register”. Accounts are stored in the file `bin/shop/data/accounts.csv` – apart from the password (which is the hashed password) you can also change the account-data manually. The following screen-shot shows the running application.



31, (DVD, 10.98€)
31, (BluRay, 17.29€)
Alison Krauss - Wind..., (CD, 12.42€)
Audioslave - Audiosl..., (download, 8.99€)
Bad Moms, (DVD, 11.73€)
Beauty and the Beast..., (BluRay, 23.14€)
Blair Witch, (DVD, 14.99€)
Blair Witch, (BluRay, 19.96€)
Bruno Mars - 24K Mag..., (CD, 11.23€)
Bruno Mars - 24K Mag..., (download, 9.49
Chris Cornell - Song..., (download, 11.49€
Chris Stapleton - Fr..., (CD, 10.98€)
Chris Stapleton - Fr..., (download, 9.49€)
Chris Stapleton - Tr..., (CD, 6.6€)
Daniel Pemberton - K..., (download, 10.49
Deadpool, (DVD, 11.13€)
Deadpool, (BluRay, 10.95€)
Deepwater Horizon, (DVD, 14.96€)
Deepwater Horizon, (BluRay, 24.96€)
Dr. Seuss - Oh, The ..., (Book, 11.39€)
Ed Sheeran - ÷ (Delu..., (download, 12.49€
Ed Sheeran - Divide ..., (CD, 11.99€)
Emily Winfield Marti..., (Book, 10.45€)
Emily Winfield Marti..., (download, 10.45€
Fifty Shades of Grey..., (DVD, 9.16€)
Finding Dory, (DVD, 17.98€)
Finding Dory, (BluRay, 22.97€)

Alison Krauss - Wind..., (CD, 12.42€)
Blair Witch, (BluRay, 19.96€)
Daniel Pemberton - K..., (download, 10.49€)
Deepwater Horizon, (DVD, 14.96€)

[✓ Standard Shipment]
[✓ Express]
[✓ Tracking]
66.83 €
DirectDebit 60000000|1234567890

1.1 First Extension: Additional Articles

Your product attributes can be accessed through the following methods:

- `public String getName()` – returns the name of the product.
- `public int getPrice()` – returns the price in Cent with taxes (as `int` for precise calculation).
- `public int getDelivery()` – returns the delivery time (in days).
- `public boolean isDigital()` – returns true, if this is downloadable product (which will cost no delivery)
- `public boolean isPhysical()` – returns true, if this is a physical product (which will cost delivery and may add tracking and express).

Now you are commissioning articles from another vendor; however these articles somewhat different from your own products The new articles (defined by the class `Article`) provide the following attributes:

- The method `getPriceWithoutTaxes()` supplies the price without value added tax and as Euro (as `double`-value).
- The method `getDescription()` returns the name of the article.

You can add the `Article`-class to your application by adding the `article.jar` to your classpath. The articles can then be used as follows

```
		// Create some articles
		Article article = new Article("Some Hardware Device", 10.00);
		System.out.println(article);
```

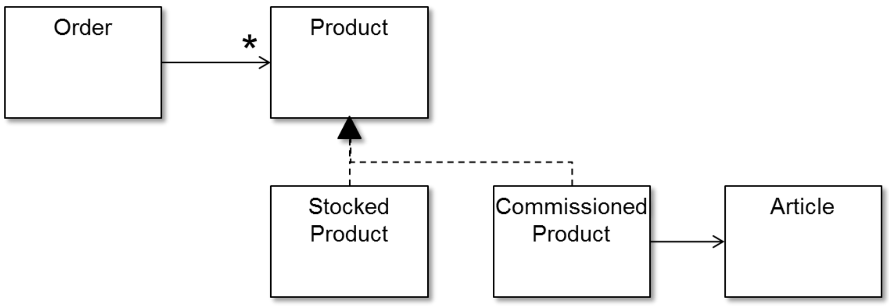
With the following output:

```
Some Hardware Device, 10.0 (without taxes)
```

The main requirement here is that

- your product method definitions should not be changed, otherwise the whole program needs to be changed and
- the definitions of the new article cannot be changed, as otherwise the dataset will no longer fit.

Your developers decide to meet these conditions by using the adapter pattern.



Do this by renaming your original `Product` to `StockedProduct` and create an interface `Product` with the above methods. Then create the article adapter that uses the methods of the article to adapt it to the product. Use the following information about the commissioned articles:

- you have to add 19% VAT for getting “your” price,

- all articles are physical,
- the delivery takes always 10 days,
- and the `toString`-Method should add the tag “`comm`” (for commissioned).

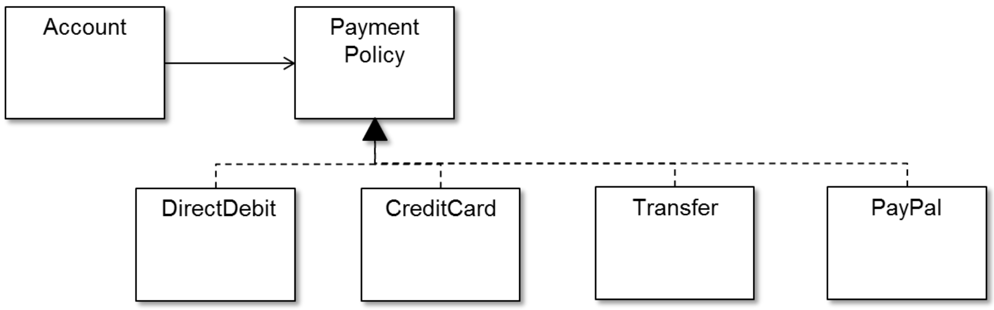
The article-adapter can be used as follows

```
    // Create and add some articles
    Article article = new Article("Some Hardware Device", 10.00);
    System.out.println(article);
    Product product = new CommissionedProduct(article);
    account.add(product);
```

When running the GUI, it should now also show the commissioned articles at the end of the list.

1.2 Second Extension: Additional Payment Types

Customers increasingly want to pay with PayPal, therefore you need to install a new form of payment in your program – you take this as an opportunity to separate pay strategies from your class (as there are perhaps more payment methods in the future):



For doing so, create a strategy interface `PaymentPolicy` with the method definition `pay(String orderId, double amount): void` and create the four implementations. Now create the data, construction and behavior for the four classes – as a blueprint use the data and behavior in the `Account`-class. PayPal-payment should store an e-mail-address as its data. Also provide the toString-methods which are used by the `getPolicyAsString()` of the account. Again use the original method as a blueprint.[2]

Then delete all occurrences of hard-coded payment in the Account; these are: the methods `setDirectDebit`, `setCreditCard` and `setTransfer` with the associated data. Instead, define a field for the strategy in the Account-class. The pay-method and the policyAsString-method of the Account should now simply delegate to the strategy. The strategy itself should be set through a setter-method. Note that the actual payment is only simulated by writing a message to `System.out`.

The strategy could be used as follows

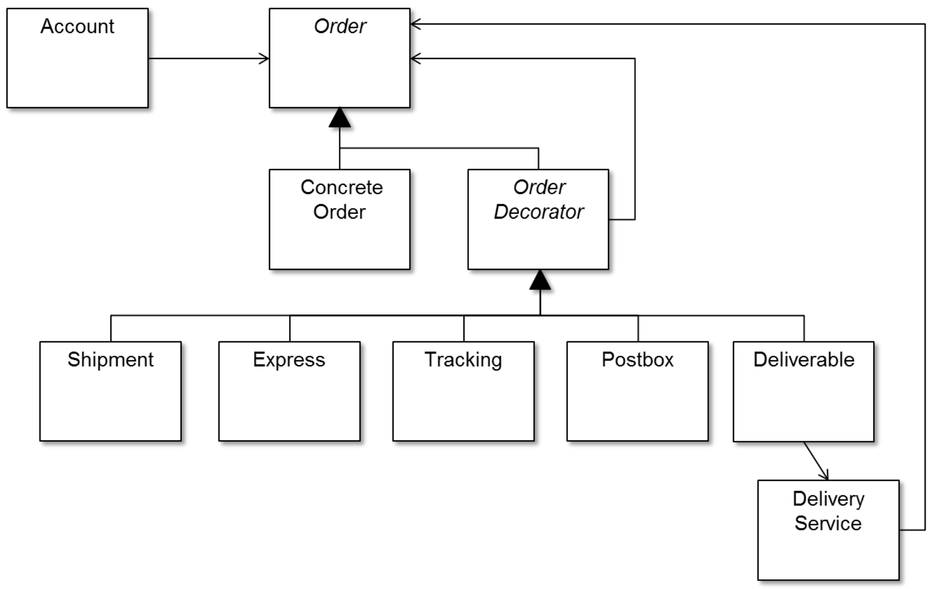
```
    Account account = new Account("John", "Doe", …);
    account.setPolicy(new DirectDebit("500600", "12345678"));
// or
    account.setPolicy(new PayPal("john.doe@gmail.com"));
```

2 The consistent usage of this method is necessary for the correct behavior of the GUI.

1.3 Third Extension: Additional Shipping Methods

Post service offers your customers to pick up goods from a post box. This kind of shipment should be combined with any of the previous types of "Express" and "Tracking". You assume, later there may other options, too. Fulfill this requirement with a Decorator Pattern. You use this opportunity and extract the DeliveryService into its own decorator.[3]

Note:



Rename the former class `Order` to `ConcreteOrder` and introduce the new abstract class `Order`. The `Order` should provide the following abstract methods:

```
    public abstract List<Product> getProducts();
    public abstract int getCosts();
    public abstract int getLatestDeliveryDate();
    public abstract boolean isEmpty();
    public abstract boolean isPhysical();
    public abstract boolean isDigital();
    public abstract void add(Product product);
    public abstract void deliver();
    public abstract List<Product> deliver(int day);
    public abstract void delivered(String message);
```

As the Method `listing` will be needed in several classes, this should also be put into the `Order`-class. Of special interest are the methods `deliver()`, `deliver(int day)` and `delivered(String message)`.

- `deliver()` starts the overall delivery.
- `deliver(int day)` delivers the products with the given delivery day (and below).
- `delivered(String message)` is executed once the delivery is finished. The parameter holds a final message.

[3] The DeliveryService is a simple java thread that will start on the method start(). The thread prints the delivery state day by day until all items are delivered. To do the exercises, it is not necessary to understand this mechanism in detail.

The following example shows how decorators should be used: The method `addExpress` (in `Account`) that originally set the `express`-flag now "decorates" the order with an `Express`-Decorator instead. The original flag and its setter can be deleted.

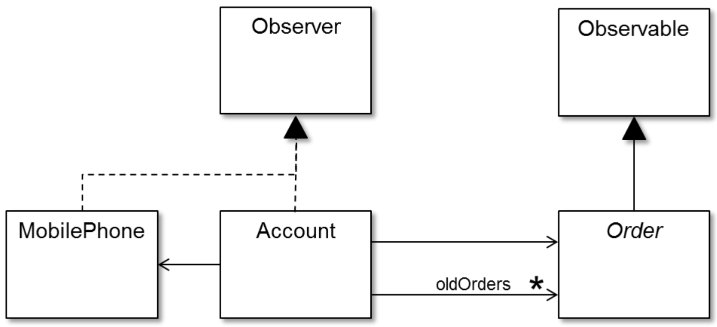
```
    public void addExpress(){
        if(order.isPhysical())
            order = new Express(order);
    }
```

Each decorator can filter / change the method outcomes to its respective terms. Otherwise the behavior is delegated to the concrete class. The decorators should provide the following behavior:

- `Shipment`: adds a basic shipment fee (3€) to the order. A physical order should always be wrapped by a "Shipment" (but only once).
- `PostBox`: free of charge, adds just additional information ("to Postbox") to the final delivery message.
- `Tracking`: adds tracking (2€) to the order and prints the current status of the delivery.
- Express: speeds up the order (4€): This crosscuts `latestDeliverDate()` and `deliver(day)`.
  - The latest delivery date will be reduced by `EXPRESS_DAYS` (so far 2). But it cannot be reduced below one day.
  - The deliver function propagates the speed up in the following way: for 0 days the delivery stays 0; for 1 day all deliveries up to `EXPRESS_DAYS` (so far 2) + 1 are executed; otherwise all orders will be sped up by `EXPRESS_DAYS`.
- `Deliverable`: Starts the actual delivery service when `deliver()` is called. This should be wrapped by the `Account`-class and when the order is completed.

1.4 Fourth Extension: Tracking with your mobile phone

The boss has a great idea: the tracking information to a customer should be delivered to a mobile device. So far, the information is only shown on the computer (via `System.out`). You take the opportunity to decouple the representation from the generation of the tracking information - of course with the Observer pattern:



All orders now generate no more output on their own, but report through the notify-mechanism (where a String object is passed) that a change has taken place and all attached observers can present the message (that is the String object) as they want.

Remark: For this task you may use the original Observer/Observable, which are now deprecated, therefore you may suppress the respective warning. Alternatively, you may use a slightly revamped version which is to be found in the initial class collection. In this case you *have to type Observer and Observable with the class of the object, which is sent as notification – in this example, it would be `String`, as we are sending String-messages.*

The `Account` class must now implement the method `update (Observable, Object)` - the second argument is the string that is sent from the Observables. This method now prints the text (via `System.out`) on the screen.

The Account has to be added as an observer to all Order-subclasses who print an output. For this you have to extend the constructors of the respective classes by **`Observer ... observers`**. These observers should be added during the construction with the method `addObserver`. When creating or decorating the order with an observer, you can provide now the Account itself (and other potential observers).

This allows solving the issue that the output should stop, when the user logs out (which still runs at the moment):

- Add a new method `logout(Observer observer)` to the Order-class. This method deletes the observer from the current order – the logout is also delegated by the `OrderDecorator`.
- As the Account creates a new (empty) order after the completion of the order the old order (which holds the observers) has to be saved in a list `oldOrders`. When calling logout, the orders of this list have to be logged out! After the logout, the list can be cleared.

Finally the smart phone: fortunately the class `MobilePhone` is already finished and will be created on the account creation.

Define the `MobilePhone` also as an "Observer" and implement the method `update(Observable,` Object), which sends the text to your mobile. To display any text on the phone the method `text.append("any text")` can be used. To open the mobile you can call the method `showMobile()` of the `Account`-class as in the following (full) example:

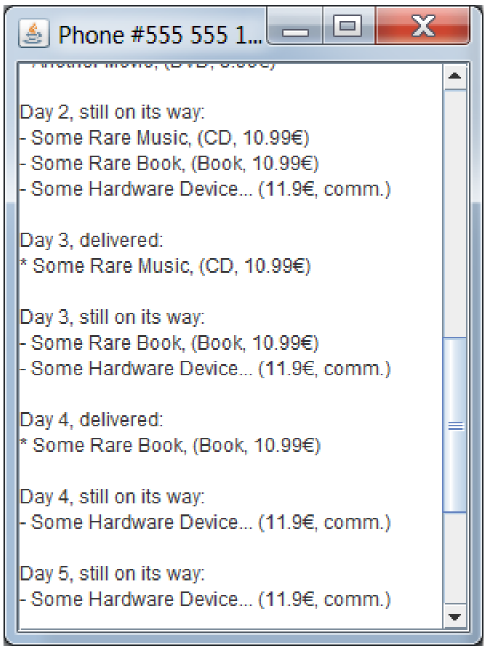
```java
public class Main {

    public static void main(String[] args) throws InterruptedException{

        // Create the account
        Account account = new Account("John", "Doe", "555 555 123",
                                      "john.doe", "secret");
        account.login("john.doe", "secret");

        account.setPolicy(new DirectDebit("500600", "12345678"));

        // Create and add some products
        account.add(new StockedProduct("download", "Java eBook", 9.99, 0));
        account.add(new StockedProduct("CD", "Some Music", 4.99, 1));
        account.add(new StockedProduct("BluRay", "A Movie", 8.99, 2));
        account.add(new StockedProduct("BluRay", "A Second Movie", 8.99, 2));
        account.add(new StockedProduct("Book", "Java Book", 6.99, 3));
        account.add(new StockedProduct("DVD", "Another Movie", 8.99, 4));
        account.add(new StockedProduct("CD", "Some Rare Music", 10.99, 5));
        account.add(new StockedProduct("Book", "Some Rare Book", 10.99, 6));

        // Create and add some articles
        Article article = new Article("Some Hardware Device", 10.00);
        Product product = new CommissionedProduct(article);
        account.add(product);
        // Add options
        account.addExpress();
        account.addTracking();
        account.addPostBox();

        // Complete the order
        account.showMobile();
        account.complete();

        // logout after some time
        Thread.sleep(10000);
        account.logout();
        System.out.println("logged out");
    }
}
```

With the following results

```
Order of Thu Jun 08 19:04:10 CEST
2017: 91.82€ will be transfered from
your account.

Sent out:
- Java eBook, (download, 9.99€)
- Some Music, (CD, 4.99€)
- A Movie, (BluRay, 8.99€)
- A Second Movie, (BluRay, 8.99€)
- Java Book, (Book, 6.99€)
- Another Movie, (DVD, 8.99€)
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)
- Some Hardware Device... (11.9€,
comm.)

Tracking started

Day 0, delivered:
* Java eBook, (download, 9.99€)

Day 0, still on its way:
- Some Music, (CD, 4.99€)
- A Movie, (BluRay, 8.99€)
- A Second Movie, (BluRay, 8.99€)
- Java Book, (Book, 6.99€)
- Another Movie, (DVD, 8.99€)
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)
- Some Hardware Device... (11.9€,
comm.)
```

```
Day 1, delivered:
* Some Music, (CD, 4.99€)
* A Movie, (BluRay, 8.99€)
* A Second Movie, (BluRay, 8.99€)
* Java Book, (Book, 6.99€)

Day 1, still on its way:
- Another Movie, (DVD, 8.99€)
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)
- Some Hardware Device... (11.9€,
comm.)

Day 2, delivered:
* Another Movie, (DVD, 8.99€)

Day 2, still on its way:
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)
- Some Hardware Device... (11.9€,
comm.)

Day 3, delivered:
* Some Rare Music, (CD, 10.99€)

Day 3, still on its way:
- Some Rare Book, (Book, 10.99€)
- Some Hardware Device... (11.9€,
comm.)

logged out
```

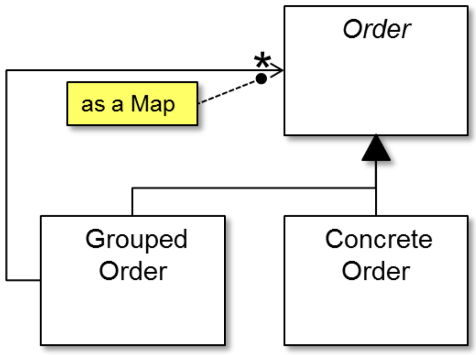
The output should now also be visible on the phone (without being logged out):



```
Day 2, still on its way:
- Some Rare Music, (CD, 10.99€)
- Some Rare Book, (Book, 10.99€)
- Some Hardware Device... (11.9€, comm.)

Day 3, delivered:
* Some Rare Music, (CD, 10.99€)

Day 3, still on its way:
- Some Rare Book, (Book, 10.99€)
- Some Hardware Device... (11.9€, comm.)

Day 4, delivered:
* Some Rare Book, (Book, 10.99€)

Day 4, still on its way:
- Some Hardware Device... (11.9€, comm.)

Day 5, still on its way:
- Some Hardware Device... (11.9€, comm.)
```

1.5 Fifth Extension: Grouping the Orders

The method `getProducts(int delivery)` in the `ConcreteOrder` returns a list of all products which should be delivered at the given delivery time. Instead of collecting these lists on demand (that is when the tracker asks for it) the orders should be grouped according to the delivery time from the beginning. To achieve this use the Composite-Pattern and add a `GroupedOrder` to your application:



The `GroupedOrder` should collect orders as a `Map<Integer, Order>` with the Integer-key being the common delivery time of all contained products of the respective order. If a new product is added to the order the `GroupedOrder` has to determine if an order with the products delivery time exists, if yes it is added to this order, if no, a new `ConcreteOrder` is created and put at this delivery-key. To initialize this behavior, the account class now creates an empty `GroupedOrder` (instead of an empty `ConcreteOrder`). To delegate the other behavior the `GroupedOrder` has to iterate over all contained orders.

1.6 Overview / Documentation [Pre-Exam-Task]

This is the pre-exam-task for this lecture. You can form groups (up to four students) – if you form a group please clearly state all group members on the document. Create a pdf with the following content and submit in Moodle until 18th June 20xx:

Document your final result:

- Create a UML diagram that represents the entire solution – mark each of the parts which constitute a pattern

Answer the following questions:

- Which pattern had a local effect (i.e. without any ripple effects) on other classes when introducing?
- Which pattern had a large effect (e.g. when renaming classes during refactoring)?
- When adding new decorators or new policies what would be the effect on your application (compared to the original version of the application)?
- Which principles (as stated in the lecture) have been fulfilled through what refactoring/pattern?