Exercises for lecture “Software Architecture“

05.1 – Design Patterns

1 Dependency Injection / Dependency Inversion

Given is the following small application for (simulated) sending of emails (also in moodle as 05.1.messaging.application.v1.initial):

```java
package messaging.app;

import messaging.infrastructure.EmailMessageSender;

public class MessagingService {

	private EmailMessageSender messageSender = new EmailMessageSender();

	public void sendMessage(String message) {
		messageSender.send(message);
	}
}
```

```java
package messaging.infrastructure;

public class EmailMessageSender {

	public void send(String message) {
		System.out.println("Sending email message: " + message);
	}
}
```

```java
package messaging.main;

import messaging.app.MessagingService;

public class Main {

	public static void main(String[] args) {
		MessagingService messagingService = new MessagingService();
		messagingService.sendMessage("Hello, World!");
	}
}
```

1.1 Step 1: Dependency Injection

Review the Initial Application: Examine the initial application structure provided in the `messaging.app`, `messaging.infrastructure`, and `messaging.main` packages. Draw an UML-diagram (classes and packages) of the application. Currently the `MessagingService` directly creates an `EmailMessageSender`.

Modify the `MessagingService` class to accept an instance of `EmailMessageSender` on construction. Update the Main class to create an instance of `EmailMessageSender` and pass it to the `MessagingService` constructor.

Run the refactored application to ensure that messages can still be sent successfully. Draw an updated UML-diagram of the application.

1.2 Step 2: Dependency Inversion

Review the modified application: The (high-level) service depends on the (low-level) infrastructure. Introduce dependency inversion by extracting an interface `MessageSender`. Use this interface instead of the original `EmailMessageSender` wherever possible. Be careful where to place the new interface to ensure the dependency inversion.

Run the refactored application to ensure that messages can still be sent successfully. Draw again an updated UML-diagram of the application.

2 Dependency Injection with Reflection

In moodle you find the code for the Java project 0.5.2.reflexive.injection. Create the project and add the code. Execute the application: it should instantiate the client with the current operation system representative and print it out. Create a UML-class/package diagram of the application.

3 Refactoring an Application with the Help of Design Patterns

This exercises intends to refactor a given application.

3.0 Preparation / Initial Version

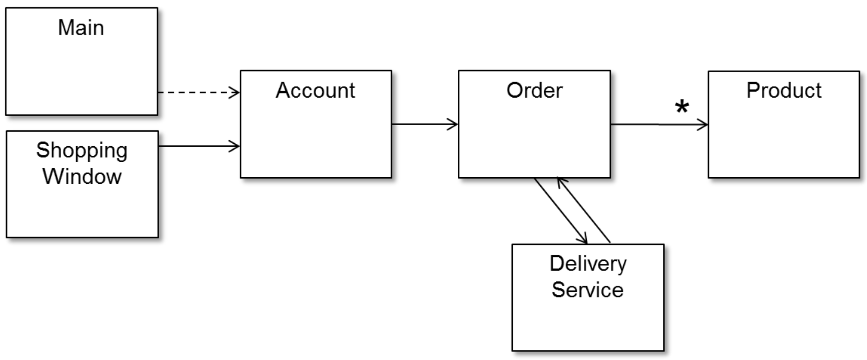
Download and unpack the zip-file. Create an general project `05.3.0.libs` and add the two jars

- `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.

Create a java project `05.3.0.shop.initial` and add the java classes. Add the two jars to your project.

The initial shop-project 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`) or with a graphical-user-interface by adding the shop-io.jar to the classpath.

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.



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).

```
public class Main {

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

        // Create the account
```

[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.

```
        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).

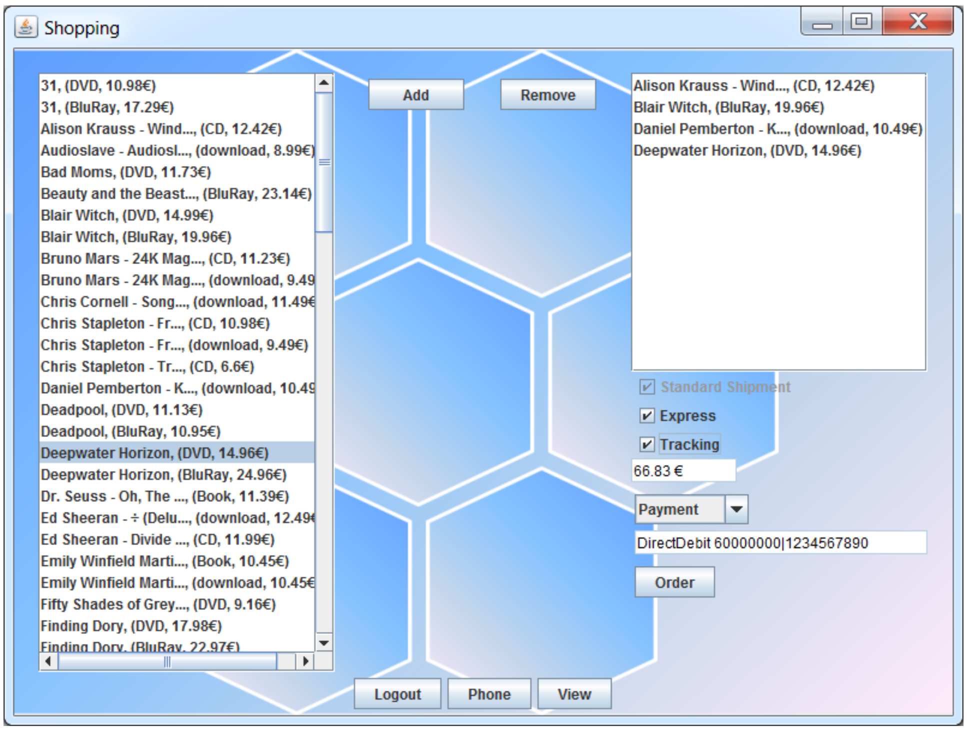
```
Order of Thu Jun 08 11:27:50 CEST
2017: 79.92€ will be transfered from
your account.

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. Upon start the gui creates a set of users (amber, ben, cecil, dan, evan, finn, Gordon, and john, all with the with password “secret”) but you can also create your own by pressing “Register”. Accounts are stored in the file `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 €

Payment

DirectDebit 60000000|1234567890

3.1 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);
```

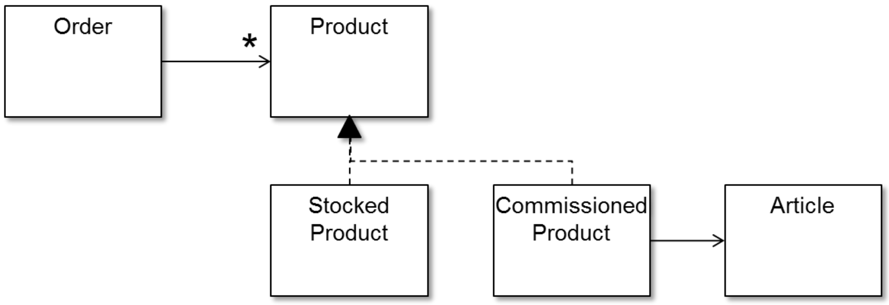
With the following output:

```
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).

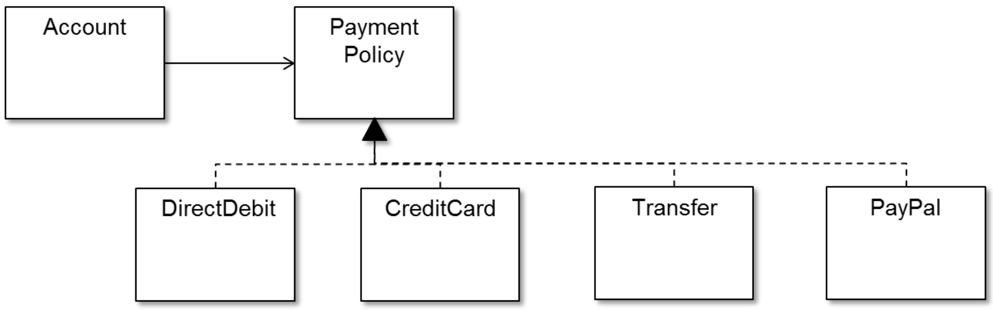
The article-adapter can be used as follows

```
      // 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.

3.2 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.

4 Bonus Task: Observer

4.1 Basic Example

Implement the Observer-Example as given in the lecture:[3] This is the Model which is observed by the view – but it has no direct connection to the view.

```
import java.beans.*;

public class ObservedModel {

	private PropertyChangeSupport support;

	public ObservedModel() {
		support = new PropertyChangeSupport(this);
	}

	public void addObserver(PropertyChangeListener pcl) {
		support.addPropertyChangeListener(pcl);
	}

	public void removeObserver(PropertyChangeListener pcl) {
		support.removePropertyChangeListener(pcl);
	}

	private String data;

	public void setData(String value) {
		support.firePropertyChange("data", this.data, value);
		this.data = value;
	}
}
```

This is the “View” which observes the Model:

```
import java.beans.*;

public class ObservingComponent implements PropertyChangeListener {

	@Override
	public void propertyChange(PropertyChangeEvent evt) {
		System.out.println("received: " );
		System.out.println("\tproperty: "+ evt.getPropertyName());
		System.out.println("\told:      "+ evt.getOldValue());
		System.out.println("\tnew:      "+ evt.getNewValue());
	}
}
```

The main-method plugs everything together – please note, that the `ObservedModel` is not depending on the view-class.

```
public class Main {

	public static void main(String[] args) {
```

[3] All code of this exercise is provided on the Cheat-Sheet in moodle.

```
        ObservedModel observable = new ObservedModel();
        ObservingComponent observer = new ObservingComponent();

        observable.addObserver(observer);
        observable.setData("new value");
    }
}
```

4.2 Extended Example

Extend the example in a way, that the `ObservingComponent` receives an id on construction which is also printed, when reacting on an event. In the main-method create now two observers (called "Observer1" and "Observer 2") and add both to the observable. Run the example again. For a test remove both observers (or don't add them) and run the example again.

4.3 An experimental Observer/Observable

In moodle you find an experimental version of a new Observer/Observable-Implementation. Both are now interfaces (in contrast to the original `Observable`). For the basic version (which should be used for this task) only a generic type has to be supplied, which specifies the send data[4]. The Implementation is of course thread-safe. The usage is like this:

```
import observer.pattern.basic.Observable;

public class ObservedModel implements Observable<String> {
      …
      public void someMethod(String data) {
          do something …
          this.notifyObservers(data);
      }
     …
     @Override
}
```

```
import observer.pattern.basic.Observer;

public class ObservingComponent implements Observer<String> {
     …
     @Override
     public void update(String data) {
          do something with data
     }
     …
}
```

Now do the above task again with this new Observer/Observable.

4 The full version allows to supply additionally an aspect (i.e. enumeration type) which allows to hand over certain aspects or states which can be switched over.