|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| Refactoring (exercises only) |

|  |
| --- |
| By Per Laursen  08-04-2019 |

[Exercises 2](#_Toc5639282)

[ReFac.1 2](#_Toc5639283)

[ReFac.2 3](#_Toc5639284)

# Exercises

|  |  |
| --- | --- |
| **Exercise** | ReFac.1 |
| **Solution** | SimpleShop |
| **Purpose** | Work with a few of the “original” code refactorings. |
| **Description** | The solution simulates a simple web shop, where customers can order various pro­ducts. For each order, an invoice is generated and sent to the customer.  The solution also contains a unit test project, which validates that the actu­al content of the sent mails match the expected content. You should use this unit test actively, to ensure that you do not introduce errors during the refactoring. |
| **Steps** | 1. Get a general overview of the project, and be sure you understand how the classes interact. Some of the code in the classes may have certain “smells”; see if you can spot some of them. 2. A first smell to address is *Duplicated Code*. It looks like the code for valida­ting an email address is present in more than one place. See if you can somehow get rid of the duplicated code. A relevant question relating to this refactoring is then: *where should the code for email validation be placed?* (Hint: maybe we should define a new class **Email** for modeling e-mail addresses?) 3. Once you have isolated the code for email address validation, take a clo­ser look at the code itself. Perhaps it also has the *Duplicated Code* smell. Try to Google the refactoring named *Consolidate Condi­tional Expression*, and see if you can rewrite the logic into a more compact form. 4. It seems that e.g. the **Order** class needs a lot of customer data… but there is no **Customer** class? Maybe we have a *Primitive Obsession* smell here. Define a **Customer** class with the relevant properties, and use it where relevant. It might also be useful to define a **CustomerCatalog** class. 5. If you defined a **CustomerCatalog** class in step 4, you will now have two catalog classes. Do they have something in common? Maybe it could be relevant to introduce a new base class for catalog classes? 6. Inside the **Invoice** class, we have some fairly detailed logic for calculating the total price of an order. Does that code really belong there? Or do we have a *Feature Envy* smell here? See if you can move this logic to the **Order** class. NB: The **Invoice** class may still need a **TotalPrice** property. 7. See if you can find additional opportunities for code refactoring, and keep using the unit test to ensure that the functionality is preserved. It is okay to refactor something, even if it doesn’t match an “official” code smell ☺. |

|  |  |
| --- | --- |
| **Exercise** | ReFac.2 |
| **Solution** | StockTradeStrat |
| **Purpose** | Try to work with the refactoring principle ***Replace Conditional Logic With Strategy*** |
| **Description** | The project contains code for implementing various **trading strategies** for stock trading. More specifically, four strategies are implemented; for buying and selling, respectively, an “aggressive” and “defensive” strategy is defined, and a total of four strategies are thus defined by combining two such partial strategies.  In the initial version of the code, the four trading strategies are implemented by means of **conditional logic**, in a method **Trade** representing the trading logic. |
| **Steps** | 1. Open the solution, and get an overview. It contains the stock trading pro­ject itself, plus a unit test project. 2. Try to run the project, and also try to run the unit test. Hopefully, you should see four unit test methods which all pass the test. 3. Take a look at the interfaces in the **Interfaces** folder. One you feel you understand the interfaces, proceed to the two implementations of the interfaces found in the **Original** folder. 4. Focus on the **Trade** method in **StockTraderOrg**. How long is it? How easy is it to understand? How does it comply with the **Open/Close** principle, i.e. what will happen if we want to define additional strategies? 5. Before proceeding, also take a look at the classes in the **Shared** folder. You do not need to understand the **StockQuote…** classes in detail, since they are just there to implement simplistic stock quote generation and analysis. In the base class **StockTraderBase**, take a look at how the **Act** method is imple­mented. 6. As the first step in refactoring the code, create a new interface in the **Inter­faces** folder, called **IStockTradeStrategy**. It should look like this:   **public interface IStockTradeStrategy**  **{**  **void Trade(IStockTrader trader, int index, List<double> quotes);**  **}**   1. Now create a new folder e.g. named **WithStrategy**. This folder should con­tain the revised implementation of the code. Add a new class **StockTra­der­FactoryStrategy** to the folder. This class should implement the inter­face **IStockTraderFactory**. 2. To start with, implement the class simply by copying the entire content from the existing class **StockTra­der­FactoryOrg** into the class. Then go to the unit test project. 3. In the unit test project, change the initialisation of the **StFac** property to use **StockTra­der­FactoryStrategy** instead of **StockTra­der­FactoryOrg**. Compile and run the unit test; it should still be in the green, since the new factory class just replicates the functionality from the old factory class. 4. Now we can begin to refactor the actual implementation. Create a new class **StockTraderWithStrategy** , which – just as the original stock trader class – will inherit from **StockTraderBase**, and implement **IStockTrader**. 5. Implement **StockTraderWithStrategy** in such a way that you can imple­ment the **Trade** method by calling **Trade** on a property of type **IStock­Trade­Strategy**. This property should be initialised in the constructor, by receiving a reference of type **IStock­Trade­Strategy** in the constructor parameters (that is, by using **Dependency Injection**). 6. Now proceed to create actual implementations of **IStock­Trade­Strategy**. You should implement four such strategies, corresponding to the four strategies used in the original implementation. 7. Once you have at least one implementation ready, try to update the code in one of the methods in **StockTra­der­FactoryStrategy**, to use the new classes (i.e. **StockTraderWithStrategy** and a **IStock­Trade­Strategy** imple­men­tation). **NB:** This is where the unit test becomes useful! If the unit test now fails, your implementation of the trading strategy must contain at least one error. See if you can find and fix your errors (if you have any), and do not proceed until the unit test is back in green again! 8. Now that you are back in green again, repeat the process for the remain­ing three strategy implementations. 9. Once you are done, take a closer look at your strategy implementations. Could they be improved somehow? Do they contain repetitive code? Feel free to try to improve the implementation further – e.g. by introducing a base class and/or a library class for common strategy methods – but keep using the unit test frequently to make sure you have not introduced any errors, or to quickly roll back if errors do occur. 10. Finally, reflect over your refactored solution. Will the new **StockTrader­WithStrategy** class need to change if new strategies are implemented? Are strategies now limited to being variations over aggressive/defensive partial strategies? Who establishes the connection between a stock trader object and specific trading strategies? |