|  |
| --- |
| EASJ Notes |
| C# Programming Exercises |
| (used in conjunction with Object-Oriented Programming With C#) |

|  |
| --- |
| By Per Laursen  28-02-2018 |

Content

[**How to use this exercise set** 4](#_Toc507616839)

[Get.1 5](#_Toc507616840)

[Pro.1.1 6](#_Toc507616841)

[Pro.1.2 7](#_Toc507616842)

[Pro.1.3 8](#_Toc507616843)

[Pro.1.4 9](#_Toc507616844)

[OOP.1.1 10](#_Toc507616845)

[OOP.1.2 11](#_Toc507616846)

[OOP.1.3 12](#_Toc507616847)

[OOP.1.4 13](#_Toc507616848)

[OOP.1.5 14](#_Toc507616849)

[OOP.1.6 15](#_Toc507616850)

[OOP.1.6a 16](#_Toc507616851)

[OOP.1.6b 17](#_Toc507616852)

[OOP.1.7 18](#_Toc507616853)

[Pro.2.1 19](#_Toc507616854)

[Pro.2.2 20](#_Toc507616855)

[Pro.2.3 21](#_Toc507616856)

[Pro.2.4 22](#_Toc507616857)

[Pro.2.5 23](#_Toc507616858)

[Pro.2.6 24](#_Toc507616859)

[Pro.2.7 25](#_Toc507616860)

[Pro.2.8 26](#_Toc507616861)

[Pro.2.9 27](#_Toc507616862)

[Pro.2.10 28](#_Toc507616863)

[Pro.2.11 29](#_Toc507616864)

[Pro.2.12 30](#_Toc507616865)

[Pro.2.13 31](#_Toc507616866)

[Pro.2.14 32](#_Toc507616867)

[Pro.2.15 33](#_Toc507616868)

[OOP.2.1 34](#_Toc507616869)

[OOP.2.2 35](#_Toc507616870)

[OOP.2.3 36](#_Toc507616871)

[OOP.2.4 37](#_Toc507616872)

[OOP.2.5 38](#_Toc507616873)

[OOP.2.6 39](#_Toc507616874)

[OOP.2.7 40](#_Toc507616875)

[GUI.1.0 41](#_Toc507616876)

[GUI.1.1 42](#_Toc507616877)

[GUI.1.2 43](#_Toc507616878)

[GUI.1.3 44](#_Toc507616879)

[GUI.1.4 45](#_Toc507616880)

[GUI.1.5 46](#_Toc507616881)

[GUI.1.6 47](#_Toc507616882)

[GUI.1.6a 48](#_Toc507616883)

[GUI.1.7 49](#_Toc507616884)

[GUI.1.8 50](#_Toc507616885)

[GUI.2.0 51](#_Toc507616886)

[GUI.2.1 52](#_Toc507616887)

[GUI.2.2 53](#_Toc507616888)

[GUI.2.3 54](#_Toc507616889)

[PARA.0 55](#_Toc507616890)

[PARA.1 56](#_Toc507616891)

[PARA.2 57](#_Toc507616892)

[PARA.3 58](#_Toc507616893)

[PARA.4 59](#_Toc507616894)

[PARA.5 60](#_Toc507616895)

[PARA.6 61](#_Toc507616896)

[PRO.3.1 62](#_Toc507616897)

[PRO.3.2 63](#_Toc507616898)

[PRO.3.3 64](#_Toc507616899)

[PRO.3.4 65](#_Toc507616900)

[PRO.3.4a 66](#_Toc507616901)

[PRO.3.4b 67](#_Toc507616902)

[PRO.3.5 68](#_Toc507616903)

[PRO.3.6 69](#_Toc507616904)

[PRO.3.7 70](#_Toc507616905)

[PRO.3.8 71](#_Toc507616906)

[PRO.3.9 72](#_Toc507616907)

[PRO.3.10 73](#_Toc507616908)

[PRO.3.11 74](#_Toc507616909)

[PRO.3.12 75](#_Toc507616910)

[DAPE.1 77](#_Toc507616911)

[DAPE.2 78](#_Toc507616912)

[DAPE.3 79](#_Toc507616913)

[DAPE.4 80](#_Toc507616914)

[DAPE.5 81](#_Toc507616915)

[DAPE.6 82](#_Toc507616916)

[DAPE.7 83](#_Toc507616917)

[DAPE.8 84](#_Toc507616918)

[DAPE.9 85](#_Toc507616919)

[MVVMStarter.1 86](#_Toc507616920)

[MVVMStarter.2 87](#_Toc507616921)

[MVVMStarter.3 88](#_Toc507616922)

[MVVMStarter.4 89](#_Toc507616923)

[MVVMStarter.5 90](#_Toc507616924)

# **How to use this exercise set**

This set of exercises is intended to be used in conjunction with the note ***Object-Oriented Programming with C#***. However, they are as such self-contained.

The formulation of each exercise follows a standard pattern:

* **Exercise**: Identifier for the exercise. The first part of the identifier is an acronymed reference to the corresponding chapter in the notes.
* **Project**: A C# project used in the exercise. The specific details of how the project is made available (.zip file, GitHub repository, etc.) may vary from course to course. The projects are self-contained.
* **Purpose**: What aspect of the learning process does this exercise concern.
* **Description**: The “setup” for the exercise, typically some sort of simplified domain-specific context.
* **Steps**: Specific steps in the exercise. The steps often become increasing­ly difficult. Some steps are marked in red. These steps are considered quite difficult.

To get around some technicalities with C# projects that are irrelevant for the begin­ner, some projects contain an extra C# file called **InsertCodeHere.cs**. In that file, an area is delimited by two comments

// The FIRST line of code should be BELOW this line

(sandbox area)

// The LAST line of code should be ABOVE this line

This area is referred to as the “sandbox area” in several exercises. If you are required to put some code in the “sandbox area”, this is the place.

|  |  |
| --- | --- |
| **Exercise** | Get.1 |
| **Project** | Sandbox |
| **Purpose** | Reality check – Visual Studio up and running |
| **Description** | The **Sandbox** project is as simple as it gets – we will just use it to verify that your installation of Visual Studio is up and running |
| **Steps** | 1. Load, compile and run the project. 2. Verify that the message ***Hello world!*** Is printed on the screen. |

|  |  |
| --- | --- |
| **Exercise** | Pro.1.1 |
| **Project** | MovieManagerV05 |
| **Purpose** | Discuss variables with regards to types and naming |
| **Description** | We imagine this project to be the very first steps in creating an appli­cation for movie management. The application could be used to keep track of relevant information for movies, e.g. a private collection of movies on DVD/Blu-ray (yes, some people still watch movies on physical media ☺). |
| **Steps** | 1. Think about what specific information it could be relevant to store for each movie. 2. For each specific piece of information, think about how you can represent this information. Think about the nature of the infor­mation; is it text, numeric, or something else. 3. In the sandbox area in the project (**InsertCodeHere.cs**), define a variable for each piece of information. You should    1. Choose a proper type for the variable    2. Find a descriptive name for the variable 4. Once you are done, pair up with another student. 5. Switch computer with your partner 6. Review the work of your partner. For each variable in the partner’s project, think about if    1. The purpose of the variable is easy to understand    2. The type seems properly chosen 7. Discuss your findings with your partner. 8. Was there any types of information that were particularly hard to find a good representation for? 9. Suppose you need to store information about a lot of movies. How would you need to change your code in order to make that possible? |

|  |  |
| --- | --- |
| **Exercise** | Pro.1.2 |
| **Project** | WebShopV05 |
| **Purpose** | Get some practice in using arithmetic operators |
| **Description** | Part of the business logic in a web shop involves calculating the total cost of an order. The logic for calculating the total cost is as follows:   1. An item has a net price 2. You pay a 10 % tax on top of the net price 3. Shipping costs 49 kr., no matter the number of items 4. There is a credit card fee of 2 % on top of the entire cost, including tax and shipping. |
| **Steps** | 1. Load and open the project – you will see that some variables for the net prices and number of items in an order have already been included. Also, the order details are printed on the screen. 2. The variable **totalPrice** is supposed to contain the total price for the order. You must add the calculations needed to do this, given the logic in the description. 3. Test your solution by varying the number of books, DVDs and games in the order (you do this by assigning new values to the **noOf…** variables, and running the application again) |
| **Extra info** | Some test examples you can use to verify your solution.   |  |  |  |  | | --- | --- | --- | --- | | **Books** | **DVDs** | **Games** | **Total price** | | 8 | 3 | 2 | 711,96 kr. | | 0 | 12 | 4 | 1171,98 kr. | | 23 | 16 | 7 | 2507,16 kr. | |

|  |  |
| --- | --- |
| **Exercise** | Pro.1.3 |
| **Project** | WebShopV06 |
| **Purpose** | Get some practice in using logical operators |
| **Description** | Another part of the business logic in a web shop involves deciding if a customer qualifies for certain special offers, based on the order. The shop has four special offers. The logic for qualifying for each offer is:   1. The net total price (no taxes, etc.) is more than 1.000 kr. 2. You have ordered more books than games 3. You have ordered at least 10 items of one kind 4. You have ordered between 10 and 20 (incl.) DVDs, or at least 5 games |
| **Steps** | 1. Load and open the project – again, some variables are already present. Note the boolean variables **receiveSpecialOffer…** 2. For each of these variables, you must specify a logical expres­sion, corre­sponding to the logic given in the description. 3. Test your solution by varying the number of books, DVDs and games in your order. 4. The web shop decides to offer an extra special offer. You qualify for the extra offer, if you qualify for exactly two of the previous offers. Update your code to include this extra offer. |
| **Extra info** | Some test examples you can use to verify your solution (SO#1 means “special offer 1”, and so on):   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **Books** | **DVDs** | **Games** | **SO#1** | **SO#2** | **SO#3** | **SO#4** | | 8 | 3 | 2 | false | true | false | false | | 0 | 12 | 4 | false | false | true | true | | 23 | 16 | 7 | true | true | true | true | | 3 | 5 | 4 | false | false | false | false | |

|  |  |
| --- | --- |
| **Exercise** | Pro.1.4 |
| **Project** | FunctionExample |
| **Purpose** | Define and use a simple function |
| **Description** | You can define a rectangle by two points (x1, y1) and (x2, y2). The area of the rectangle is then:  the absolute value of (x1 – x2)\*(y1 – y2)  The absolute value just means that if the value is negative (e.g. -4), the absolute value is the corresponding positive value (e.g. 4 in this example). If the value is already positive, it just says positive. |
| **Steps** | 1. Review the code in **InsertCodeHere.cs**. It calculates the area of two rect­angles, and prints out the values. Make sure you understand the calcula­tion, including the use of **Math.Abs**. 2. An (incomplete) method **AreaOfRectangle** is also part of **InsertCode­Here.cs**. See if you can implement it correctly (Hint: try to move the calcu­lation logic from the existing code into the method, and rename the variable names to match the parameter names used in **AreaOfRectangle**). 3. Once you have implemented the method, use it to perform the area calculations in the code above the method. Check that you get the same results as before. 4. Why is it a good idea to replace the area calculations with calls to the method **AreaOfRectangle**? |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.1 |
| **Project** | MovieManagerV10 |
| **Purpose** | Observe how to use an existing class.  Implement simple use of an existing class. |
| **Description** | In this version of the movie manager application, a class named **Movie** has been added (in the file **Movie.cs**). It contains an absolute mini­mum of information about a specific movie. The class is put to use in the sandbox area, where some **Movie** objects are created and used. |
| **Steps** | 1. Load the project, and go directly to the sandbox area. You will see that some code is already present. See if you can figure out what goes on in each line of code. If you hover the mouse cursor over a specific element, you should see some useful information pop up. Make sure you under­stand where    * Objects are created    * Arguments to the constructors are specified    * Properties are used    * Methods are called 2. Feel free to create additional **Movie** objects, and exercise them a bit (call methods, use properties, etc.) 3. We can only change the value of **NoOfViews** by calling the **Watch** method. Consider why it would be a bad idea to allow a user of the class **Movie** to change the value of **NoOfViews** directly, e.g. like this: **movieA.NoOfViews = movieA.NoOfViews + 1;** |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.2 |
| **Project** | BankV05 |
| **Purpose** | Implement minor additions to an existing class. |
| **Description** | The project contains a minimal **BankAccount** class. The class is put to use in the sandbox area, where a **BankAccount** object is created and used. |
| **Steps** | 1. Load the project, and take a look at the **BankAccount** class. Make sure you understand all the elements in the class definition. Then take a look at how the **Bank­Account** class is used in the sandbox area. 2. We now want to add an extra property to the **BankAccount** class: the name of the account holder. Add this feature to the class. This will probably involve:    1. Adding a new instance field    2. Adding a property which uses the new instance field    3. Updating the constructor, such that the name of the account holder must be specified when a **BankAccount** object is created. 3. Once the class has been updated, make sure to test the new feature by updating the code in the sandbox area. More specifically, you should test that you must now specify a name when creating a **BankAccount** object, and that you can retrieve the name by using the property added in step 2. |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.3 |
| **Project** | RolePlayV10 |
| **Purpose** | Implement non-trivial additions to an existing class |
| **Description** | The project contains a **Warrior** class, which is initially very simple – it only contains a **Name** property. We now need to extend the class with an additional feature. |
| **Steps** | 1. Start out by taking a look at the **Warrior** class. Make sure you understand all the elements in the class definition. Specifically, make sure you can identify:    1. An instance field    2. A property    3. The constructor 2. Next, take a look at how the **Warrior** class is used in the sandbox area (**InsertCode­Here.cs**). This is a very small test of the class. 3. We must now extend the **Warrior** class with a “level” feature. The require­ments for this feature are:    1. All warriors start at level 1.    2. The level can be retrieved freely, but not changed freely.    3. It must be possible to increment the level, i.e. increase the value of the level by 1. 4. Implement this feature in the **Warrior** class. You will need to consider if    1. An additional instance field is needed (Hint: we need to store the current level of a **Warrior** somewhere)    2. An additional property is needed (if so, do we need both the **get** and the **set** part? Hint: We only require that the value of the level can be retrieved through the property, not changed)    3. The constructor should be updated (Hint: The construc­tor should initialise all instance fields with a well-defined value. What would that value be for level, given requirement 3a?).    4. A method for incrementing the level is needed (Hint: the level should always just be increased by 1). 5. Test the updated **Warrior** class, by adding some code to the sandbox area. |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.4 |
| **Project** | RolePlayV11 |
| **Purpose** | Implement non-trivial additions to an existing class |
| **Description** | The project contains a **Warrior** class, including the functionality described in the previous exercise (a “level” feature). |
| **Steps** | 1. We must now extend the class with a “hit points” feature. Details of this feature are:    1. Hit points are set individually when a warrior is created.    2. Hit points can be retrieved freely, but not changed freely.    3. It must be possible to decrease hit points by a specified amount. This corresponds to the warrior being damaged by someone 2. Implement this feature in the **Warrior** class. You will need to consider if    1. An additional instance field is needed (Hint: we need to store the current hit points of a **Warrior** somewhere)    2. An additional property is needed (if so, do we need both the **get** and the **set** part? Hint: We only require that the value of the hit points can be retrieved, not changed)    3. The constructor should be updated (Hint: The construc­tor should initialise all instance fields with a well-defined value. What would that value be for hit points, given requirement 1a? How can we provide this value to the constructor? Extra hint: How is the value of the warrior’s name provided to the constructor?).    4. A method for decreasing the hit points is needed (Hint: how can the specific amount to decrease the hit points with be provided to the method?). 3. Implement a property called **Dead**, which returns a boolean value. The property should return **true** if hit points are equal to or below zero (Hint: we only need to include the **get** part of this new property). 4. Test the updated **Warrior** class, by adding some code to the sandbox area. 5. A **Warrior** should not only receive damage, but also deal damage! Implement a method **DealDamage**, with these requirements:    1. Returns an integer value    2. Does not take any parameters    3. The returned integer value should be a random number between 10 and 30 (Hint: Figure out how the **Random** class from the .NET class library works). 6. See if you can make two warriors battle against each other, to the death! |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.5 |
| **Project** | ClockV10 |
| **Purpose** | Implement a class from scratch, including use of the class |
| **Description** | This project contains an empty class definition **Clock**. Your job is to implement the class **Clock**, given the below requirements:   1. The clock should be able to display (i.e. print on the screen) hours and minutes. 2. The clock should use the 24-hour system. 3. It must be possible to set the clock to a specific time. 4. It must be possible to retrieve the current time from the clock. 5. It must be possible to advance the clock by a single minute.   **NB!** To avoid confusion: the class should only be able to “simulate” a clock, i.e. it should not in any way contain code that retrieves the real-world clock from the computer. A clock will only “tick” when you call a method which changes the time, as stated in requirement 5). |
| **Steps** | 1. Implement requirements 1-4. This will involve figuring out what instance fields, constructor, properties and methods you need for this. Remember to include code for testing the class. 2. Implement requirement 5. In this case, it becomes quite important to choose relevant test cases! Note that you may need to rethink how you represent time in the **Clock** class (Hint: maybe a single instance field is enough…?). |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.6 |
| **Project** | DiceGame |
| **Purpose** | Work with a project containing collaborating classes |
| **Description** | This project contains three classes: **RandomNumberGenerator**, **Die** and **DiceCup**.   * The **RandomNumberGenerator** class contains code which can generate random numbers within a given interval. The class is already completed, and we will not be concerned with the details of this class. * The **Die** class represents a 6-sided die, and is already completed. * The **DiceCup** class needs a bit of work to be complete. The **Dice­Cup** class uses the **Die** class. |
| **Steps** | 1. Take a look at the **Die** class. It is complete, and fairly simple. Note how the **Die** class uses the **RandomNumberGenerator** class. 2. Open the **DiceCup** class. Note how the class contains two instance fields of type **Die**. Also note the constructor – what happens there? 3. The **DiceCup** class is not complete. Implement the **Shake** method and the **TotalValue** property, as specified in the comments in the code. Test that your code works as expected, by creating and using a **DiceCup** object in the sandbox area. 4. How much would we need to change in **DiceCup** in order to have a dice cup with three dice? 5. When we create a **DiceCup** object, we would also like to be able to specify the number of sides the dice should have. Implement the changes in **Die** and **DiceCup** needed to enable this feature. |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.6a |
| **Project** | RolePlayV12 |
| **Purpose** | Work with a project containing collaborating classes.  Reflect over class responsibilities. |
| **Description** | The project contains two classes: **Warrior** and **Sword**. The **Sword** class is initially not used for anything. |
| **Steps** | 1. Take a look at the **Warrior** class. It is very similar to what we have seen in the previous **RolePlayV…** projects. Here, we focus in particular on the me­thod **DealDamage**. What elements in the **Warrior** class definition (instance fields, properties, etc.) are needed in order to implement **DealDamage**? 2. The **DealDamage** method contains the logic for calculating the damage a specific **Warrior** object deals. We would like to move this logic to another class; the **Sword** class. Take a look at the **Sword** class, and make sure you understand how it works. 3. The next step is to connect the two classes. Do this by adding an instance field of type **Sword** to the **Warrior** class. The instance field should be initia­lised by an additional parameter (of type **Sword**) to the constructor (why is this a better solution than just creating a new **Sword** object directly in the **Warrior** constructor?). 4. Now that a **Warrior** object references a **Sword** object, the **DealDamage** method should be modified. Modify the method, simply by letting it call the **DealDamage** method on the **Sword** object. 5. When the code for **DealDamage** has been modified, we can remove several instance fields from the **Warrior** class. Figure out which instance fields we don’t need anymore, and remove them. 6. The code in the sandbox area contains a small test. This code should also be modified, such that each **Warrior** object is provided a **Sword** object at crea­tion. Run the test, to confirm that the **Warrior** objects are still capable of dealing damage. 7. Which class now contains the logic for specific damage calculation? Why is this a better division of responsibilities than before? 8. If you have time left, you could update the **Warrior** class further: 9. Make it possible to use two swords. 10. Make it possible to change a sword after creation. 11. Add a **Warrior**-specific damage factor, such that the damage generated by the sword is multiplied with this factor. This could e.g. represent that the warrior is stronger/weaker than average. |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.6b |
| **Project** | StockPortfolio |
| **Purpose** | Work with a project containing collaborating classes.  Reflect over class responsibilities. |
| **Description** | The project contains three classes: **Stock**, **Portfolio** and **StockMarket**. Together, they form a very simple simulation of a stock market, and how it influences a portfolio of stocks. Further details about each class is provided as comments in the code. |
| **Steps** | 1. Take a look at the **Stock** class. The class is completed, and you need not add any code to it. Make sure you understand the meaning of the data stored in the **Stock** class. 2. Take a first look at the **Portfolio** class. The class is not complete yet, since the property **TotalEarningsPercentage** and the method **UpdateCurrent­Prices** are not implemented yet. Still, it should be possible (e.g. by reading the comments) to understand what data the class contains, and what is intended with the methods and properties. Make sure you also understand how the **Portfolio** class and the **Stock** class are related. 3. Take a look at the **StockMarket** class. The class is completed, and you need not add any code to it. Make sure you understand how the class is intended to be used. Make sure you also understand how the **StockMarket** class and the **Portfolio** class are related. 4. Take a look at the code in **InsertCodeHere.cs**, where the **Stock­Market** class is used. If you run the application now, it will always report the earnings to be 0.0 %, since the **Portfolio** class is incomplete. 5. Now return to the **Portfolio** class, and implement the property **Total­Earnings­Percentage** and the method **UpdateCurrent­Prices**, as described in the comments. A correct implementation will typically report earnings of a few percent. 6. Once the application works as intended, you can reflect a bit over the structure of the application. See if you can answer these questions:    1. What class holds information about stock prices?    2. What class decides how stock prices are changed?    3. What class decides what stocks we have in the portfolio?    4. If we wanted more than three stocks in the portfolio, where would we need to change the code?    5. Could some parts of the code become simpler, if additional methods or properties were added to some of the classes? |

|  |  |
| --- | --- |
| **Exercise** | OOP.1.7 |
| **Project** | StaticExamples |
| **Purpose** | Defining and using static methods and instance fields. |
| **Description** | The project contains the class **ListMethods**, which defines two methods **FindSmallestNumber** and **FindAverage**. The names should hopefully indicate what the methods do. Code that tests the class is included in the sandbox area. The class is tested in the traditional way; create an object, and call methods on the object. |
| **Steps** | 1. Change the methods in the **ListMethods** class into static methods, by adding the keyword **static** to the method definitions. 2. Modify the code in the sandbox area, such that it uses the methods in **ListMethods** as static methods. The output of running the application should of course be as before. 3. The project also contains a simple class **Car** (see the code). We would now like to track how the class is used. More specifically, we wish to track the number of    1. **Car** objects that have been created    2. Uses of the property **LicensePlate**    3. Uses of the property **Price** 4. Add static instance fields to the **Car** class, to enable the tracking described above. Increment the value of each variable at the appropriate places in the class. 5. Add a static method that can print out the values of the static instance fields. It could be called **PrintUsageStatistics**. 6. Test that your additions work, by including some test code in the sandbox area. Create and use some **Car** objects, and finally call the static method created in step 5) to observe the **Car** usage statistics. |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.1 |
| **Project** | BankV10 |
| **Purpose** | Use simple **if**-statements in an existing class. |
| **Description** | This project contains a minimal **BankAccount** class, that sort of works. However, it has some problems… |
| **Steps** | 1. Test the **BankAccount** class, by adding code in the sandbox area. Specifi­cally, make some tests that make the balance go negative. 2. Now change the code in the **Withdraw** method, such that a withdrawal is only done if the balance is larger than or equal to the given amount. Remem­ber to test that the change works as expected. 3. This makes the **BankAccount** class more realistic, but there are still pro­blems – you can call both **Withdraw** and **Deposit** with negative amounts (try it), which does not make much sense. Make changes to both methods, such that they only perform a withdrawal/deposit if the given amount is positive. Remember that for the **Withdraw** method, the change made in part 2 must still work! 4. Test that all your changes work as expected. 5. If we call **Withdraw** or **Deposit**, and detect an error situation, we don’t do anything… What should we do? |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.2 |
| **Project** | WTF |
| **Purpose** | Use **if-else**-statements. Use method calls creatively. |
| **Description** | This project contains a class called **MysticNumbers**, with a single method **ThreeNumbers**. All that is known about **ThreeNumbers** is that it takes three integers as input, and returns one of them |
| **Steps** | 1. By reading the code for **ThreeNumbers**, try to figure out what it does. Write some test code to see if you are right. 2. Write and test a new method **TwoNumbers**, that does the same thing as **ThreeNumbers**, but now only for two numbers. 3. Write and test a new method **FourNumbers**, that does the same thing as **ThreeNumbers**, but now for four numbers (hint: you can probably use the method **TwoNumbers** to make the code fairly short and easy). 4. Rewrite **ThreeNumbers** to use the **TwoNumbers** method. What code do you like best – the original code or the new code? 5. All the **…Numbers** methods only perform operations on the method para­meters (also note that the **MysticNumbers** class does not contain any instance fields). How can we then change the **…Numbers** methods, such that they are a bit easier to use (hint: we should add a single extra key­word to the definition of each method)? |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.3 |
| **Project** | WeatherStationV10 |
| **Purpose** | Use multi-**if-else**-statements. |
| **Description** | This project contains a class called **Barometer**, containing two pro­perties **Pressure** and **WeatherDescription**. The latter property gives an old-fashioned description of the weather, as a function of the pressure, according to this table:   |  |  | | --- | --- | | **Pressure** | **WeatherDescription** | | Below 980 | Stormy | | 980-1000 | Rainy | | 1000-1020 | Changing | | 1020-1040 | Fair | | Above 1040 | Very dry | |
| **Steps** | 1. Implement the property **WeatherDescription**, according to the table in the description. 2. Test your code. |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.4 |
| **Project** | WhileLoopsBaseCamp |
| **Purpose** | Get some experience with **while**-loops |
| **Description** | The project contains some counter-controlled **while**-loops, and some number sequences that should be generated using **while**-loops. |
| **Steps** | 1. In the sandbox area, four **while**-loops (Case 1-4) are given. Try to figure out what the output from each loop will be. When ready, uncomment the line in each loop that prints the current value of the counter variable, and see if you were right. 2. Next follows Case 5-8. Here you must implement a **while**-loop yourself, to produce the number sequence given in the comment for each case. |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.5 |
| **Project** | CorrectChangeAutomat |
| **Purpose** | Use **while**-loops for a more complicated problem |
| **Description** | This exercise is about calculating the correct change when a customer pays a due amount with too much cash (yes, some people still pay with cash…).  Example: A customer has to pay 266 kr., and pays with a 500 kr. bill. The custo­mer must then receive 234 kr. in change. The tricky part is to figure out how to pay this amount using ordinary bills and coins, and paying back as few bills and coins as possible. In this example, the correct way to pay back correct change would be:   * One 200-kr bill * One 20-kr coin * One 10-kr coin * Two 2-kr coins |
| **Steps** | 1. Implement code to calculate and print out the correct change. To keep things simple, we assume that you only use 100-kr bills, 10-kr coins and 1-kr coins. Remember to test your code with a couple of different values for change. You can just add the code in the sandbox area. 2. Once the above problem is solved, include some more bills and coins, like 50-kr bills, 5-kr coins, etc.. 3. If you used while-loops for solving the problem: Try to solve the problem without using loops. |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.6 |
| **Project** | RolePlayV20 |
| **Purpose** | Get further experience with **while**-loops. Work with a project involving several classes. |
| **Description** | The project is supposed to model a very simple game, where a hero can battle against a beast, until either beast or hero is dead! The project contains four classes, which are described in general terms here – see the code for more details:   * The **NumberGenerator** class, with the method **Next**. This is a helper class for generating random numbers. * The **BattleLog** class, where individual strings can be “saved”, and later on printed out on the screen. * The **Hero** class, which models a game character. It is a very simple model, since it just has a number of hit points. * The **Beast** class, which also models a game character, in a way similar to the **Hero** class.   Even though this is a very simple setup, it does include fundamental game mechanics from many popular role-playing games. |
| **Steps** | 1. Study the classes in details, so you are sure of what they can do and how they work. Note how the **Hero** and **Beast** classes make use of the **Number­Generator** and **BattleLog** classes. 2. See if you can figure out how to code a battle between a **Hero** and a **Beast** (until the death!). A bit of code is present in the sandbox area, but it obviously needs to be extended. 3. When you can make the two game characters battle each other, there are a number of things to consider afterwards:    1. It seems like the **Hero** wins most of the time (depending of course on how you coded the battle…). Why is that? How could we make the battle more fair?    2. The damage dealt by the **Hero** is always between 10 to 30 points. How could we change that? Could we even let the creator of the **Hero** object decide this interval? Could this also be done for the number of initial hit points?    3. Do we really need separate classes for **Hero** and **Beast**? |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.7 |
| **Project** | DrawShapes |
| **Purpose** | Get some experience with **for**-loops |
| **Description** | This exercise is about drawing some simple shapes on the screen, using **for**-loops to get the job done. A very simple class **DrawingTool** is provided to help with this. |
| **Steps** | 1. Study the class **DrawingTool**. As you can see, it is very simple. Why are the methods static? 2. Using **for**-loops and the **DrawingTool** class, see if you can create code to draw the shapes A to E, as defined in the comments in the sandbox area. NOTE: The shapes get increasingly hard to draw… |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.8 |
| **Project** | ListBaseCamp |
| **Purpose** | Get some experience with methods in the **List** class. |
| **Description** | This exercise is about predicting the result of applying some methods in the **List** class to a **List** object, and also about writing some code to use a **List** object |
| **Steps** | 1. In the sandbox area, a **List** object is created, and some elements are added and removed. At four points in the code (Case 1-4), you must predict the outcome of the **WriteLine** statement. When ready, you can uncomment the **WriteLine** statement, and see if your prediction was correct. 2. Following the cases above, four more cases are given (Case 5-8), where you must write code that uses the **List** object, to retrieve various informa­tion about the elements in the list. Details for each case are given as com­ments in the code. |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.9 |
| **Project** | RolePlayV21 |
| **Purpose** | Get some experience with the **List** class. Work with a relatively complex setup of collaborating classes. |
| **Description** | In a previous exercise, we designed a battle between a Hero and a Beast. In this exercise, a Hero will now fight against an “army” of Beasts, represented by a new **BeastArmy** class.  An “army” is essentially a **List** of **Beast** objects. In order to keep the original battle logic intact, we will try to implement properties and methods in the **BeastArmy** class, that correspond to properties and methods in the **Beast** class. However, the implementation does become more complex when dealing with several beasts… |
| **Steps** | 1. Review the classes **Hero** and **Beast**. They are fairly similar to the **Hero** and **Beast** classes from the previous exercise, but do contain a few improve­ments, mostly in the form of more parameters to the constructors. 2. Review the battle logic code found in **InsertCodeHere.cs**. This code manages a 1-on-1 battle. Make sure you understand the logic of the code. 3. Implement the methods **DealDamage** and **ReceiveDamage**, plus the pro­perties **Dead** and **BeastsAlive**, in the class **BeastArmy**. The specification of the methods and properties are found as comments in the code. 4. Change the code in **InsertCodeHere.cs**, to do a battle between a single Hero and an army of Beasts. It may take a bit of balancing (tuning the number of Beasts, their hit points, etc.), to make the battle reasonably fair. 5. Implement some sort of statistics, to be able to measure if the battle setup is fair. You should e.g. be able to run 100 battles, and print out the per­cent­age of battles won by the Hero and the Beast army, respectively. |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.10 |
| **Project** | LibraryV10 |
| **Purpose** | Use the **List** class. Implement linear search. |
| **Description** | This exercise illustrates the concept of a catalog. A catalog is a class that can store and use data of a certain type, without revealing the specific representation of data to the user of the catalog.    The project contains the simple domain class **Book** (we consider the isbn number to be a “key” for **Book**, i.e. no two **Book** objects can have the same isbn number). Also, it contains the (incomplete) catalog class **BookCatalog**. The three public methods in **BookCatalog** allow the user to add, look up and delete **Book** objects in a simple way (see the com-ments in the code for more details about each method). In this ver­sion of the exercise, the **BookCatalog** class uses a **List** to store **Book** objects internally.  The solution also contains a Unit Test project, which tests the public methods in the **BookCatalog** class. |
| **Steps** | 1. Complete the three methods in the **BookCatalog** class, using the com­ments for each method as a guideline for implementation. 2. Build the application, run the Unit tests, and see if the Unit Test is “all green” (if not, you will have to review your code, and maybe debug it…). 3. Is there anything in your code that prevents a user from adding two **Book** objects with the same isbn value? 4. How could you prevent that **Book** objects with the same isbn value are added to the catalog? |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.11 |
| **Project** | LibraryV11 |
| **Purpose** | Use the **Dictionary** class. |
| **Description** | ***NOTE: This exercise is intentionally almost identical to Pro.2.10***  This exercise illustrates the concept of a catalog. A catalog is a class that can store and use data of a certain type, without revealing the specific representation of data to the user of the catalog.    The project contains the simple domain class **Book** (we consider the isbn number to be a “key” for **Book**, i.e. no two **Book** objects can have the same isbn number). Also, it contains the (incomplete) catalog class **BookCatalog**. The three public methods in **BookCatalog** allow the user to add, look up and delete **Book** objects in a simple way (see the com-ments in the code for more details about each method). In this ver­sion of the exercise, the **BookCatalog** class uses a **Dictionary** to store **Book** objects internally.  The solution also contains a Unit Test project, which tests the public methods in the **BookCatalog** class. |
| **Steps** | 1. Complete the three methods in the **BookCatalog** class, using the com­ments for each method as a guideline for implementation. 2. Build the application, run the Unit tests, and see if the Unit Test is “all green” (if not, you will have to review your code, and maybe debug it…). 3. Is there anything in your code that prevents a user from adding two **Book** objects with the same isbn value? 4. How could you prevent that **Book** objects with the same isbn value are added to the catalog? |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.12 |
| **Project** | SchoolAdministrationV10 |
| **Purpose** | Use the **Dictionary** class. Work with an application containing several classes. |
| **Description** | The project contains the class **Student**. This is a simple representation of a student, with three instance fields; id, name and test scores. The first two are simple, but the “test scores” field is a **Dictionary**, holding key-value pairs of course names (string) and scores (int).  The project also contains the class **StudentCatalog**. This class is sup­posed to be able to retrieve various information about the students; for this purpose, an instance field **\_students** of type **Dictionary** is used to hold key-value pairs consisting of ids and **Student** objects (since a student is uniquely identified by an id)  The solution also contains a Unit Test project, which tests the public methods in the **StudentCatalog** class. |
| **Steps** | 1. The class **Student** is complete, and you need not change anything in it. However, take a good look at the **Student** class anyway, and make sure you understand how the methods work. Pay particular attention to the property **ScoreAverage**. 2. Look in the class definition of **StudentCatalog**. It contains four methods (**AddStudent**, **GetStudent**, **GetAverageForStudent**, **GetTotalAverage**) that are not completed. Add code to complete these methods, according to the specification given in the comments in the code. 3. Build the application, run the Unit tests, and see if the Unit Test is “all green” (if not, you will have to review your code, and maybe debug it…). |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.13 |
| **Project** | Flinter |
| **Purpose** | Use enumerations |
| **Description** | **Flinter** is supposed to be the start of a new dating app. You can create profiles for those you are interested in meeting.  In the project, the class **Profile** has been included. The class contains instance fields for gender, eye color, hair color, and height. You can thus create a **Profile** object by specifying values for each of these four fields in the class constructor. Furthermore, you can get a text description of a **Profile** object by using the property **GetDescription**. |
| **Steps** | 1. Code that tests the **Profile** class is included in sandbox area. Examine the code in the **Profile** class definition, and see if you can predict the outcome of running the test. 2. Running the test reveals some problems. In two cases, we have specified hair color where we should have specified eye color, and vice versa (unless you really want a partner with white eyes and blue hair…), and in one case, we have specified a height category that doesn’t exist. Change the **Profile** class definition by adding enumerated types for gender, eye color, hair color and height category. Use these new types for the four instance variables. The constructor needs some changes as well. Also consider if you still need the properties **GenderDescription** and **HeightDescription**. 3. Change the code in the sandbox area , so it is compatible with the rede­signed **Profile** class. Observe how it is now only possible to specify legal values for each type. 4. Reflect a bit on the changes. Is there anything in the new code that is more complicated than it was in the original code? Was it always relevant to use an enumerated type? |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.14 |
| **Project** | CalculationSimulation |
| **Purpose** | Improve code structure by replacing values with constants, instance fields and parameters |
| **Description** | The project contains a simple simulation of a calculation. The inten­tion is to simulate a calculation that takes about half a second. The calculation takes two values x and y, and returns an integer value.  In order to speed up the calculation, a “cache” class is also provided. The idea is that once a calculation has been done, the result can be stored in the cache, from which it can be retrieved very quickly. See the code for further details. |
| **Steps** | 1. The code is set up to do calculations in a 5x5 table (that is, x and y can be numbers between 0 and 4, both included). How many places in the project would you have to change something, if you want to do calculations in a 10x10 table instead? 2. Change the code, such that you get rid of all the instances of the number 5 in the methods. This could be done by using constants, instance fields and parameters. 3. It seems like -1 means “no value”. Change the code, such that the value -1 does not occur in the methods. 4. Are there other values that are candidates for being replaced with con­stants or parameters? If so, make the necessary updates to the code. |

|  |  |
| --- | --- |
| **Exercise** | Pro.2.15 |
| **Project** | WebShopV10 |
| **Purpose** | Improve code structure by creating new methods |
| **Description** | Part of the business logic in a web shop involves calculating the total cost of an order. The logic for calculating the total cost is found in the code in the project.  In the project, the **Order** class contains an item list. For simpli­city, the item list just contains the net price for each item in the order. The class also contains a property **TotalOrderPrice** for calculating the total price for the order.  The solution also contains a Unit Test project, which tests the proper­ty **TotalOrderPrice** in the **Order** class. |
| **Steps** | 1. The implementation of the **TotalOrderPrice** property is less than optimal. Rewrite it, with the intent of:    1. Removing duplicate code    2. Making the method easier to understand 2. Make sure to regularly build the application, run the Unit tests, and con­firm that the Unit Test is still “all green”. If not, you will know that your most recent change caused the tests to fail. |

|  |  |
| --- | --- |
| **Exercise** | OOP.2.1 |
| **Project** | EmployeesV10 |
| **Purpose** | See inheritance in action. Reorganise existing code to use inheritance. Call base class constructors. |
| **Description** | The project contains two existing classes **Teacher** and **ITSupporter**. They have quite a lot in common, so there is a lot of code duplication to get rid of. |
| **Steps** | Reorganise the code using inheritance     1. Create a new class **Employee**, that contains the common parts from **Teacher** and **ITSupporter**. 2. Let **Teacher** and **ITSupporter** inherit from **Employee**. The code in InsertCodeHere.cs should work as before. Remember that the derived classes will need to call the base class constructor. |

|  |  |
| --- | --- |
| **Exercise** | OOP.2.2 |
| **Project** | RolePlayV23 |
| **Purpose** | Override existing methods in derived class |
| **Description** | The project contains a working role-play system. Any character in the game is represented by an object of the class **Character**. |
| **Steps** | 1. Get an overview of the application. The most interesting class is the **Cha­racter** class, which implements a generic game charac­ter. Also note the code in InsertCodeHere.cs, where two teams with two members are set up for battle. 2. Create a class **Defender**, which derives from **Character**. A **Defender** has a 50 % chance of having the received damage reduced by 40 %. This means that the **ReceiveDamage** method must be overrided. Once you have crea­ted the class, update the code in InsertCodeHere.cs to include a **Defender** on each team. 3. Create a class **Damager**, which derives from **Character**. A **Damager** has a 40 % chance of dealing double damage. This means that the **DealDamage** method must be overrided. Once you have created the class, update the code in InsertCode­Here.cs to include a **Damager** on each team. 4. Can we organise the code better, in order to e.g. make calls to the base class methods for dealing and receiving damage? (Maybe the calculation and the logging should be separated into separate methods). |

|  |  |
| --- | --- |
| **Exercise** | OOP.2.3 |
| **Project** | SimpleGeometry |
| **Purpose** | Override abstract methods. See polymorphic behavior in action. |
| **Description** | The project contains the (abstract) class **Shape**, with an abstract property **Area**. The class also contains a static method **FindTotalArea**, that should calculate the total area of a list of shapes |
| **Steps** | 1. Create two classes **Circle** and **Rectangle**. Both classes should inherit from **Shape**, and therefore implement the abstract property **Area**. You also need to figure out what instance fields, etc. the two classes need (if you need the value of π (pi), you can get it by writing **Math.PI**). 2. Implement the **FindTotalArea** method properly, such that it finds the total area of a list of shapes. 3. In the sandbox area, fill in some shapes in the given list, and see if your implementation works as expected |

|  |  |
| --- | --- |
| **Exercise** | OOP.2.4 |
| **Project** | FilteringV10 |
| **Purpose** | Use interfaces to generalise code |
| **Description** | The project contains a class **Filter**, with a **FilterValues** method. The method filters out values higher than 10 from a list of integers. The project also contains an interface ***IFilterCondition***. |
| **Steps** | 1. Figure out how you can use the interface ***IFilterCondition*** to change the **FilterValues** method, into a method that can filter a list of integers accor­ding to any condition. That is, the condition itself has to somehow become a parameter to the method. Try out your solution with a couple of condi­tions. 2. Figure out how you can apply several filter conditions to a list in a single method call. 3. Filtering is a very generic operation. Maybe some of the .NET collection classes already support filtering…? |

|  |  |
| --- | --- |
| **Exercise** | OOP.2.5 |
| **Project** | CarDealershipV05 |
| **Purpose** | Override methods from **Object** class. |
| **Description** | The project contains a simple class **Car**, which contains a few proper­ties. In the sandbox area, we attempt to print out **Car** objects, and perform some comparisons between **Car** objects. |
| **Steps** | 1. Run the program as-is, and observe the result. Can you figure out when the comparisons return **true**? 2. In the **Car** class, uncomment the **Equals** method (only that method), and run the program again. What has changed? 3. Uncomment the rest of the code in the **Car** class, and run the program again. What has changed? 4. The printing of **Car** objects is still not very satisfying. In the **Car** class, over­ride the **ToString** method, so that it returns a string giving a reasonable description of the **Car** object. Run the program again, and see what difference it makes. |

|  |  |
| --- | --- |
| **Exercise** | OOP.2.6 |
| **Project** | CompanyV10 |
| **Purpose** | Use inheritance when creating several classes |
| **Description** | The project contains very little from the start. There is an **Employee** class with a **Name** property, and some abstract properties. Concerning salary calculation, only some very general rules exist:   * Part of the salary is a fixed amount * Part of the salary is a bonus amount * The bonus amount is paid if a certain condition is met   Specific definitions of the rules should be defined in classes that inherit from **Employee** |
| **Steps** | 1. Create a **Worker** class. The class should inherit from **Employee**. For a worker, the below rules apply:  * A worker works a fixed amount of hours per month * A worker is paid a fixed amount per hour * A worker does not receive any sort of bonus  1. Create a **Manager** class, also inheriting from **Employee**. The rules for salary calculation are more vague for a manager:  * A manager has a fixed monthly base salary * A manager has a fixed monthly bonus   The condition for when the bonus is paid out may vary, depending on the specific type of manager (NB: This implies that **Manager** also becomes an abstract class).   1. Create a **JuniorManager** class, that inherits from **Manager**. A junior mana­ger will have the bonus paid out if (s)he has worked at least 180 hours during the month. 2. Create a **SeniorManager** class, that inherits from **Manager**. A senior mana­ger will have the bonus paid out if (s)he has a performance evaluation of at least 6 during the month |

|  |  |
| --- | --- |
| **Exercise** | OOP.2.7 |
| **Project** | BankWithExceptions |
| **Purpose** | Add exceptions and exception handling to a project |
| **Description** | The project contains a class **BankAccount**. It defines a fairly straight­forward bank account, but there are a few restrictions in it (see the code). One restriction is that the balance must not become negative.  The project also contains three additional classes:   * **IllegalInterestRateException** * **NegativeAmountException** * **WithdrawAmountTooLargeException**   They are all exception classes, i.e. they inherit from **Exception**. The specific purpose of each exception class is described in the code. The **BankAccount** class already uses the **WithdrawAmount­TooLarge­Exception** class, to prevent that the balance becomes negative (see the **Withdraw** method) |
| **Steps** | 1. Modify the code in the **BankAccount** class, such that the additional exception classes are used properly. 2. Study the code in **InsertCodeHere.cs**. Make sure you understand why the try-catch statements are included in the code. 3. Run the application, and test that the exceptions are now thrown and handled properly. |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.0 |
| **Project** | ExamAdmV10 |
| **Purpose** | Create data bindings between GUI controls |
| **Description** | The project contains a simple GUI for an exam administration system. In this version, you can just type in a name, a subject and a test score for an exam. The data is entered through two text boxes and a slider control. |
| **Steps** | 1. Open the project, and open the MainPage.xaml file. Even though the file contains quite a bit of XAML, we only need to focus on the three named controls, with the names **student­Name**, **subject** and **score** (two **TextBox** controls and a **Slider** control). Make sure you can find these three controls in the XAML code. 2. We want to bind three **TextBlock** controls to the value of the three named controls. The three **TextBlock** controls are all part of the top line of the GUI, which consists of a total of six **TextBlock** controls. For each of the three relevant **TextBlock** controls, figure out which specific named control to bind to. 3. Now create the actual bindings, using the syntax described in the notes (for a **Slider**, you bind to the **Value** property; for a **TextBox**, you bind to the **Text** property). 4. Test that your bindings work as expected |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.1 |
| **Project** | ExamAdmV11 |
| **Purpose** | Create data bindings between GUI controls and a domain object |
| **Description** | The project is identical to the project from ExamAdmV10, except that a class **Student** has been added. Right now, the constructor in **Student** just sets the properties to some fixed values. |
| **Steps** | 1. Open the MainPage.xaml file, and add a data context to the **Page** control, specifying **Student** as the data context. See the notes for the syntax for adding a data context. 2. Bind the three relevant **TextBlock** controls (the same as in the previous exercise) to the corresponding properties on the **Student** class. Again, see the notes if you cannot remember the syntax for this. 3. Also create bindings for the three named controls, such that each control – or more precisely; the relevant property in each control – is bound to the corresponding **Student** property. 4. Run the application, and check that the bindings work as expected. Try to change the values as well. Are the changes reflected in the text line at the top? |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.2 |
| **Project** | ExamAdmV12 |
| **Purpose** | Create two-way data bindings between GUI controls and a domain object |
| **Description** | The project starts off where the previous exercise left off. The project does contain data bindings, but changes in the values are still not reflected in the rest of the GUI. |
| **Steps** | 1. Open and run the application. Confirm that changes in the values are not reflected in the top text line. 2. Open the **Student** class. All three properties now have a **set**-part as well. Now let **Student** inherit from the **INotifyProperty­Changed** interface, and implement the **OnPropertyChanged** method (if ReSharper if installed, Visual Studio can generate the code for you. If not, you can simply copy-paste the code from the notes). 3. Run the application again – are value changes now reflected in the text line? 4. For each property in **Student**, add a call to **OnPropertyChanged** to the **set**-part of the property, after the value has been set. 5. Run the application again – are value changes now reflected in the text line? 6. For each of the three bindings for the named controls (not the **TextBlock** control), update the binding mode to **TwoWay**. 7. Run the application again – are value changes now reflected in the text line? 8. Why don’t we need to update the three **TextBlock** bindings to being **TwoWay**? |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.3 |
| **Project** | ExamAdmV13 |
| **Purpose** | Create data bindings between collection-oriented GUI controls and domain object collections |
| **Description** | We now introduce a **StudentCollection** into the application. For now, it contains a **Student** list (with one entry), and a subjects list (initia­lised with five entries). The class also has a property **SelectedStudent**, that for now just returns the single entry in the **Student** list. |
| **Steps** | 1. Open the **StudentCollection** class, and make sure you under­stand the instance fields and properties it contains (except the **NewSubject** property) 2. Open the MainPage.xaml file. The bindings are now a bit more complex, since the data context is now **StudentCollection**. Most properties are now bound to the corresponding property on the **SelectedStudent** property (i.e. **Student** object). Make sure you understand the new bindings. 3. Run the application (ignoring the two extra lines beneath “Score”). The application does work, since updates to **name**, **score** and **subject** are reflected in the top text line (try it!). 4. In the “New subject” line, the intention is that when a new subject is entered, it should show up in the Subject combo-box. Confirm that this is not the case right now (remember that you must leave the text box, before the update is triggered). 5. The “No. of subjects” field tells how many entries the **\_subjects** list in **StudentCollection** contains. Right now, the number stays at 5. Figure out how to create a binding for the text box next to the “New subject” text, such that a new entry is indeed added to **\_subjects** (Hint: Take a look at the **NewSubject** property in **StudentCollection**). 6. Once this binding works, the number should increase every time a new subject is added. Still, the new subjects do not show up in the combo-box. Figure out why this is the case, and fix it. (Hint: Are we using the correct collection class for \_**subjects**?) |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.4 |
| **Project** | ExamAdmV14 |
| **Purpose** | Create a data template for presenting objects in a **ListView** |
| **Description** | The application main view contains a **ListView** control, where the **Items­Source** property is bound to the **Students** property on the **Student­Collection** class. The list contains five students. However, the presen­tation of the students in the list view is not optimal. |
| **Steps** | 1. Try to improve the presentation of **Student** objects in the list view, by providing an implementation of **ToString** in the **Student** class. 2. Improve the presentation further by defining a data template for the **Student** class (Tip: you can probably use the properties in the **Student** class for this purpose). |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.5 |
| **Project** | ExamAdmV15 |
| **Purpose** | Create a Master/Details view |
| **Description** | The application main view again contains a **ListView** control, where the **Items­Source** property is bound to the **Students** property on the **Student­Collection** class. The list contains five students, and a reason­able data template has been provided for presentation. The **Student** class has however been extended with several additional properties. |
| **Steps** | 1. Create the Details part of a Master/Details view (the **ListView** is the Mas­ter part), such that all details of a given **Student** object are shown in the Details part. The Details part should show the details of the **Student** which is currently selected in the list view. (Tip: use the example in the notes for inspiration). 2. Use the styles **TextBlockStyle** and **TextBoxStyle** to specify the appearance of the Details part. |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.6 |
| **Project** | ExamAdmV16 |
| **Purpose** | Add deletion functionality to a Master/Details view |
| **Description** | The application contains a working Master/Details view for the **Student** class. We now wish to add functionality to delete a student |
| **Steps** | The steps needed to create deletion functionality are very similar to the steps described in the notes. Almost all changes are done in the **StudentCollection** class. It can be assumed that the **Name** property can be used as a key for **Student** objects.  In the **StudentCollection** class:   1. Change the **\_students** instance field, such that it uses **ObservableCollec­tion** instead of **List** 2. Add a **DoDelete** method, similar to the **DoDelete** method in the notes (it should call the existing **Delete** method) 3. Add a **DoDeleteRelay** method, similar to the **DoDeleteRelay** method in the notes 4. Add a **StudentIsSelected** method, similar to the **CarIsSelected** method in the notes 5. Add a **\_deleteCommand** instance field, of type **RelayCommand** 6. Initialise the **\_deleteCommand** instance field in the constructor, using **DoDeleteRelay** and **StudentIsSelected** as parameters 7. Add a **DeletionCommand** property, similar to the **Deletion­Command** property in the notes 8. Update the **set** part of the **SelectedStudent** property, such that it calls \_**deleteCommand.RaiseCanExecuteChanged**()   In the **MainPage.xaml** file:   1. Add a **Delete** button in a proper place in the view, and bind its **Command** property to **DeletionCommand** 2. Check that you can now delete students from the view! |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.6a (NOTE: Very similar to GUI.1.6, does not use RelayCommand) |
| **Project** | ExamAdmV16a |
| **Purpose** | Add deletion functionality to a Master/Details view |
| **Description** | The application contains a working Master/Details view for the **Student** class. We now wish to add functionality to delete a student |
| **Steps** | The steps needed to create deletion functionality are very similar to the steps described in the notes. Most changes are done in the **StudentCatalog** class. You can assume that the **Name** property can be used as a key for **Student** objects.  In the **StudentCatalog** class:   1. Change the **\_students** instance field, such that it uses **ObservableCollec­tion** instead of **List** 2. Add a **\_deleteCommand** instance field, of type **DeleteCommand** 3. Initialise the **\_deleteCommand** instance field in the constructor, by creating a new **DeleteCommand** object. 4. Add a **DeletionCommand** property, similar to the **Deletion­Command** property shown in the notes 5. Update the **set** part of the **SelectedStudent** property, such that it calls \_**deleteCommand.RaiseCanExecuteChanged**()   In the **DeleteCommand** class:   1. Implement a constructor for **DeleteCommand**, which takes a reference to **StudentCatalog** as parameter. 2. Add an instance field **\_catalog** of type **StudentCatalog**, and initialise it with the parameter from the constructor. 3. Implement **Execute** and **CanExecute**. The implementations will be almost identical to the implementations shown in the notes.   In the **MainPage.xaml** file:   1. Find the **Button** controlwhich implements the **Delete** button, and bind its **Command** property to **DeletionCommand** 2. Check that you can now delete students from the view! |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.7 |
| **Project** | ExamAdmV17 |
| **Purpose** | Consider how insertion/editing functionality can be added to the application |
| **Description** | The application contains a working Master/Details view for the **Student** class, with deletion functionality. A natural extension of the application could be to add functionality for editing existing students, and adding new students |
| **Steps** | 1. Consider what it would take in order to add insertion and editing function­ality to the view. Consider for instance:    * How can we enable editing of specific fields?    * Should all fields be editable?    * Should editable fields be editable all the time?    * How can we manage the “editability” of fields in the Details view?    * What are the detailed steps needed in order to create a new student?    * What sort of validation will be needed when creating a new student (remember we assume that student names are unique)? 2. If you are up to the challenge, feel free to start on the actual implemen­tation of the functionality 3. Finally, consider if the **Student** and **StudentCollection** classes are appro­priate classes for containing all this functionality. Can you envision a better distribution of the functionality? |

|  |  |
| --- | --- |
| **Exercise** | GUI.1.8 |
| **Project** | ExamAdmV18 |
| **Purpose** | Save the day at StudentSoft A/S (see memo below) |
| **Description** | **MEMO**: Finish the Show/Hide details feature in the Student view  **From**: Maurice Fischer (StudentSoft A/S CTO)  **TO**: EASJ Intern (can’t remember the name…)  Hi,  Unfortunately, our main developer on the Exam Administration application died yesterday, due to an unfortunate incident involving a hamster, three small oranges and a large piece of brown cardboard. We would therefore like you to finish up the Show/Hide Details feature he was working on in the Students view. I think it was something about being able to toggle the visibility of parts of the Details view on and off, using a Toggle­Switch or something… Anyway, you can probably figure it out by looking in the C# project, as he said he was “almost done” with it, and he always puts…uhh, used to put comments in the code. I would like a demo of it later today, as we are shipping a new version of the application tomorrow.  Regards,  M. Fischer |
| **Steps** | Do as you’re told… |

|  |  |
| --- | --- |
| **Exercise** | GUI.2.0 |
| **Project** | ExamAdmV20 |
| **Purpose** | Change the given application from a Model-View (MV) architecture to a Model-View-ViewModel (MVVM) architecture. |
| **Description** | The project initially contains a class **Student**, which acts both as a domain class and a “provider” to the main view (via data bindings in MainPage.xaml). |
| **Steps** | Add a new class **StudentViewModel** to the project, which will acts a the ViewModel in an MVVM architecture. This involves:   1. Create the class **StudentViewModel**. 2. Add an instance field **\_domainObject** of type **Student** to **StudentView­Model**, and initialise it to refer to a new **Student** object in the constructor. 3. Let **StudentViewModel** inherit from **INotifyPropertyChanged**, and generate the code needed (Tip: click the lightbulb ☺). If the includes are not generated automatically, add to the top of the file:   using System.ComponentModel;  using System.Runtime.CompilerServices;   1. Add properties **Name**, **Subject** and **Score** to **StudentView­Model**, in the style described in the notes. 2. Clean up the **Student** class, such that it no longer inherits from **INotifyPropertyChanged** 3. Change the data context in **MainPage.xaml**, and check that the new bindings work as expected. 4. Now create a new property **TopLineText** in **StudentView­Model**, of type **string**. The intention is that this property should provide enough information to enable you to delete the six **TextBlocks** in the top line of the GUI, and replace them with a single **TextBlock**, that binds to **TopLineText**. 5. Delete the six **TextBlocks**, replace them with a single **TextBlock**, and bind the new **TextBlock** to **TopLineText**. Are changes to the data reflected in the top text line? 6. Add extra calls of **OnPropertyChanged** to the **Name**, **Subject** and **Score** properties, in the style described in the notes. Check that changes are now reflected in the top text line |

|  |  |
| --- | --- |
| **Exercise** | GUI.2.1 |
| **Project** | ExamAdmV21 |
| **Purpose** | Add and use view model classes in an application |
| **Description** | The given application provides simple read-only functionality for a collection of students (only the Master part of a Master/Details view). However, the view uses the StudentCatalog class as Data Context. For now, the class **StudentItemViewModel** is not used. |
| **Steps** | 1. Add a new class **StudentMasterDetailsViewModel** to the project. The intention is that MainPage.xaml should use this class as its new data context. The class must contain a constructor that does not take any parameters, 2. Add an instance field **\_****studentCatalog** of type **StudentCatalog** to the class, and initialise it in the constructor, to refer to a new **StudentCatalog** object. 3. Take a look at the **StudentItemViewModel** class. Note that the construc-tor takes a parameter of type **Student**. 4. Add a private method **CreateStudentItemViewModelCollection()** to the **StudentMasterDetailsViewModel** class (it’s okay to use a shorter name ☺). The return type of the method should be **List<StudentItemView­Model>**. The method should build the list by looping through the list of **Student** objects in the catalog (Hint: use the **Students** property in the **StudentCatalog** class), and create a new **Student­ItemViewModel** object for each **Student** object. 5. Add a property**StudentItemViewModelCollection** to **Student­Master­View­Model**. Only the **get**-part of the property is needed; it should look like this: **{ return CreateStudentItemViewModelCollection(); }**. 6. Change the data context in MainPage.xaml to use **Student­MasterDetails­ViewModel** instead of **StudentCatalog**, and change the binding of the **ListView** property **ItemsSource** from **Students** to **StudentItemViewModel­Collection**. 7. Rebuild the application, and check that the data is still shown properly when running the application. 8. Do these changes enable you to clean out any properties from **Student** and **StudentCatalog**, that were only there to supply GUI-specific data? |

|  |  |
| --- | --- |
| **Exercise** | GUI.2.2 |
| **Project** | ExamAdmV22 |
| **Purpose** | Add deletion functionality to a working read-only Master/Details view. |
| **Description** | The application contains a functional read-only Master/Details view, where students can be viewed. The application uses the MVVM archi­tecture. We now want to add deletion functionality to the application |
| **Steps** | In the **StudentMasterDetailsViewModel** class (in the **ViewModel** folder):   1. Initialise the \_**deleteCommand** instance field, by creating a new **Delete­Command** object. Note that the **DeleteCommand** constructor requires two parameters; one of type **StudentCatalog**, one of type **StudentMaster­Details­View­Model**. 2. Add a new property **DeletionCommand** (only the **get**-part is needed) of type **ICommand**, that returns **\_deleteCommand**.   In MainPage.xaml (in the **View** folder)   1. Find the **Delete** button (just after the **ListView** control), and bind its **Command** property to **DeletionCommand** 2. Build and run the application. Does the **Delete** button work as it should (probably not…)   Back in the **StudentMasterDetailsViewModel** class:   1. At the end of the **set**-part of the **Student­Item­ViewModelSelected** proper­ty, add a call of **RaiseCanExecuteChanged** on \_**deleteCommand**. 2. Build and run the application again. Does the **Delete** button now work as it should (Hopefully it does ☺)   Consider the below questions:   1. Take a look at the classes **StudentItemViewModel** and **StudentDetails­View­Model**. What are their similarities? What are their differences? What are their specific purposes (Hint: Also take a look at MainPage.xaml)? 2. Which class in the project has knowledge about all the steps involved in deleting a student (deleting a **Student** object from the catalog, and upda­ting the GUI accordingly)? What do we usually call classes of this kind? 3. Imagine we had to add a new domain class **Teacher** to the application, with the same functionality (Master/Details view, Delete functionality). How many new classes would we have to create for that purpose? How similar (or different) will these classes be compared to their **Student**… counterpart? |

|  |  |
| --- | --- |
| **Exercise** | GUI.2.3 |
| **Project** | ExamAdmV23 |
| **Purpose** | Rewrite domain-specific classes to use provided base classes |
| **Description** | The project contains a working Master view with delete function­a­li­ty (the Details part has been omitted). All domain-specific classes are located in the folder **DomainClasses**. A number of base classes are available in the folder **BaseClasses**, but are not used yet. |
| **Steps** | 1. Let the **Student** class inherit from **DomainClassBase**. This will require that you override the property **Key** (you just add the keyword **override** to the pro­perty) 2. Let the **StudentCatalog** class inherit from **CatalogBase<Student>**. You can then delete everything else from the **StudentCatalog** class, except the con­structor. In the constructor, change the calls of **Create** to use the base class method **Add**. 3. Let the **StudentItemViewModel** class inherit from **ItemViewModel­Base<Student>**. Remove the existing lines of code from the constructor body. The constructor must then call the base class constructor with **obj** as parameter. Also delete the instance field **\_domainObject** and the property **DomainObject**, and replace the use of **\_domainObject** with **DomainObject** in the other properties (this will then refer to **DomainObject** from the base class). 4. Let the **StudentViewModelFactory** class inherit from **ViewModelFactory­Base<Student>**. Delete the existing two properties, and implement the abstract method **CreateItemViewModel**. It should just return a new **Student­Item­View­Model** object, taking **obj** as parameter. 5. Let the **StudentMasterDetailsViewModel** class inherit from **Master­Details­­ViewModelBase<Student>** (it no longer needs to inherit from **INotifyPropertyChanged**). Delete everything else from the class definition, except the constructor. 6. Implement the constructor for **StudentMasterDetailsViewModel** like this**:**   public StudentMasterDetailsViewModel()  : base(new StudentCatalog(), new StudentViewModelFactory())  {}   1. Let the **DeleteCommand** class inherit from **DeleteCommandBase<Student, StudentMasterDetailsViewModel>**. Delete everything else from the class definition, except the constructor. Remove the existing lines of code from the constructor body. The constructor must then call the base class constructor with **catalog** and **viewModel** as parameter. 2. In **MainPage.xaml**, change the binding of **ItemsSource** to **ItemView­Model­Collection**, and the binding of **SelectedItem** to **ItemViewModel­Selected** 3. Make sure all files are saved, then build and run the application |

|  |  |
| --- | --- |
| **Exercise** | PARA.0 |
| **Project** | GenericCatalog |
| **Purpose** | Create and use a Catalog class based on Generics |
| **Description** | The project contains three simple domain classes **Car**, **Employee** and **Computer**, and two catalog classes **CarCatalog** and **EmployeeCatalog**. |
| **Steps** | 1. Examine the implementation of **CarCatalog** and **EmployeeCatalog**. Take note of the similarities and differences between the classes. 2. There is currently no catalog class for **Computer**. The next step is there­fore to create a catalog class for storing **Computer** objects. You can choose between two paths:    1. ***The Path of Darkness***: Create a class called **ComputerCatalog**, copy/paste code from one of the existing catalog classes into the new class, and modify it to be able to handle **Computer** objects. Add code to **Main** to test your new class.    2. ***The Path of Light***: Create a type-parameterised class called **Catalog**, which can be used for any domain class. Rewrite the code in **Main** to use the new class for all three domain classes. 3. We now also want to be able to print out the content of (i.e. the objects stored in) a catalog. Add this functionality to your catalog class(es) by adding a method called **PrintAll**, and use it to print out the content of all three catalogs. 4. We now also want to be able to retrieve the number of objects stored in a catalog. Add this functionality to your catalog class(es) by adding a proper­ty called **Count**. 5. Add a new domain class **Phone**, and repeat steps 2, 3 and 4 again. If you chose the Path of Darkness, feel free to reconsider your allegiance… |

|  |  |
| --- | --- |
| **Exercise** | PARA.1 |
| **Project** | GenericsDogsAndCircles |
| **Purpose** | Increase cohesion and decrease coupling in the given project, by adding a type-parameterised class |
| **Description** | The project contains two unrelated domain classes **Dog** and **Circle**. The project also contains the class **ObjectComparer**, which contains methods for finding the “largest” **Dog** and **Circle** object out of three given objects. |
| **Steps** | 1. Examine the three given classes, with particular focus on the **ObjectCom­parer** class. What are the problems with this class? 2. Let **Dog** inherit from **IComparable<Dog>** and implement the **CompareTo** method, as described in the notes. Compare according to **Weight**. 3. Let **Circle** inherit from **IComparable<Circle>** and implement the **Compare­To** method, as described in the notes. Compare according to **Area**. 4. Add a new class **BetterObjectComparer** to the project. The class should take one type parameter **T**, and have the constraint **where T : IComparable<T>** 5. Implement a method **Largest**, that takes three parameters of type **T**, and returns a reference to the “largest” object (hint: remember that you can now call **Compare­To** on a domain object, with another domain object as argument). 6. Rewrite the test code in **Program.cs** to use the new **Better­Object­Compa­rer** class. Test that your new code works as expected. 7. Why does this approach decrease coupling? Is there any coupling left between **BetterObjectComparer** and the domain classes? |

|  |  |
| --- | --- |
| **Exercise** | PARA.2 |
| **Project** | GenericsDogsAndCircles (same project as used in previous exercise) |
| **Purpose** | Achieve further decoupling by using the **IComparer<T>** interface. |
| **Description** | The project starts out just in the previous exercise, with two domain classes **Dog** and **Circle**, and the **ObjectComparer** class. |
| **Steps** | 1. Implement a class **DogCompareByHeight**, which inherits from **IComparer<Dog>**. Implement the **Compare** method as outlined in the notes, and compare dogs by **Height**. 2. Implement a class **CircleCompareByX**, which inherits from **IComparer<Circle>**. Implement the **Compare** method as outlined in the notes, and compare circles by **X** (x-coordinate). 3. Add a new class **EvenBetterObjectComparer** to the project. Note that the class does not need any type parameters. 4. In the **EvenBetterObjectComparer** class, implement a method **Largest<T>** (i.e. a method which takes a type parameter), which takes three para­meters of type **T** and one parameter of type **IComparer<T>**. The method should return a reference to the “largest” object (hint: use the **Compare** method, which is available on the parameter of type **IComparer<T>**). 5. Rewrite the test code in **Program.cs** to use the new **EvenBetter­Object­Comparer** class. Test that your new code works as expected. 6. What are the advantages of this solution, compared to the **BetterObject­Comparer** used in the previous exercise (Hint: Does **Dog** and **Circle** need to implement any interfaces now)? |

|  |  |
| --- | --- |
| **Exercise** | PARA.3 |
| **Project** | GenericsVariance |
| **Purpose** | Illustrate practical benefits from declaring type parameters as co-variant or contra-variant |
| **Description** | The project contains a simple class system for animals: An **Animal** base class, and two derived classes **Bird** and **Cat**. Furthermore, the project contains interfaces and classes for collections and collection processing. |
| **Steps** | 1. Examine the two interfaces **ICollectionGet<T>** and **ICollectionSet<T>**. Pay particular attention to how the type parameter **T** is used in each interface. 2. Examine the class **Collection<T>**. It is a very simple collection class, that implements the two interfaces mentioned above. 3. Examine the **AnimalProcessor** class, which contains four methods. Pay particular attention to the type of the parameter to each method, and to the operations performed inside the methods. 4. Now open **Program.cs**, and examine the code. Notice the commented-out code, which contains 8 method calls (Case A to H). Before un-commenting the code, see if you can work out which method calls are valid, and which are not (Hint: Pay close attention to the specific type of the parameter in each call). 5. Un-comment the code. How many cases did you get right? 6. Now open the **ICollectionGet<T>** interface. Declare the type parameter **T** to be co-variant, by adding the keyword **out** just before the **T**, like this: **ICollectionGet<out T>**. 7. Go back to **Program.cs**. Which case(s) that were previously invalid are now valid? See if you understand why… 8. Now open the **ICollectionSet<T>** interface. Declare the type parameter **T** to be contra-variant, by adding the keyword **in** just before the **T**, like this: **ICollectionSet<in T>**. 9. Go back to **Program.cs**. Which case(s) that were previously invalid are now valid? See if you understand why… 10. Two cases remain invalid. Do you think we in any way could fix this by further adjustments of the interfaces? 11. Since the **Collection** class implements both **ICollectionGet<T>** and **ICollectionSet<T>**, wouldn’t it be easier just to have a single interface **ICollection<T>**, containing all methods from the two interfaces? What would the consequences be? |

|  |  |
| --- | --- |
| **Exercise** | PARA.4 |
| **Project** | LambdaAnimals |
| **Purpose** | Write a couple of small lambda expressions, and use them to filter out certain items in a collection |
| **Description** | The project contains a single class **Dog**. In **Main**, a number of **Dog** objects are created and inserted into a **List<Dog>**. Just below **Main**, a method **ConditionalPrint** is defined, which takes a list and a filtering condition as arguments. |
| **Steps** | 1. At the indicated places in the code, call **ConditionalPrint** with a suitable lambda expression as argument, i.e. an expression which evaluates to true according to each of the three descriptions in the comments. 2. Create a new method **ConditionalPrint2**, which takes two lambda expres­sions as arguments. It should only print out those items that match both lambda expressions. 3. Imagine we have a list of conditions (in the form of lambda expressions) that we wish to use for filtering. See if you can create a **MultiCondi­tional­Print** method for that purpose. |

|  |  |
| --- | --- |
| **Exercise** | PARA.5 |
| **Project** | ClockV20 |
| **Purpose** | Use events to connect two domain objects. |
| **Description** | The project contains two classes:  **PulseGenerator**: This class can generate an event at regu­lar time intervals, and allow other objects to be notified of these events.  **Clock**: This class simulates a simple 24-hour clock |
| **Steps** | 1. Study the **PulseGenerator** class. Note in particular the **Pulse** event. What methods (with regards to parameters and return values) can be attached to this event? 2. Study the **Clock** class. Note in particular the methods **Tick** and **PrintTime**. How do you think they will be related to the **Pulse** event? 3. In **Program.cs**, a **PulseGenerator** object is created. Further down, the method call **theGenerator.Start(200)** is invoked. Between those two lines, create a **Clock** object, e.g. with a Danish text. 4. Just after creating the **Clock** object, attach the relevant methods from the object to the **Pulse** event on the **PulseGenerator** object. 5. Start the application, and see if the clock progresses as expected. 6. Create some additional **Clock** objects with different texts – and perhaps different tick factors – and attach them to the **Pulse** event. 7. Start the application again, and see if the additional clock objects behave as expected. |

|  |  |
| --- | --- |
| **Exercise** | PARA.6 |
| **Project** | StockTrade |
| **Purpose** | Use events to connect various domain objects. |
| **Description** | The project contains classes for (very simplified) simulation of stock trading. Some objects will generate events, while other objects will need to be notified of these events. |
| **Steps** | 1. Study the **TradeLog** class – it is quite simple ☺. 2. Study the **PulseGenerator** class. This class can generate an event at regu­lar time intervals, and allow other objects to be notified of these events. 3. Study the **Stock** class. It simulates a real stock, including price changes. Notice the method **GenerateNewPrice**. Who do you think should call this method? Also notice the event **PriceChanged**. What is it used for? 4. Study the class **StockTrader**. It simulates a real stock trader, in a very simplified way. Note the method **DoTrade**. How is this method related to the **PriceChanged** event in the **Stock** class?   We now want to connect the pieces to create a stock trade simulation. This is done in **Program.cs**.   1. After the creation of the **PulseGenerator** object – but before the call of **thePulseGenerator.Start** – create a few **Stock** objects. Choose some reasonable upper and lower limits for stock prices. 2. Create some **StockTrader** objects. Each stock should be traded by at least one stock trader. 3. Make sure that the **Stock** objects have their **GenerateNewPrice** method called, whenever the **PulseGenerator** object generates a **Pulse** event. 4. Make sure that the **StockTrader** objects are notified about changes in stock prices for the relevant stocks. 5. On each **Pulse** event, print out the current price of all stocks (hint: Create a lambda expression which prints out the stock prices, and attach it to the **Pulse** event). 6. On the **LastPulse** event, print out the entire trade log. 7. Run the application. Check to see if the trades obey the limits set for each stock trader. 8. See if you can extend the stock trader model, maybe by letting each stock trader trade more than one stock, have more advanced criteria for buying or selling, etc.. |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.1 |
| **Project** | DataStructureCompare |
| **Purpose** | Observe a comparative test of three collection classes, when exposed to various types of use. |
| **Description** | The project contains classes that enable you to measure the perform­ance (in terms of run-time) of various operations, when performed on various collection classes. |
| **Steps** | 1. The class **TimedTester** is a general class for measuring the run-time of a method invocation. Have a look at the class, and notice the very useful class **Stopwatch** from the .NET library. 2. The interface **IDataStructureTester** and the class **DataStructureTesterBase** are general-purpose classes for collection class test. Study the classes, until you feel you understand the general structure of the test. 3. The classes **ListTester**, **LinkedListTester** and **HashSetTester** contain the specific test code for each collection class, i.e. the methods which have a specific implementation for this particular class. Compare how the test is done for each class, i.e. how are the **…Statement** methods implemented for each collection class. 4. **Program.cs** contains the code which executes the test. Get an overview of the test, and try to run the application. 5. Given the discussion about pros and cons of the various collection classes, do the actual run-times reported by the tests make sense? Or are there any surprising results? Note that it is not the absolute run-times that are of interest here – it is the relative measurements of performing the same operation of different collection classes, or different operations on the same collection class. 6. Try to increase the value of **noOfInserts**, in order to increase the number of times the various operations are invoked. What are your expectations to the running time of the various operations, if you e.g. double the value of **noOfInserts**? Do the results match your expectations? If not, try to think about plausible reasons for this. |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.2 |
| **Project** | Palindrome |
| **Purpose** | Solve a simple problem using a recursive approach |
| **Description** | A **palindrome** is a phrase that reads the same backwards and forwards, like “Racecar” or “Amore Roma”. Note that spaces and upper/lowercase is ignored in this definition.  The **Palindrome** project contains an interface **IPalindromeChecker** and a class **PalindromeChecker**. |
| **Steps** | 1. Study the interface **IPalindromeChecker** and the class **Palin­drome­Checker**. They are both quite simple. 2. In **Program.cs**, some test code has been provided. The test code makes it easy to check if your palindrome checker works properly. Try to run the application, and see the results. 3. In **PalindromeChecker**, the method **IsPalindromeInternal** is not imple­men­ted properly. Implement a version that actually works, using a recursive approach.    1. Think about how you can divide the original problem into small­er problems, and also about when the problem is trivially solved.    2. You will probably need to use the method **Substring**, which can be called on variables of type **string**.    3. Once you think the implementation is correct, you can just run the application again, and study the test output. |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.3 |
| **Project** | BackPacking |
| **Purpose** | Solve a real-life problem using a recursive approach |
| **Description** | The project contains several classes to support solving of the so-called **Backpacking** problem:  Given   * A **backpack** with limited weight capacity * An **item vault** with a set of **items**, each with a weight and a value   Select a set of items from the item vault, such that:   * The items can fit into the backpack (i.e. the weight capacity of the backpack is not exceeded). * The items have as high a total value as possible. |
| **Steps** | 1. Study the single class **BackPackItem** in the **Item** folder. It should be fairly straightforward. 2. Study the classes in the **Containers** folder (start with **BackPackItem­Container**), until you understand their purpose and functionality. 3. Study the classes in the **Algorithms** folder (start with I**BackPackingSolver**), until you understand their purpose and functionality. Where does recursion come into play? 4. Study the code in **Program.cs** – it uses the “stupid” solver to solve a specific backpacking problem. 5. Run the program, and study the output. Are there some obvious indica­tions that the algorithm does not produce the best possible result? 6. Now create a new class **BackPackingSolverSmart**, which should inherit from **BackPackingSolverBase**. Implement a smarter version of **Solve**, i.e. an algorithm which is smarter than the one found in **BackPackingSolver­Stupid**. The crucial step is to figure out a better way to pick the next item from the vault. See if you can beat the result produced by the stupid algorithm. 7. Once you have implemented a better algorithm, reconsider if the structure for the **BackPackingSolver…** classes is optimal. Could you move some (dupli­cated) code into the **BackPackingSolverBase** class? 8. Try out other criteria for picking the “best” item of the remaining items, and see if you can beat your first attempt. |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.4 |
| **Project** | LINQDrink |
| **Purpose** | Use LINQ queries on a single collection of objects |
| **Description** | The project contains a **Drink** class, which is fairly straightforward. In **Program.cs**, a **List** of **Drink** objects is created. |
| **Steps** | For each of the below cases, do the following:   * Write a LINQ query that returns the speci­fied result * Print out the result of the query, using e.g. a **foreach**-loop:  1. The names of all drinks. 2. The names of all drinks without alcohol. 3. The name, alcohol part and alcohol amount for all drinks with alcohol. 4. The names of all drinks in alphabetical order. 5. The total amount of alcohol in the drinks. 6. The average amount of alcohol in drinks with alcohol. 7. The name and alcohol amount of each drink, grouped by name of alcohol part (NB: We have not discussed grouping in class! Seek information about the **group** LINQ operator online in order to solve this case ☺) |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.4a |
| **Project** | LINQHotels |
| **Purpose** | Use LINQ queries on two collections of objects |
| **Description** | The project contains the classes **Hotel** and **Room**, which have a one-to-many relationship. In **Program.cs**, a **List** of **Hotel** objects and a **List** of **Room** objects are created. |
| **Steps** | For each of the below cases, do the following (Note the two helper methods **PrintEnumerableQueryResult** and **PrintNumericQueryResult**):   * Write a LINQ query that returns the speci­fied result * Print out the result of the query, using e.g. a **foreach**-loop:  1. The full details of all hotels 2. The full details of all hotels in Roskilde 3. Names of all hotels in Roskilde 4. All double rooms with a price below 400 kr. 5. All double or familiy rooms with a price below 400 kr., in order of price 6. All hotels which start with “P” 7. The number of hotels 8. The number of hotels in Roskilde 9. The average price of hotel rooms 10. The total price of all double hotel rooms |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.4b |
| **Project** | SchoolAdministrationV15 |
| **Purpose** | Rewrite old-school procedural logic to new, shiny declarative logic using LINQ! |
| **Description** | The project is essentially a solution to an old exercise (Pro.2.12, using the SchoolAdministrationV10 project). In its current form, the project uses classic, procedural logic to answer questions about score avera­ges for individual students, and for all students. |
| **Steps** | 1. Study the project in its current form. Pay particular attention to the pro­per­ties **ScoreAverage** in **Student**, and **TotalAverage** in **StudentCatalog**. 2. Run the application, and take note of the output. 3. Now rewrite the logic in the two properties mentioned above, using LINQ where you find it appropriate (Hint: concentrate on the **else**-part of both of the properties mentioned above). 4. Run the application again – the output should of course be exactly the same as before. |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.5 |
| **Project** | LINQCocktails |
| **Purpose** | Use LINQ queries – including **join** – on two collections of objects |
| **Description** | The project contains an **Ingredient** class and a **Cocktail** class. A **Cock­tail** object contains a collection of (references to) **Ingredient** objects. In **Program.cs**, a **List** of **Ingredient** objects and a **List** of **Cocktail** objects are created. |
| **Steps** | First study the **Ingredient** class and the **Cocktail** class, to ensure you understand their structure. How does a **Cocktail** object refer to **Ingredient** objects? It might help to create an ER-diagram to understand the relation between cocktails and ingredients.  Then, for each of the below cases, write a LINQ query that returns the speci­fied result, and print out the result of the query (NB: Note that some queries will return collec­tions of collections, so you may need a nest­ed loop to print the query result properly):   1. The names of all cocktails. 2. For each cocktail: The name of the cocktail, and the name and amount of all ingredients 3. For each cocktail: The name of the cocktail, and the name of all ingredi­ents with an alcohol percentage above 10 % 4. For each cocktail: The name and the price of the cocktail (note that the price (per cl.) for an ingredient can be found in the **Ingredient** object collection). 5. For each cocktail: The name and the alcohol percentage of the cocktail. |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.6 |
| **Project** | CrossTalk |
| **Purpose** | See how the **Task** class can be used to execute some very simple operations in parallel |
| **Description** | The project contains the class **Reciter**, which contains a couple of fairly simple methods. The method **ReciteAllTheWords** executes a recitement of the numbers 1 to 8, in three different languages. Each recitement is done by calling the method **Recite**. |
| **Steps** | 1. Run the application. You will see that the recitements are done sequen­tially, i.e. one recitement is completed before the next recitement is executed. 2. Change the code in the **ReciteAllTheWords** method, such that the recitements are done in parallel. The effect should be that the printouts on the screen are a mix of the three languages (Hint: each recitement should be turned into a task). 3. A side-effect of the changes made in step 2 is that the message *"Press any key to close application"* is now printed long before the last word is printed. Why does this happen? |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.7 |
| **Project** | NumericalPi |
| **Purpose** | Change the execution of a time-consuming calculation to use **Tasks**, to decrease the absolute running time. |
| **Description** | The project contains the class **PiCalc**, which contains an algorithm for calculating an approximate value of π (the exact value of π can be retrieved from **Math.PI**).  The method **Iterate** will do this for the specified number of iterations:   * Generate a random point within the square [0;1[ x [0;1[ (a.k.a. the unit square) * Count the number of times the point falls within the circle with center at (0,0), and radius 1 (a.k.a. the unit circle).   The ratio between the parameter **iterations** and the returned number (**inside­Unit­Circle**) will approximate π/4. The higher the number of iterations, the closer the ratio will come to π/4. The final estimate is then easy to calculate, as is done in the **Calculate** method.  Suppose you had to do this calculation manually. You could e.g. draw the square and circle on a piece of paper, and throw a dart at the paper e.g. 100 times. You should then count the number of times the dart has hit within the circle. Say the dart hit within the circle 77 times. Your estimate of π would then be (4.0 \* 77) / 100 = 3.08. |
| **Steps** | 1. You invite three friends over to help with your experiment. A total of four persons can now throw darts. How will you utilise this to speed up the experiment? 2. See if you can translate your redesigned experiment into a new version of **Calculate**, where you use **Task** objects (Hint: wrap the code corresponding to what one person should do, into a **Task** object, like **Task task1 = Task.Run( () => {…});**) 3. Compare the running time of your new version of **Calculate** with the original version. How much faster is your version? 4. See if you can figure out how many cores your CPU has. How does this number relate to what you observed in step 3? |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.8 |
| **Project** | SlicesOfPi |
| **Purpose** | See the usefulness of using the **async**/**await** programming model |
| **Description** | In a previous exercise, we saw a an algorithm for calculating an approximate value of π. The algorithm is fairly easy to speed up using tasks, but it still has to run to comple­tion, before a value is available. It could be useful to – at any time during the calculation – be able to:   * See how good the currently calculated value of π is. * Stop the calculation, and use to current value as the final result.   The project contains two approaches to solving this.   * A synchronous approach (i.e. no use of tasks or **async**/**await**), where the user can request that small “slices” of the calculation are done. A calcu­lation slice can not be interrupted. After each slice, the user can choose to accept the calculated value, or perform additional slices. * An asynchronous approach, where the main calculation loop is wrapped into a task, which is awaited. This makes it possible to interrupt the calcu­lation at any time. |
| **Steps** | 1. Study the class **PiCalcData** in the project. It holds a bit of data related to the calculation, and is fairly simple. 2. Study the method **Calculate** in the class **PiCalcAlgorithm**. It uses a tradi­tional, sequential style and cannot be interrupted. 3. Study the methods **RunPiCalculation** and **MainUILoop** in the class **PiCalc­UI**. They manage the user interaction for the synchronous approach. Note how the **MainUILoop** method calls **Calculate**, with the number of itera­tions specified by the user. 4. Try to run the application (make sure that **RunPiCalculation** is called in **Main**). How do you feel about this way of interaction? Try to enter a large (VERY large, like 100,000,000) number of iterations. Can you interact with the application during the calculation? 5. Now study the methods **RunPiCalculationAsync** and **MainUILoopForAsync** in the class **PiCalc­UI**. They manage the user interaction for the asynchron­ous approach. In particular, note how the **RunPiCalculationAsync** method calls **CalculateAsync**. That call returns a **Task<double>** object; where is that object awaited? What happens just before that object is awaited? 6. Try to run the application, using **RunPiCalculationAsync** in **Program.cs** instead. How has the interaction changed, compared to step 4? Can you now interact with the application while the calculation is ongoing? |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.9 |
| **Project** | ProducerConsumer |
| **Purpose** | Work with a typical producer/consumer setup, involving Tasks and data locking |
| **Description** | The project contains several classes, which participate in a so-called **producer/consumer** scenario. A “producer” produces objects and inserts them into a data structure, while a “consumer” consumes objects by removing them from the same data structure. The production and consumption should be executed in parallel |
| **Steps** | 1. Study all of the classes in the project. Note that this a a fairly large project, and some of the classes are a bit complex, so be prepared to spend some time on this steps. 2. Team up with another student, and discuss your understanding of the overall application, and the individual classes. If you disagree on something, you must investigate the code further. 3. Try to run the application as-is. You will see that the reporting looks strange (sort of jumps up and down a bit on the screen). Why do you suppose this happens? Try to fix the problem (Hint: Maybe only one thread should try to print on the screen at any time…) 4. Try to change the reporting mode (last parameter in the **Scenario** constructor) to **ReportMode.silent**, and re-run the application. Why is the message *"Press any key to abort the run..."* printed almost immediately after starting the app? 5. The above run probably gave the result that all balances were good. Try to change the first five parameters to the **Scenario** constructor to 1000, 500, 1000, 3, 2 (also change the report mode to **silent**). Re-run the app. Do you see any bad balances now? Try to run the app a few times. Do the results change from run to run? 6. See if you can figure out how to get rid of the bad balances problem (Hint: What data can be accessed by both the producer and consumer?). 7. See if you can extend the application to include several producers and/or consumers. |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.10 |
| **Project** | TestExampleA |
| **Purpose** | Implement a method, using existing unit tests as guidance. |
| **Description** | The solution contains two projects:   * The project **TestExampleA** con­tains the class **ListMethods**, with the single method **SumOfSquaresOf­Positives**. * The project **UnitTestProject** contains the class **ListMethods­UnitTest**, which contains test cases for testing the method **SumOfSquaresOf­Positives**. |
| **Steps** | 1. Study the **SumOfSquaresOf­Positives** method. The method itself is initially empty, so focus on understanding what the method should do. This is described in the comments. 2. Study the test cases in **ListMethods­UnitTest**. Try to run the tests (open the **Test Explorer** window, by choosing **Test** | **Windows** | **Test Explorer**, and click on **Run All** in the **Test Explorer** window). Note that some of the tests actually pass, even though the method is clearly not correctly implemen­ted yet. 3. Implement **SumOfSquaresOf­Positives** correctly, such that all test cases pass. 4. Do you find the existing test cases to be sufficient? Can you think of some test cases it would be useful to add? |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.11 |
| **Project** | TestExampleB |
| **Purpose** | Implement a unit test for an existing class. |
| **Description** | The solution contains two projects:   * The project **TestExampleB** con­tains the class **Warrior**. * The project **UnitTestProject** contains the class **Warrior­UnitTest**, which should contain test cases for testing the **Warrior** class. Initially, the unit test only contains a few test cases, and needs to be extended. |
| **Steps** | 1. Study the **Warrior** class, until you have a detailed understan­ding of how it is intended to work. 2. Study the existing test cases in **Warrior­UnitTest**. They are clearly insuffi­cient… 3. Add new test cases to **Warrior­UnitTest**, until you feel you have covered all aspects of the functionality. You can use the existing test cases for inspira­tion, with regards to how to struc­ture test cases. |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.12 |
| **Project** | TestExampleC |
| **Purpose** | Implement both a class and associated unit test, given a requirement specification |
| **Description** | The solution contains two projects:   * The project **TestExampleC** con­tains the class **Currency­Exchange**. * The project **UnitTestProject** contains the class **Currency­Exchange­Test**.   None of the classes are complete, however. The starting point is the below requirement specification (see next page), from which the **CurrencyExchange** class and the **CurrencyExchangeTest** class must be completed. |
| **Steps** | 1. Implement the **CurrencyExchange** class and the **Currency­Exchange­Test** class, given the below requirement specification. If you are in doubt about a specific requirement detail, you must make a decision about how to interpret it, and work forward from that. Note that it is perfectly fine – even encouraged – to define interfaces, helper methods classes, etc. in order to create the implementation. 2. When you are done with the requirements specified below, you can try to extend the class (and the test class) along these lines:  * If the currency cross **AAABBB** is specified, you can calculate exchanges from **AAA** to **BBB**. However, that should also make it possible to calcu­late exchanges from **BBB** to **AAA**. * Suppose you wish to make an exchange from **AAA** to **CCC**. This curren­cy cross has not been specified, but **AAABBB** and **BBBCCC** have. This can be utilised to calculate the **AAACCC** exchange rate. * Exchange rates should be consistent. If you have specified **AAABBB** = 2 and **BBBCCC** = 3, it should not be possible to set **AAACCC** to e.g. 7 (it should be 6). |

|  |  |
| --- | --- |
| **Exercise** | PRO.3.12 (continued) |
| **Requirement**  **Specification** | **CurrencyExchange – Definitions**   * A **currency identifier** is defined as being a three-character acronym for a currency. Examples are **USD** (US Dollars), **EUR** (Euro), **DKK** (Danish Kroner), and so on. * A **currency cross** is a combination of two different currency identifiers. Examples are **EURUSD** (Euro to US Dollars), **DKKEUR** (Danish Kroner to Euro), and so on. * An **exchange rate** is a currency cross and a positive decimal number. Example: (**EURUSD**, 1.20), meaning that 1.00 Euro is worth 1.20 US Dollar.   **CurrencyExchange – Requirement specifications**   * It must be possible to set a number of exchange rates. * Trying to specify an illegal exchange rate (see Definitions), should cause an exception to be thrown. * It is permitted to change an existing exchange rate, simply by specifying it again. * Given a currency cross **AAABBB** and a positive amount of currency **AAA**, it must be possible to calculate the amount obtained by exchanging the amount to currency **BBB**. Example: Given **USDDKK** = 6.50 and an amount of 200 **USD**, the result should be 1300 **DKK**. * Trying to perform the calcuation with either an illegal (or non-existing) currency cross or illegal amount, should cause an exception to be thrown. |

|  |  |
| --- | --- |
| **Exercise** | DAPE.1 |
| **Project** | NoteBookV10 |
| **Purpose** | Add Load- and Save-functionality to an MVVM application |
| **Description** | The given application contains a very simple system for creating notes. A note consists of a title and some content. It is not allowed to have two notes with the same title. However, the application does not support saving and loading of notes yet. |
| **Steps** | In the **NoteMasterDetailsViewModel** class:   1. Add a new instance field **\_loadCommand**, of type **Relay­Command** 2. Add a new property **LoadCommand**, of type **ICommand**. It should just return the instance field **\_loadCommand**, in the same style as e.g. the **AddCommand** property. 3. Add a new method **Load**, in the same style as in the notes. That is, it should call **Load** on the **\_model** instance field. 4. In the constructor, initialise **\_loadCommand** in the same style as in the notes. 5. In the method **NotifyCommands**, add a call of **RaiseCanExecuteChanged** on the **\_loadCommand** instance field 6. Repeat steps 1-5 for the Save functionality   In the **MainPage.xaml** file:   1. Add two new buttons **Load** and **Save** to the view, and bind them to the **LoadCommand** and **SaveCommand** property, respectively. 2. Rebuild the application, and see if you can now load and save notes. Create some notes, click **Save**, close the application, start it again, click **Load**, and see if the saved notes reappear. |

|  |  |
| --- | --- |
| **Exercise** | DAPE.2 |
| **Project** | NoteBookV20 |
| **Purpose** | Use exceptions for error-handling in an MVVM application |
| **Description** | The given application checks that notes cannot have the same title (try it!), but the implementation is quite a mess… The handling is all done in the **set**-part of the **Title** property in **NoteDetailsViewModel**, with several calls to the model and the master-details view model |
| **Steps** | Our aim is to clean up the error handling. This involves using exceptions for error signaling and handling, and also to distribute various responsibilities to the proper classes. An exception class **TitleExistsException** is included in the project.   1. In the **NoteModel** class, add checks to the methods **Add** and **UpdateTitle**, such that a **TitleExistsException** is thrown if the new title exists 2. In the **NoteMasterDetailsViewModel** class, uncomment the method **UpdateTitle**. See if you understand why the method is structured in this particular way. 3. In the **NoteDetailsViewModel** class, go to the **set**-part of the **Title** property. Remove ALL the code in the **set**-part, and replace it with a single line of code:   \_masterDetailsViewModel.UpdateTitle(value);   1. Clean up the **NoteDetailsViewModel** class a bit, since it no longer needs a reference to the model (remove the instance field, and remove the parameter from the constructor) 2. Rebuild the application and run it. See if the validation of titles still works as before. 3. See if you can answer the below questions:    1. Which class **detects and signals** the error?    2. Which class **assumes responsibility** for handling the error?    3. Which class **reports** the error to the user? 4. If you have more time, see if you can update the application such that the **Title** and **Content** fields in the view are only enabled if the user has selected a Note in the list view. |

|  |  |
| --- | --- |
| **Exercise** | DAPE.3 |
| **Project** | DBandEFMovie |
| **Purpose** | Use the Entity Framework to establish a connection to a local relational database, and to retrieve and alter data in the database. |
| **Description** | In this exercise, we create a local relational database, and create tables and data in the database, by using a given database script. We then try to connect to the database through the Entity Framework, and access the data in the tables. |
| **Steps** | The project contains a text file called **MovieDBScript.txt**. This script can be used to generate tables in a database, and insert some sample data into the tables.   1. Create a new local database called **MovieDB**. 2. Run the given script file on the database (Right-click on the database in the **SQL Server Object Explorer** window, choose **New Query**, copy the content of the script file into the query window, and execute the query). The data­base should now contain two tables **Movie** and **Studio**, and both tables should contain some sample data (seven records in **Movie**, three records in **Studio**). 3. In the **DBandEFMovie** project, use the Entity Framework to create a class model corresponding to the **MovieDB** database, by following the guide­lines in the notes closely. This process should add three new classes named **MovieDBContext**, **Movie** and **Studio** to the project. 4. In order to make it easy to print out **Movie** and **Studio** objects, add an override of **ToString** to each class. The specific implementation is up to you (tip: call **TrimEnd(' ')** on string properties to get rid of trailing spaces for strings). 5. Implement the methods **PrintAllMovies** and **PrintAllStudios** in the class **MovieDBTester**, using the style suggested in the notes. The methods are already being called from **Main** in **Program.cs** - test that the methods produce the expected result. 6. Experiment with creating, updating and deleting some **Movie** and **Studio** objects (you can always recreate the original data by running the database script again). Remember that any changes you make will not be reflected in the database before calling **SaveChanges** on the **MovieDBContext** object. |

|  |  |
| --- | --- |
| **Exercise** | DAPE.4 |
| **Project** | DBandEFHotel |
| **Purpose** | Use the Entity Framework to establish a connection to a local relational database, and to retrieve and alter data in the database. |
| **Description** | In this exercise, we create a local relational database, and create tables and data in the database, by using a given database script. We then try to connect to the database through the Entity Framework, and access the data in the tables. |
| **Steps** | The project contains a text file called **HotelDBScript.txt**. This script can be used to generate tables in a database, and insert some sample data into the tables.   1. Create a new local database called **HotelDB**. 2. Run the given script file on the database (Right-click on the database in the **SQL Server Object Explorer** window, choose **New Query**, copy the content of the script file into the query window, and execute the query). The data­base should now contain four tables **Booking, Guest, Hotel** and **Room**, and all tables should contain some sample data. 3. In the **DBandEFHotel** project, use the Entity Framework to create a class model corresponding to the **HotelDB** database, by following the guide­lines in the notes closely. This process should add five new classes named **HotelDBContext**, **Booking, Guest, Hotel** and **Room** to the project. 4. Verify that you are now capable of retrieving and altering the data from the database, e.g. by writing some methods similar to the methods used in the previous exercise. 5. Take a closer look at the classes generated by the Entity Framework. Note how some of the classes have relations to other classes (for instance **Hotel** and **Room**). Compare the relations between classes with the information in the database script, more specifically the information about foreign key relationships. How does a foreign key relationship “translate” into a class relationship? |

|  |  |
| --- | --- |
| **Exercise** | DAPE.5 |
| **Solution** | DBEFandWSMovie (contains two projects) |
| **Purpose** | Use the Entity Framework and a RESTful Web Service to enable a client to access data in a local database |
| **Description** | In this exercise, we create a local relational database, and create tables and data in the database, by using a given database script. We then try to connect to the database through the Entity Framework, and expose the data by using a RESTful Web Service. |
| **Steps** | The project contains a text file called **MovieDBScript.txt**. This script can be used to generate tables in a database, and insert some sample data into the tables.   1. Create a new local database called **MovieDB** (if you don’t have it already). 2. Run the given script file on the database. The data­base should now con­tain two tables **Movie** and **Studio**, and both tables should contain some sample data. 3. In the **DBEFandWSMovieServer** project, use the Entity Framework to create a class model corresponding to the **MovieDB** database. Make sure that the classes **Movie** and **Studio** are generated. Also remember to update the **MovieDBContext** constructor with the two calls to the base class described in the notes. 4. Still in the **DBEFandWSMovieServer** project, now create a controller for both the **Movie** and **Studio** classes (one controller for each class), by fol­lowing the guidelines in the notes. This should generate a **MoviesControl­ler** and a **StudiosController** class in the **Controllers** folder. 5. Once the controllers are generated, the web service is ready to run. You can test it by right-clicking on the **DBEFandWSMovieServer** project, and choose **View | View in browser**. You should see the standard ASP.Net start page. If you choose **API** in the menu, you should see the web service API for **Studio** and **Movie**, both containing five methods. 6. Once the web service is up-and-running, you should be able to compile and run the **DBEFandWSMovieClient** project. That project makes use of the web service server, by calling some of the methods in the web API. The test is performed by using the class **WebAPITest**, which in turn uses the class **WebAPIAsync**, which contains a general implementation of a wrap­per around the web API method calls. 7. Feel free to explore the classes **WebAPITest** and **WebAPIAsync** in more detail, and use them to conduct further tests of the web service. |

|  |  |
| --- | --- |
| **Exercise** | DAPE.6 |
| **Solution** | DBEFandWSHotel (contains two projects) |
| **Purpose** | Use the Entity Framework and a RESTful Web Service to enable a client to access data in a local database |
| **Description** | In this exercise, we create a local relational database, and create tables and data in the database, by using a given database script. We then try to connect to the database through the Entity Framework, and expose the data by using a RESTful Web Service. |
| **Steps** | The project contains a text file called **HotelDBScript.txt**. This script can be used to generate tables in a database, and insert some sample data into the tables.   1. Create a new local database called **HotelDB** (if you don’t have it already). 2. Run the given script file on the database. The data­base should now con­tain four tables **Booking**, **Guest**, **Hotel** and **Room**, and all tables should contain some sample data. 3. In the **DBEFandWSHotelServer** project, use the Entity Framework to create a class model corresponding to the **HotelDB** database. Make sure that the classes **Booking**, **Guest**, **Hotel** and **Room** are generated. 4. Still in the **DBEFandWSHotelServer** project, now create a controller for each of the classes mentioned above (one controller for each class), by fol­lowing the guidelines in the notes. Make sure that all controller classes are generated in the **Controllers** folder. 5. Once the controllers are generated, the web service is ready to run. You can test it by right-clicking on the **DBEFandWSHotelServer** project, and choose **View | View in browser**. You should see the standard ASP.Net start page. If you choose **API** in the menu, you should see the web service API for all of the classes. 6. Once the web service is up-and-running, you should be able to compile and run the **DBEFandWSHotelClient** project. Using a style similar to the style used in the previous exercise, add some code that tests the web service APIs. You should of course try to run the tests, and verify that the methods return the expected data. |

|  |  |
| --- | --- |
| **Exercise** | DAPE.7 |
| **Solution** | DataSourceExample (contains two projects) |
| **Purpose** | Illustrate the benefits of using interface-based programming, using the problem of accessing different data source as an example |
| **Description** | This exercise focuses on the client side of a client-server setup, where the client needs to interact with the server in order to work with domain data. The exercise does not require that you do much pro­gramming, or even run the program ☺  The **Client** project contains several files, organised in three folders.   * The **EFClasses** folder contains classes which were auto-generated by the Entity Framework, corresponding to the **MovieDB** database used in previous exercises (containing a **Movie** and **Studio** table). They are not as such important for this exercise. * The **ModelClasses** contains interfaces and classes defining a very simple framework for a domain data model. * The **DataSourceClasses** folder contains interfaces and classes for accessing data sources. |
| **Steps** | 1. Study the classes/interfaces in the **ModelClasses** folder. Make sure you understand the responsibilities for each class. 2. Study the classes/interfaces in the **DataSourceClasses** folder. Make sure you understand the responsibilities for each class. Why is it useful to de­fine an interface **IDataSourceAsync** for data sources? What is the interface used for? 3. Study the code in **Program.cs**. What happens in the **SelectSource** method? What parts of the client application as a whole knows the specific data source we are currently using? Do the **Model** and **Catalog** classes know? 4. Suppose we have to support a third way of accessing data, e.g. from a file. How would you implement that functionality in the client application? What needs to be added/changed? 5. Suppose we for some reason forbid to use interface (and base classes in general) in the client application. How would that impact the structure of the code? |

|  |  |
| --- | --- |
| **Exercise** | DAPE.8 |
| **Solution** | AzureWebServiceA (contains two projects) |
| **Purpose** | Deploy a RESTful web service to Azure, and connect to it from a console application |
| **Description** | The solution contains the project **AzureCarRetailDBWebService**, which is a RESTful web service application (or rather; we will turn it into a web service). The project **ConsoleClient** contains code for a simple test of the web service. |
| **Steps** | 1. Go to your Azure account, and create a new database called **CarRetailDB**. Part of that process may also involve creating a new SQL Server, if you do not have one running already. 2. Open the database in Visual Studio (note that you may need to configure the firewall settings for your database server – see the notes), and run the script **CarRetailDBScript.txt** on it. This will create four tables **Car**, **Custo­mer**, **Employee** and **Sale**, and insert some sample data into the tables. 3. Go back to the **AzureCarRetailDBWebService** project, and turn it into a RESTful web service. That is: First perform the steps which connects the new database to the application using the Entity Framework (follow the steps described in the notes), and then add controllers for each table to the application (follow the steps described in the notes). 4. Now publish the web service to Azure. Again, do this by following the steps in the notes. Once the service is published, you can test it a bit by trying to call the API in a browser. 5. Open **Program.cs** in the **ConsoleClient** project. Change the value of the variable **serverURL** in **Main**, to refer to the URL of your own web service. 6. Run the **ConsoleClient** project (Note that you may be prompted by a virus checker to allow the application to make outgoing calls) . It contains a simple test of the web service: The data from all four tables is loaded and printed on the screen. Note that it may take several seconds before data is returned by the web service, so be a little patient ☺. |

|  |  |
| --- | --- |
| **Exercise** | DAPE.9 |
| **Solutions** | MVVMStarterLibrary and MVVMStarterDemoB (**NOTE**: These solu­tions are found as separate repositories on GitHub). |
| **Purpose** | See a complete example of a UWP application using a RESTful web service as a data source |
| **Description** | The **MVVMStarterLibrary** solution contains a fairly large number of small, general-purpose class libraries for developing UWP applications based on the MVVM architecture. The **MVVMStarterDemoB** solution is an example of how the class libraries can be used, when using a web service as data source |
| **Steps** | 1. Download the **MVVMStarterLibrary** solution, and rebuild it. This may involve restoring some NuGet packages for the solution. Do not proceed until the solution can rebuild entirely. 2. Download the **MVVMStarterDemoB** solution, and try to rebuild it. This will probably result in several compilation errors. In the **Solution Explorer**, open the **References** node. For each “broken” reference, re-establish it by removing the reference, and browse to the .dll file on your own computer (see the **MVVMStarter Class Library Guide** for details). Do not proceed until all references are re-established, and the entire solu­tion can compile without errors. 3. In the **MVVMStarterDemoB** solution, open the file **AppConfig.cs** in the folder **Configuration/App**. Change the variable **ServerURL** to refer to your own web service for the CarRetailDB on Azure, as defined in the previous exercise. If you did not create such a web service, you can leave the URL as it is, and hope there is a web service at the other end ☺. 4. Run the **MVVMStarterDemoB** application, choose **Load**, and see if the data from the Azure database is loaded, by opening the views related to each of the four data types (**Car**, **Customer**, **Employee** and **Sale**) 5. Try e.g. to delete a couple of data entries, and verify that the data is actu­ally deleted permanently by checking directly in the database (open the database in Visual Studio, navigate down to a specific table, right-click on the table, and choose **View Data**). |

|  |  |
| --- | --- |
| **Exercise** | MVVMStarter.1 |
| **Solutions** | MVVMExample01 |
| **Purpose** | Take initial steps in using the **MVVMStarter** class libraries for creating a UWP application based on the MVVM architecture |
| **Description** | Before starting, you must have downloaded and successfully rebuilt the **MVVMStarterLibrary** solution. This may involve restoring some NuGet packages for the solution |
| **Steps** | 1. Download the **MVVMExample01** solution, and try to rebuild it. If some references to the class libraries are broken, you should re-establish them. 2. Once the solution can be built, try to run it. The functionality is very limi­ted, since you can only browse a few hard-coded **Movie** objects. 3. Open the **Movie.cs** file (in the **Model** folder).    1. What indicates that the **Movie** class uses the class library?    2. What is the consequence of the fact that the **Movie** class also acts as a Domain View Model class (Hint: Bindings…) 4. Open the **MovieCatalog.cs** file (in the **Model** folder).    1. What indicates that the **MovieCatalog** class uses the class library?    2. The **MovieCatalog** acts as Data Context for the view where you can browse **Movie** objects. What **MovieCatalog** properties do you think the view needs to bind to, in order to implement this?    3. Try to highlight the word **InMemoryCatalog** and press **F12**. See what information you can obtain in this way. 5. Open the **MainPage.xaml** file.    1. The **ListView** control contains three data bindings. Why do these bindings work?    2. The **(Details)** sectioncontains two data bindings. Why do these bindings work? 6. If you have a bit of time left, see how much you need to change in order to add e.g. a **Country** property to the **Movie** class. This property should of course also be visible in the view when browsing **Movie** objects. |

|  |  |
| --- | --- |
| **Exercise** | MVVMStarter.2 |
| **Solutions** | MVVMExample02 |
| **Purpose** | Take further steps in using the **MVVMStarter** class libraries for creating a UWP application based on the MVVM architecture  (focus on implementation of a deletion functionality) |
| **Description** | Before starting, you must have downloaded and successfully rebuilt the **MVVMStarterLibrary** solution. This may involve restoring some NuGet packages for the solution |
| **Steps** | 1. Download the **MVVMExample02** solution, and try to rebuild it. If some references to the class libraries are broken, you should re-establish them. 2. Once the solution can be built, try to run it. Compared to the previous version, you can now delete a **Movie** object. 3. Open the **MovieCatalog.cs** file (in the **Model** folder). See if you can figure out what parts of the code that relate to the deletion functionality. 4. What would it take to separate the deletion functionality from the **Movie­Catalog** class, e.g. as a separate class called **MovieDeleteController**? 5. Open the **MainPage.xaml** file. Where in the code is the deletion function­ality tied to an element in the view? 6. Catalog classes are in general intended to manage domain data objects. How does that correspond to the actual responsibilities of the **Movie­Catalog** class? 7. If you have a bit of time left, see if you can move the deletion function­ality into a separate class, as suggested in step 4. |

|  |  |
| --- | --- |
| **Exercise** | MVVMStarter.3 |
| **Solutions** | MVVMExample03 |
| **Purpose** | Take further steps in using the **MVVMStarter** class libraries for creating a UWP application based on the MVVM architecture  (focus on introducing View Model classes) |
| **Description** | Before starting, you must have downloaded and successfully rebuilt the **MVVMStarterLibrary** solution. This may involve restoring some NuGet packages for the solution |
| **Steps** | 1. Download the **MVVMExample03** solution, and try to rebuild it. If some references to the class libraries are broken, you should re-establish them. 2. Once the solution can be built, try to run it. With regards to functionality, nothing has changed compared to the previous version. 3. Open the **MovieCatalog.cs** file (in the **Model** folder). Does the class defini­tion look simpler or more complex than in the previous version? Can you see any functionality relating to views, or presentation in general? 4. Open the **MovieViewModel.cs** file (in the **Transformation** folder). It looks almost identical to the **Movie** domain class. What are the differences specifically? What is the purpose of these differences (Hint: Take a look at the **MovieViewModelFactory** as well)? 5. Open the **ViewModel** folder. Can you remember the specific purposes of each of the four classes? If not, have a closer look at them. 6. Open the **MasterDetailsViewModel.cs** file. How is the deletion function­ality implemented? Where in the code are properties/methods from the base class used? 7. The **MasterDetailsViewModel** constructor takes a new **MovieCatalog** object as one of its parameters. Why could that be problematic? 8. Open the **MainPage.xaml** file. What are the most important changes compared to the **MainPage.xaml** file from the previous version? 9. If you have a bit of time left, think about what it would take to implement an insertion functionality in this version. What classes would primarily need to be extended? |

|  |  |
| --- | --- |
| **Exercise** | MVVMStarter.4 |
| **Solutions** | MVVMExample04 |
| **Purpose** | Take further steps in using the **MVVMStarter** class libraries for creating a UWP application based on the MVVM architecture  (focus on introducing full CRUD functionality) |
| **Description** | Before starting, you must have downloaded and successfully rebuilt the **MVVMStarterLibrary** solution. This may involve restoring some NuGet packages for the solution |
| **Steps** | 1. Download the **MVVMExample04** solution, and try to rebuild it. If some references to the class libraries are broken, you should re-establish them. 2. Once the solution can be built, try to run it. With regards to functionality, you should hopefully experience that you can now create, read, update and delete movies, i.e. full CRUD functionality (NOTE: There is NO error handling in the project ☺). 3. Compare the structure of this project with the structure of the project for the previous version. Have any files been added/deleted? 4. Take a look at the **MovieCatalog** class. How has it changed from before? What problem are we solving by turning the class into a Singleton? If you don’t know what a Singleton is, try to Google *“Singleton design pattern”* 5. Take a look at the **MasterDetailsViewModel** class. Does the class defini­tion look simpler or more complex than in the previous version? What is the most significant change? Where do you suppose the implementation of CRUD functionality is “hidden”? 6. The last two parameters to the **MasterDetailsViewModel** constructor are two lists of strings. See if you can figure out what the purpose of these lists are. What happens if you move an element from one list to the other? (Hint: Run the application, and try to update an existing **Movie** object). 7. Open the **MainPage.xaml** file. What are the most important changes compared to the **MainPage.xaml** file from the previous version? 8. If you have a bit of time left, think about what it would take to implement the same functionality for another domain class, say a **Studio** class. |

|  |  |
| --- | --- |
| **Exercise** | MVVMStarter.5 |
| **Solutions** | MVVMExample05 |
| **Purpose** | Take further steps in using the **MVVMStarter** class libraries for creating a UWP application based on the MVVM architecture  (focus on structure of a more complex application) |
| **Description** | Before starting, you must have downloaded and successfully rebuilt the **MVVMStarterLibrary** solution. This may involve restoring some NuGet packages for the solution |
| **Steps** | 1. Download the **MVVMExample05** solution, and try to rebuild it. If some references to the class libraries are broken, you should re-establish them. 2. Once the solution can be built, try to run it. We are now a bit closer to a “real” application, and some extra elements have been added:    1. One more domain class: **Studio**    2. A main view for navigation to domain-specific views, and for invoking general functionality like Load, Save and Quit.    3. Data persistence; data is saved to a text file when choosing Save, and loaded from the file when choosing Load. 3. Take a bit of time to explore the project structure. How much of it can you relate to the structure of the previous versions? 4. Open the **AppConfig.cs** file (in the **Configuration**/**App** folder). Even if you do not understand all details, see if you can get an idea about what pur­poses this class serves. 5. Open the **AppViewModel.cs** file (in the **ViewModels**/**App** folder). What kind of class does this class look like? See if you can figure out where the class is used (Hint: take a look at **MainPage.xaml**). 6. Open the **MainPage.xaml** file, and also the **MainPage.xaml.cs** file. See if you can get an idea about the purpose and workings of this view. 7. (NB: This question is a bit hard, and will require some exploration of both the project and the **MVVMStarter** class libraries) This version of the applica­tion uses file persistency. Exactly where in the code do we “lock” the application to use file persistency? How much would it require to switch to a different specific data source, e.g. a web service? |