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

|  |
| --- |
| By Per Laursen  08-03-2018 |

Content

[MP.1 2](#_Toc508216791)

[MP.2 3](#_Toc508216792)

[MP.3 4](#_Toc508216793)

[MP.4 5](#_Toc508216794)

|  |  |
| --- | --- |
| **Title** | MP.1 |
| **Project** | WesternStrike |
| **Purpose** | Clean up a larger project, by using inheritance, type parameterisation and other means to make the project more DRY. |
| **Description** | The **WesternStrike** project is – yet another – example of a role-playing game, this time with indians against pale-faces. You should imagine that the project has been completed by inexperienced software deve­lopers, who were also working against a tight deadline. The conse­quen­ces are that the project is poorly structured and documented.  Your company has now taken over the project, and wish to improve the structure of the project, since they plan to extend the game considerably in the future. |
| **Tasks** | The task is only defined in broad terms: clean up the code, by elimina­ting duplicate code in various ways (e.g. by inheritance and type para­meterisation).  The first steps should probably be to obtain an understanding of the structure and functionality of the project. After that, you can start by focusing on a single aspect of the game (e.g. weapons) and then pro­ceed to another aspect.  Keep in mind that the game is poorly documented, and you may at some points need to make assumptions about the gameplay, based only on what you can observe in the code. |

|  |  |
| --- | --- |
| **Title** | MP.2 |
| **Project** | SimpleCraft |
| **Purpose** | Work with an example of an event-driven game project |
| **Description** | The **SimpleCraft** project is in its current form a sort of simulator for certain kinds of role-playing game magic. This magic has the below characteristics:   * The magic is applied in the form of a “spell cast”. A spell cast consists of a spell with certain properties, a center (see below), and a time at which the spell is casted. * A spell cast affects all characters within a certain area (e.g. within a cer­tain radius from the center of the spell cast). * It has a duration, e.g. 10 seconds. * Affected characters receive a certain amount of damage per second, e.g. 5 damage points per second.   The game setup is very simple. The world is defined as a 10x10 grid, and each character can be positioned at a grid, defined by an (x,y) coordinate, e.g. (6,4). Currently, a character cannot be moved, which is obviously not very realistic…  **SimpleCraft** has a console-based GUI. After starting the application, you can cast spells by typing in a code consisting of one letter follow­ed by two digits. A code could e.g. be **p68**, which should be read as “cast the spell with code **p**, at the position (**6,8**)”. You can then observe the effect of the spell cast on the characters currently in the world. |
| **Tasks** | The first task is to obtain an overview of the project. It contains a fairly large number of classes, and is not documented very well... A very important aspect of the application is that events are used to manage the interaction between spell casts and characters.  Once you feel you have a reasonable overview, you can experiment in various ways, like adding more spells and characters, adding more advanced types of damage calculation, improving the GUI, being able to move characters, etc.. You could also try to improve the structure of the application itself. |

|  |  |
| --- | --- |
| **Title** | MP.3 |
| **Project** | FinanceSimulator |
| **Purpose** | Work with an event-driven application without a well-defined speci­fi­cation and initial structure |
| **Description** | The **FinanceSimulator** project contains various classes related to simulation of a financial system. The central class in the existing project is **StockSimulationModel**, which contains functionality for setting up a simulation of stock prices. A client class can then subscribe to a continuous “stream” of stock quotes (a “quote” is the current price of e.g. a stock). Initially, the project only contains a very small test example of a stock quote client, found in **MainPage.xaml.cs**. |
| **Tasks** | The project is (intentionally) fairly weakly structured, and it is up to you to try to organise the initial content of the project. It is also up to you to come up with ideas for what to do with such a stream of quotes. Examples of ideas could be:   * A graphical overview of the current (and maybe also historical) prices of all stocks being simulated. * A stock trader class, which can react to certain conditions by selling or buying certain stocks. * A stock trading GUI, where you simulate that the user can buy or sell stocks. This could also include definition of classes for representing a stock portfolio and an account with funds for stock trading. |

|  |  |
| --- | --- |
| **Title** | MP.4 |
| **Project** | CryptoCrowns |
| **Purpose** | Try to speed up a needle-in-a-haystack calculation by using tasks |
| **Description** | The **CryptoCrowns** project is in itself quite small, and only contains the (small) class **MinerSequential**. The project does however use the library **CryptoCrownLib**, found in the folder **CryptoLibrary**.  The setting of the project is as follows: CryptoCrowns is yet another crypto-curren­cy, which can be “mined” by using the **CryptoCrownLib** library. The library exposes the class **MinerBase**, from which a custom miner class can be created by using inheritance. This is more specifically done by overriding the virtual method **Mine­SingleCryptoCrown**, which should contain the logic for mining a single Crypto­Crown (an example of this is found in **MinerSequential**).  Mining of a single Crypto­Crown boils down to guessing the key (which is of type **long**, i.e. a 64-bit integer) for the next valid Crypto­Crown. The value of the key for the next Crypto­Crown is a number between 0 and the current value of the property **CurrentMaxKey**. For the first Crypto­Crown, **CurrentMaxKey** is set to 1,000,000. Since it should become harder and harder to mine subsequent Crypto­Crown, the value of **CurrentMaxKey** is increased by 10 % for each subsequent Crypto­Crown. The effort needed for guessing the next key will thus also increase by 10 %.  The validity of a candidate key can be tested by calling the method **AttemptTo­MineSingleCryptoCrown**. This method will return true if the key is valid, otherwise false. The algorithm for finding the next CryptoCrown is therefore quite simple: try out all numbers from 0 to **CurrentMaxKey**, until the next valid key is hit. When a valid key is hit, the new CryptoCrown is added to the miner’s “wallet” (which is hidden inside the **CryptoCrownLib** library). A message will be printed on the screen whenever a new CryptoCrown is found. |
| **Tasks** | The main task is fairly simple to formulate: use a Task-based approach to speed up the discovery of new valid keys. You can implement your own algorithm in the class **MinerWithTasks**, and then compare it with the sequential approach simply by running the application. If you feel more adventurous, you could try to exploit a potential weakness in the system (hint: the sequence of valid keys is not random…) |