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

|  |
| --- |
| By Per Laursen  07-04-2018 |

Content

[MP.0 2](#_Toc510884731)

[MP.1 5](#_Toc510884732)

[MP.2 6](#_Toc510884733)

[MP.3 7](#_Toc510884734)

[MP.4 8](#_Toc510884735)

[MP.5 9](#_Toc510884736)

[MP.6 10](#_Toc510884737)

|  |  |
| --- | --- |
| **Title** | MP.0 |
| **Project** | Yatzy |
| **Purpose** | Understand and extend a C# application, intended to implement the dice game Yatzy. |
| **Description** | Yatzy is a dice game with fairly simple elements (a set of six dice with six sides each), and rules. The **Yatzy** project contains the first steps towards implementing the game in C#.  Initially, the **Yatzy** project contains these classes (further details on each class can be found in the source code comments):   * **Die**: Represents a single die. * **DiceCup**: Represents a set of dice. * **IDiceEvaluator**: Defines a (very small) interface for a ”dice eva­lu­ator”. A dice evaluator is a class which can evaluate a set of dice, according to one of the point-scoring combinations in the Yatzy game, like ”Three of a Kind”. * **ChanceEvaluator** and **OnePairEvaluator** (located in the folder **Evaluators**): Two implementations of **IDiceEvaluator**, which evaluate a set of dice according to the combinations “Chance” and “One Pair”. * **GameManager**: Class which manages the Yatzy gameplay.   All classes are as such functional, but the gameplay is very limited ini­tially. If the application is executed in its initial state, it will just per­form 10 different “rolls” of the dice in the dice cup, and evaluate them against the two implemented evaluators (look in **Program.cs** to see how the game is more specifically started).  Section A below contains a number of questions to the project, which you can use as a self-test to see how well you understand C# code. Following in Section B are a number of increasingly difficult suggest­ions for how to extend the application. |

|  |  |
| --- | --- |
| **Title** | MP.0 |
| **Project** | Yatzy |
| **Section A** | Open the **Die** class:   * How many **instance fields** does the class contain? What is the **type** of each instance field? * How many **methods** does the class contain? How many **properties**? What are the differences between methods and properties? * What happens in the **constructor**? What is the purpose of a constructor?   Open the **DiceCup** class:   * What is the type of the instance field **\_dice**? Who decides how many **Die** objects we add to the list? * What is the purpose of the method **ToString**? What does the keyword **override** indicate? * See if you can explain to yourself what happens in the properties **Dice­Face­Values** and **DiceCountByFaceValues**.   Open the **IDiceEvaluator** class:   * What are the main differences between an **interface** definition and a **class** definition? * Can you create an **object** of type **IDiceEvaluator**? * Can you declare a **variable** of type **IDiceEvaluator**? * Why are interfaces useful?   Open the classes **ChanceEvaluator** and **OnePairEvaluator**:   * How are these two classes related to the other classes/interfaces in the project? How is their coupling to the classes **Die** and **DiceCup**? * See if you can explain to yourself what happens in the **Evaluate** method in **ChanceEvaluator**. * See if you can explain to yourself what happens in the **Evaluate** method in **OnePairEvaluator**.   Open the **GameManager** class:   * What is the type of **\_diceEvaluators**? What is the purpose of this instance field? * See if you can explain to yourself what happens in the **GameManager** constructor. * See if you can explain to yourself what happens in the method **Run**. |

|  |  |
| --- | --- |
| **Title** | MP.0 |
| **Project** | Yatzy |
| **Section B** | Implement additional dice evaluator classes, i.e. classes which:   * Implement the interface **IDiceEvaluator**. * Implements the logic for evaluating a set of dice against one of the scoring combinations in Yatzy, e.g. “Two Pairs”.   The choice of specific evaluators to implement is up to you. Remember to test the evaluators you choose to implement!  An important part of playing Yatzy is – of course – to keep track of the points scored by each player. Think about which classes it could make sense to define to implement this, and how they would be related. Examples could be:   * **ScoreBoardEntry**: Class representing a single entry on a score boad, e.g. the points scored for the combination “Three of a Kind” * **ScoreBoard**: Class for representing all score board entries for one player. * **GameManager**: Extended with a collection of **ScoreBoard** objects; one for each participating player.   A player should be able to interact with the Yatzy application, at certain steps of the general gameplay. A player should be able to:   1. Roll the dice in the dice cup. 2. Select a set of dice to save. 3. (again) Roll the remaining dice in the dice cup. 4. (again) Select a set of dice to save. 5. (again) Roll the remaining dice in the dice cup. 6. Choose the combination by which the final set of dice values should be scored (including the possibility to score zero, if the dice no not match any unscored combinations).   Think about how this algorithm might be implemented in the **GameManager** class. How should the player more specifically interact with the application? Try to create some draft proposals for a GUI for the application. If you have learned about how to actually create a GUI in an application (by using XAML, Data Binding, etc.), feel free to try to implement some of your suggestion as well. |

|  |  |
| --- | --- |
| **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…) |

|  |  |
| --- | --- |
| **Title** | MP.5 |
| **Project** | PvPSimulator |
| **Purpose** | See an example of a UWP application which has a different nature than the usual CRUD-type applications, but still follows the MVVM layering model. |
| **Description** | The **PvPSimulator** project is closely related to the **RolePlayV…** series of console projects, but comes as a UWP application with a (simple) GUI. The application can simu­late a simple Player-vs-Player (PvP) battle, where individuals of a few character classes can be pitted against each other.  The project contains a fair number of classes and interfaces, but the four most important (and also largest) classes are:   * **Player** (in folder **Players**): Models a participant in a PvP battle. The most interesting elements are probably the methods **DealDamage** and **Receive­Damage**. The rest of the class is mostly simple properties. * **BattleModel** (in folder **Model**): Manages the PvP simulation. Refers to two player objects, and maintains battle state data. The class also contains an event, which is invoked whenever the battle state data changes. * **PlayerViewModel** (in folder **ViewModel**): Exposes various properties rela­ting to the state of a single player object. The properties are used for Data Binding in the main view, are all of a simple type, and are all read-only. * **BattleViewModel** (in folder **ViewModel**): Exposes various properties rela­ting to the entire PvP simulation. The properties are used for Data Binding in the main view. Several of these properties are of a **Dictionary** type. The view model has a reference to a **BattleModel** object, and subscribes to changes in that object. |
| **Tasks** | The application is intended to be a demo of an (atypical) UWP application, so there are no specific tasks to perform, other than studying the structural details of the appli­cation. An agenda for this could be:   * Get an overview of the application. What do the individual folders contain? How are they related? * What classes and interfaces can be said to belong to the **Model** layer? * What classes and interfaces can be said to belong to the **ViewModel** layer? * How are Data Bindings implemented in the view (**MainPage.xaml**)? How are bindings to properties of a **Dictionary** type implemented? What advantages do you gain by such bindings? * The main view contains a lot of nested structures. How is that achieved in XAML? Where are properties like margins, border colors, etc. specified? |

|  |  |
| --- | --- |
| **Title** | MP.6 |
| **Project** | JustPullTheTrigger |
| **Purpose** | Unravel the story behind a successful (?) agent mission |
| **Description** | A story about an agent mission is told partly as back-story, partly in the form of the project **JustPullTheTrigger**. The project is intended to illustrate use of interfaces and a couple of Design Patterns. However, the project is also set up in a way that reflects that there is something suspicious about the mission… |
| **Tasks** | Read the back story (follows after the exercise text), and run the project when indi­cated in the text. Hopefully, you can see that something is not quite right. See if you can work out where something suspicious happens, try to fix it, and see how the mission now plays out. The final line printed by the application will indicate if you have found and fixed the suspicious parts ☺.  Once you have fixed the mission, consider the below questions:   * How much information can the agent retrieve about his weapon? * Who can load ammunition into the agent’s weapon? * Why is the **Government** class a Singleton? * How much does the government know about the mission target? * Who knows that the target is a Russian Gangster? * What information can a target retrieve about ammunition he was hit by? * What classes implement the **IParticipant** interface? * Why is the method **Report** in **ParticipantBase** declared as **virtual**? Does anybody override it? * Where is the Proxy pattern used? Is it used in the traditional way? If no, what is different from how you usually apply the pattern? * Where is the Factory pattern used? Is it used in the traditional way? If no, what is different from how you usually apply the pattern? |

|  |
| --- |
| **MP.6 - Backstory** |
| MISSION BRIEFING  “Enter!”  The Agent hesitated for a second. He was always slightly unnerved when entering the Director’s office. It was a bit paradoxical, really. After many years in the service, he took down high-profile targets without second thought, even if it came down to a standoff, where he could look his target straight in the eyes. Still, this dull, grey-haired administrator could hurt him more than any AK47-wielding mad-eyed terrorist ever could, just with few strokes on a keyboard… He pulled himself together, and entered the Director’s office.  The Director sat behind his desk, wearing his standard thin smile. There were no other chairs in sight. The Agent smiled to himself; the Director had always enjoyed such petty power plays. “So, Agent, you’re up for a new mission. About time too, eh?”, the Director said, just a bit too exitedly. The Agent nodded slightly. “Yeah, it’s been a couple of months now…”, he said. “Yeah, since that thing in Caracas”, the Director replied. “Didn’t go down too well, that one…”, he added with a slight frown. The Agent frowned back. “The target was eliminated, yes?”, he replied. “Yeah, two hours after being hospitalised!”, the Director replied. “Pure luck that she didn’t regain consciousness. Who knows what she might have spilled!”, the Director continued, with a somewhat angry tone in his voice. The Agent admitted to himself that the Director had a point. That target had proven to be remarkably resillient. He was down to his last bullet, before she finally dropped. And yes, she was only down, not dead…  The Director switched back to his thin smile. “Well, this new one should be a walk in the park. Gorkij Park, perhaps, heh heh…”. The Director had no talent for humor. He pulled out a standard-issue mission suitcase from behind the desk, and placed it in front of him. The Agent walked up to the desk, and placed his thumb on the fingerprint scanner. *They really need to upgrade these suit­cases soon*, he thought. The suitcase snapped open. Inside was – not surprisingly – two items: a standard grey-metal weapon box, and a small, agency-grade USB memory drive, which would no doubt contain information on the target. The Agent looked back at the Director, who was still wearing his thin smile, showing his yellow-stained teeth. *God, I understand why she left him*, he thought. The Director had not talked much about it, and if it ever came up, he usually brushed it aside with a masculine Julia-was-no-good-in-bed-anyway comment. Well, the Agent knew better…  The Agent touched the memory drive with a single finger. “So, Russian target…?”, he asked. The Director broadened his smile a bit, in a sarcastic way. “Well, how would I know? But the name of the target is Terpentin, so it would be a good guess, eh?”. “Hmm…”, the Agent said, “the intel’s good?”. The Director replaced his smile with an angry look, as if the Agent was a stupid schoolboy who still couldn’t get his French verbs right. “Jesus, Agent, you know how this works! Some secret gover­ment comitee gets information from our operatives, and select targets for, uh, processing. They hand information about the target down to us here at HQ, and we take it from there. Just as it has been done since God knows when!”. The Director calmed down a bit. “I’m guessing this intel came from one of the Russian operatives, and they tend to be top-notch. So, yeah, the intel’s pro­bably rock-solid.”.  The Agent contemplated a bit. The russian operatives were indeed rather famous for their consis­tent performance, but information is never better than the media by which it is transferred. The Agent had never been impressed by the Government (and HQ, for that matter) IT departments. Mostly a bunch of ex-Black Hat twenty-somethings, all balancing on the edge of autism. Sure, they knew their network and encryption protocols by heart, but could not really be bothered with such routine tasks as system backup, email security and other “boring” stuff. There were rumors about the best-and-brightest IT guys being recruited to covert departments, that had their own agendas and were not entirely under government control. Of course, nothing of any substance was ever produced to prove these rumors…  “Hey, Agent, you with us?”, the Director called out. The Agent snapped back. “Yeah, sure”, he repli­ed. He closed the suitcase. “Okay, so, I guess that’s it”, he said. He took the suitcase, and started to turn around. “Yeah, that’s it, Agent”, the Director replied, “Just get over there, pull the trigger, and head on home…and, Agent…?”, he continued. The Agent stopped, and looked back at the Director over his shoulder. “Don’t run out of bullets this time…”, the Director said. They locked eyes for a brief moment. “See you at the debriefing, Agent”, he finally said, looking back at some papers on his desk. The Agent left the office, closing the door silently behind him.  *What was that about!?*, the Agent thought. Why did the Director feel the need to dip the Agent’s nose in that Caracas shit again? And was there something about his standard thin smile that was different at that final exchange? The Agent did have a nagging feeling that the Director grew stead­ily more hostile towards him. *Did he know…?* The Agent and Julia had been extraordinarily careful to keep their relation a secret, but how do you keep secrets from a husband who also happens to be the Director of a government cloak-and-dagger agency…? The Agent decided to put those thoughts on hold, using his well-proven, agent-grade mental discipline. *Now, let’s focus on the mission at hand…*  **[Run the JustPullTheTrigger project, and see how the mission plays out…]**  MISSION DEBRIEFING  “Enter!”  The Agent hesitated for a second. For no reason, actually. The mission went smoothly, almost too smoothly. The target dropped after the first bullet (the Agent did wonder what the bullets in the weapon contained. Small, high-powered grenades!?), and yelled out a pathetic death cry before dropping to the ground like a sack of corn. Job done, get out, get home. A walk in the park indeed!  The Director sat behind his desk, wearing his standard thin smile, but he also seemed unusually tense. A chair was placed in front of the Director’s desk, and he motioned the Agent to come and sit down. “Well, quite the success, this mission, eh?”, the Director started, with a strange tension in his voice. “One headshot, and that was it, right?”. The Agent shrugged. “I’m not even sure it was a headshot, actually”, the Agent replied. “Sure it was!”, the Director said abruptly. “A job well done! I’m sure the world is a better place now, with one less megalomaniac Russian in it, ha ha…”, he con­tinued, his face expression completely out of sync with his upbeat tone of voice.  *What the fuck is wrong with him…?*, the Agent thought. The Director flipped aimlessly through some papers on his desk. “Well, uh, I’m putting you on a well-deserved two-week holiday, starting as of today. When you come back, I’ll probably have another mission ready for our top marksman here, eh”. The Director got out of his chair, went around the desk, and gave the Agent a couple of hard pats on the back, trying to deliver it in the best we’re-best-buddies style, but it came out awkward… The Director mo­tioned the Agent to get out of the chair, his hand still on the Agent’s shoulder. He led the Agent to the office door. “Look, I want you to enjoy yourself and relax for a couple of weeks. Get any ten­sions out of the body and mind, right?”. He fumbled around in his trouser pocket, and produced a small business card. He handed it to the agent. “It’s a masseuse I use myself, she’s really skilled. She can work those tensions out of the body…and mind, if you know what I mean”. The Director had an almost boyish expression on his face, as if he had just found a stack of porn magazines under his dad’s bed. He gave the Agent a last pat on the shoulder, and almost pushed him out of the door. It closed quickly behind him.  *What was that about!?*, the Agent thought. Why this sudden over-the-top camaraderie? It’s not like it was the first time he came back after a successful mission. Something was definitely off… The Agent sighed. The prospect of a couple of kick-back weeks was not so bad, all things considered. He made a mental note about investigating the circumstances of the mission after his vacation. *Well, let’s make the best of it*, he thought. He looked at the business card, and pulled out his phone. |
| <Route-To: [akn@b-ops.(icept).govdark.(icept).gov](mailto:akn@b-ops.(icept).govdark.(icept).gov)>  <Route-From: [ssu@b-ops.(icept).govdark.(icept).ru](mailto:ssu@b-ops.(icept).govdark.(icept).ru)>  <Time: 0244Z23FEB2019>  <Mail-Server: BOPS-INTERCEPTOR.govdark>  <Encryption: 2048-qbit>  <Decrypt-Token: shfgaeksncvnFFA112NXV45kskasfaf567kb5bb2, checksum=OK, range=6HRS>  <MSG>  TERPENTIN SAFE -> DAMASCUS < 36 HRS  PREP SURGICAL TEAM  IRL PROXY DELETED  RESET MAIL ROUTING < 12 HRS  </MSG> |