Refactors

Friday, March 13, 2015 9:42 PM

1) Removed Keyboard class and replaced it with the PromptUser with a Scanner object. All user input is run through this object and regexs are used to check the input data. The old outdated keyboard class worked fine when the project was created but now we have access to scanner and can use the scanner class that comes with many of the methods we need in the API. Using an API is better than using a public class, public vs published. BY moving all of the prompts to the user out into its own class allows for only one class to care about using system.in. This helps with the single responsibility principle. I could have made this class a singleton since I only want a single object at any time, but I ended up using the class in the hero class for asking the user to rename the character. I could have passed this in as a parameter but decided against it.

2) Moved related classes to their own packages to make it more logically organized. Not sure if this counts as a refactor. Since it does help with the structure of the project it could be considered a refactor even though it does not affect the code. I find it slightly more revealing to group classes together and you can change the modifier to package to limit who can see the underlying set methods, etc. Applying the only talk to those you that you know and no one should have access to change the characters fields but a character.

3) Added package sets to the DungeonCharacter class to allow for subclasses to easily set each field without sending massive parameters. This helps make some of the magic numbers by using intent revealing set method names. The numbers are still a little magic and should be made final when needed or passed in using a factory. Making it easier to create more classes on the fly. Half the battle with writing code is making it intent revealing. There are a multitude of ways we can do this. I decided to stick with how the code was originally written and added setters to keep a lot of code that would have been duplicated throughout each class had I made DungeonCharacter an interface.

4) Changed the dvc for the Hero class to only require the hero's chanceToBlock since all other parameters are handled in setters. This also helps remove the magic numbers in the hero creation and can also be considered the same as the next refactor since they cascade down the inheritance hierarchy.

5) Removed all the excess parameters being passed from the hero classes to DungeonCharacter and replaced them with calls to super setters. As stated in the previous refactor this can be considered the same as 4. I separated them into separate numbers to help document changes as I made them and since they are two separate classes they might be counted as two but I see why they shouldn't.

6) Changed the dvc for the Monster class to only require chanceToHeal, minHeal, and maxHeal. Super class handles all other parms. Same with the hero class I removed a lot of the magic numbers in the code and replaced them with more revealing setters.

7) Removed all the excess parameters passed from each monster class and replaced them with super class setters. Should be counted the same as 6 and wrote it down to help document all the changes I made to the original code.

8) Created a builder for Hero Characters. This class just needs the type of hero you want and it will create it and return it. This allows the driver class to only have to care that it has a hero and not what type. By using a generic class reference in the main driver class makes the code more flexible to change when more characters are added in the future. This is a simple factory to handle the logic in creating a hero object.

9) Created a builder for Monster Characters. When you create a new instance of MonsterBuilder, a random number is generated, this number is used as the index of a String [] of all the currently implemented monsters. When the buildMonster method is called the apporiate monster object is returned. This is another simple factory class and could be combined with the hero factory. I am working off the idea that a character factory should only care about creating a specific character type and leaves it open to expansion in the future. But it does go against the "YAGNI" principle. As the code sits there isn't a need for two separate factories and can be refactored down to one. Yet if they were placed into one factory there may be some logic complexity in if statements to go through heroes and random monsters. I personally like only calling a buildHero or buildMonster method and having them handle each individually.

10) Abstracted out the battle method into its own class. The DVC for the object requires a hero and monster, then sets these as global variables. This allows the driver class to not have to worry about handling the battle logic. It just tells the battle object to start the battle. Single responsibility of the battle falls on this object and the driver class can just tell the object to do its thing for battles. A single object also allows for another dev to quickly find the method they need to modify quickly. Smaller classes equal faster turn around.

11) Removed all used imports in all classes. If it is not used, there is no need to have it. It can add extra overhead when running a program as well. Since it imports everything when the code is compiled.

12) Removed some excess commenting. There were some comments used to separate methods in classes. I found this to be not needed and cluttered the code a little. Not much of an improvement but it is a code smell. I think I may have also removed some java doc without noticing what it was prior to deletion.

13) Changed the visibility of the DungeonCharacter's fields to private since it is now using setters for all parameters. This helps to lock down the code for future devs. If a field is set to public but really should be accessed through a getter to prevent setting the value when trying to read a value, the field should be private.

14) Replaced plain calls to super fields with get methods to keep all fields private. This helps make the code more revealing and prevents fields from changing on the fly when just reading the data.

15) Changed the play again to a boolean instead of checking for a specific character in the Dungeon class. A boolean requires less complex logic in a loop or if statement. Since both are expecting a boolean anyway, it seems more appropriate for it to hold a boolean. This extracts the logic out to another class/method.

16) Changed the variable for when the user wants to bail out to a boolean. Extracting out the logic to another method that only cares about the logic of bailing out.

17) Used regexs when scrubbing using input data. More of a security patch than a refactor but it does simplify the logic complexity needed. By creating a regex method I also made it easier to scrub future data faster. This also allows a single method to handle all of the equals logic in the game. Single method with many uses.

As I stated in some of the refactors I did, there are still plenty of other ways I can refactor this code. I could clean up the dungeonCharacter class to remove some of the calls to super from subclasses and into an interface that all other classes implement. This would give a baseline guarantee of methods. I didn't do this since each class extends on DungeonCharacter and leaves all of the classes smaller. I could convert the PromptUser class into a singleton since there should only be one object that cares about prompting the user and passing the object to other classes that need it. This would make closing the

Scanner object easier since it would only need to be called once. But since the class only cares about prompting the user for information, modifications to the code won't harm the rest of the classes any more than deleting a vital method would.