Conforming to the Open-Closed Principle

The Open-Closed Principle states that code should be written in such a way that it minimises the amount of change when adding extra functionality to the program. This means that software entities such as classes should be open for extension but closed for modification.

Stage 2 of my program splits the default package up into 2 separate layers, an application and a presentation layer. The application layer is responsible for what happens when the user interacts with the user interface (UI). Whereas, the presentation layer is responsible for the definition of the look-and-feel of the application and the Graphical User Interface (GUI) elements on screen.

The presentation layer also contains a Java Interface and a class containing a 'Factory Method', alongside the UI Class. The interface is responsible for eliminating the 'tight-coupling' relationship between the layers and instead creating a 'loose-coupling'. However, the factory class is what allows confirmation of the Open-Closed Principle by returning Interface type Objects and invoking abstract polymorphic methods from the IAnimal Interface. This allows the software developer to easily add new types of predators into the program without having to edit the code in the interface or abstract Animal, Predator or Prey classes. A new predator class and an extension to the if-else statement in the PredatorFactory is all that is required.

Pros & Cons of My Program

One good aspect of my program is the use of abstract classes within the application layer. I have created a parent abstract class called Animal which contains all the common attributes and methods for every type of animal. I then have 2 child abstract classes, Predator and Prey. These 2 extend the Animal class and each contain attributes and methods specific to Predators and Prey respectively. This drastically minimises the amount of duplication of code and therefore redundancy in the program as the concrete classes simply inherit the common attributes and methods from the abstract classes.

Another good aspect of my program is the use of the IAnimal Interface that inverts the dependency between the application layers' abstract classes and the presentation layers' user interface class. This also conforms to the Dependency Inversion Principle (DIP). As mentioned in the section above, the program is therefore open for extension and closed for modification which ultimately reduces the amount of work that must be done to add extra functionality to the program.

One bad aspect of my program would be the nature in which the pet-prey pairs are designed. When re-working the code for stage 2, I had to move all the code for the prey into their respective predator classes, shifting the dependency on the predator classes themselves. I feel that separate interfaces and factory classes could have been used for predators and prey to break that relationship and ensure that the second factory method is used to return prey objects and invoke relevant methods through its own interface.

References

OODesign, n.d. *Open Close Principle.* [Online]
Available at: http://www.oodesign.com/open-close-principle.html
[Accessed 05 03 2017].

SpringFrameworkGuru, n.d. *Open Closed Principle.* [Online]
Available at: https://springframework.guru/principles-of-object-oriented-design/open-closed-principle/
[Accessed 05 03 2017].