Abstract factory program

Program description

The program allows the user to select an animal factory type (land/sea animals) and fetch either a docile or ferocious animal. Animals have 2 methods – to return their type (breed) and to produce sounds. All actions are logged in the ListBox at the bottom of the form.

Object description

1. AnimalFactory – main factory used to encapsulate factories for fetching different kinds of animals

* currentFactory – stores the *AnimalFactory* subclass currently in use
* CurrentFactory – read-only property to access *currentFactory*
* getDocileAnimal() – encapsulates the *getDocileAnimal()* method of the *AnimalFactory* subclass stored in *currentFactory*
* getFerociousAnimal() – encapsulates the *getFerociousAnimal()* method of the *AnimalFactory* subclass stored in *currentFactory*
* setAnimalFactory(String factoryType) – creates a new *currentFactory* based on the *factoryType* string (*LandAnimalFactory*/*SeaAnimalFactory*

1. LandAnimalFactory – *AnimalFactory* subclass used to fetch land animals

* getDocileAnimal() – overrides the *AnimalFactory* method *getDocileAnimal*; returns an *Animal* instance of type *Dog*
* getFerociousAnimal() - overrides the *AnimalFactory* method *getFerociousAnimal*; returns an *Animal* instance of type *Cat*

1. SeaAnimalFactory - *AnimalFactory* subclass used to fetch sea animals

* getDocileAnimal() – overrides the *AnimalFactory* method *getDocileAnimal*; returns an *Animal* instance of type *Starfish*
* getFerociousAnimal() - overrides the *AnimalFactory* method *getFerociousAnimal*; returns an *Animal* instance of type *Shark*

1. Animal – interface used to define the methods that all animals in the program will implement

* returnType() – method will return a string that identifies the animal’s breed in all classes that implement the interface
* speak() - method will return a string corresponding to the sound the animal produces in all classes that implement the interface

1. Dog – implements the *Animal* interface; Created by the *LandAnimalFactory.getDocileAnimal()* method

* returnType() – returns a string identifying the breed of the animal (dog)
* speak() – returns a string identifying the sound the animal produces

1. Cat – implements the *Animal* interface; Created by the *LandAnimalFactory.getFerociousAnimal()* method

* returnType() – returns a string identifying the breed of the animal (cat)
* speak() – returns a string identifying the sound the animal produces

1. Starfish – implements the *Animal* interface; Created by the *SeaAnimalFactory.getDocileAnimal()* method

* returnType() – returns a string identifying the breed of the animal (starfish)
* speak() – returns a string identifying the sound the animal produces

1. Shark – implements the *Animal* interface; Created by the *SeaAnimalFactory.getFerociousAnimal()* method

* returnType() – returns a string identifying the breed of the animal (shark)
* speak() – returns a string identifying the sound the animal produces

1. Form1 – the form/client of the program, manages all activities related to the program’s operation

* animal – stores the most newly fetched *Animal* object
* animalFactoryMain – an instance of the *AnimalFactory* class; used to access the other 2 factories and fetch *Animal* objects;
* Form1() – form constructor; used to also initialize the *AnimalFactoryMain* instance
* radioButton1\_CheckedChanged() – if the radio button is checked, sets *AnimalFactoryMain.currentInstance* to *LandAnimalFactory* using the *setAnimalFactor()* method, nulls the current *animal* instance and displays the factory change in the ListBox in the form
* radioButton2\_CheckedChanged() – if the radio button is checked, sets *AnimalFactoryMain.currentInstance* to *SeaAnimalFactory* using the *setAnimalFactor()* method, nulls the current *animal* instance and displays the factory change in the ListBox in the form
* getDocile\_Click() – if there is an animal factory selected *(AnimalFactoryMain.CurrentInstance* is not *null*), fetches a new *Animal* object into the *animal* variable using the *AnimalFactoryMain.getDocileAnimal()* method and displays that a docile animal was fetched in the ListBox; displays a message instead if there’s no factory currently selected
* getFerocious\_Click() – if there is an animal factory selected *(AnimalFactoryMain.CurrentInstance* is not *null*), fetches a new *Animal* object into the *animal* variable using the *AnimalFactoryMain.getFerociousAnimal()* method and displays that a docile animal was fetched in the ListBox; displays a message instead if there’s no factory currently selected
* btnIdentify\_Click() – if there is a current *animal* instance, calls the *animal.returnType()* method and passes its returned string to the ListBox; displays a message instead if there’s no factory or animal currently selected;
* btnMakeSounds\_Click() – if there is a current *animal* instance, calls the *animal.speak()* method and passes its returned string to the ListBox; displays a message instead if there’s no factory or animal currently selected;

3 problems

Reusability – The program could, for example, be extended to also store animal data input by the user and store a list of *Animal* objects, so as to reuse it in an animal shelter or store

Maintainability – The pattern can result in unnecessary complexity, especially in cases where lots of small factories/objects are implemented, so maintainability could prove difficult

Extendibility – *Animal* and *AnimalFactory* objects can be seamlessly interchanged, so extending the program with more of the same objects would be easy. Adding more functionality would result in an exponential increase in the work needed though, especially since every existing factory/object would need to have the new functionality added to it, so in that case extendibility can prove difficult.