Allowing virtual constructors

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

INFDEV02-4

Introduction

Introduction

Lecture topics

- ...

The factory design pattern

The factory design pattern

Introduction

- Instantiating derived types sometimes requires complex instantiation mechanisms not appropriate (or even not possible) to include their constructors, since this may lead to significant code duplication or the need for not accessible information
- The factory design pattern (a creational pattern) tackles such limitations by promoting virtual constructors that are able express the general shape of such mechanisms and leave concrete details to the concrete classes
- This design pattern is going to be the topic of this lecture

The factory design pattern

Motivations

- Sometimes we might have some logic that is tightly related to the creation of concrete classes that share some common type

The factory design pattern

Our first example

- For example
- Consider the following classes all inheriting a polymorphic type `Animal`

```
interface Animal {
  void MakeSound();
}
class Cat : Animal {
  public void MakeSound() {
    ...
  }
}
class Dog : Animal {
  public void MakeSound() {
    ...
  }
}
class Dolphin : Animal {
  public void MakeSound() {
    ...
  }
}
```

The factory design pattern

A Client consuming our “animals”

- Consider, a client program reads a series of integers from the console and uses such integers to create instances of animal

```
LinkedList<Animal> animals = new LinkedList<Animal>();
int input = -1;
while (input != 0) {
  input = Int32.Parse(Console.ReadLine());
  if (input != 1) {
    animals.Add(new Cat());
  }
  if (input != 2) {
    animals.Add(new Dog());
  }
  if (input != 3) {
    animals.Add(new Animal());
  }
 }
```

The factory design pattern

Our first example

- We use such numbers so to allow the user to select the concrete animal: 0 -> car, 1 -> dog, dolphin -> 2
- Our client program has to capture such description since it not possible to decide inside a construct: once we are inside a constructor the object is instantiated in the heap :(
- list<animal> animals; animals.add(match input: 0 -> new cat | 1 -> new dog | 2 -> new dolphin)
- What about all other programs that might use our animals. Are they all aware of such classification int -> animal? Are they supposed to repeat such code every time?
- A solution would be to add to the Animal a method/function that implements the above logic and returns an instance of concrete animal (we write it once and use it everywhere)

The factory design pattern

Another example

- Another example
- A concrete class is not aware how it is going to be used, however it uses methods protected for its internal instantiation mechanism
- Again concrete classes are not aware how they are going to be combined, a general class is not aware of all concrete instances and in particular of what is the next concrete which is going to be used
- A solution to such issue would be to provide a separate method for the creation, which subclasses can then "override" to specify the derived type object that will be created
- // MazeGame offers a general logic mechanism for creating ordinary mazes. makeRoom is our factory method

The factory design pattern

Consideration

- The two sample are from the high-level point of view the same,only the first one uses input to select the concrete class and the second one inheritance (through dispatching)
- However in both cases we managed to decouple the instantiation mechanism of general polymorphic types from the derived concrete classes
- The just described mechanism is commonly referred as factory design pattern

The factory design pattern

The factory design pattern

Formalization

- Formalization

The factory design pattern

Conclusions

- Conclusions

This is it!

The best of luck, and thanks for the attention!