Allowing virtual constructors

The INFDEV team

Hogeschool Rotterdam
Rotterdam, Netherlands

INFDEV02-4

Introduction

Introduction

Lecture topics

- Abstract classes
- The necessity for constructors at interface level
- The factory design patter
- Conclusions

Abstract classes

Abstract classes

Lecture topics

- ...

The factory design pattern

The factory design pattern

Introduction

- Sometimes, we know which interface to instantiate, but not its concrete class
- Interfaces specify no constructors, external code is necessary to express such mechanism
- This leads to conditionals in client code to determine which concrete class to instantiate

- `switch classToInstantiate ...`
  - Hard to read
  - Repeated[a] wherever instantiation happens

[a]Error prone, hard to modify and maintain.

The factory design pattern

Introduction

- In particular, we will study the factory design pattern (a creational pattern) meant to tackles such issues by promoting virtual constructors
- This design pattern is going to be the topic of this lecture

The factory design pattern

Motivations

- We wish to standardize our application that works across different domains, so to provide a unique way for instantiating their entities
- Every entity may have a logic that is not shared with others, but they all share a 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

Consuming our “animals” issue with constructors

- Consider, a scenario where animals are selected and instantiate based on unique number read from the console
- Unfortunately, such logic cannot be encapsulated inside the constructors of our animals, since once we enter a constructor we have to returns its instance
- Imagine we are inside the constructor of `Dog` and the read number is 3, the system will anyways return an instance of `Dog` instead of `Dolphin`
- So we need the caller to encapsulate such mechanism

The factory design pattern

Consuming our "animals" from the caller

- Consider, a client program that 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

- What about all other programs that potentially might use our animals. Are they all aware of such classification (number to animal)?
- Are all programs supposed to repeat such mechanism every time?
- What if the map changes and now 2 maps to `Dolphin` ad 3 to `Dog`. How do we notify all clients of such change?
- Evidently such solution does not take into consideration maintainability and is not flexible for changes

The factory design pattern

Achieving flexibility and maintainability

- 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)
- To avoid clients to use different, potentially wrong, entry points for our animal we define all constructors private

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

HOGESCHOOL ROTTERDAM

The factory design pattern

Formalization

- Formalization

The factory design pattern

Conclusions

- Conclusions

This is it!

The best of luck, and thanks for the attention!