Creational patterns

Giảng viên: Huỳnh Tuấn Anh
Khoa CNTT - Đại học Nha Trang

Creational patterns

- Creational patterns
  - Thiết kế các khuôn mẫu tạo lập.
  - Thiết kế các khuôn mẫu tạo lập → Trừu tượng hoá tiến trình khởi tạo.
  - Giúp tạo ra một hệ thống các đối tượng độc lập với việc thao tác trên các đối tượng đó.
- Lớp & Đối tượng:
  - Lớp → Thừa kế
  - Đối tượng → Đại diện (interface, abstract classs)
- Các khuôn mẫu khởi tạo sẽ có trách nhiệm khởi tạo các đối tượng cụ thể thích hợp và gán cho đại diện của đối tượng

Creational patterns

- Sử dụng những lớp căn bản để xây dựng những lớp lớn hơn → Thao tác khởi tạo không còn đơn giản là khởi tạo cho 1 đối tượng nữa.
- Với 1 hàm khởi tạo cho 1 đối tượng cụ thể, khi cần tái sử dụng:
  - Viết lại một số thao tác → Phức tạp
  - Cách khác → Sử dụng Patterns.

Creational patterns

- Factory method
- Abstract Factory
- Builder Pattern
- Prototype pattern
- Singleton Pattern

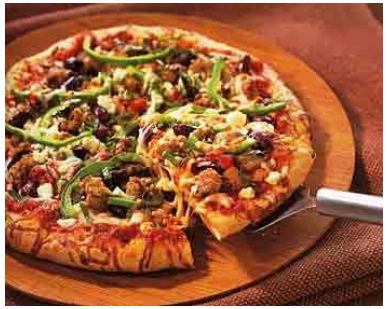
Factory method

Factory method

- Toán tử new: Dùng để tạo thể hiện của một đối tượng
  - Duck duck = new MallardDuck();
- Trường hợp có thêm một ConcreteClass được định nghĩa:
  - Việc sử dụng giao diện cho phép định nghĩa thêm lớp mới thực thi nó rất dễ dàng → *"Open for extension"*
  - Sử dụng toán tử new ở phần client để tạo các đối tượng đòi hỏi phải viết lại mã lệnh →mã lệnh không có tính chất *"close for modification"*.
- *Làm thế nào để tách rời phần khởi tạo thể hiện đối tượng của chương trình với các phần còn lại của chương trình?* → **Factory method**

Pizza Example

```
Pizza orderPizza()
{
        Pizza pizza = new Pizza();
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```



Nhiều loại Pizza?

Pizza Example

```
Pizza orderPizza(String type)
{
        Pizza pizza;
        if (type.equals("cheese"))
        {
                pizza = new CheesePizza();

        }
        else if (type.equals("greek")
            {
                pizza = new GreekPizza();
            }
            else if (type.equals("pepperoni")
                {
                        pizza = new PepperoniPizza();
                }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
}
```

Phần mã lệnh thường xuyên thay đổi

Phần thay đổi và phần cố định của mã lệnh nằm chung → khó khăn trong việc nâng cấp, không có tính reuseable

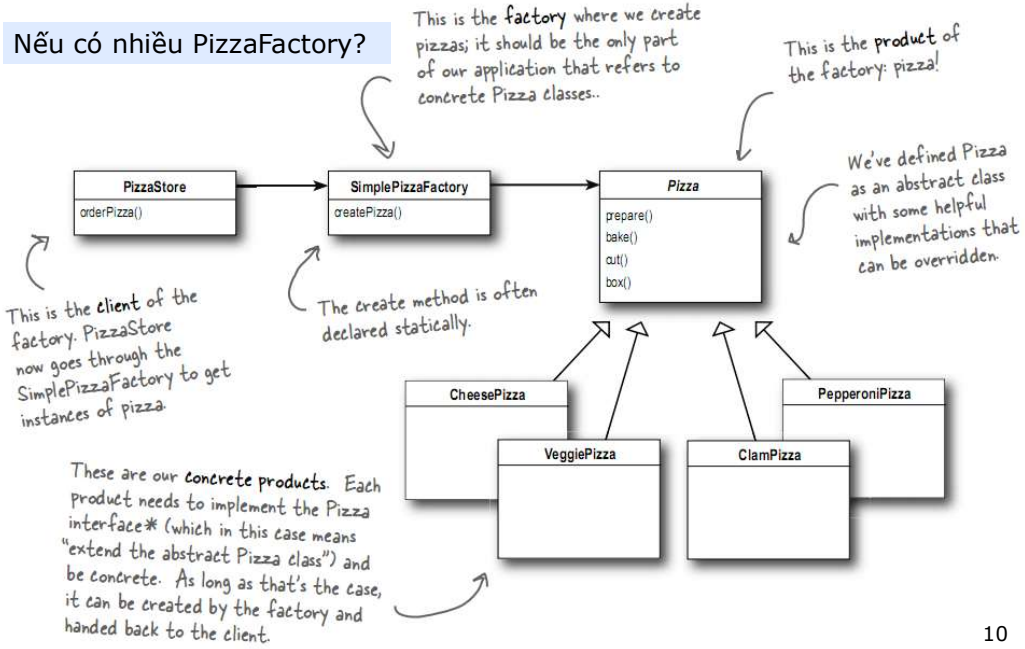
Pizza Example


```
public class SimplePizzaFactory
{
        public Pizza createPizza(String type)
        {
                Pizza pizza = null;
                if (type.equals("cheese"))
                        pizza = new CheesePizza();
                else if (type.equals("pepperoni"))
                        pizza = new PepperoniPizza();
                else if (type.equals("clam"))
                        pizza = new ClamPizza();
                else if (type.equals("veggie"))
                        pizza = new VeggiePizza();
                return pizza;
        }
}
```

```
public class PizzaStore
{
        SimplePizzaFactory factory;

        public PizzaStore(SimplePizzaFactory factory)
        {
                this.factory = factory;
        }

        public Pizza orderPizza(String type)
        {
                Pizza pizza;
                pizza = factory.createPizza(type);
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }
        // Các phương thức khác
}
```


Pizza Example: simple factory

Nếu có nhiều PizzaFactory?

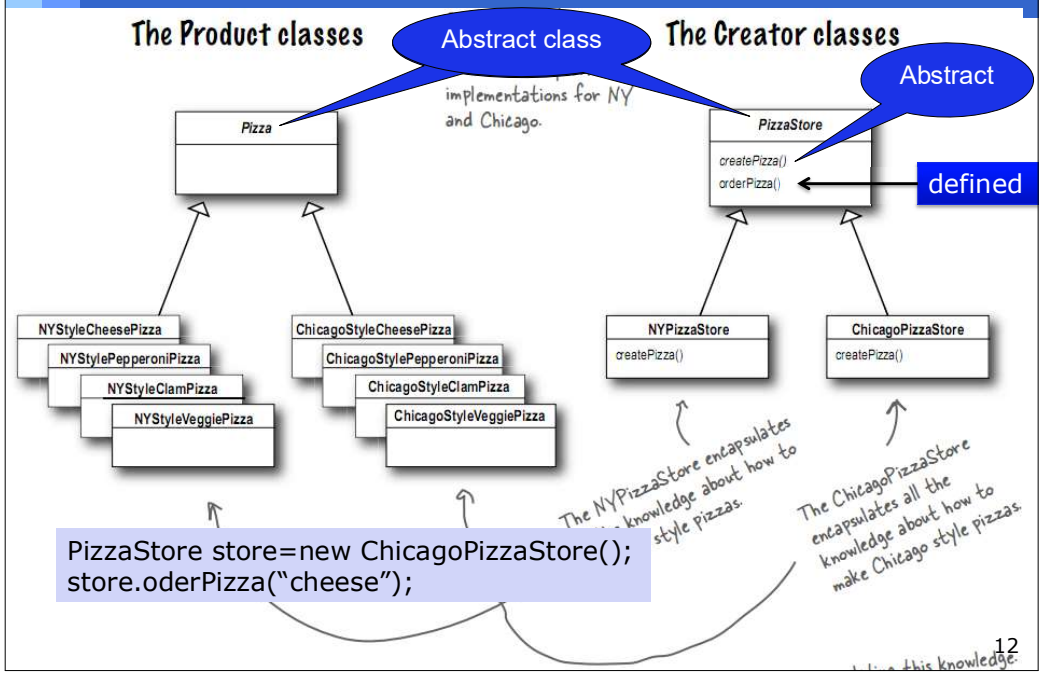


pizza example: simple factory

- 1st idea: Xây dựng nhiều cặp lớp tương tự như SimplePizzaFactory và PizzaStore
  - Mã lệnh ở client:

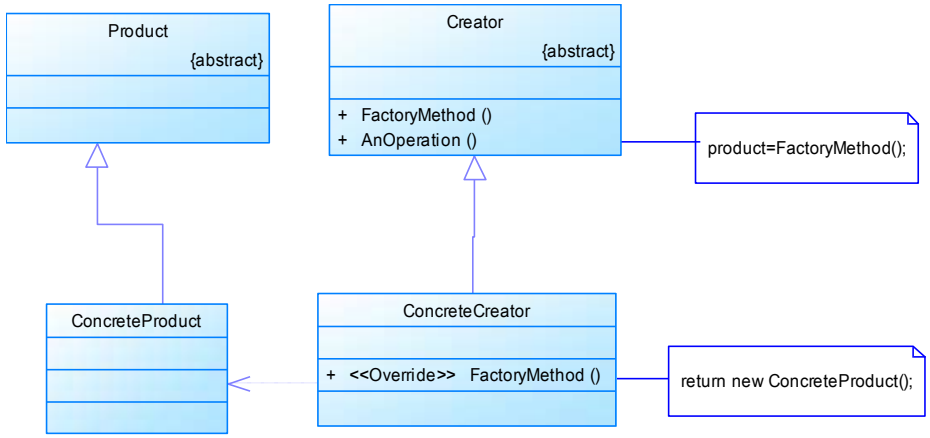
    ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
    PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
    chicagoStore.order("cheese");
  - *Mã lệnh không mềm dẻo:* Do có sự liên hệ chặt chẽ giữa các PizzaStore và PizzaFactory
- 2nd idea: Gộp 2 lớp PizzaFactory và PizzaStore
  - Xây dựng lớp trừu tượng PizzaStore
  - Các subclass thực thi interface PizzaStore
  - Các subclass quyết định loại pizza nào được tạo ra

Pizza Example: Factory method diagram



Factory method

- **Mục đích:** Định nghĩa một giao diện để tạo đối tượng, nhưng để cho các lớp con quyết định lớp nào sẽ được khởi tạo.
- **Cấu trúc:**



Questions

- Factory method có ưu điểm gì khi chỉ có một ConcreteCreator?
- Trong trường hợp nào ta phải tách riêng quá trình khởi tạo các đối tượng ra khỏi việc xử lý các đối tượng đó?
- Nêu ưu và khuyết điểm khi cài đặt phương thức FactoryMethod là static.
- FactoryMethod và Creator chỉ có thể luôn luôn là Abstract?
- Có thể thay các lớp Creator và Product là các interface được không?

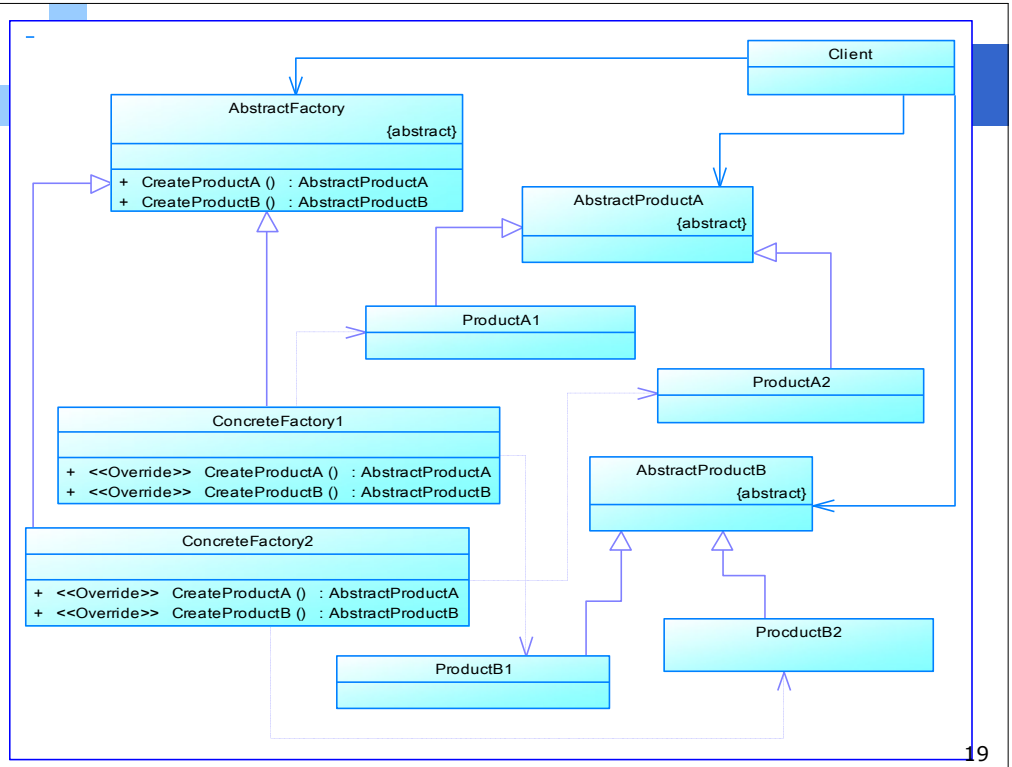
Design Principle

Depend upon abstractions. Do not depend upon concrete classes.

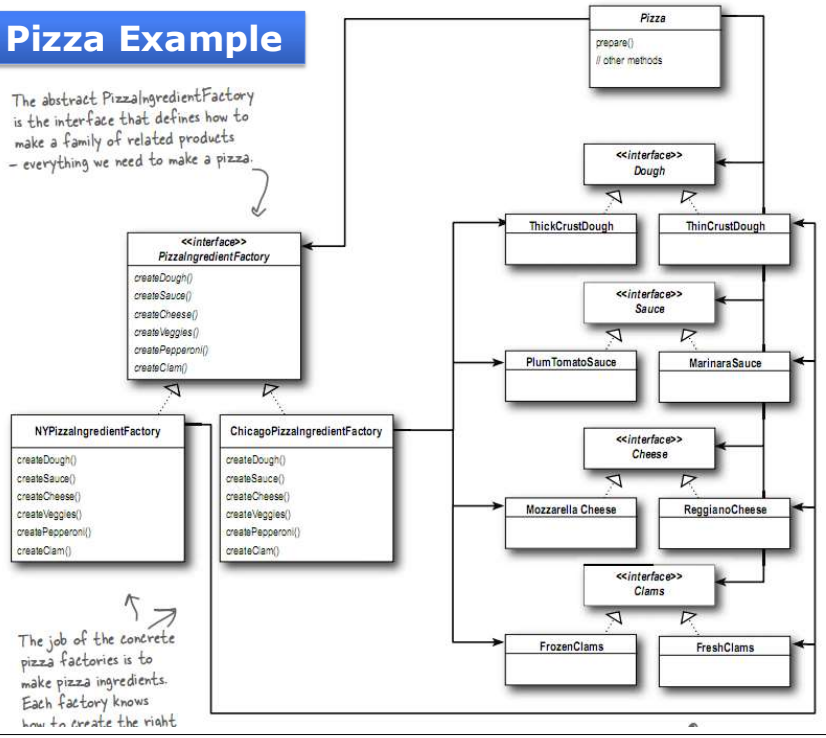
Abstract factory

Abstract factory

- Mục đích:
  - Cung cấp một giao diện (interface) cho việc tạo một tập đối tượng có liên quan hay phụ thuộc nhau mà không cần chỉ định các lớp cụ thể của chúng



Pizza Example



Abstract factory vs Factory method

Abstract Factory	Factory method
Tạo đối tượng mà không chỉ ra lớp cụ thể của nó	
Tách rời mã lệnh giữa client và các concrete class cần để tạo đối tượng cho các lớp đó	
Tạo một tập các đối tượng	tạo một đối tượng
Sử dụng khi client tạo một tập các đối tượng cùng với nhau	Sử dụng khi client chỉ tạo một đối tượng
Các đối tượng được tạo từ một đối tượng concreteFactory	Đối tượng được tạo từ lớp thực thi của lớp Creator
interface AbstractFactory thay đổi nếu thêm một loại product mới	interface Creator không thay đổi khi thêm product mới

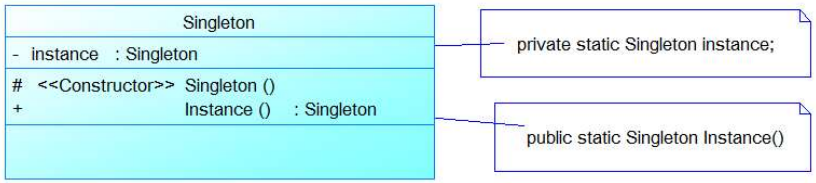
- → phương thức CreateProduct() thường được thực thi như một Factory method
- Sử dụng Abstract Factorry khi tạo một tập các loại product ít thay đổi

Singleton pattern

Singleton patern

- Mục đích:
  - Đảm bảo chỉ có một thể hiện của một lớp được tạo, và cho phép sự truy cập toàn cục đến đối tượng đó.
  - Việc tạo đối tượng của một lớp phải do chính lớp đó đảm nhận

- Cấu trúc



Singleton pattern

- Mã lệnh ()

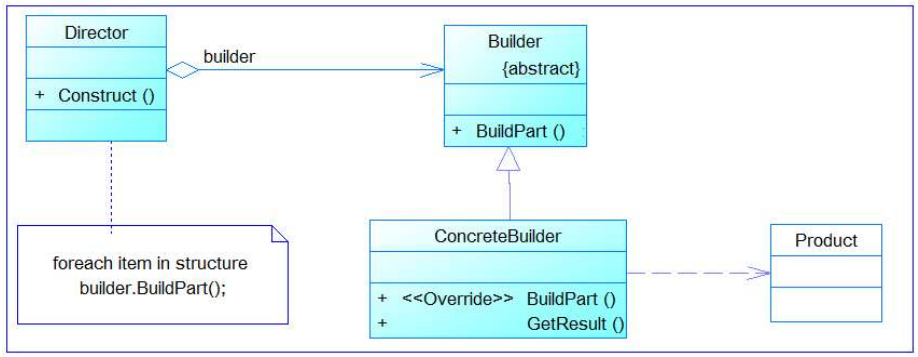
```
class Singleton
{
  // Fields
  private static Singleton instance;

  // Constructor
  protected Singleton() {}

  // Methods
  public static Singleton Instance()
  {
    // Uses "Lazy initialization"
    if( instance == null )
      instance = new Singleton();

    return instance;
  }
}

Sử dụng: Singleton s = Singleton.Instance();
```

Questions

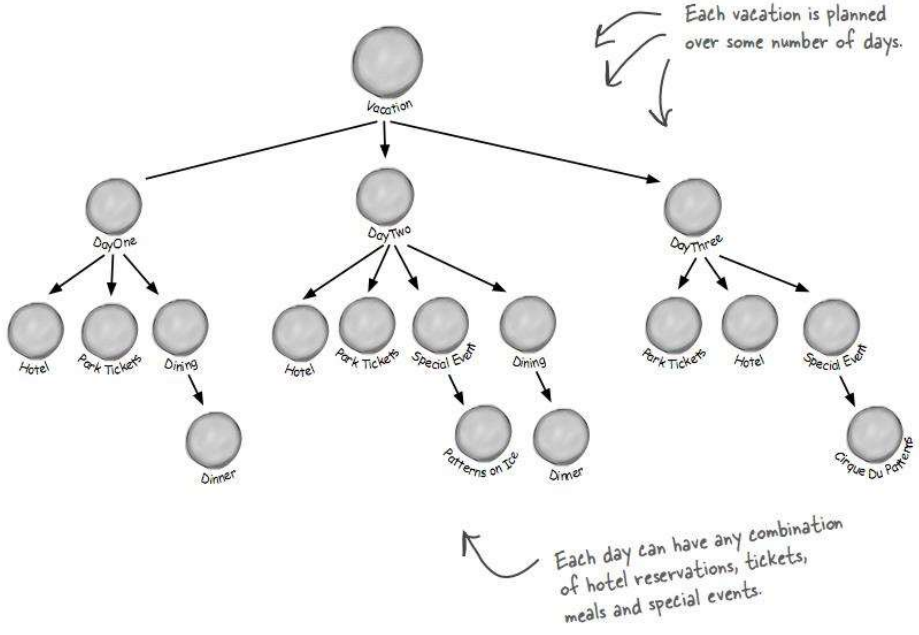
- Tại sao phương thức khởi tạo của lớp Singleton lại được khai báo *protected* ?
- Tại sao biến instance được khai báo *static* ?
- Phương thức Instance() có thể không khai báo *static* được không?
- Nêu một số ví dụ về các trường hợp sử dụng Singleton pattern

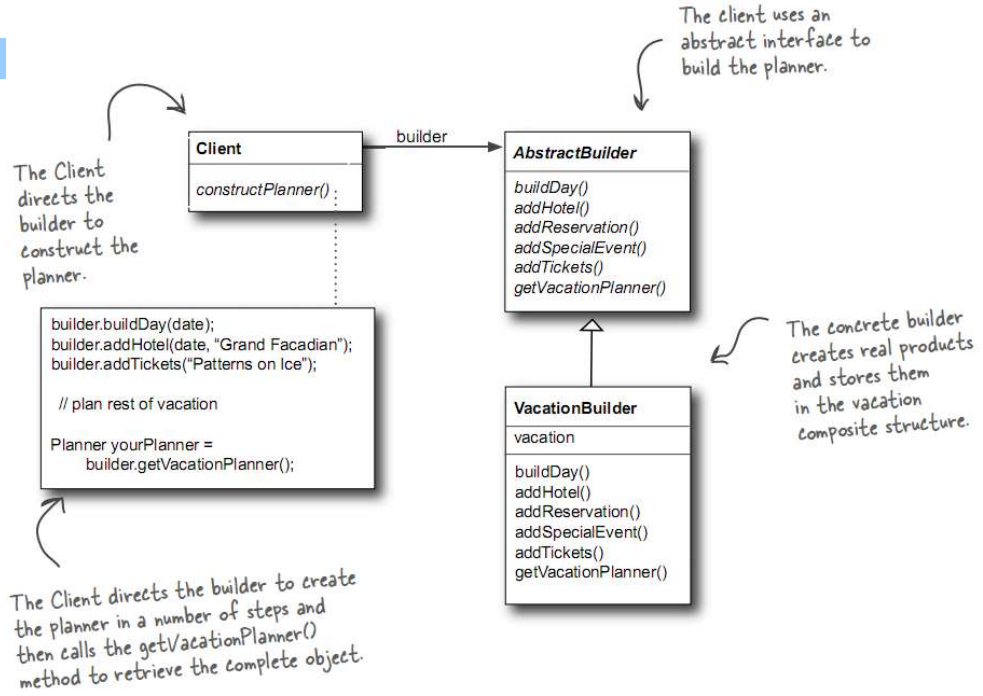
Builder pattern

- **Mục đích**: Đóng gói việc xây dựng một product và cho phép nó được xây dựng qua nhiều bước



Example: Vacation planner





Đánh giá Builder pattern

- Ưu điểm
  - Đóng gói cách xây dựng một đối tượng phức tạp
  - Cho phép một đối tượng được xây dựng qua nhiều bước và tiến trình khác nhau
  - Giấu biểu diễn bên trong của product đối với client
  - Sự thực thi một product có thể được hoán chuyển in, out bởi vì client chỉ thấy một giao diện trừu tượng
- Khuyết điểm
  - Việc xây dựng các đối tượng đòi hỏi nhiều tri thức hơn khi dùng Builder pattern so với dùng Factory
- Sử dụng
  - Được sử dụng cho việc xây dựng các cấu trúc phức tạp bao gồm nhiều thành phần

Questions

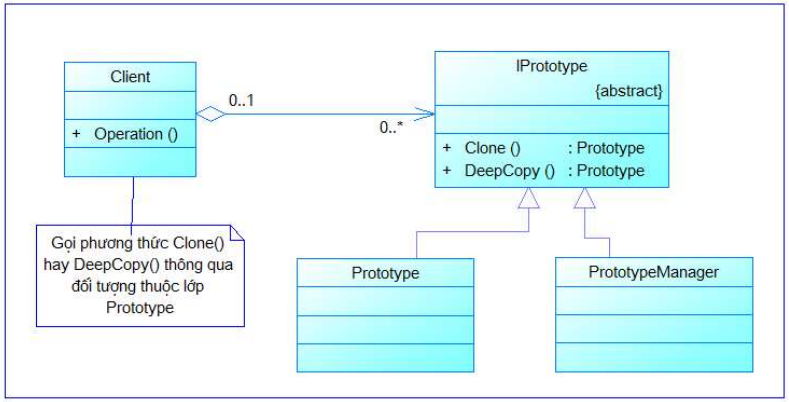
- So sánh 2 pattern Abstract factory và Builder
- Có thể sử dụng Builder pattern để thay thế cho Abstract factory được không?

Prototype pattern

Prototype pattern

- Mục đích:
  - Tạo một đối tượng của một lớp phức tạp hay một lớp mà việc khởi tạo tốn kém nhiều chi phí
  - Cho phép tạo một đối tượng mới bằng cách sao chép một đối tượng sẵn có

prototype pattern



Prototype pattern

- Lớp trừu tượng IPrototype
  - Phương thức Clone(): Chỉ tạo một tham chiếu tới một đối tượng
    - Không lưu giữ được các giá trị của một đối tượng khi đối tượng đó bị thay đổi.
  - Phương thức DeepCopy(): Tạo một bản sao thật sự của đối tượng
    - Lưu giữ được các giá trị của đối tượng
  - Các phương thức Clone(), DeepCopy() nên được cài đặt trong lớp trừu tượng IPrototype

Prototype pattern

- Ưu điểm
  - Che giấu sự phức tạp của việc tạo một đối tượng mới đối với Client
  - Cung cấp sự chọn lựa chọn Client để tạo các đối tượng mà không cần biết dạng của chúng.
  - Trong nhiều trường hợp việc copy một đối tượng hiệu quả hơn là việc khởi tạo nó
- Khuyết điểm
  - Đôi khi việc tạo bản copy của một đối tượng có thể rất phức tạp
- Sử dụng:
  - Prototype pattern được sử dụng khi việc tạo thể hiện của một lớp là phức tạp và tốn kém nhiều chi phí

Questions

- Có thể dùng interface để thay thế cho lớp Abstract IPrototype hay không?

Tài liệu tham khảo

- Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. Head First Design pattern. O'Reilly 2006.
- **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley 1995
- http://www.dofactory.com/Patterns/Patterns.aspx