Instituto de Computação uff

Técnicas de Programação Avançada

TCC-00.174

Prof.: Anselmo Montenegro

www.ic.uff.br/~anselmo

anselmo@ic.uff.br

Conteúdo: Padrão Factory

Documento baseado no material preparado pelo Prof. Luiz André (http://www.ic.uff.br/~lapaesleme/)

Estamos realmente programando para tipos?

E o que dizer sobre a criação de instâncias???

Como podemos criar instâncias de modo seguro sem criar problemas de ligação

Quando usamos **new**, criamos imediatamente uma instância de uma classe concreta

Naturalmente isso é uma implementação e não uma interface

Ligar seu código a uma classe concreta o torna mais frágil e menos flexível

Quando temos um conjunto de classes concretas e precisamos instanciá-las de acordo com uma condição, normalmente usamos um código semelhante ao código abaixo:

```
Duck duck;

If (picnic) {
   duck = new MallardDuc();
else if (hunting) {
   duck = new DecoyDuck();
else if (inBathTube) {
   duck = new RubberDuck();
...
}
```

Padrões de Projeto

O Padrão Factory

Temos várias classes sendo instanciadas e a decisão de qual instanciar é tomada em tempo de execução conforme algumas condições

Que ocorrerá quando for necessário alterações ou extensões?

O código terá que ser reaberto e reexaminado

Como esse tipo de código pode ocorrer em várias partes do aplicativo, a manutenção fica comprometida

Há algo de errado com **new**?

O problema é como lidar com o uso do **new** em face a mudanças

Quando codificamos para um interface o código fica protegido contra alterações

Ele sempre funcionará para novas classes que implementem a interface por causa do polimorfismo

Quando usamos (dependemos de) classes concretas o código deve ser alterado quando novas classes concretas são adicionadas

Isso significa que ele não fica fechado para modificações

Vamos lembrar do princípio de projeto O.O:

Identificar as partes que variam e separar do que continua igual

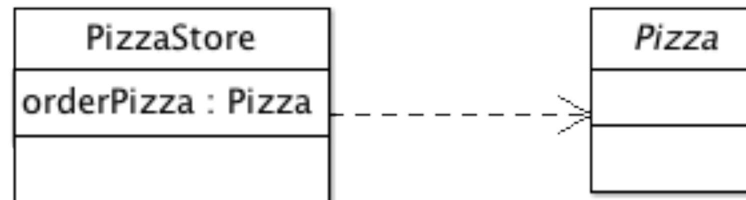
Padrões de Projeto

O Padrão Factory

Como pegar as partes que instanciam classes concretas e encapsulá-las do resto do aplicativo?

Vamos ver um exemplo e uma primeira tentativa de solução

Considere um aplicativo que simula uma fábrica de pizzas contendo uma classe PizzaStore que disponibiliza um método denominado orderPizza



Uma solução simples para orderPizza:

```
public void PizzaStore{

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

Qual o problema com essa solução?

```
public void PizzaStore{

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

Qual o problema com essa solução?

```
public void PizzaStore{

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

Não podemos instanciar Pizza que é uma classe abstrata...

Instituto de computação uff

Logo precisamos instanciar classes concretas

```
public void PizzaStore{
   Pizza pizza;

   Pizza orderPizza(String type ){

      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();
   }
}
```

Mas e se for necessário adicionar ou remover tipos de Pizza?

```
public void PizzaStore{
  Pizza pizza;

  Pizza orderPizza(String type){
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek")) {
      pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
    } else if (type.equals("clam"){
      pizza = new ClamPizza();
    }
    ...
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
  }
}
```

Mas e se for necessário adicionar ou remover tipos de Pizza?

```
public void PizzaStore{
  Pizza pizza;

  Pizza orderPizza(String type){
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek")) {      // (riscado)
      pizza = new GreekPizza();             // (riscado)
    } else if (type.equals("pepperoni") {
      pizza = new PepperoniPizza();
    } else if (type.equals("clam"){
      pizza = new ClamPizza();
    }
    ...
    pizza.prepare();
    pizza.bake();
    pizza.box();
    pizza.cut();
  }
}
```

Isso varia

Isso não varia

Lidar com a escolha do tipo de classe que deve ser instanciada no método orderPizza() faz com que o método não fique fechado para modificação

Uma ideia é remover o código de criação e levá-lo para outra classe

Padrões de Projeto

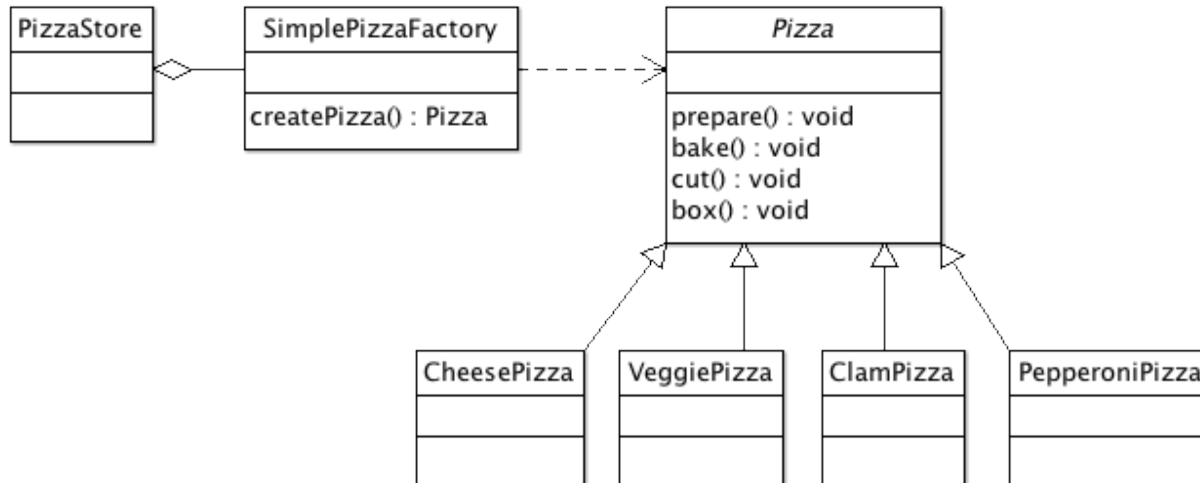
A Simple Factory

```
public void PizzaStore{
  SimplePizzaFactory factory;

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

  Pizza orderPizza(String type ){
     Pizza pizza = factory.createPizza(type)
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
  }
}
```

```
public class SimplePizzaFactory{

  public Pizza createPizza(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();
    } else if (type.equals("clam"){
      pizza = new ClamPizza();
    }
    return pizza;
  }
}
```

Padrões de Projeto

O Padrão Simple Factory



A Simple Factory não é exatamente um padrão de projetos

A Simple Factory é um *programming idiom*

Mas o que foi ganho com essa reorganização de código?

Não passamos o problema para outro objeto?

A diferença agora é que vários clientes podem usar a fábrica e as alterações ficam localizadas no código encapsulado

Que ocorre se quisermos ter franquias de nossa pizzaria?

Podemos criar diferentes classes para cada franquia e usar diferentes fábricas usando diferentes classes SimpleFactory

Padrões de Projeto

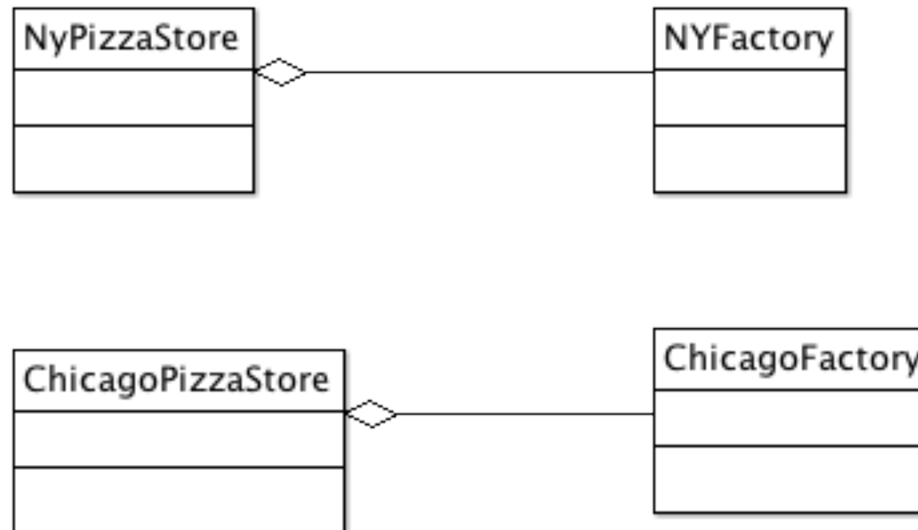
O Padrão Factory Method

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order(“veggie”);

NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order(“veggie”);
```

Instituto de computação uff

Padrões de Projeto

O Padrão Factory Method



Observe que uma SimpleFactory poderia ser usada com um outro modelo de PizzaStore

Não há garantia que as franquias sigam o modelo de pizzaria proposto no início (elas podem usar seu próprio modelo de loja)

Isto descaracteriza a ideia de uma franquia... (A classe PizzaStore e a criação das pizzas não estão mais acopladas como o esperado)

Pensando abstratamente, não há um acoplamento entre a classe que usa a fábrica (cliente) e a própria fábrica

Como contornar este problema?

Originalmente a pizzaria mantinha a instanciação dentro da própria classe PizzaStore

Problema: não era flexível e o código não era fechado a modificações

A SimpleFactory tornou o código da pizzaria fechado a modificações mas quebrou o vínculo entre a Pizza e a Pizzaria

Vamos criar um forma de estabelecer um vínculo entre a Pizzaria e a forma de instanciar as pizzas mas mantendo a flexibilidade

Padrões de Projeto

O Padrão Factory

```
public abstract class PizzaStore{

  public Pizza orderPizza(String type){
    Pizza pizza;
    pizza = createPizza(type);
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
  }

  abstract createPizza(String type);
}
```

PizzaStore agora é abstrata

```
public abstract class PizzaStore{

  public Pizza orderPizza(String type){
    Pizza pizza;
    pizza = createPizza(type);
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
  }

  abstract createPizza(String type);
}
```

A instanciação voltou para dentro da classe PizzaStore (cliente), mas como método abstrato.

Cabe às subclasses decidirem o que de fato é instanciado

PizzaStore agora é uma classe abstrata

orderPizza é um método fechado para modificações (poderíamos usar palavra static)

Mas é flexível. Por que?

PizzaStore agora é uma classe abstrata

orderPizza é um método fechado para modificações (poderíamos usar palavra static)

Mas é flexível. Por quê?

As subclasses de PizzaStore (as franquias) é que decidem que tipo de Pizzas criar

Mas a forma do pedido da pizza segue o padrão estabelecido para todas as franquias

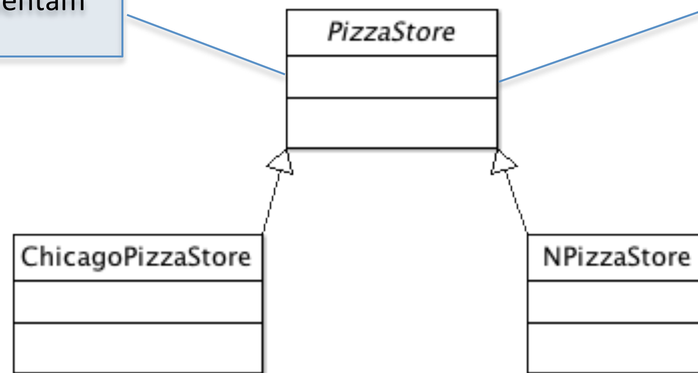
orderPizza é um método cliente do instanciador createPizza :)

Instituto de computação uff

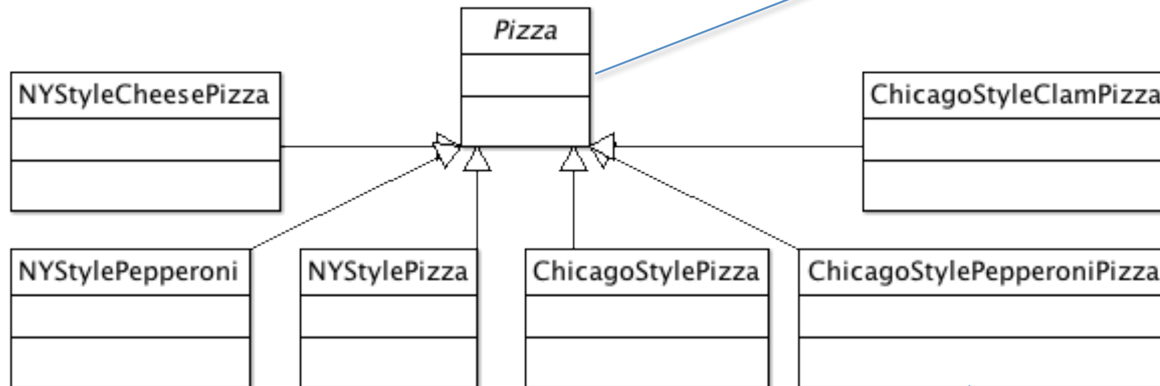
Criador abstrato – contém um método que depende de um produto abstrato

Como cada franquia recebe sua própria subclasse de PizzaStore fica livre para criar seu próprio estilo de pizza implementando createPizza()

Classes concretas – implementam createPizza



Produto abstrato



Exemplo de produto concreto

Padrão Factory Method

Define uma interface para criar um objeto, mas permite às classes decidir qual classe instanciar. **O Factory Method permite uma classe deferir a instanciação para subclasses**

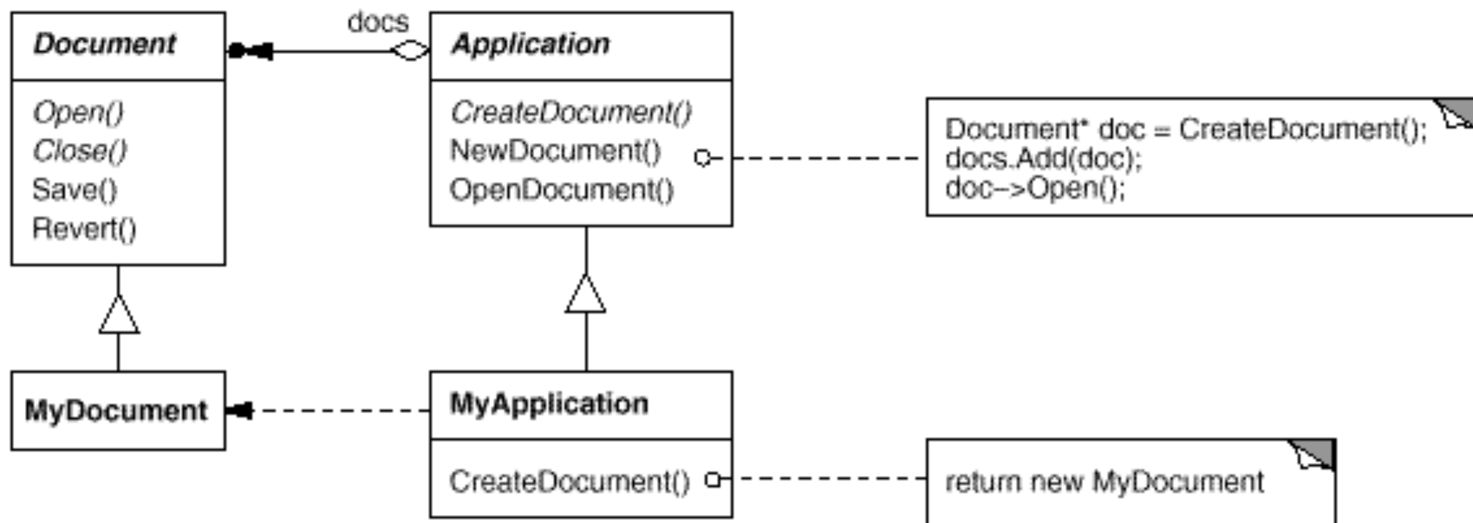
Utilize o padrão quando uma classe precisa instanciar subclasses de uma classe C que ainda não foram definidas.

Utilize esse padrão quando precisar delegar a responsabilidade de criar objetos

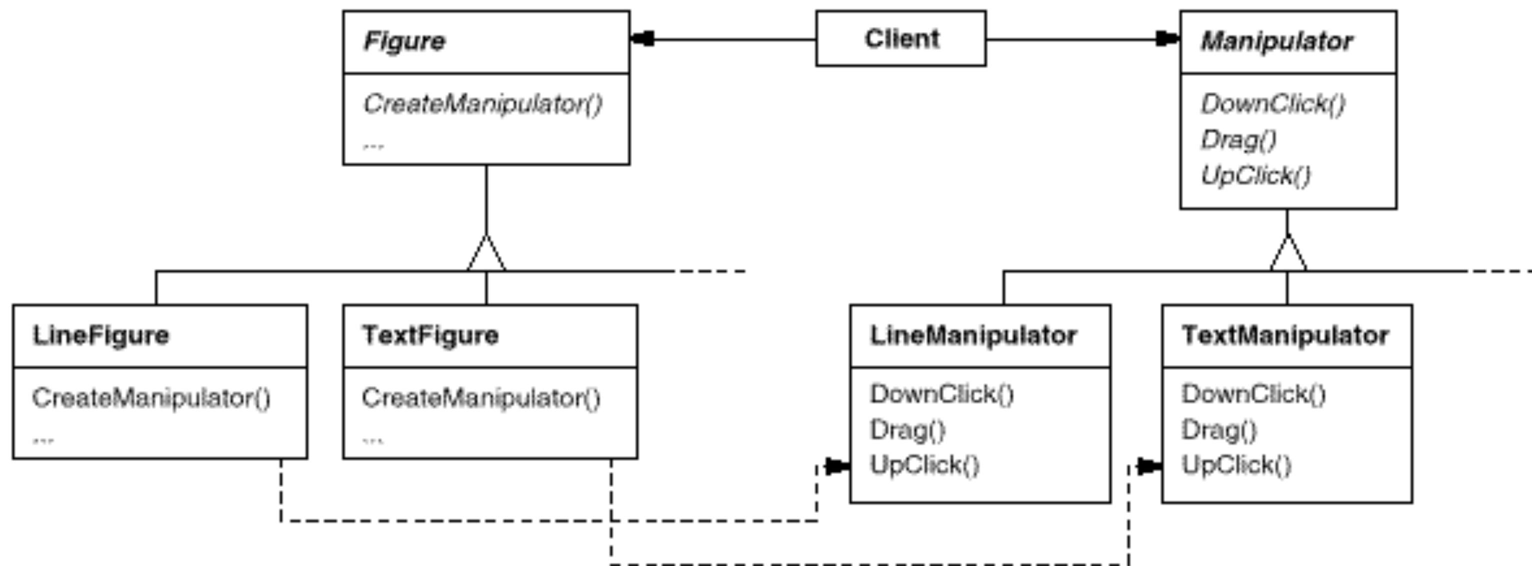
Padrões de Projeto

Exercícios I

Considere uma classe abstrata *Application* que é cliente de Documentos, cujas versões concretas precisam de algum modo ser instanciadas. Proponha uma solução usando o Padrão Factory Method



No exemplo abaixo, figura tem um conjunto de operações que dependem de manipuladores que podem ser diferentes subtipos (LineManipulator e TextManipulator). Proponha uma solução para instanciar os manipuladores apropriados para cada subtipo de figura.



Oferece um modo de encapsular instanciações de tipos concretos

O criador abstrato oferece uma interface com um método para criar objetos (*Factory Method*)

Outros métodos que são implementados no criador abstrato operam sobre produtos fabricados pelo *FactoryMethod*

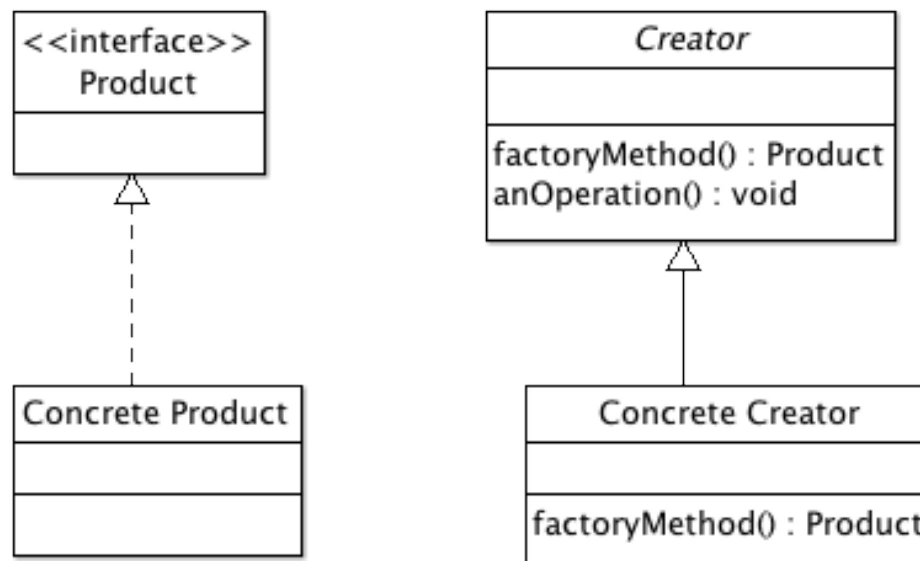
Apenas as subclasses implementam o *FactoryMethod*

A classe criadora é escrita sem conhecer que produtos reais serão criados

Somente as subclasses escolhem que produtos irão criar

Por este motivo diz-se que as subclasses decidem quais produtos instanciar

Entretanto essa decisão não é dinâmica (em tempo de execução)


<<interface>>
Product
Creator
factoryMethod() : Product
anOperation() : void
Concrete Product
Concrete Creator
factoryMethod() : Product

Vamos esquecer momentaneamente o conceito de fábricas

Como fica o grau de dependência entre as classes de uma solução que não se baseia em padrões?

Vejamos nos próximos slides

Padrões de Projeto

Uma solução sem padrões

```
public class DependentPizzaStore{
  Public Pizza createPizza(String style, String type){
    Pizza pizza = null;
    if (style.equals("NY")){
     if (type.equals("cheese")) {
       pizza = new CheesePizza();
     } else if (type.equals("greek")) {
       pizza = new GreekPizza();
     } else if (type.equals("pepperoni")
       pizza = new PepperoniPizza();
     }
    }
   else if (style.equals("Chicago")){
      if (type.equals("cheese")) {
       pizza = new CheesePizza();
     } else if (type.equals("greek")) {
       pizza = new GreekPizza();
     } else if (type.equals("pepperoni")
       pizza = new PepperoniPizza();
     }
   }
   else {
      System.out.println("Error: invalid type of pizza");
      return;
   }
```

```
   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box();

   return pizza;
}
```

Padrões de Projeto

Dependência entre as classes da solução anterior

PizzaStore

NYStyleClamPizza

NYStyleVeggiePizza

NYStyleCheesePizza

NYStylePepperoniPizza

ChicagoStyleVeggiePizza

ChicagoStyleClamPizza

ChicagoStyleCheesePizza

ChicagoStylePepperoniPizza

Instituto de Computação

Podemos observar componentes de alto nível dependendo de componentes de mais baixo nível...

Componente de alto nível – classe cujo comportamento é definido por componentes de mais baixo nível

Exemplo: o comportamento de PizzaStore é definido em termos de Pizza (nível mais baixo)

Um componente de alto nível que depende de componentes de baixo nível concretos se torna mais vulnerável a mudanças

Mudanças no comportamento de instâncias concretas de Pizza irão potencialmente deflagrar mudanças em PizzaStore

Quinto Principio – Princípio da inversão de dependência

Dependa de abstrações e não de classes concretas

Como aplicar o princípio para o problema das pizzas?

Criar uma abstração Pizza

Evitar que PizzaStore instancie diretamente implementações (versões concretas de Pizza) através de Factory Method

PizzaStore

NYStyleClamPizza

ChicagoStyleVeggiePizza

Pizza

NYStyleVeggiePizza

ChicagoStyleClamPizza

NYStyleCheesePizza

ChicagoStyleCheesePizza

NYStylePepperoniPizza

ChicagoStylePepperoniPizza

Porque a palavra inversão no princípio?

Tem a ver com a forma como pensamos no projeto

A ordem convencional seria pensar na PizzaStore e posteriormente nos tipos de Pizza (CheesePizza, VeggiePizza, etc)

Mas e se invertermos e pensarmos nas classes concretas identificando o que pode ser abstraído (o conceito de Pizza)...

Então, PizzaStore passa a se preocupar somente com o conceito abstrato de Pizza e não nas versões concretas

Mas para isso precisamos liberar PizzaStore da instanciação direta de classes concretas

O objetivo é que PizzaStore dependa da abstração

Factory Method é um padrão que resolve esse problema, mas não é o único...

Diretrizes para seguir o princípio de inversão de dependências:

Nenhuma variável deve conter uma referência para uma classe concreta

Nenhuma classe deve derivar de uma classe concreta

Nenhum método deve substituir um método implementado em uma de suas classes base

Voltando as Pizzarias...

As PizzaStores estão todas usando o *framework* de Pizzaria proposto anteriormente

Todas preparam, assam, cortam, empacotam do jeito estabelecido

Vamos olhar de perto as Pizzas...

Padrões de Projeto

Como criar famílias de produtos relacionados

Algumas franquias tomara a liberdade de usar ingredientes mais baratos para aumentar os lucros...

Como evitar esse problema?

Vamos fornecer uma fábrica que produz os ingredientes para as PizzaStores

Padrões de Projeto

Como criar famílias de produtos relacionados

Problema: cada franquia usa suas próprias variações locais de ingredientes apesar da receita da Pizza ser a mesma: massa, molho, queijo, etc

Então temos apenas que garantir um modelo de fábrica que garanta a qualidade

Como fazer isso?

Vamos definir uma fábrica abstrata, isto é uma interface que indica que ingredientes devem ser produzidos para criar uma Pizza

```
public interface PizzaIngredientFactory {
  public Dough createDough();
  public Sauce createSauce();
  public Cheese createCheese();
  public Veggies[] createVeggies();
  public Pepperoni createPepperoni();
  public Clams createClam();
}
```

Como implementar as diferenças regionais?

Criar um fábrica para cada região (criar uma subclasse que implementa a interface fábrica abstrata PizzaIngredientFactory)

Criar um conjunto de classes filhas para cada classe abstrata produto para serem usadas como ingredientes. Ex.:ReggianoCheese, RedPeppers, and ThickCrustDough.

Conectar as fábricas ao antigo código PizzaStore

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {
  public Dough createDough() {
    return new ThinCrustDough();
  }
  public Sauce createSauce() {
    return new MarinaraSauce();
  }
  public Cheese createCheese() {
    return new ReggianoCheese();
  }
  public Veggies[] createVeggies() {
    Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
    return veggies;
  }
  public Pepperoni createPepperoni() {
    return new SlicedPepperoni();
  }
  public Clams createClam() {
    return new FreshClams();
  }
}
```

Padrões de Projeto

O Padrão Abstract Factory

As Pizzas agora são classes abstratas – um produto abstrato formado um conjunto de componentes abstratos (interfaces que definem os ingredientes)

```
public abstract class Pizza {
  String name;
  Dough dough;
  Sauce sauce;
  Veggies veggies[];
  Cheese cheese;
  Pepperoni pepperoni;
  Clams clam;
  abstract void prepare();

 void bake() {
    System.out.println("Bake for 25 minutes at 350");
  }
  void cut() {
    System.out.println("Cutting the pizza into diagonal slices");
  }
void box() {
    System.out.println("Place pizza in official PizzaStore box");
  }
  void setName(String name) {
    this.name = name;
  }
  String getName() {
    return name;
  }
  public String toString() {
    // code to print pizza here
  }
}
```

Uma pizza concreta implementa uma Pizza abstrata

Os tipos de pizza podem ser regionalizados (particularizados) via a especificação de uma fábrica de ingredientes específica

```
public class ClamPizza extends Pizza {
  PizzaIngredientFactory ingredientFactory;

   public ClamPizza(PizzaIngredientFactory ingredientFactory) {
     this.ingredientFactory = ingredientFactory;
   }
   void prepare() {
     System.out.println("Preparing " + name);
     dough = ingredientFactory.createDough();
     sauce = ingredientFactory.createSauce();
     cheese = ingredientFactory.createCheese();
     clam = ingredientFactory.createClam();
  }
}
```

Uma pizza concreta implementa uma Pizza abstrata

Os tipos de pizza podem ser regionalizados (particularizados) via a especificação de uma fábrica de ingredientes específica

```
public class ClamPizza extends Pizza {
  PizzaIngredientFactory ingredientFactory;

   public ClamPizza(PizzaIngredientFactory ingredientFactory) {
     this.ingredientFactory = ingredientFactory;
   }
   void prepare() {
    System.out.println("Preparing " + name);
    dough = ingredientFactory.createDough();
    sauce = ingredientFactory.createSauce();
    cheese = ingredientFactory.createCheese();
    clam = ingredientFactory.createClam();
  }
}
```

Como são cridadas as diferentes pizzarias?

```
public class NYPizzaStore extends PizzaStore {
  protected Pizza createPizza(String item) {
    Pizza pizza = null;
    PizzaIngredientFactory ingredientFactory = new NYPizzaIngredientFactory();

    If (item.equals(“cheese”)) {
      pizza = new CheesePizza(ingredientFactory);
      pizza.setName(“New York Style Cheese Pizza”);
    } else if (item.equals(“veggie”)) {
      pizza = new VeggiePizza(ingredientFactory);
      pizza.setName(“New York Style Veggie Pizza”);
    } else if (item.equals(“clam”)) {
      pizza = new ClamPizza(ingredientFactory);
      pizza.setName(“New York Style Clam Pizza”);
    } else if (item.equals(“pepperoni”)) {
      pizza = new PepperoniPizza(ingredientFactory);
      pizza.setName(“New York Style Pepperoni Pizza”);
    }
  return pizza;
 }
}
```

Instituto de computação uff

Padrões de Projeto

O Padrão Abstract Factory

A fábrica abstrata especifica como criar uma família de produtos relacionados

O Cliente é escrito em termos da fábrica abstrata. Em tempo de execução é ligado a uma fábrica concreta

Família de produtos

Cada fábrica concreta implementa um família diferente de produtos

Padrão Abstract Factory

Provê uma interface para a criação de uma família de objetos relacionados ou dependentes sem especificar suas classes concretas.

Instituto de computação uff

Padrões de Projeto

Exercício

obtido e baseado em http://pt.wikipedia.org/wiki/Abstract_Factory

Considere uma aplicação que necessita criar interfaces gráficas em diferentes versões, por exemplo, Motif e QT, dependendo da configuração que seja escolhida.

As interfaces são representadas pela classe abstrata *Widget* que pode conter inúmeros elementos como *Windows*, *Frames*, *RadioButtons, Buttons*, e etc, também abstratos. Cada interface gráfica concreta é formada por uma coleção de elementos de interface escolhidos pelo designer.

Considerando a existência de uma classe Configuration com um método getConfiguration, o qual retorna um tipo enumerado com valores MOTIF e QT, Escreva uma solução para construção de uma interface gráfica que instância os elementos necessários.

Suponha que para elemento de interface exista os correspondentes contretos na versão MOTIF e QT: QTWindow, QTFrame, QTRadioButton, QTButton, Motif Window, MotifFrame, MotifRadioButton, MotifButton, etc.

Considerando a existência de uma classe Configuration com um método getConfiguration( ), o qual retorna um tipo enumerado com valores MOTIF e QT, escreva uma solução para construção de uma interface gráfica que instância os elementos necessários.

Padrões de Projeto

Um outro exemplo de aplicação do Abstract Factory

WidgetFactory

Cliente

Botao

BotaoQt

BotaoMotif

MotifWidgetFactory

QtWidgetFactory

Janela

JanelaMotif

JanelaQt

http://pt.wikipedia.org/wiki/Ficheiro:Abstract_Factory.gif

Padrões de Projeto

Bibliografia

- Use a Cabeça ! Padrões de Projetos (design Patterns) - 2ª Ed. Elisabeth Freeman e Eric Freeman. Editora: Alta Books
- Padroes de Projeto – Soluções reutilizáveis de software orientado a objetos. Erich Gamma, Richard Helm, Ralph Johnson. Editora Bookman