Abstract Factory

Objetivo

- Prover uma interface para criar uma família de objetos relacionados ou dependentes sem especificar suas classes concretas

Também chamado de

- Kit

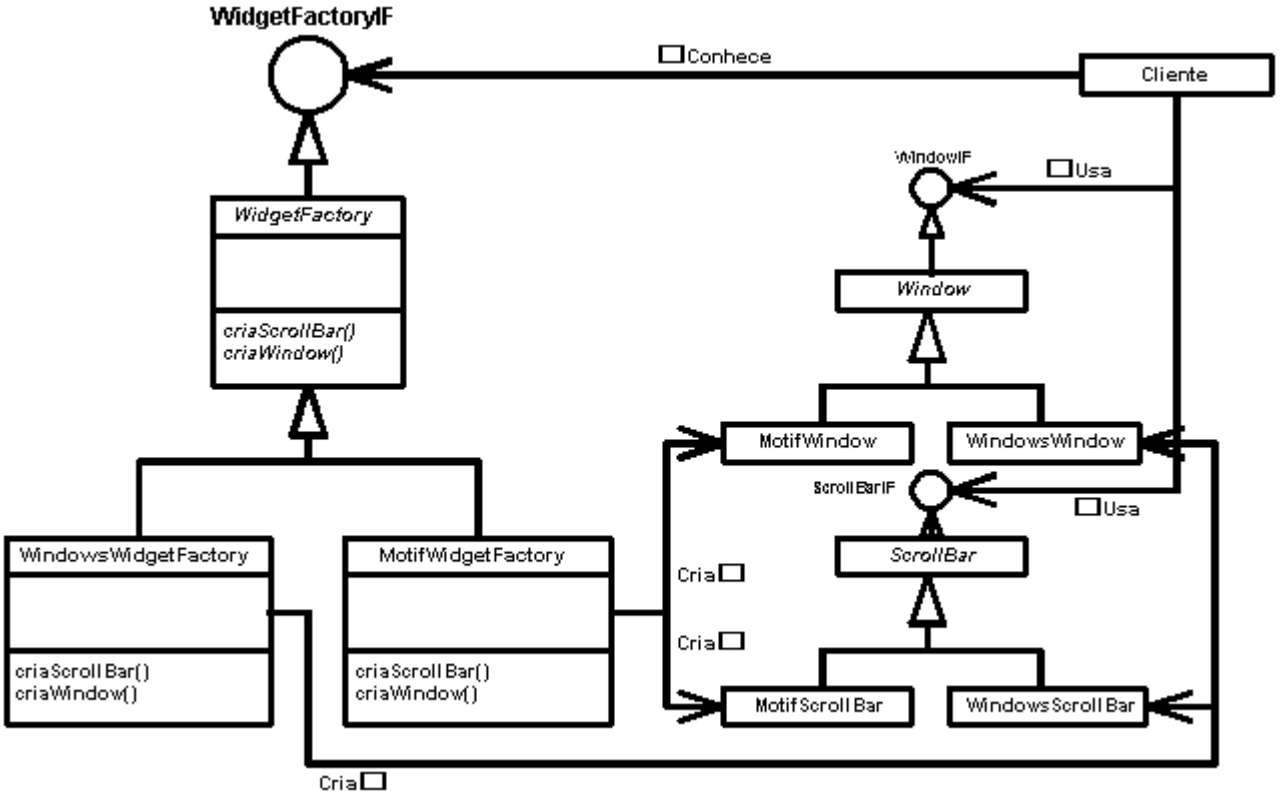
Resumo

- Parece semelhante ao padrão Factory Method, mas
  - Em vez do cliente (que quer criar objetos sem saber as classes exatas) chamar um método de criação (Factory Method), ele de alguma forma *possui* um objeto (uma Abstract Factory) e usa este objeto para chamar os métodos de criação
    - Onde Factory Method quer que você *seja* diferente (via herança) para criar objetos diferentes, o Abstract Factory quer que você *tenha* algo diferente
  - Se ele possuir uma referência a uma Abstract Factory diferente, toda a criação será diferente
  - O fato de todos os métodos de criação estarem na mesma subclasse de uma Abstract Factory permite satisfazer a restrição de criar apenas objetos relacionados ou dependentes

Exemplo: look-and-feel de GUIs

- Para look-and-feel diferentes (Motif, Windows, Mac, Presentation Manager, etc.) temos formas diferentes de manipular janelas, scroll bars, menus, etc.
- Para criar uma aplicação com GUI que suporte qualquer look-and-feel, precisamos ter uma forma simples de criar objetos (relacionados) de uma mesma família

- Os objetos são dependentes porque não posso criar uma janela estilo Windows e um menu estilo Motif
- Java já resolveu este problema internamente no package awt usando Abstract Factory e você não precisa se preocupar com isso
  - Porém, você poderia estar usando C++ e precisaria cuidar disso você mesmo
- Uma classe (abstrata) (ou interface, em Java) "Abstract Factory" define uma interface para criar cada tipo de objeto básico (widgets no linguajar GUI)
- Também tem uma classe abstrata para cada tipo de widget (window, scroll bar, menu, ...)
- Há classes concretas para implementar cada widget em cada plataforma (look-and-feel)
- Clientes chamam a Abstract Factory para criar objetos
  - Uma Concrete Factory cria os objetos concretos apropriados
- Ver o diagrama de classes abaixo

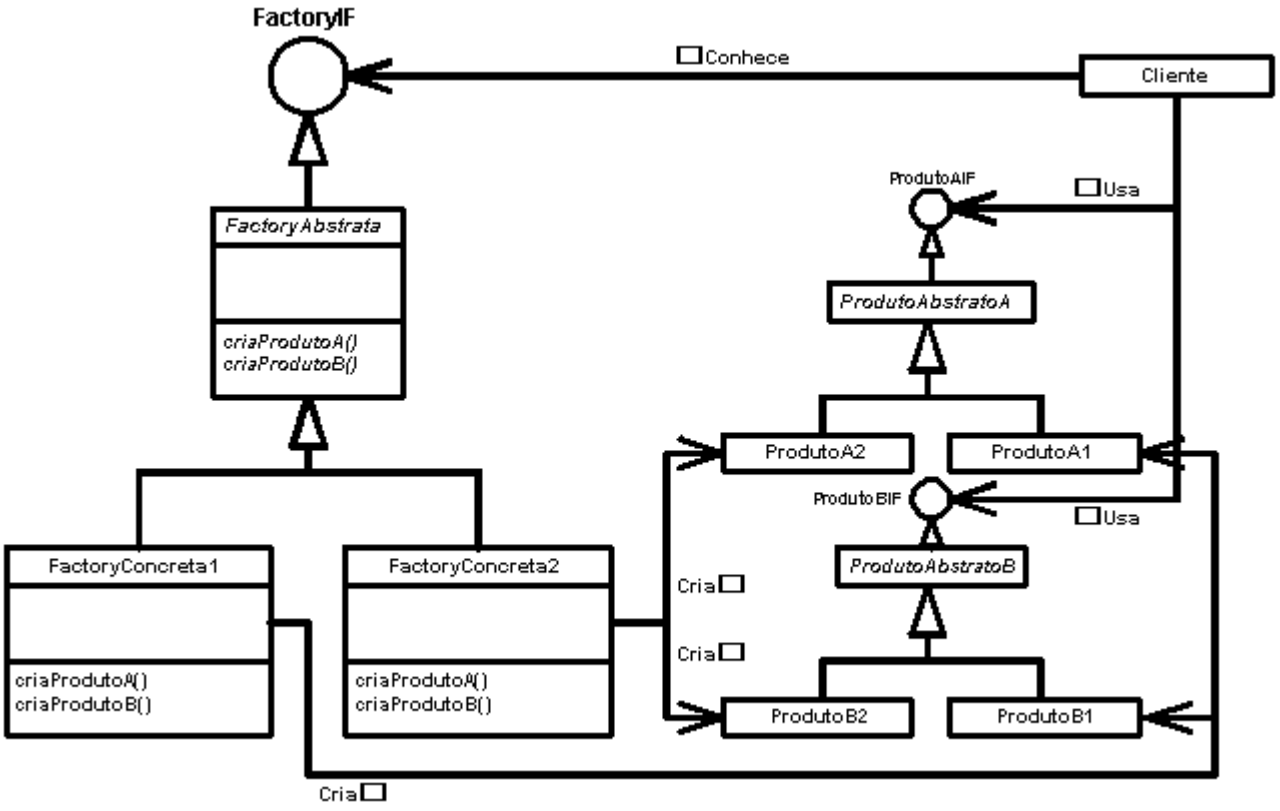


O padrão Abstract Factory

Quando usar o padrão Abstract Factory?

- Quando um sistema deve ser independente de como seus produtos são criados, compostos e representados
- Quando um sistema deve ser configurado com uma entre várias famílias de produtos
- Quando uma família de produtos relacionados foi projetada para uso conjunto e você deve implementar essa restrição
- Quando você quer fornecer uma biblioteca de classes e quer revelar sua interface e não sua implementação
  - Não permita portanto que objetos sejam diretamente criados com new

Estrutura genérica



Participantes

- **FactoryIF** (WidgetFactoryIF)
  - Define uma interface para as operações que criam objetos como produtos abstratos

- **FactoryAbstrata** (WidgetFactory)
  - Possível classe abstrata para fatorar o código comum às FactoryConcretas
- **FactoryConcreta** (MotifWidgetFactory, WindowsWidgetFactory)
  - Implementa as operações para criar objetos para produtos concretos
- **ProdutoXIF** (WindowIF, ScrollBarIF)
  - Define uma interface para objetos de um tipo
- **ProdutoAbstrato** (Window, ScrollBar)
  - Possível classe abstrata para fatorar o código comum aos ProdutosConcretos
- **ProdutoConcreto** (MotifWindow, MotifScrollBar)
  - Define um objeto produto a ser criado pela FactoryConcreta correspondente
  - Implementa a interface de ProdutoAbstrato
- **Cliente**
  - Usa apenas interfaces definidas por FactoryIF e ProdutoXIF

Colaborações entre objetos

- Normalmente uma única instância de uma classe FactoryConcreta é criada em tempo de execução
- Essa FactoryConcreta cria objetos tendo uma implementação particular
- Para criar produtos diferentes, clientes devem usar uma FactoryConcreta diferente
- FactoryAbstrata depende de suas subclasses FactoryConcreta para criar objetos de produtos

Consequências do uso do padrão Abstract Factory

- O padrão isola classes concretas
  - Uma factory encapsula a responsabilidade e o

processo de criação de objetos de produtos
    - Isola clientes das classes de implementação
    - O cliente manipula instâncias através de suas interfaces abstratas
- Facilita o câmbio de famílias de produtos
    - A classe da FactoryConcreta só aparece em um lugar: onde ela é instanciada
    - Uma mudança numa única linha de código pode ser suficiente para mudar a FactoryConcreta que a aplicação usa
        - A família inteira de produtos muda de uma vez
- Promove a consistência entre produtos
    - Produtos de uma determinada família devem funcionar conjuntamente e não misturados com produtos de outra família
    - O padrão permite implementar esta restrição com facilidade
- Do lado negativo: dar suporte a novos tipos de produtos é difícil
    - O motivo é que a FactoryAbstrata fixa o conjunto de produtos que podem ser criados
    - Dar suporte a mais produtos força a extensão da interface da factory o que envolve mudanças na FactoryAbstrata e em todas suas subclasses FactoryConcreta

Considerações de implementação

- Factory como padrão Singleton
    - Uma aplicação normalmente só precisa de uma única instância de uma FactoryConcreta por família de produtos
    - O padrão Singleton ajuda a controlar a instância única
- Criação dos produtos
    - A FactoryIF apenas define a *interface* de criação
    - Quem cria os objetos são as FactoryConcreta

- Tais subclasses são freqüentemente implementadas usando o padrão Factory Method
- Uma FactoryConcreta faz override do Factory Method de cada produto

Exemplo de código: criação de labirintos

- Todos os Factory Methods do padrão anterior saem da classe Jogo e entram na factory abstrata FactoryDeLabirinto
  - Também possuem um default, o que significa que FactoryDeLabirinto também é uma factory concreta
  - Também usamos o design pattern Singleton aqui

```
public interface FactoryDeLabirintoIF {
  public LabirintoIF criaLabirinto();
  public SalaIF criaSala(int númeroDaSala);
  public ParedeIF criaParede() {
  public PortaIF criaPorta(SalaIF sala1, SalaIF sala2) {
}

public class FactoryDeLabirinto implements FactoryDeLabirint
  private static FactoryDeLabirintoIF instânciaÚnica = null;

  private FactoryDeLabirinto() {}

  public static FactoryDeLabirintoIF getInstance(String tipo
    if(instânciaÚnica == null) {
      if(tipo.equals("perigoso")) {
        instânciaÚnica = new FactoryDeLabirintoPerigoso();
      } else if(tipo.equals("encantado")) {
        instânciaÚnica = new FactoryDeLabirintoEncantado();
      } else {
        instânciaÚnica = new FactoryDeLabirinto();
      }
    }
    return instânciaÚnica;
  }

  // Factory Methods
  // Tem default para as Factory Methods
  public LabirintoIF criaLabirinto() {
    return new Labirinto();
  }
```

```
  public SalaIF criaSala(int númeroDaSala) {
    return new Sala(númeroDaSala);
  }

  public ParedeIF criaParede() {
    return new Parede();
  }

  public PortaIF criaPorta(SalaIF sala1, SalaIF sala2) {
    return new Porta(sala1, sala2);
  }
}
```

- A nova versão de montaLabirinto recebe um FactoryDeLabirintoIF como parâmetro e cria um labirinto

```
public class Jogo implements JogoIF {
  // Observe que essa função não tem new: ela usa uma Abstra
  // Esta é a *única* diferença com relação à versão origina
  // Observe como montaLabirinto acessa a factory (através d
  public LabirintoIF montaLabirinto(FactoryDeLabirintoIF fac
    LabirintoIF umLabirinto = factory.criaLabirinto();
    SalaIF sala1 = factory.criaSala(1);
    SalaIF sala2 = factory.criaSala(2);
    PortaIF aPorta = factory.criaPorta(sala1, sala2);

    umLabirinto.adicionaSala(sala1);
    umLabirinto.adicionaSala(sala2);

    sala1.setVizinho(NORTE, factory.criaParede());
    sala1.setVizinho(LESTE, aPorta);
    sala1.setVizinho(SUL, factory.criaParede());
    sala1.setVizinho(OESTE, factory.criaParede());

    sala2.setVizinho(NORTE, factory.criaParede());
    sala2.setVizinho(LESTE, factory.criaParede());
    sala2.setVizinho(SUL, factory.criaParede());
    sala2.setVizinho(OESTE, aPorta);

    return umLabirinto;
  }
}
```

- Para criar um labirinto encantado, criamos uma factory concreta como subclasse de

FactoryDeLabirinto

```
public class FactoryDeLabirintoEncantado extends FactoryDeLa
  public SalaIF criaSala(int númeroDaSala) {
    return new salaEncantada(númeroDaSala, jogaEncantamento(
  }
  public PortaIF criaPorta(SalaIF sala1, SalaIF sala2) {
    return new portaPrecisandoDeEncantamento(sala1, sala2);
  }
  protected EncantamentoIF jogaEncantamento() {
    ...
  }
}
```

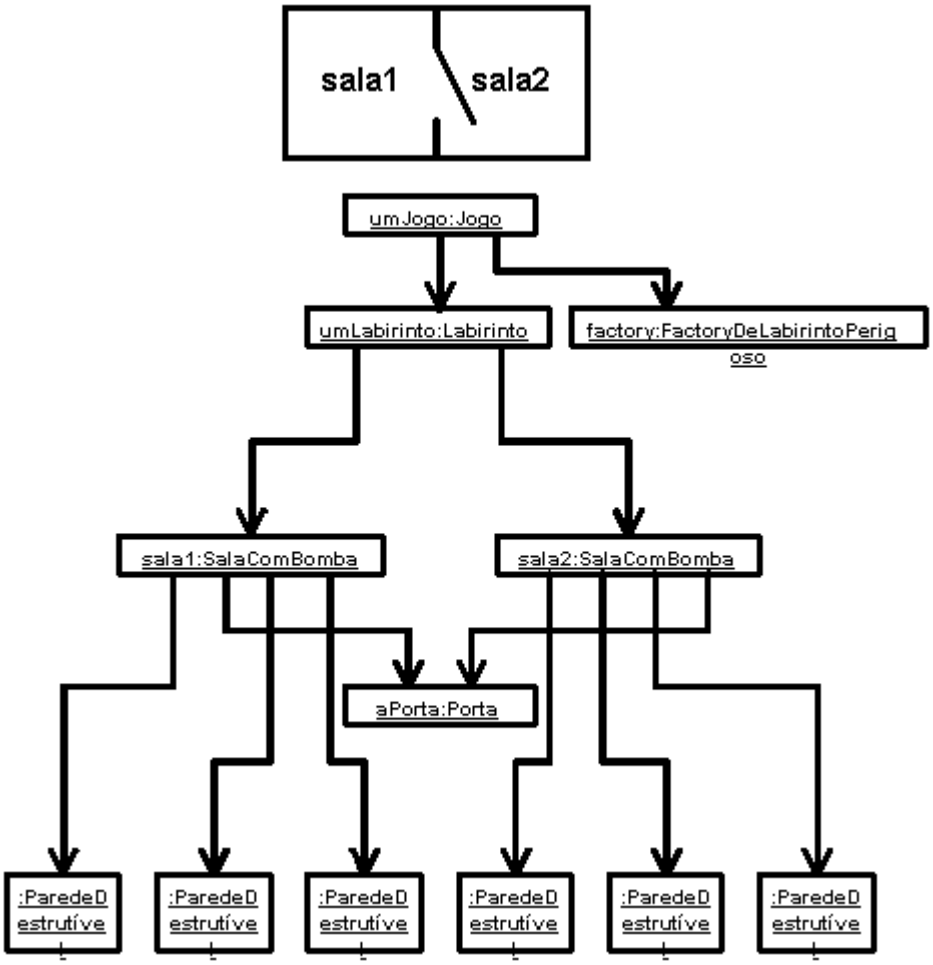
- Para criar um labirinto perigoso, criamos uma *outra* factory concreta como subclasse de FactoryDeLabirinto

```
public class FactoryDeLabirintoPerigoso extends FactoryDeLab
  public ParedeIF criaParede() {
    return new paredeDestruível();
  }
  public SalaIF criaSala(int númeroDaSala) {
    return new salaComBomba(númeroDaSala);
  }
}
```

- Finalmente, podemos jogar:

```
JogoIF umJogo = new Jogo();
FactoryDeLabirinto factory = FactoryDeLabirinto.getInstanc
jogo.montaLabirinto(factory);
```

- Poderíamos jogar um jogo encantado com uma mudança muito simples ao código acima
- O diagrama de objetos tem um objeto a mais comparado com a versão original



Discussão geral de padrões de criação

- Ajudam a deixar o sistema independente de como seus objetos são criados, compostos e representados
- São dois tipos:
  - Padrões de criação via classes
    - Usam herança para variar a classe que é instanciada
    - Exemplo: Factory Method
  - Padrões de criação via objetos
    - Delegam a instanciação para outro objeto
    - Exemplo: Abstract Factory
- Composição é usada mais que herança para estender funcionalidade e padrões de criação ajudam a lidar com a complexidade de criar comportamentos
  - Em vez de codificar um comportamento estaticamente, definimos pequenos

comportamentos padrão e usamos composição para definir comportamentos mais complexos
    - Isso significa que instanciar um objeto com um comportamento particular requer mais do que simplesmente instanciar uma classe
    - Eles escondem como instâncias das classes concretas são criadas e juntadas para gerar "comportamentos" (que podem envolver vários objetos compostos)
    - Os padrões mostrados aqui mostram como encapsular as coisas de forma a simplificar o problema de instanciação
- Os padrões de criação discutem temas recorrentes:
    - Eles encapsulam o conhecimento das classes concretas que são instanciadas
        - Lembre que preferimos nos "amarrar" a interfaces (via interface ou classes abstratas) do que a classes concretas
        - Isso promove a flexibilidade de mudança (das classes concretas que são instanciadas)

Perguntas finais para discussão

- Na seção de implementação deste padrão, os autores (Gamma et al.) discutem a idéia de *definir factories extensíveis*. Já que uma Abstract Factory consiste de Factory Methods e que cada Factory Method tem apenas uma assinatura, isto significa que o Factory Method só pode criar um objeto de uma única forma?
- Considere o exemplo de construção de labirintos. O FactoryDeLabirinto contém um método criaSala() que recebe um inteiro como parâmetro, representando o número da sala. O que acontece se você também quer especificar o tamanho e cor da sala? Isto significa que você deveria criar um novo Factory Method para o FactoryDeLabirinto, permitindo passar como parâmetros o número da sala, tamanho e cor para um outro método criaSala()?
- Claro que nada impediria que alterasse a cor e tamanho da sala *depois* que tiver sido instanciada, mas isso deixaria o código mais "cheio e sujo",

especialmente se você estiver criando e configurando muitos objetos. Como reter o FactoryDeLabirinto e usar apenas um método criaSala() e acomodar diferentes parâmetros usados por criaSala() para criar e configurar objetos do tipo Sala?

- No exemplo acima (FactoryDeLabirinto), não há estado a ser mantido na instância única do singleton. Você acha que um singleton é realmente necessário aqui? O que ocorreria se permitissemos a criação de mais de uma instância?
- No exemplo acima (FactoryDeLabirinto), o que ocorre se getInstance("...") for chamado mais de uma vez com parâmetros diferentes?

programa