Design Patterns
ABSTRACT FACTORY

EMERSON BARROS DE MENESES

Breve Histórico Sobre Design Patterns

A origem dos Design Patterns (Padrões de Desenho ou ainda Padrões de Projeto) vem do trabalho de um arquiteto chamado Christopher Alexander, no final da década de 70. Ele escreveu dois livros, inicialmente, A Pattern Language [Alex77] e A Timeless Way of Building [Alex79], nos quais ele exemplificava o uso e descrevia seu raciocínio para documentar os padrões. Em 1995, um grupo de quatro profissionais escreveu e lançou o livro "Design Patterns: Elements of Reusable Object-Oriented Software" [Gamma95] e traduzido para “Padrões de Projeto: soluções reutilizáveis de software orientado a objetos” [2000], um catálogo com 23 padrões de desenho (design patterns). Os autores: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, ficaram mais conhecidos como A Gangue dos Quatro (Gang Of Four ou GoF), considerados os maiores entusiastas dos Design Patterns. Cujo mesmo livro foi utilizado na criação deste trabalho.

Breve resumo sobre “o que é um Design Patterns”?

Os Design Patterns são padrões de classes e de relacionamentos entre as mesmas que aparecem de forma freqüente em projetos de software. Tais padrões são categorizados para atender a soluções específicas. Sua utilização é uma atividade que simplifica a reutilização de software. Os padrões aparecem em situações especiais dentro de um sistema como “descrições de objetos e classes comunicantes que são customizadas um contexto particular”. Este descreve uma solução comprovada para um problema, são mais abstratos, menores e menos específicos que frameworks que por sua vez é um conjunto de classes que pode ser utilizado para um tipo específico de projeto de software (análise de domínio, biblioteca de reuso).

Uma primeira visualização (entendimento) corresponde à tríade modelo, vista e controlador (MVC). Procuram estabelecer uma distinção rígida entre estes elementos de modo a torná-los independentes. Ou seja, modelo, vista e controlador podem ser modificados de forma independente (desacoplamento).

Patterns são dispositivos que permitem que os programas compartilhem conhecimento sobre o seu desenho. Quando programamos, encontramos muitos problemas

que ocorrem, ocorreram e irão ocorrer novamente. A questão que nos perguntamos agora é *como nós vamos solucionar este problema desta vez?* Documentar um padrão (pattern) é uma maneira de poder reusar e possivelmente compartilhar informação que aprendeu sobre a melhor maneira de se resolver um problema de desenho de software. O catálogo de padrões do GoF (Gang Of Four), como dito, contém 23 padrões e está, basicamente, dividido em três seções: Creational (Criacional), Structural (Estrutural), Behavioral (Comportamental). Neste trabalho iremos elencar pontos referentes ao Padrão Criacional Abstract Factory.

Abstract Factory (Design Patterns Creational)

Introdução

- Padrão que cria um conjunto de famílias de objetos com características comuns. A escolha pela adoção de uma determinada família e a conseqüente instanciação dos objetos pode ser definida em tempo de execução.

Objetivo

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

Escopo

- De Objeto

Propósito

- De criação

Também chamado de

- Kit

Resumo

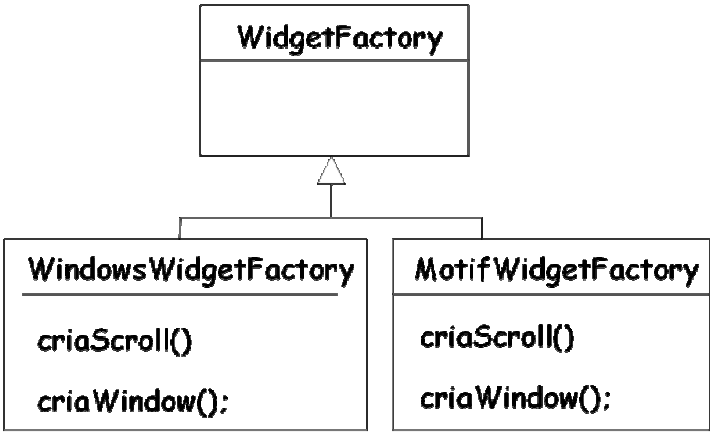
- Parece semelhante ao padrão Factory Method, mas
    - Em vez do cliente (que quer criar objetos sem saber as classes exatas) chamar um método de criação (Factory Method, escopo de Classe), ele de

alguma forma *possui* um objeto (uma Abstract Factory) e usa este objeto para chamar os métodos de criação

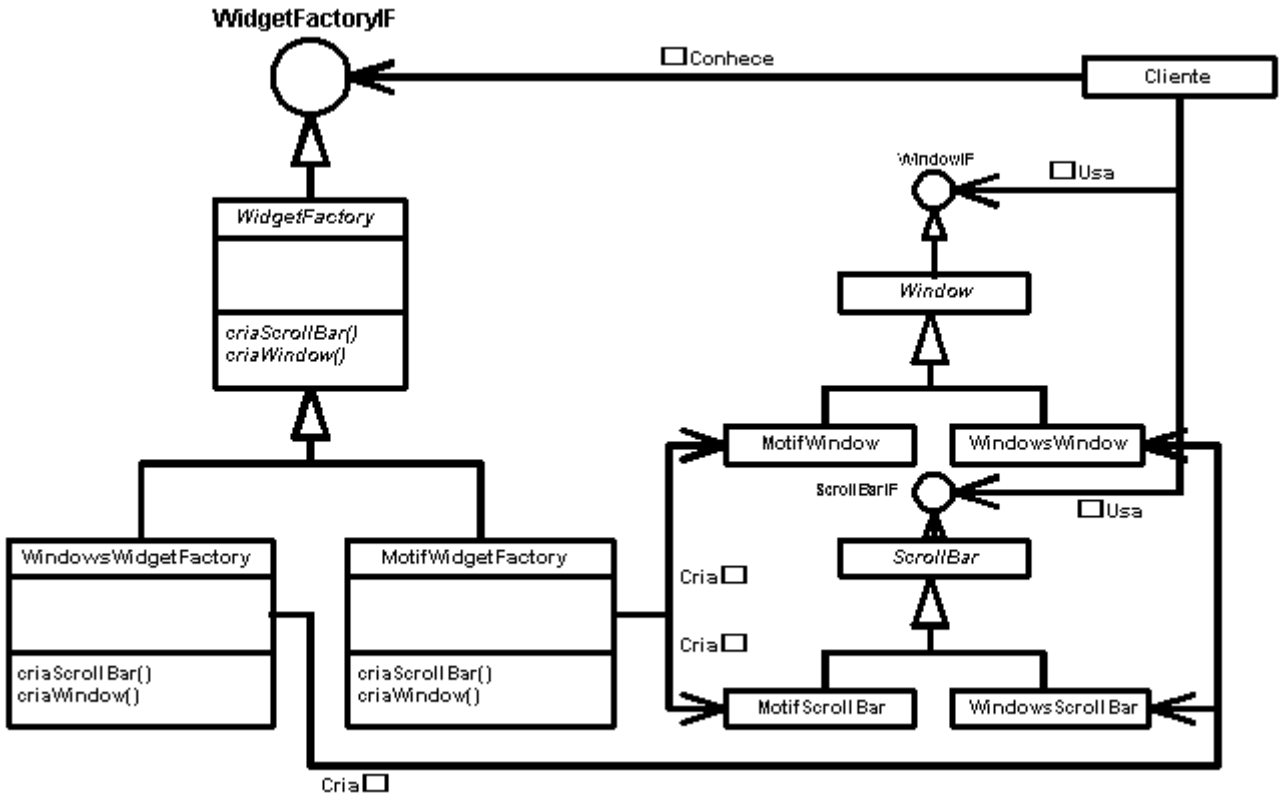
        - Onde Factory Method quer que você *seja* diferente (via herança) para criar objetos diferentes, o Abstract Factory quer que você *tenha* algo diferente
    - Se ele possuir uma referência a uma Abstract Factory diferente, toda a criação será diferente
    - O fato de todos os métodos de criação estarem 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 pode criar uma janela estilo Windows e um menu estilo Motif
- Java já resolveu este problema internamente no package awt usando Abstract Factory e não precisa se preocupar com isso
    - Porém, poderia estar usando C++ e precisaria cuidar disso pessoalmente
- 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:


WidgetFactory
WindowsWidgetFactory
criaScroll()
criaWindow();
MotifWidgetFactory
criaScroll()
criaWindow();

Mais detalhadamente seria:

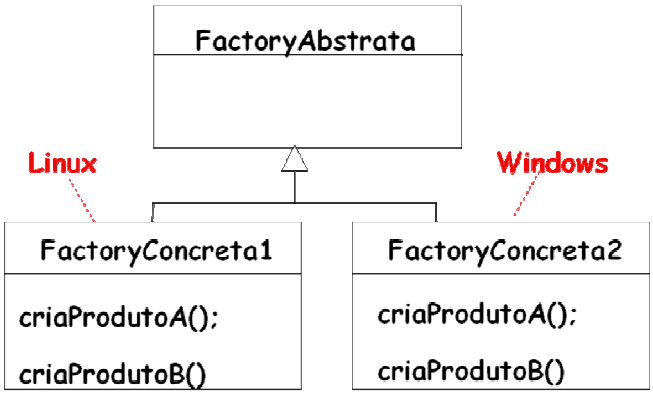


Quando usar o padrão Abstract Factory?

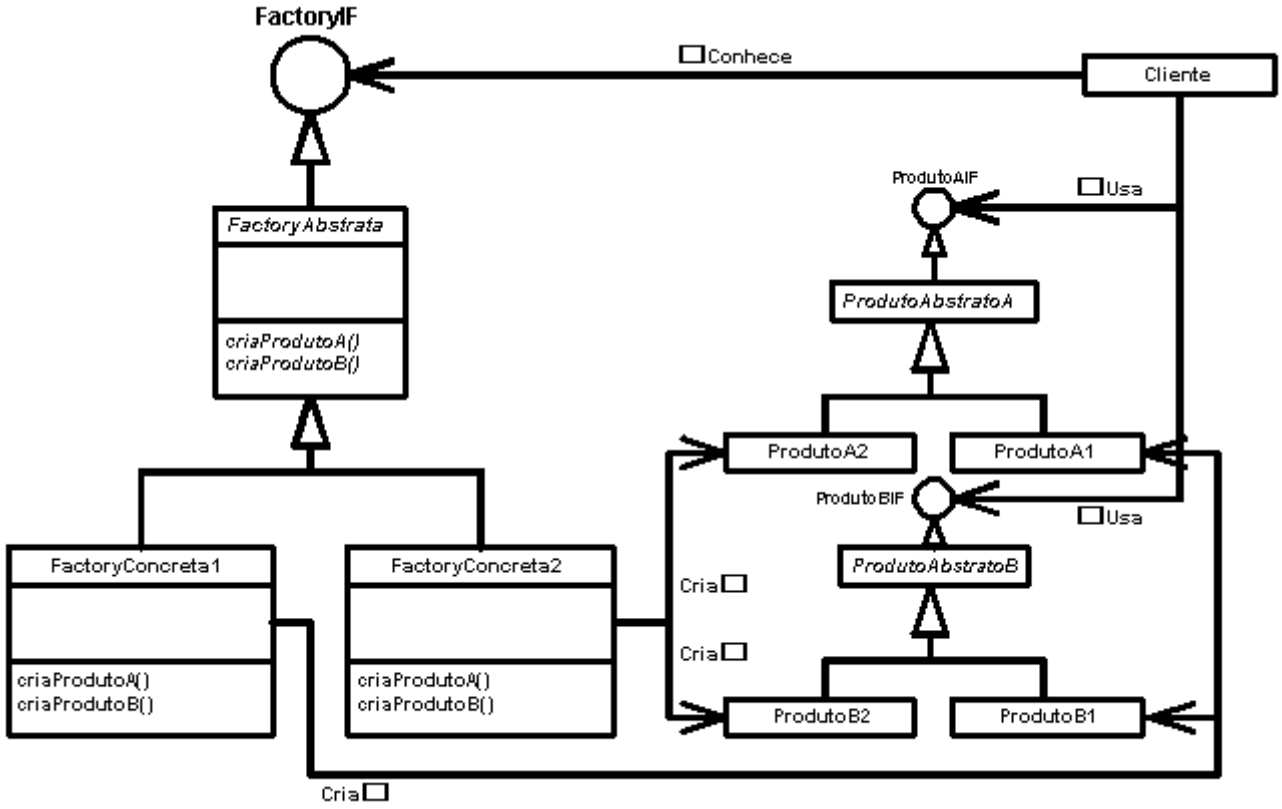
- Quando um sistema deve ser independente de como seus produtos são criados, compostos e representados
- Quando um sistema deve ser configurado com uma entre várias famílias de produtos
- Quando uma família de produtos relacionados foi projetada para uso conjunto e você deve implementar essa restrição
- Quando deseja 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



Mais detalhadamente seria:



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 da classe Jogo entram na factory abstrata FactoryDeLabirinto
    - Também possuem um default, o que significa que FactoryDeLabirinto também é uma factory concreta

```
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 FactoryDeLabirintoIF {
  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);
  }
}
```

- montaLabirinto recebe um FactoryDeLabirintoIF como parâmetro e cria um labirinto

```
public class Jogo implements JogoIF {
  // Essa função não tem new: ela usa uma Abstract Factory
  public LabirintoIF montaLabirinto(FactoryDeLabirintoIF factory) {
    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, foi criado uma factory concreta como subclasse de FactoryDeLabirinto

```
public class FactoryDeLabirintoEncantado extends FactoryDeLabirinto {
  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() {
    ...
  }
```

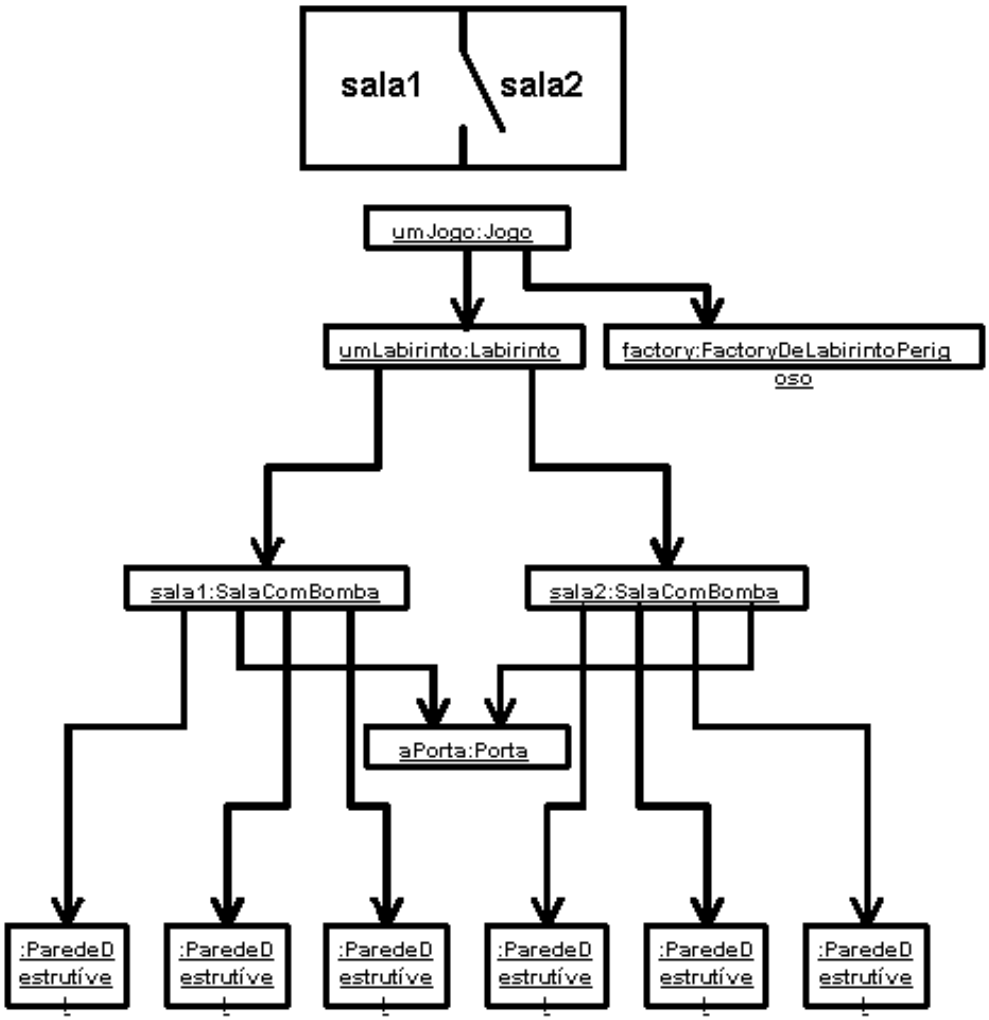
```
}
```

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

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

- Finalmente, é possível jogar:

```
  JogoIF umJogo = new Jogo();
  FactoryDeLabirinto factory =
FactoryDeLabirinto.getInstance("perigoso");
  jogo.montaLabirinto(factory);
```



Questões gerais 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)

Bibliografia:

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

http://www.guj.com.br/articles/137

http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/pat/abstractfactory.htm

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Padrões de Projeto: soluções reutilizáveis de software orientado a objetos. Ed. 1. Bookman, 2000. ISBN 9788573076103.

Design Patterns - ABSTRACT FACTORY

Por Emerson Barros de Meneses