Padrões de Projeto de Software Orientado a Objetos

ACH 2006 – Engenharia de Sistemas de Informação I

Baseado nos slides dos Profs. Fabio Kon - IME/USP e Jan Vitek -Purdue University.

O problema

- Projetar software para reúso é difícil porque deve-se procurar por:
  - Uma boa decomposição do problema e a abstração correta.
  - Flexibilidade, modularidade e elegância.
- Projetos freqüentemente emergem de um processo iterativo (tentativas e muitos erros).

O problema

- A boa notícia é que bons projetos existem:
  - na verdade, eles apresentam características recorrentes,
  - mas eles quase nunca são idênticos.
- Os projetos podem ser descritos, codificados ou padronizados?
- Se sim,
  - isto iria diminuir a fase de tentativa e erro e
  - sofwares melhores seriam produzidos mais rápidos.

Padrões de Projeto de Software OO

- Também conhecidos como
  - *Padrões de Desenho de Software OO*
  - ou simplesmente como *Padrões*.

A Inspiração

- A idéia de padrões foi apresentada por Christopher Alexander em 1977 no contexto de Arquitetura (de prédios e cidades):

  Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca implementa-la duas vezes da mesma forma.

A Inspiração

- Livros
  - *The Timeless Way of Building*
  - *A Pattern Language: Towns, Buildings, and Construction*
    - serviram de inspiração para os desenvolvedores de software.

Catálogo de soluções

- Um padrão encerra o conhecimento de uma pessoa muito experiente em um determinado assunto de uma forma que este conhecimento pode ser transmitido para outras pessoas menos experientes.
- Outras ciências (p.ex. química) e engenharias possuem catálogos de soluções.
- Desde 1995, o desenvolvimento de software passou a ter o seu primeiro catálogo de soluções para projeto de software: o livro GoF.

Gang of Four (GoF)

- E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Versão em português disponível na biblioteca da EACH.

Gang of Four (GoF)

- Passamos a ter um vocabulário comum para conversar sobre projetos de software.
- Soluções que não tinham nome passam a ter nome.
- Ao invés de discutirmos um sistema em termos de pilhas, filas, árvores e listas ligadas, passamos a falar de coisas de muito mais alto nível como Fábricas, Fachadas, Observador, Estratégia, etc.

Gang of Four (GoF)

- A maioria dos autores eram entusiastas de Smalltalk, principalmente Ralph Johnson.
- Mas acabaram baseando o livro em C++ para que o impacto junto à comunidade de CC fosse maior. E o impacto foi enorme, o livro vendeu centenas de milhares de cópias.

O Formato de um padrão

- Todo padrão inclui
  - Nome
  - Problema
  - Solução
  - Conseqüências / Forças

- Existem outros tipos de padrões mas vamos nos concentrar no GoF.

O Formato dos padrões no GoF

- Nome (inclui número da página)
  - um bom nome é essencial para que o padrão caia na boca do povo
- Objetivo / Intenção
- Também conhecido como
- Motivação
  - um cenário mostrando o problema e a necessidade da solução
- Aplicabilidade
  - como reconhecer as situações nas quais o padrão é aplicável

O Formato dos padrões no GoF

- Estrutura
  - uma representação gráfica da estrutura de classes do padrão (usando OMT91).
  - Em algumas vezes diagramas de interação (Booch 94) são utilizados
- Participantes
  - as classes e objetos que participam e quais são suas responsabilidades
- Colaborações
  - como os participantes colaboram para exercer as suas responsabilidades

O Formato dos padrões no GoF

- Conseqüências
  - vantagens e desvantagens, *trade-offs*
- Implementação
  - com quais detalhes devemos nos preocupar quando implementamos o padrão
  - aspectos específicos de cada linguagem
- Exemplo de Código
  - no caso do GoF, em C++ (a maioria) ou Smalltalk

O Formato dos padrões no GoF

- Usos Conhecidos
  - exemplos de sistemas reais de domínios diferentes onde o padrão é utilizado
- Padrões Relacionados
  - quais outros padrões devem ser usados em conjunto com esse
  - quais padrões são similares a este, quais são as dierenças

Tipos de Padrões de Projeto

- Categorias de Padrões do GoF
  - Padrões de Criação
  - Padrões Estruturais
  - Padrões Comportamentais

- Vamos ver exemplos de cada um deles.

- Na aula de hoje:
  - Fábrica de método (Factory Method (112)
  - Fábrica Abstrata (Abstract Factory (87))
    - padrão de Criação de objetos

Labirinto (maze)

```
class Maze {
Maze create() {
Maze maze = new Maze();
Room r1 = new Room(1);
Room r2 = new Room(2);
Door door = new Door(r1, r2);
maze.addRoom(r1); maze.addRoom(r2);
r1.setSide(North, new Wall()); r1.setSide(East, door);
r1.setSide(South, new Wall()); r1.setSide(West, new
  Wall());
// falta inserir as paredes e prota da sala r2
return maze;
}
}
```

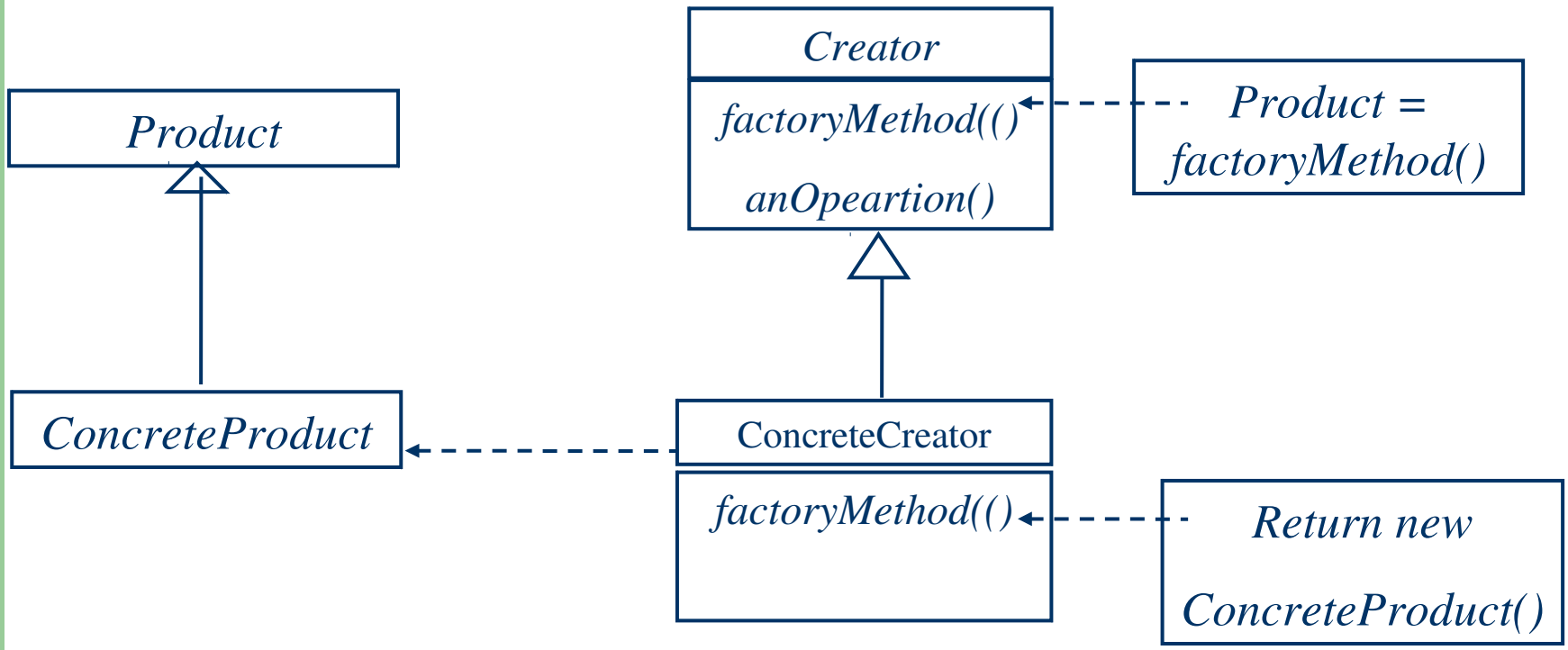
Factory Method (112)

- Intenção
  - Definir uma interface para criar um objeto, mas deixar por conta das subclasses decidirem qual objeto criar.
- Motivação
  - Os *frameworks* freqüentemente precisam instanciar classes. Porem, para usos diferentes do framework, classes concretas diferentes podem ser utilizadas para criar os objetos.

Factory Method (112)

- Aplicabilidade
  - Use o padrão FM se uma classe não consegue antecipar os objetos que ela tem que criar ou a classe quer que suas subclasses especifiquem os objetos que ela cria.

Factory Method - Estrutura



Factory Method (112)

- Conseqüências:
  - Evitando especificar o nome da classe concreta e os detalhes da sua criação o código do cliente fica mais flexível .
  - O cliente é dependente apenas da interface.
  - A construção de objetos requer uma classe a mais em alguns casos.

Factory Method (112)

- Implementação
  - A classe criadora é abstrata e não implementa métodos de criação; por isso, deve ser estendida por uma subclasse.
  - A classe criadora é concreta e fornece uma implementação *default*. Se for necessário uma classe diferente então a classe criadora deve ser estendida.

Factory Method (112)

- Implementação
  - Um Factory Method deve ser capaz de criar variantes?
    - Se sim, o seu método deve receber um parâmetro.

Factory Method (112)

- Implementação
  - Um exemplo de *Factory Method* parametrizado.

```
class Creator {
    public Product create(ProductID id) {
      if (id == MINE) return new MyProduct();
      if (id == YOURS) return new YourProduct();
      return null;
}
}
```

Factory Method (112)

- Implementação
  - Que pode ainda ser estendido para adicionar mais casos.

```
class MyCreator extends Creator {
    public Product create(ProductID id) {
      if (id == YOURS) return new MyProduct();
      if (id == THEIRS) return new OurProduct();
      return super.create(id);
}
```

Factory Method (112)

- Implementação
  - Com genéricos os tipos podem ser usados para implementar o factory method.

```
abstract class Creator<T extends Product> {public T
  create();}
class MyCreator<T extends Product> {
     public T create() { return new T();}
}
MyCreator<YourProduct> factory = new
  MyCreator<YourProduct>();
```

Factory Method (112)

- Revisitando o exemplo do labirinto.

```
class Maze {
    public Maze create();
    public Maze makeMaze() { return new Maze(); }
    public Maze makeRoom(int n) { return new
   Room(n); }
    public Maze makeWall() { return new Wall(); }
    public Maze makeDoor(Room r, Room r2) {return new
   Door(r, r2);}
}
```

Factory Method (112)

- Revisitando o exemplo do labirinto.

```
Maze create() {
  Maze maze = makeMaze();
  Room r1 = makeRoom(1);
  Room r2 = makeRoom(2);
  Door door = makeDoor(r1, r2);
  maze.addRoom(r1); maze.addRoom(r2);
  r1.setSide(North, makeWall(); r1.setSide(East,
  door);
 (continua)
```

Factory Method (112)

- Revisitando o exemplo do labirinto.

```
    r1.setSide(South, makeWall());
    r1.setSide(West, makeWall());
    r2.setSide(North, makeWall();
    r2.setSide(East, makeWall());
    r2.setSide(South, makeWall());
    r2.setSide(West, door);
    return maze;
}
```

Factory Method (112)

- Revisitando o exemplo do labirinto.

```
class MazeBombsGame extends Maze {
public Maze makeRoom(int n)
{
return new RoomWithABomb(n);
}
public Maze makeWall()
{
return new BombedWall(); }
}
```

Fábrica Abstrata
Abstract Factory (95)

Objetivo:

- prover uma interface para criação de famílias de objetos relacionados sem especificar sua classe concreta.

Abstract Factory - Motivação

- Considere uma aplicação com interface gráfica que é implementada para plataformas diferentes (Motif para UNIX e outros ambientes para Windows e MacOS).
- As classes que implementam os elementos gráficos não podem ser definidas estaticamente no código. Precisamos de uma implementação diferente para cada ambiente. Até em um mesmo ambiente, gostaríamos de dar a opção ao usuário de implementar diferentes aparências (*look-and-feels*).

Abstract Factory - Motivação

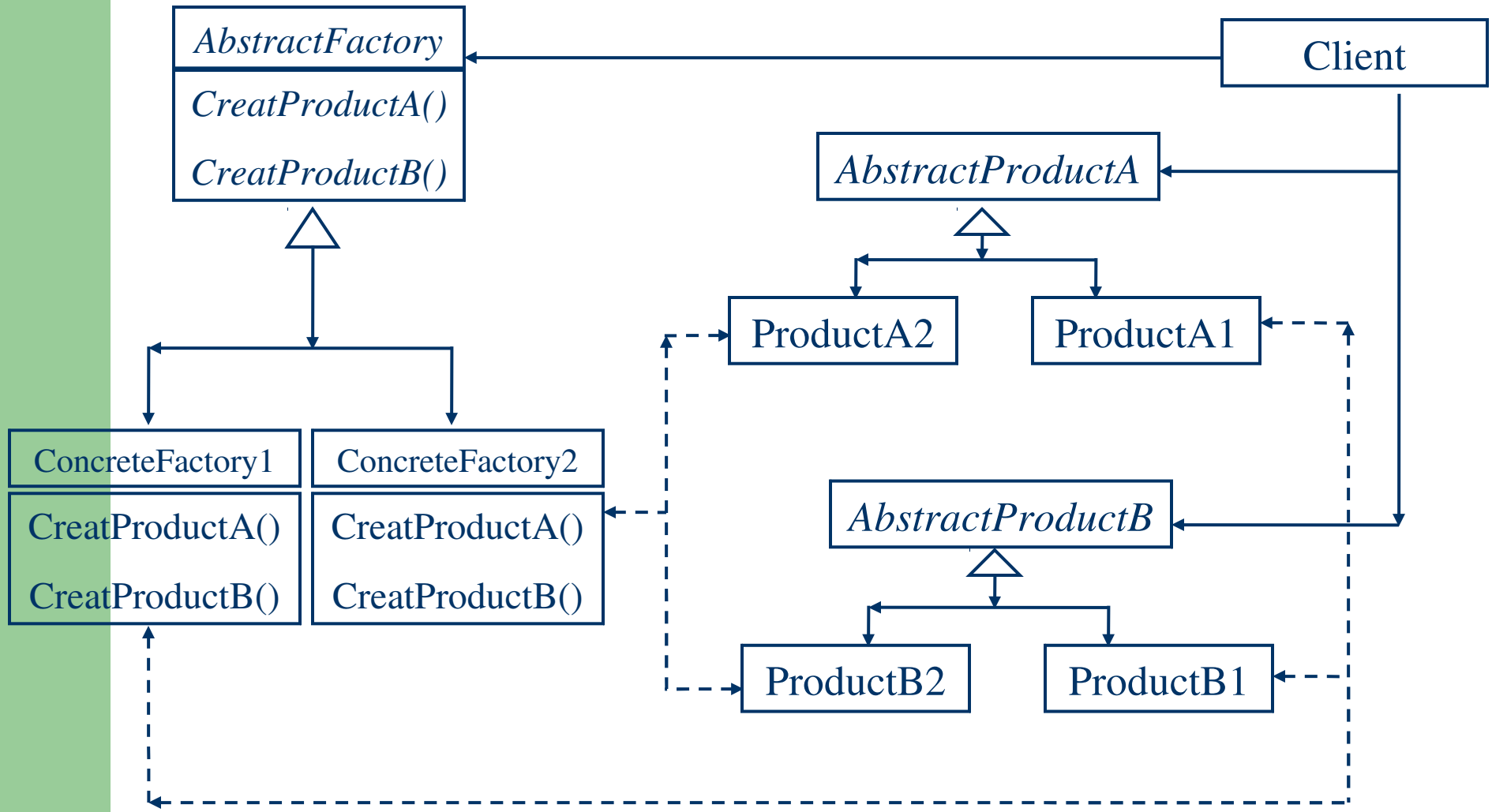
- Podemos solucionar este problema definindo uma classe abstrata para cada elemento gráfico e utilizando diferentes implementações para cada aparência ou para cada ambiente.
- Ao invés de criarmos as classes concretas com o operador **`new`**, utilizamos uma Fábrica Abstrata para criar os objetos em tempo de execução.
- O código cliente não sabe qual classe concreta utilizamos.

Abstract Factory - Aplicabilidade

Use uma fábrica abstrata quando:

- um sistema deve ser independente da forma como seus produtos são criados e representados;
- um sistema deve poder lidar com uma família de vários produtos diferentes;
- você quer prover uma biblioteca de classes de produtos mas não quer revelar as suas implementações, quer revelar apenas suas interfaces.

Abstract Factory - Estrutura



Abstract Factory - Participantes

- **`AbstractFactory`** (WidgetFactory)
- **`ConcreteFactory`** (MotifWidgetFactory, WindowsWidgetFactory)
- **`AbstractProduct`** (Window, ScrollBar)
- **`ConcreteProduct`** (MotifWindow, MotifScrollBar, WindowsWindow, WindowsScrollBar)
- **`Client`** - usa apenas as interfaces declaradas pela AbstractFactory e pelas classes AbstratProduct

Abstract Factory - Colaborações

- Normalmente, apenas uma instância de **`ConcreteFactory`** é criada em tempo de execução.
- Esta instância cria objetos através das classes **`ConcreteProduct`** correspondentes a uma família de produtos.
- Uma **`AbstractFactory`** deixa a criação de objetos para as suas subclasses **`ConcreteFactory`**.

Abstract Factory - Conseqüências

O padrão

- **isola as classes concretas dos clientes**;
- **facilita a troca de famílias de produtos** (basta trocar uma linha do código pois a criação da fábrica concreta aparece em um único ponto do programa);
- **promove a consistência de produtos** (não há o perigo de misturar objetos de famílias diferentes);
- **dificulta a criação de novos produtos ligeiramente diferentes** (pois temos que modificar a fábrica abstrata e todas as fábricas concretas).

Abstract Factory - Implementação

- **Fábricas abstratas em geral são implementadas com *Factory method* (112).**
- **Na fábrica abstrata, cria-se um método fábrica para cada tipo de produto. Cada fábrica concreta implementa o código que cria os objetos de fato.**

Abstract Factory - Implementação

Se tivermos muitas famílias de produtos, teríamos um excesso de classes “fábricas concretas”.

Para resolver este problema, podemos usar o Prototype (121): criamos um dicionário mapeando tipos de produtos em instâncias prototípicas destes produtos.

Então, sempre que precisarmos criar um novo produto pedimos à sua instância prototípica que crie um clone (usando um método como clone() ou copy()).

Abstract Factory - Implementação

- Em linguagens dinâmicas como Smalltalk onde classes são objetos de primeira classe, não precisamos guardar uma instância prototípica, guardamos uma referência para a própria classe e daí utilizamos o método new para construir as novas instâncias.

Abstract Factory - Implementação

5. Definindo fábricas extensíveis.

- normalmente, cada tipo de produto tem o seu próprio método fábrica; isso torna a inclusão de novos produtos difícil.

- solução: usar apenas um método fábrica
  - Product make (string thingToBeMade)

- isso aumenta a flexibilidade mas torna o código menos seguro (não teremos verificação de tipos pelo compilador).

Revisitando o labirinto

```
class MazeFactory {

public Maze makeMaze() { return new Maze(); }

public Room makeRoom(int n) { return new
  Room(n); }

public Wall makeWall() { return new Wall(); }

public Door makeDoor(Room r1, Room r2) { return
  new Door(r1, r2);}

}
```

Revisitando o labirinto (código cliente)

```
Maze create(MazeFactory factory) {

Maze maze = factory.makeMaze();

Room r1 = factory.makeRoom(1);

Room r2 = factory.makeRoom(2);

Door door = factory.makeDoor(r1, r2);

maze.addRoom(r1); maze.addRoom(r2);

r1.setSide(North, factory.makeWall());

  r1.setSide(East, door); (continua)
```

Revisitando o labirinto

```
r1.setSide(South, factory.makeWall());
  r1.setSide(West, factory.makeWall());
r2.setSide(North, factory.makeWall());
  r2.setSide(East, factory.makeWall());
r2.setSide(South, factory.makeWall());
  r2.setSide(West, door);
return maze;
}
```

Revisitando o labirinto

```
class EnchantedMazeFactory extends MazeFactory

{public EnchantedMazeFactory();

Room makeRoom(int n) { return new EnchantedRoom(n); }

Door makeDoor(Room r1, Room r2){ return new
   MagicDoor(r1,r2);}

class BombedMazeFacotry extends MazeFactory {

public BombedMazeFactory();

Room makeRoom(int n) { return new RoomWithABomb(n); }

Door makeWall(){ return new BombedWall(r1,r2);}
```

Abstract Factory - Usos Conhecidos

- InterViews usa fábricas abstratas para encapsular diferentes tipos de aparências para sua interface gráfica
- ET++ usa fábricas abstratas para permitir a fácil portabilidade para diferentes ambientes de janelas (XWindows e SunView, por exemplo)
- Sistema de captura e reprodução de vídeo feito na UIUC usa fábricas abstratas para permitir portabilidade entre diferentes placas de captura de vídeo.
- Em linguagens dinâmicas como Smalltalk (e talvez em POO em geral) classes podem ser vistas como fábricas de objetos.

Abstract Factory - Padrões Relacionados

- Fábricas abstratas são normalmente implementadas com métodos fábrica (FactoryMethod (107)) mas podem também ser implementados usando Prototype (117).
- O uso de protótipos é particularmente importante em linguagens não dinâmicas como C++ e em linguagens "semi-dinâmicas" como Java.
- Uma fábrica concreta é normalmente um Singleton (127)

Os 23 Padrões do GoF
Criação

- ***Abstract Factory***
- Builder
- ***Factory Method***
- Prototype
- Singleton

Os 23 Padrões do GoF

Estruturais

- Adapter
- Bridge
- *Composite*
- Decorator
- *Façade*
- Flyweight
- Proxy

Os 23 Padrões do GoF
Comportamentais

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- *Observer*
- State
- *Strategy*
- Template Method
- Visitor

Para próxima aula

- Dar uma olhada no GoF
  - a biblioteca possui algumas cópias
- Buscar por “GoF patterns” no google

Recapitulando

- **Voltando ao Christopher Alexander:**

  Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca implementa-lá duas vezes da mesma forma.

- **Talvez a última parte não seja sempre desejável.**

Referências

- Duel, Michael -- “Non-Software Examples of Software Design Patterns”, http://wwwswt.informatik.uni-rostock.de/deutsch/Lehre/Uebung/Beispiele/PatternExamples/patexamples.htm.
- Houston, Vince -- “Design Patterns”, http://www.vincehuston.org/dp/.
- E. Gamma, R. Helm, R. Johnson e J. Vlissides. *Padrões de projetos: soluções reusáveis de software orientado a objetos*. Porto Alegre: Bookman, 2000.