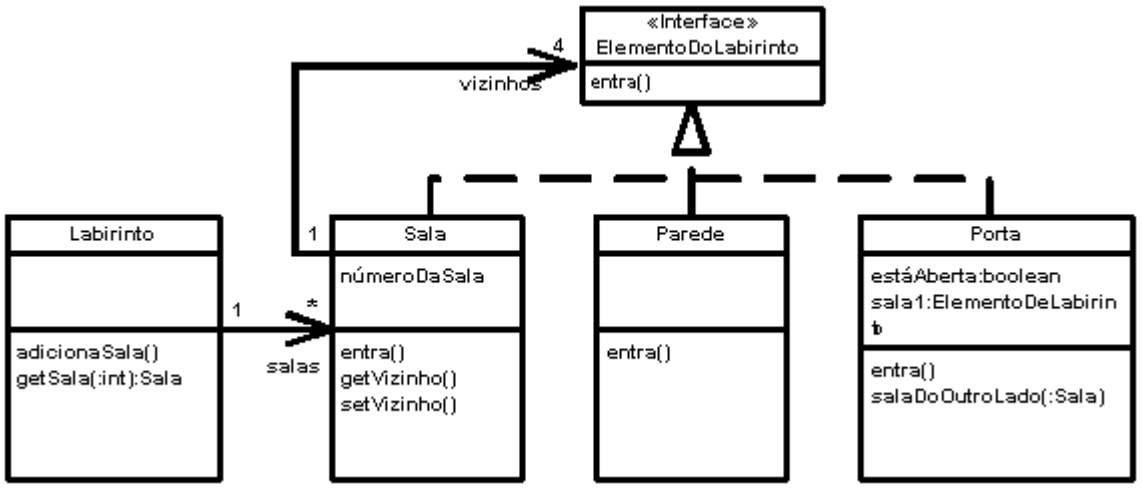
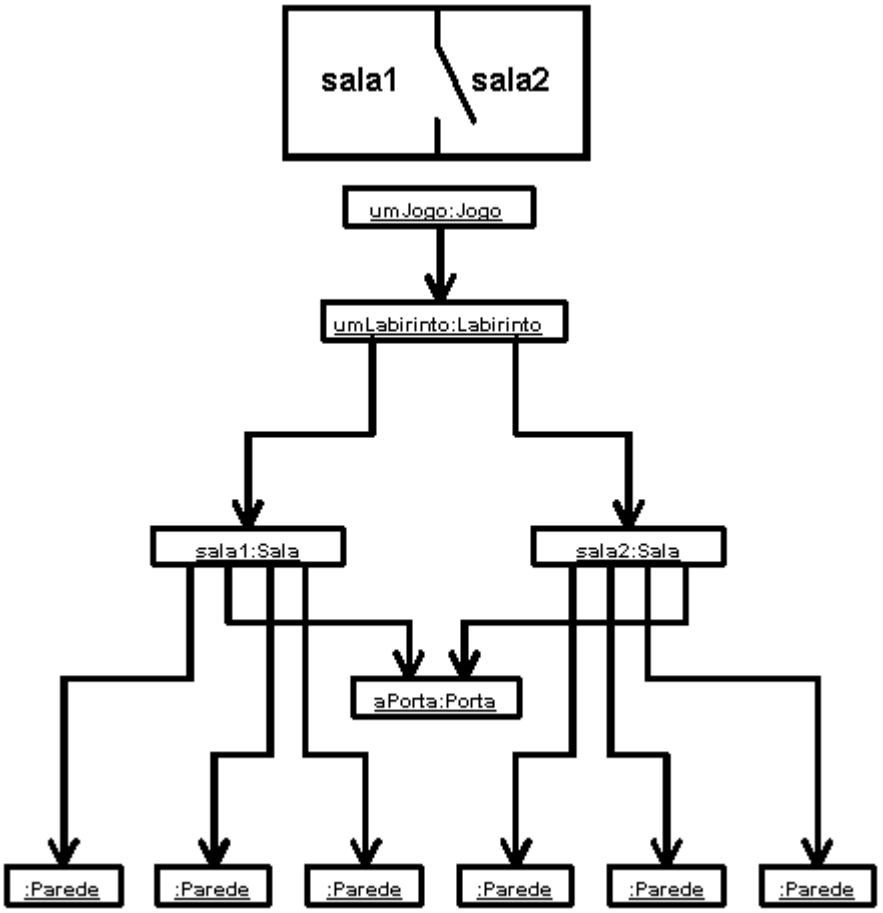
Factory Method

Introdução aos Padrões de Criação: Construindo Labirintos

- GoF classifica os padrões em Padrões de Criação, Estruturais e de Comportamento
- Padrões de criação abstraem o processo de instanciação de objetos
- Usaremos a construção de labirintos para um jogo via computador para mostrar alguns padrões de criação
  - Ignoraremos muitos detalhes do labirinto (o que pode estar no labirinto, os jogadores, etc.)
  - Foco na *criação* dos labirintos
- Um labirinto é um conjunto de salas
  - Uma sala conhece seus quatro vizinhos
  - Vizinhos podem ser outra sala, uma parede ou uma porta para outra sala
- As classes importantes são Sala, Porta e Parede
  - Só trataremos as partes das classes que interessam para a criação do labirinto
- O diagrama de classes segue abaixo (em UML)
  - Se precisar de um resumo de UML, ver aqui



- Um diagrama de objetos segue para um pequeno labirinto



- Cada sala tem quatro vizinhos
  - Usamos Norte, Sul, Leste, Oeste para referenciá-los
- A interface ElementoDeLabirinto é implementada por todos os componentes de um labirinto
  - Tem um método entra() cujo significado depende onde se está entrando
    - Se for uma sala, a localização do jogador muda
    - Se for uma porta aberta, você vai para outra sala, caso contrário se machuca
    - Se for uma parede, você se machuca

```
public interface ElementoDeLabirinto {
  public void entra();
}
```

- Exemplo: se você estiver numa sala e quiser implementar a operação "vá para o Leste", o jogo determina qual ElementoDeLabirinto está do lado Leste e chama entra() deste objeto

- O método entra() da subclasse específica determina o que ocorre
- Num jogo real, entra() poderia aceitar o objeto jogador como parâmetro

- Sala é a classe que implementa ElementoDeLabirinto e define as relações-chave entre objetos
  - Mantém referências para 4 outros ElementoDeLabirinto
  - Armazena um número de sala para identificar as salas do labirinto

```
public class Sala implements ElementoDeLabirinto {
  private ElementoDeLabirinto[] vizinhos =
          new ElementoDeLabirinto[4];
  private int númeroDaSala;

  public Sala(int númeroDaSala) {
    ...
  }
  public void entra() {
    ...
  }
  public ElementoDeLabirinto getVizinho(int direção) {
    ...
  }
  public void setVizinho(int direção,
                         ElementoDeLabirinto vizinho) {
    ...
  }
}

public class Parede implements ElementoDeLabirinto {
  public Parede() {
    ...
  }
  public void entra() {
    ...
  }
}

public class Porta implements ElementoDeLabirinto {
  private ElementoDeLabirinto sala1, sala2;
  private boolean estáAberta;

  public Porta(ElementoDeLabirinto sala1,
```

```
                ElementoDeLabirinto sala2) {
    ...
  }
  public void entra() {
    ...
  }
  public ElementoDeLabirinto salaDoOutroLado(ElementoDeLabir
      ...
  }
}
```

- Também precisamos de uma classe Labirinto para representar uma coleção de salas
  - A classe Labirinto pode localizar uma sala dado seu número com o método getSala()

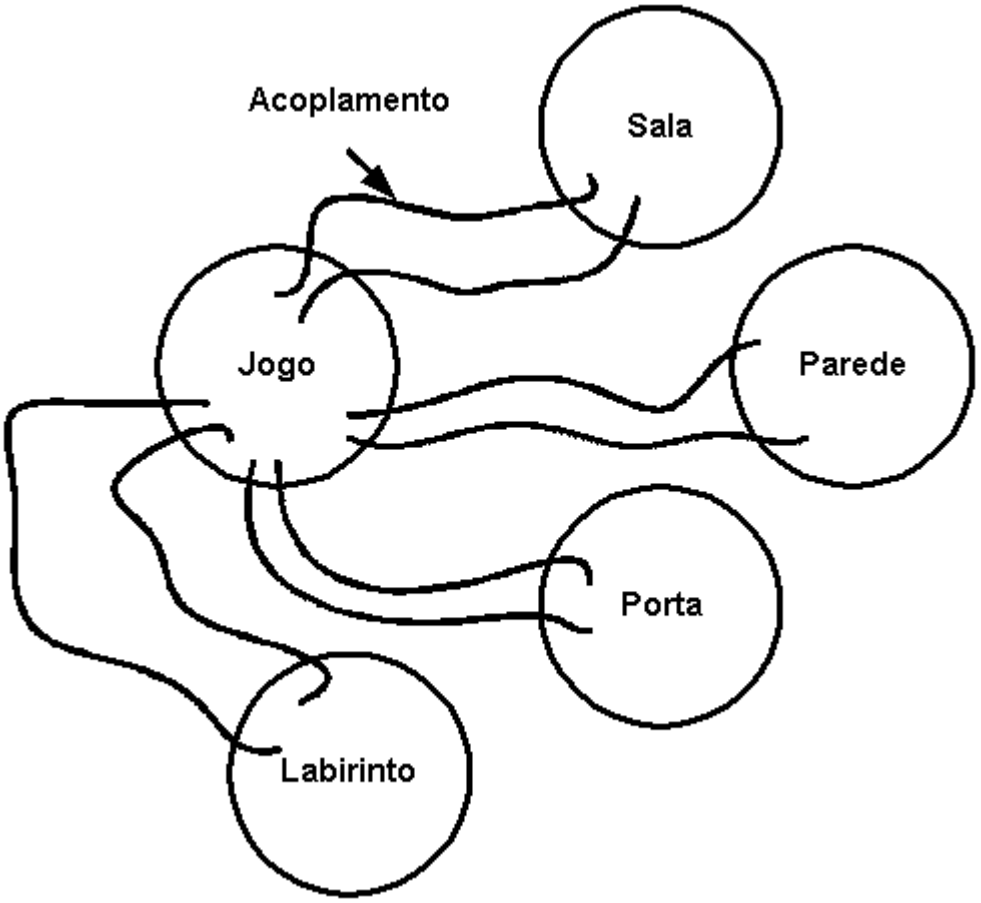
```
public class Labirinto {
  private Collection salas = new Vector();

  public Labirinto() {
    ...
  }
  public void adicionaSala(ElementoDeLabirinto sala) {
    ...
  }
  public ElementoDeLabirinto getSala(int númeroDaSala) {
    ...
  }
}
```

- Também definimos uma classe Jogo que cria o labirinto
  - Uma forma simples de criar um labirinto é de criar os objetos, adicioná-los ao labirinto e interconectá-los
  - Exemplo da criação de um labirinto com 2 salas e uma porta entre elas

```
public class Jogo {
  ...
  public Labirinto montaLabirinto() {
    Labirinto umLabirinto = new Labirinto();
    Sala sala1 = new Sala(1);
    Sala sala2 = new Sala(2);
```

```
    Porta aporta = new Porta(sala1,sala2);

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

    sala1.setVizinho(NORTE, new Parede());
    sala1.setVizinho(LESTE, aporta);
    sala1.setVizinho(SUL, new Parede());
    sala1.setVizinho(OESTE, new Parede());

    sala2.setVizinho(NORTE, new Parede());
    sala2.setVizinho(LESTE, new Parede());
    sala2.setVizinho(SUL, new Parede());
    sala2.setVizinho(OESTE, aporta);

    return umLabirinto;
  }
  ...
}
```

- O problema desta solução é sua *inflexibilidade*
  - O método montaLabirinto() não é reutilizável em outras situações
  - Motivo: montaLabirinto() mistura a questão da *estrutura* do labirinto com a questão dos tipos exatos de elementos que compõem o labirinto
    - New cria um forte acoplamento entre a classe Jogo e as classes dos objetos criados porque implica num compromisso (amarração) com uma determinada implementação



- Veremos agora como mudar o projeto para criar diferentes tipos de labirintos
  - Labirintos encantados
    - Com portas travadas que precisam de um encantamento para abrir
    - Salas contendo encantamentos que podem ser apanhados
  - Labirintos perigosos
    - Salas com bombas que podem ser explodidas para danificar as paredes (e talvez o jogador!)
- Como alterar montaLabirinto() para facilmente criar estes novos tipos de labirintos?
  - O maior problema é que a solução atual nos força a colocar em código as classes concretas que serão instanciadas
- Usaremos *padrões de criação* para tornar o projeto mais flexível (mais reusável)

O padrão Factory Method

Objetivo

- Definir uma interface para criar objetos de forma a deixar subclasses decidirem qual classe instanciar
- Factory Method deixa que subclasses façam a instanciação

Também conhecido como

- Construtor Virtual

Resumo

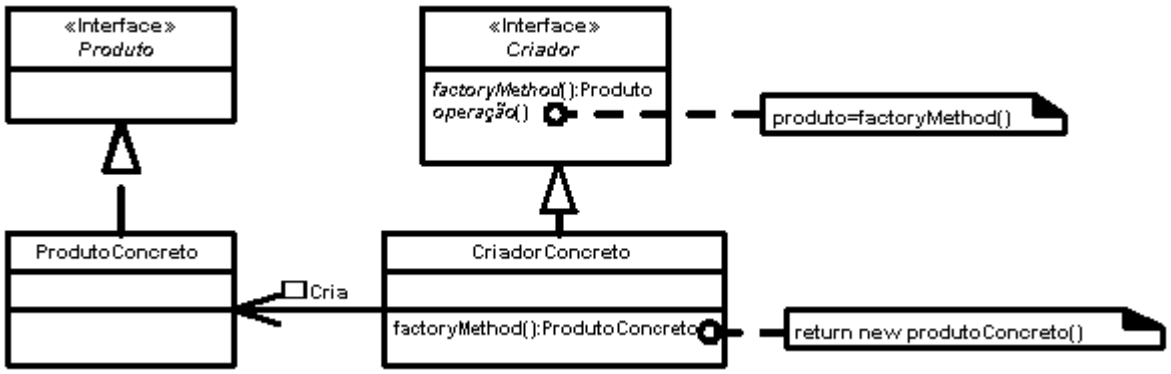
- A idéia é simples: em vez de um cliente que precisa de um objeto chamar new e assim especificar a classe concreta que ele instancia, o cliente chama um método abstrato (Factory Method) especificado em alguma classe abstrata (ou interface) e a subclasse concreta vai decidir que tipo exato de objeto criar e retornar
  - Mudar a subclasse concreta que cria o objeto permite mudar a classe do objeto criado sem que o cliente saiba
  - Permite estender a funcionalidade através da construção de subclasses sem afetar os clientes
- Resumindo:
  - Crie objetos numa operação separada de forma que subclasses possam fazer override da forma de criação

Quando usar o padrão Factory Method?

- Quando uma classe (o criador) não pode antecipar a classe dos objetos que deve criar
- Quando uma classe quer que suas subclasses especifiquem os objetos criados
- Quando classes delegam responsabilidade para uma entre várias subclasses de apoio e queremos localizar

num ponto único a conhecimento de qual subclasse está sendo usada

Estrutura genérica



Participantes

- **Produto**: define a interface dos objetos criados pelo Factory Method
- **ProdutoConcreto**: implementa a interface Produto
- **Criador**: declara o Factory Method que retorna um objeto do tipo Produto
  - Às vezes, o Criador não é apenas uma interface mas pode envolver uma classe concreta que tenha uma implementação *default* para o Factory Method para retornar um objeto com algum tipo ProdutoConcreto default
  - Pode chamar o Factory Method para criar um produto do tipo Produto
- **CriadorConcreto**: faz override do Factory Method para retornar uma instância de ProdutoConcreto

Colaborações

- Criador depende de suas subclasses para definir o Factory Method para que ele retorne uma instância do ProdutoConcreto apropriado

Consequências do uso do padrão Factory Method

- Factory Methods eliminam a necessidade de colocar classes específicas da aplicação no código
  - O código só lida com a interface Produto
  - O código pode portanto funcionar com qualquer classe ProdutoConcreto
- Provê ganchos para subclasses
  - Criar objetos dentro de uma classe com um Factory Method é sempre mais flexível do que criar objetos diretamente
  - O Factory Method provê um gancho para que subclasses forneçam uma versão estendida de um objeto
  - Exemplo num editor de documentos
    - Uma classe Documento poderia ter um Factory Method criaFileDialog para criar um objeto file dialog default para abrir um documento existente
    - Uma subclasse de Documento poderia criar um file dialog especial através do override do Factory Method default
    - Neste caso, o Factory Method não é abstrato mas fornece um default razoável
  - Exercício: como estruturar o código de uma aplicação bancária para que você não fique maluco quando seu gerente pedir que todas as *novas* contas de poupança criadas a partir de segunda-feira tenham algo novo implementado nelas mas sem afetar código antigo que trata das contas antigas?

Considerações de implementação

- É boa prática usar uma convenção de nomes para alertar para o fato de que se está usando Factory Methods
  - Exemplo: makeAbc(), makeXyz()
  - Exemplo: criaAbc(), criaXyz()

Exemplo de código: criação de labirintos

- A criação de labirintos já vista não ficou flexível pois criamos (com new) os objetos especificando as classes concretas na função montaLabirinto()
- Usaremos Factory Methods para deixar que subclasses escolham que objetos criar
- Usaremos o seguinte projeto
  - Produto: Sala, Parede, Porta
  - ProdutoConcreto: Sala, SalaEncantada, SalaPerigosa, Parede, ParedeComBomba, Porta, PortaComChave
  - Criador: Jogo
    - Seu método montaLabirinto() cria o labirinto chamando Factory Methods
    - Ele também é um CriadorConcreto pois oferece uma implementação default para os Factory Methods (para criar um labirinto simples)
  - CriadorConcreto: Jogo, JogoEncantado, JogoPerigoso que serão subclasses de jogo
- Iniciamos com o criador Jogo que contém os Factory Methods

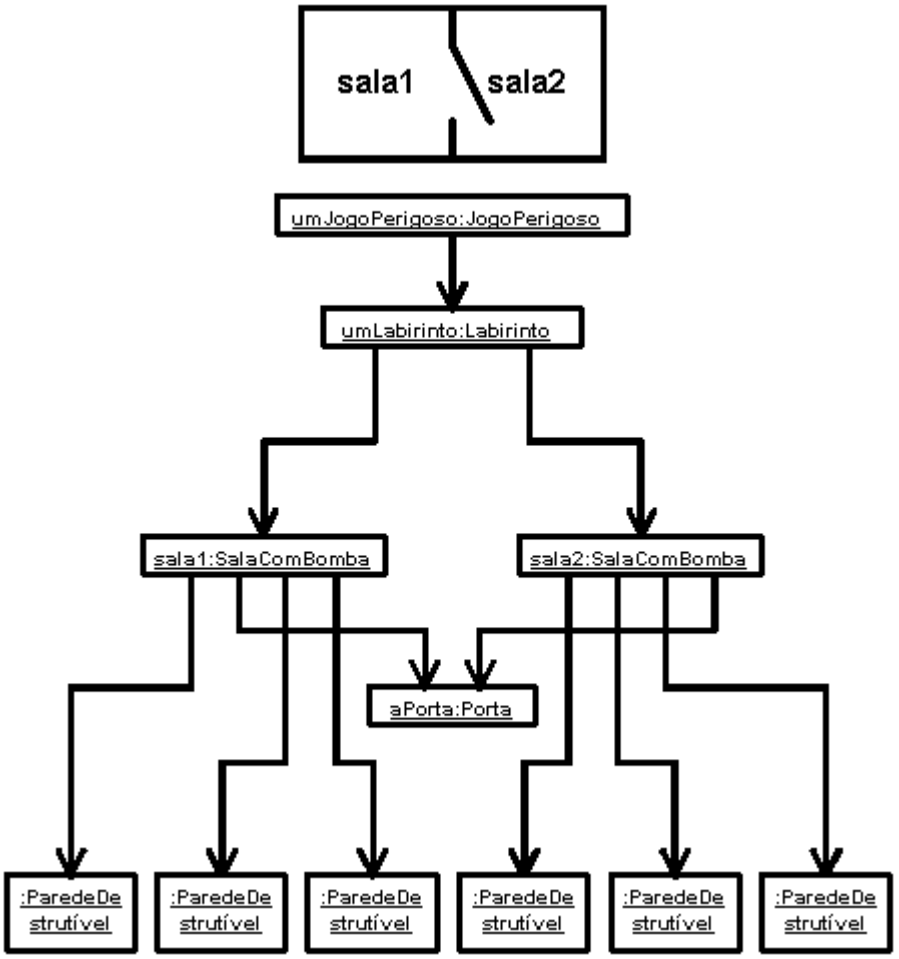
```
public class Jogo {
  // Factory Methods com default
  public Labirinto criaLabirinto() {
    return new Labirinto();
  }
  public Sala criaSala(int númeroDaSala) {
    return new Sala(númeroDaSala);
  }
  public Parede criaParede() {
    return new Parede();
  }
  public Porta criaPorta(Sala sala1, Sala sala2) {
    return new Porta(sala1, sala2);
  }

  // Observe que essa função não tem new:
  // ela usa Factory Methods
  // Esta é a *única* diferença com relação
  // à versão original
  // Observe como o método só trata da estrutura do labirint
  // e não do tipo de elemento que o compõe
```

```
  public Labirinto montaLabirinto() {
    Labirinto umLabirinto = criaLabirinto();
    Sala sala1 = criaSala(1);
    Sala sala2 = criaSala(2);
    Porta aPorta = criaPorta(sala1, sala2);

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

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

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

    return umLabirinto;
  }
}
```

- Observe como não há acoplamento (conhecimento) entre montaLabirinto e outras classes
- Para criar um jogo perigoso, criamos uma subclasse de Jogo e redefinimos alguns Factory Methods

```
// um novo CriadorConcreto
public class JogoPerigoso extends Jogo {
  public Parede criaParede() {
    return new ParedeDestrutível();
  }
  public Sala criaSala(int númeroDaSala) {
    return new SalaComBomba(númeroDaSala);
  }
}
```

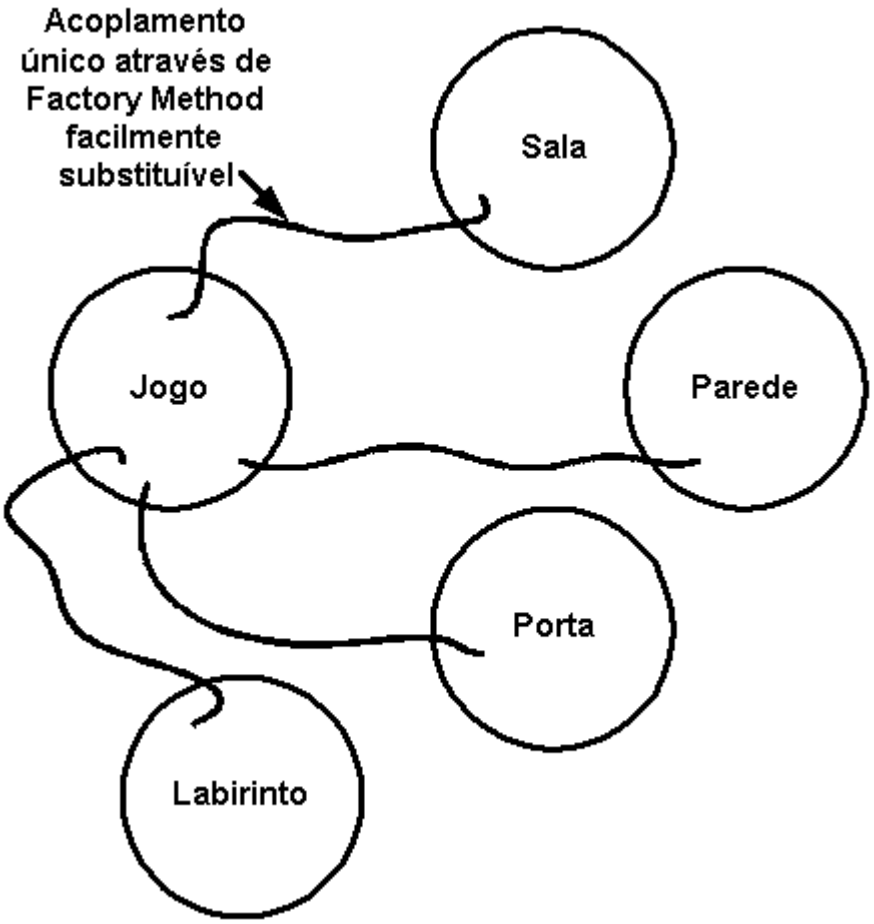
- Comparando o diagrama de objetos com a versão inicial, não há objeto adicional
  - Só há uma mudança do objeto umJogo para um umJogoPerigoso



- Para criar um jogo encantado, procedemos de forma análoga

```
// um novo CriadorConcreto
public class JogoEncantado extends Jogo {
  public Sala criaSala(int númeroDaSala) {
    return new salaEncantada(númeroDaSala, jogaEncantamento(
  }
  public Porta criaPorta(Sala sala1, Sala sala2) {
    return new portaPrecisandoDeEncantamento(sala1, sala2);
  }
  protected Encantamento jogaEncantamento() {
    ...
  }
}
```

- O acoplamento *depois* do Factory Method



Pergunta final para discussão

- De que forma a Factory Method ajuda a produzir código fracamente acoplado?

Ver também

- http://www.javaworld.com/javaworld/javaqa/2001-05/02-qa-0511-factory.html

programa