Padrões de Projeto de Criação

Padrões de Projeto Orientados a Objetos

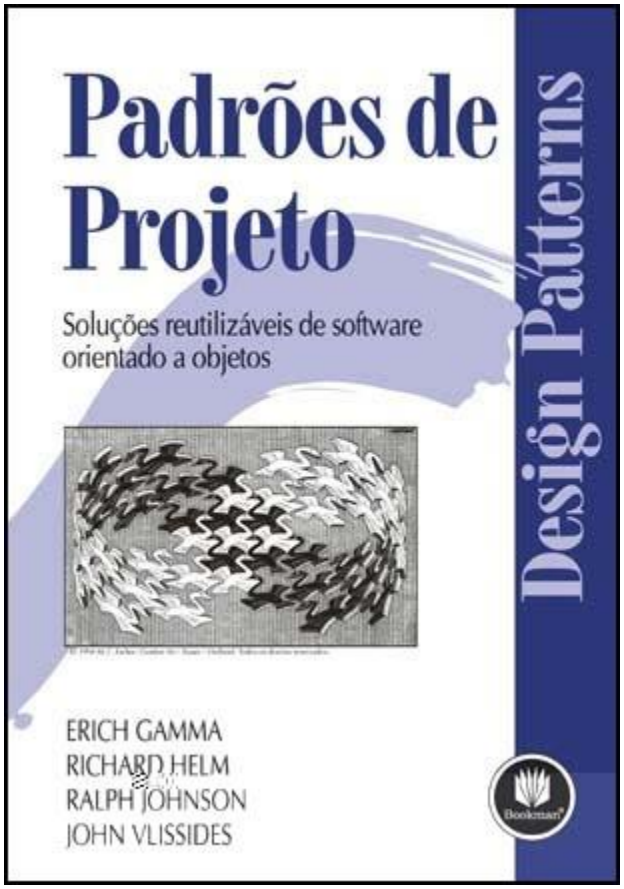
Profª. Danielle Martin

Universidade de Mogi das Cruzes

Padrão de Projeto

- Descrição da essência de uma solução comum apropriada a um problema conhecido e recorrente.
- Define boas práticas de programação.
- Facilita o reuso, flexibilidade e simplicidade do software.
- Deve ser suficientemente abstrato para ser reutilizado em diferentes aplicações.

23 Padrões de projeto – Gang of Four (GoF)



Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos

Gang of four:

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

Publicado em 1994. Referência no assunto de design patterns.

Classificação dos padrões GoF

- Criacionais
  - De criação: Dizem respeito à criação de objetos.
- Estruturais
  - De estrutura: Tratam da composição de classes e objetos.
- Comportamentais
  - Do comportamento: Tratam da colaboração (interação e responsabilidade) entre classes e objetos.

Classificação dos padrões GoF

Os padrões GoF classificados por categorias são:

	Criação	Estrutura	Comportamento
Classe	Factory	Class Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Classificação dos padrões GoF

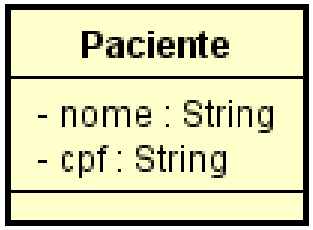
Os padrões GoF classificados por categorias são:

	Criação	Estrutura	Comportamento
Classe	Factory	Class Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

PADRÃO BUILDER

O problema

- Considere uma classe Paciente, com os atributos nome e cpf. Se ambos os atributos forem obrigatórios, como garantir que um objeto Paciente não possa ser criado sem eles?



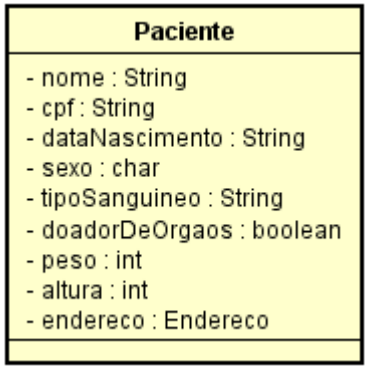
- Pode-se obrigar a inicialização destes atributos direto no construtor:

```
public class Paciente {

    private String nome;
    private String cpf;

    public Paciente(String nome, String cpf) {
        this.nome = nome;
        this.cpf = cpf;
    }
}
```

```
public class AppClinica {

    public static void main(String[] args) {

        Paciente p = new Paciente("Maria", "123456");
    }
}
```

O problema

- E se a classe tiver muitos atributos, podemos obrigar que todos sejam inicializados no construtor?



Problemas:

- Construtor extenso
- Complexidade
- Se houver a necessidade de adicionar/remover um atributo do construtor, todas as chamadas de inicialização de objetos devem ser modificadas.

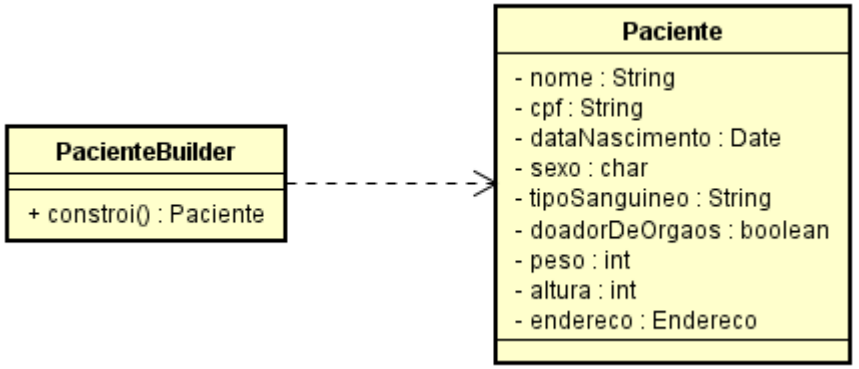
```
public class Paciente {

    private String nome;
    private String cpf;
    private String dataNascimento;
    private char sexo;
    private String tipoSanguineo;
    private boolean doadorDeOrgaos;
    private int peso;
    private int altura;
    private Endereco endereco;

    public Paciente(String nome, String cpf, String dataNascimento, char sexo, String tipoSanguineo, boolean doadorDeOrgaos, int peso, int altura, Endereco endereco) {
        this.nome = nome;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
        this.sexo = sexo;
        this.tipoSanguineo = tipoSanguineo;
        this.doadorDeOrgaos = doadorDeOrgaos;
        this.peso = peso;
        this.altura = altura;
        this.endereco = endereco;
    }

}
```

A solução

- Podemos definir uma classe Builder, responsável por criar o objeto Paciente com todos os atributos obrigatórios.

- A aplicação só recebe o objeto Paciente quando ele estiver completo.

```
public class AppClinica {

    public static void main(String[] args) {

        PacienteBuilder builder = new PacienteBuilder();
        builder.comNome("Maria");
        builder.comCpf("123456");
        builder.comDataNascimento(01,01,1990);
        builder.comAltura(170);
        builder.comPeso(60);
        builder.comTipoSanguineo("A+");
        builder.doSexoFeminino();
        builder.doadorDeOrgaos();
        builder.comEndereco("Rua do Norte", 100);
        Paciente p = builder.constroi();
    }
}
```



A classe Builder

- A classe PacienteBuilder pode mimicar os atributos de Paciente, com os métodos para inicializá-los e um método constroi() que irá efetivamente retornar o objeto Paciente.

```
public class PacienteBuilder {
    private String nome;
    private String cpf;
    private Date dataNascimento;
    private char sexo;
    private String tipoSanguineo;
    private boolean doadorDeOrgaos;
    private int peso;
    private int altura;
    private Endereco endereco;

    public void comNome(String nome) {
        this.nome = nome;
    }

    public void comCpf(String cpf) {
        this.cpf = cpf;
    }

    public void comDataNascimento(int dia, int mes, int ano) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        String data = dia + "/" + mes + "/" + ano;
        try {
            this.dataNascimento = sdf.parse(data);
        } catch (Exception ex) {
        }
    }

    public void doSexoMasculino() {
        this.sexo = 'M';
    }

    public void doSexoFeminino() {
        this.sexo = 'F';
    }

    public void comTipoSanguineo(String tipoSanguineo) {
        this.tipoSanguineo = tipoSanguineo;
    }

    public void doadorDeOrgaos() {
        this.doadorDeOrgaos = true;
    }

    public void comPeso(int peso) {
        this.peso = peso;
    }

    public void comAltura(int altura) {
        this.altura = altura;
    }

    public void comEndereco(String rua, int numero) {
        Endereco endereco = new Endereco(rua, numero);
        this.endereco = endereco;
    }

    public Paciente constroi() {
        return new Paciente(nome, cpf, dataNascimento, sexo, tipoSanguineo, doad
    }
}
```

Vantagens da classe Builder

- É possível criar métodos com formatos diferente para inicializar atributos complexos (ex. fazer conversão de valores, setar boolean automaticamente, setar atributo para um valor fixo).

- Se necessário, o método constroi() pode validar se algum atributo obrigatório foi deixado em branco e só retornar o objeto criado se estiver tudo certo.

- A classe Paciente não precisa ter todos os métodos sets criados. Após a definição dos atributos pelo Builder, eles nunca mais serão modificados, criando um objeto imutável.

Alternativas de implementação de um builder

- #1 – com interface fluente

```
public class AppClinica {

    public static void main(String[] args) {

        Paciente paciente = new PacienteBuilder()
                                .comNome("Maria")
                                .comCpf("123456")
                                .comDataNascimento(01,01,1990)
                                .comAltura(170)
                                .comPeso(60)
                                .comTipoSanguineo("A+")
                                .doSexoFeminino()
                                .doadorDeOrgaos()
                                .comEndereco("Rua do Norte", 100)
                                .constroi();
    }
}
```

```
public class PacienteBuilder {
    private String nome;
    private String cpf;
    private Date dataNascimento;
    private char sexo;
    private String tipoSanguineo;
    private boolean doadorDeOrgaos;
    private int peso;
    private int altura;
    private Endereco endereco;

    public PacienteBuilder comNome(String nome) {
        this.nome = nome;
        return this;
    }

    public PacienteBuilder comCpf(String cpf) {
        this.cpf = cpf;
        return this;
    }

    public PacienteBuilder doSexoMasculino() {
        this.sexo = 'M';
        return this;
    }
```

Alternativas de implementação de um builder

- #2 – como classe interna

```
public class AppClinica {

    public static void main(String[] args) {

        Paciente paciente = Paciente.getBuilder()
                            .comNome("Maria")
                            .comCpf("123456")
                            .comDataNascimento(01,01,1990)
                            .comAltura(170)
                            .comPeso(60)
                            .comTipoSanguineo("A+")
                            .doSexoFeminino()
                            .doadorDeOrgaos()
                            .comEndereco("Rua do Norte", 100)
                            .constroi();
    }
}
```

```
public class Paciente {
    private String nome;
    private String cpf;
    private Date dataNascimento;
    private char sexo;
    private String tipoSanguineo;
    private boolean doadorDeOrgaos;
    private int peso;
    private int altura;
    private Endereco endereco;

    private Paciente() {
    }

    public static PacienteBuilder getBuilder() {
        return new PacienteBuilder();
    }

    public static class PacienteBuilder {

        private Paciente pac;

        public PacienteBuilder comNome(String nome) {
            pac.nome = nome;
            return this;
        }
        //...
        public Paciente constroi() {
            return pac;
        }
    }
}
```

PADRÃO FACTORY

Exemplo

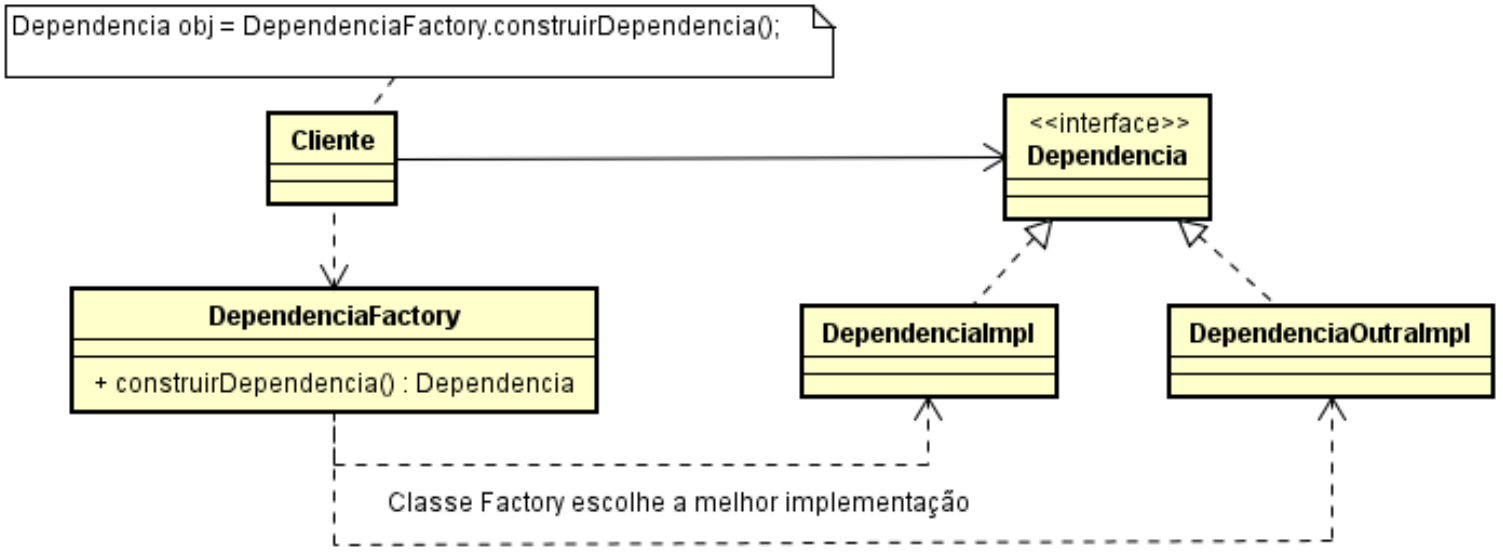
- Uma aplicação depende de objetos Pessoa, porém precisa-se diferenciar as pessoas preferenciais (mais de 65 anos). O método imprimirDados irá fazer essa impressão de forma diferenciada.
- De quem é a responsabilidade de definir se uma pessoa deve ser comum ou preferencial?

```
public class Aplicacao {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String nome = scan.next();
        int idade = scan.nextInt();

        Pessoa p;
        if (idade >= 65) {
            p = new PessoaPreferencial(nome, idade);
        } else {
            p = new PessoaComum(nome, idade);
        }

        p.imprimirDados();
    }
}
```

A aplicação não deveria conhecer a regra que diferencia pessoa comum de preferencial. Há um alto acoplamento entre a Aplicacao e as classes concretas de Pessoa, deixando a solução inflexível.

A solução: padrão factory

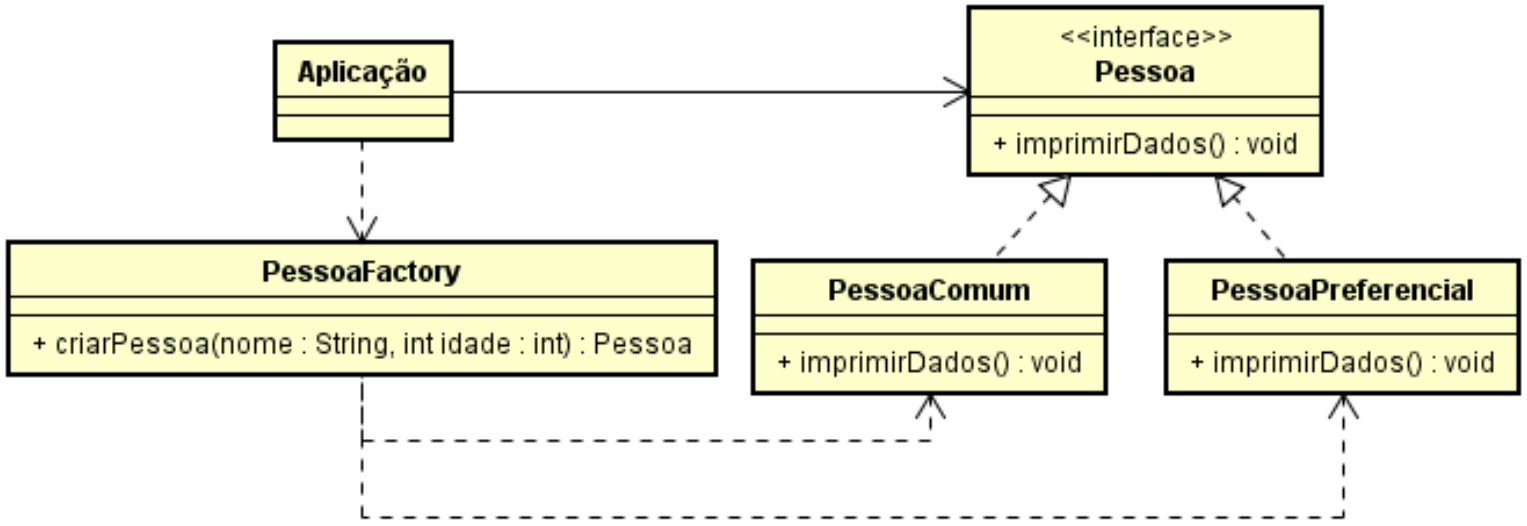
- Pode se criar uma classe Factory que adota a responsabilidade de instanciar objetos das classes dentro de uma hierarquia.
- A classe aplicação depende apenas da referência, e desconhece sua implementação.
- As implementações podem ser facilmente trocadas se necessário a regra que diferencia os objetos está encapsulada na classe Factory.



A solução

- Utilizando uma classe PessoaFactory, podemos delegar a ela a responsabilidade de criação dos objetos Pessoa.

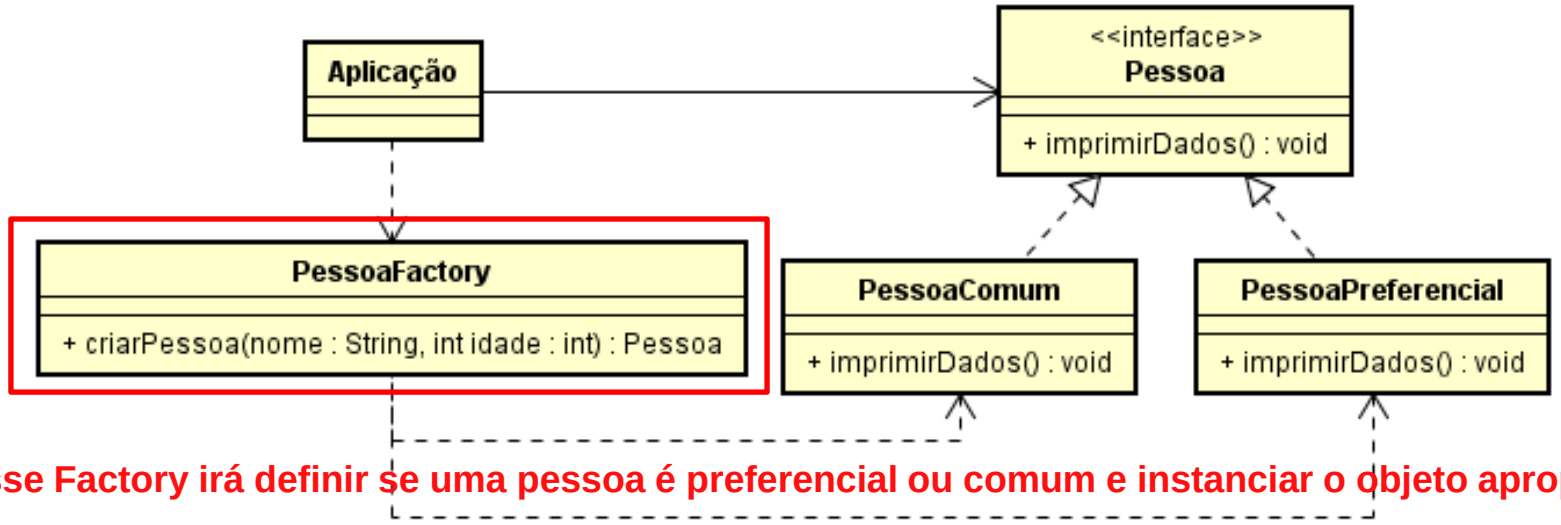
- Qualquer classe que precise de um objeto pessoa pode chamar a factory sem se preocupar em qual é a regra que diferencia a pessoa comum e preferencial.



A solução

- Utilizando uma classe PessoaFactory, podemos delegar a ela a responsabilidade de criação dos objetos Pessoa.

- Qualquer classe que precise de um objeto pessoa pode chamar a factory sem se preocupar em qual é a regra que diferencia a pessoa comum e preferencial.



A classe Factory irá definir se uma pessoa é preferencial ou comum e instanciar o objeto apropriado.

Classes Aplicacao e PessoaFactory

```
public class Aplicacao {

    public static void main(String[] args) {

        Scanner scan = new Scanner(System.in);
        String nome = scan.next();
        int idade = scan.nextInt();

        PessoaFactory fabrica = new PessoaFactory();
        Pessoa p = fabrica.criarPessoa(nome, idade);

        p.imprimirDados();

    }
}
```

A Aplicacao desconhece quais tipos de pessoas existem. Novos tipos de pessoas podem ser criados sem que a aplicação precise ser modificada.

```
public class PessoaFactory {

    public Pessoa criarPessoa(String nome, int idade) {

        Pessoa p;

        if (idade >= 65) {
            p = new PessoaPreferencia(nome, idade);
        } else {
            p = new PessoaComum(nome, idade);
        }

        return p;
    }
}
```

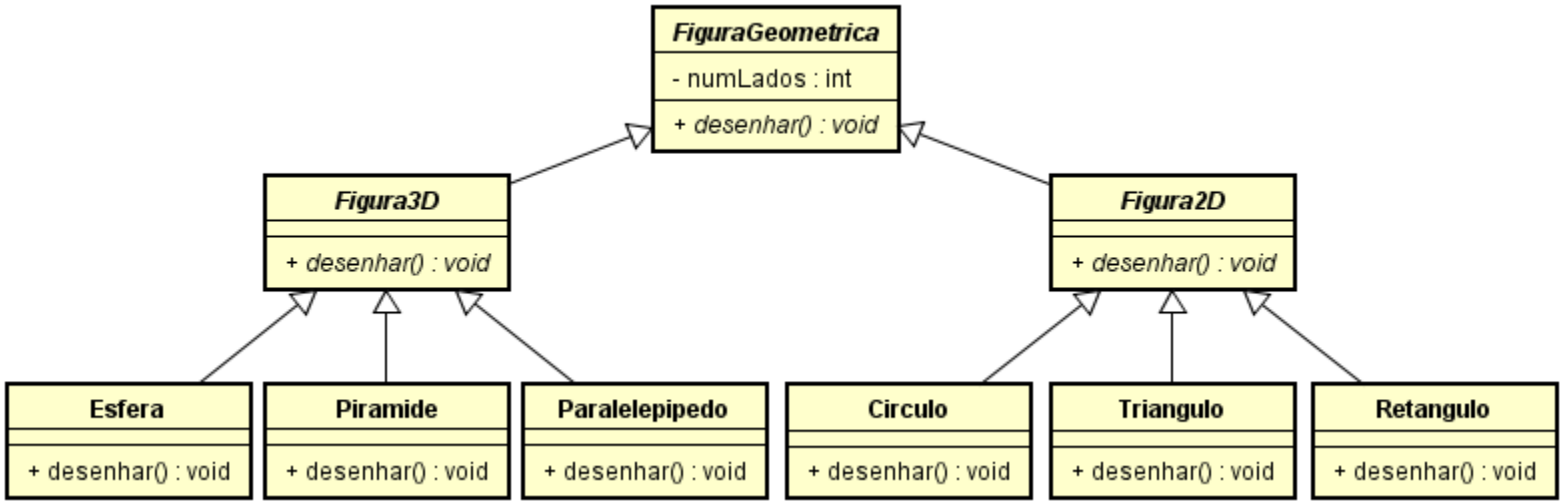
A classe PessoaFactory assume a responsabilidade de instanciar Pessoas. Somente ela conhece a regra para diferenciar tipos de pessoas.

PADRÃO ABSTRACT FACTORY

O problema

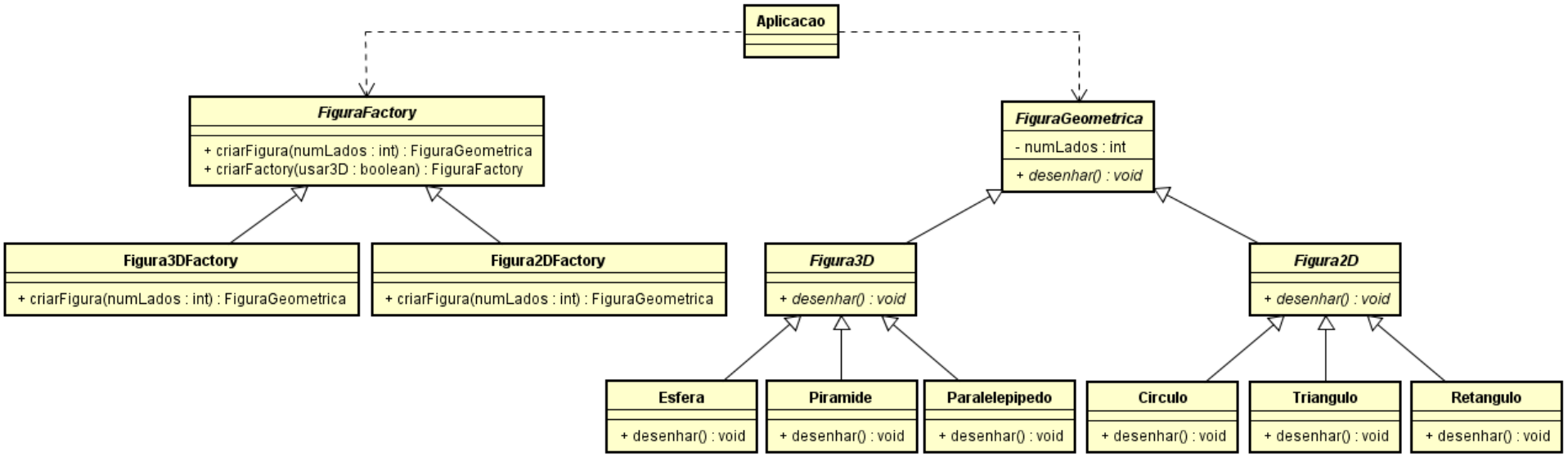
- Assim como o padrão Factory, o Abstract Factory pode instanciar objetos de uma mesma hierarquia dinamicamente. A diferença é que o Abstract Factory pode criar Factories apropriadas para tipos diferentes de objetos em uma hierarquia complexa.

- Ex: Dentro de Figuras Geométricas, temos tipos de figuras 2D e figuras 3D. Na aplicação que desenha essas figuras, se uma figura de 3 lados for desenhada sem 3D, deve-se desenhar um triângulo. Com 3D ,desenha-se automaticamente uma pirâmide.



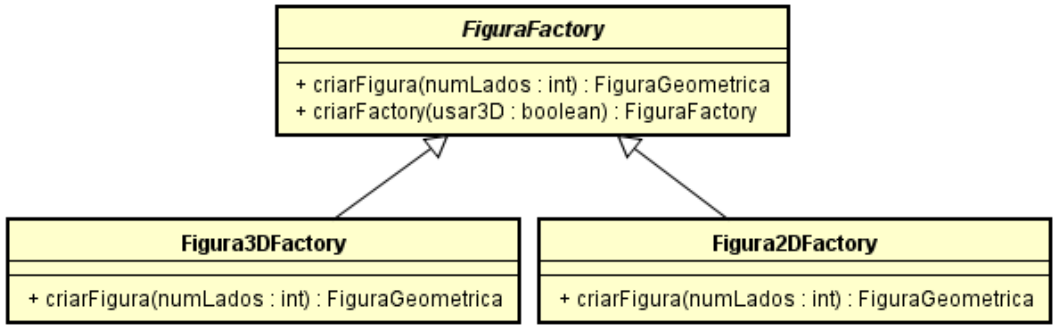
A solução

- Podemos criar duas classes Factory diferentes, uma que irá criar somente figuras 2D e outras para figuras 3D. As duas classes têm como super classe uma Factory abstrata, que pode definir qual factory apropriada usar dependendo da opção do usuário - 3D ou 2D.



As factories

```
public class Figura2DFactory extends FiguraFactory {

    @Override
    public FiguraGeometrica criarFigura(int numLados) {
        if (numLados == 1) {
            return new Circulo();
        } else if (numLados == 2) {
            return new Retangulo();
        } else {
            return new Triangulo();
        }
    }
}

public class Figura3DFactory extends FiguraFactory {

    @Override
    public FiguraGeometrica criarFigura(int numLadosDaFace) {
        if (numLadosDaFace == 1) {
            return new Esfera();
        } else if (numLadosDaFace == 2) {
            return new Paralelepipedo();
        } else {
            return new Piramide();
        }
    }
}
```



A aplicação

```
14  public class Aplicacao {
15
16      public static void main(String[] args) {
17
18          Scanner scan = new Scanner(System.in);
19
20          System.out.println("Digite o numero de lados da face da figura: ");
21          int numLados = scan.nextInt();
22
23          System.out.println("É 3D: ");
24          boolean usar3D = scan.nextBoolean();
25
26          FiguraFactory ff = FiguraFactory.criarFactory(usar3D);
27          FiguraGeometrica figura = ff.criarFigura(numLados);
28
29          figura.desenhar();
30      }
31  }
```

Output - ProjetoAbstractFactory (run) | HTTP Server Monitor

```
run:
Digite o numero de lados da face da figura:
1
É 3D:
true
Desenhando esfera
BUILD SUCCESSFUL (total time: 5 seconds)
```