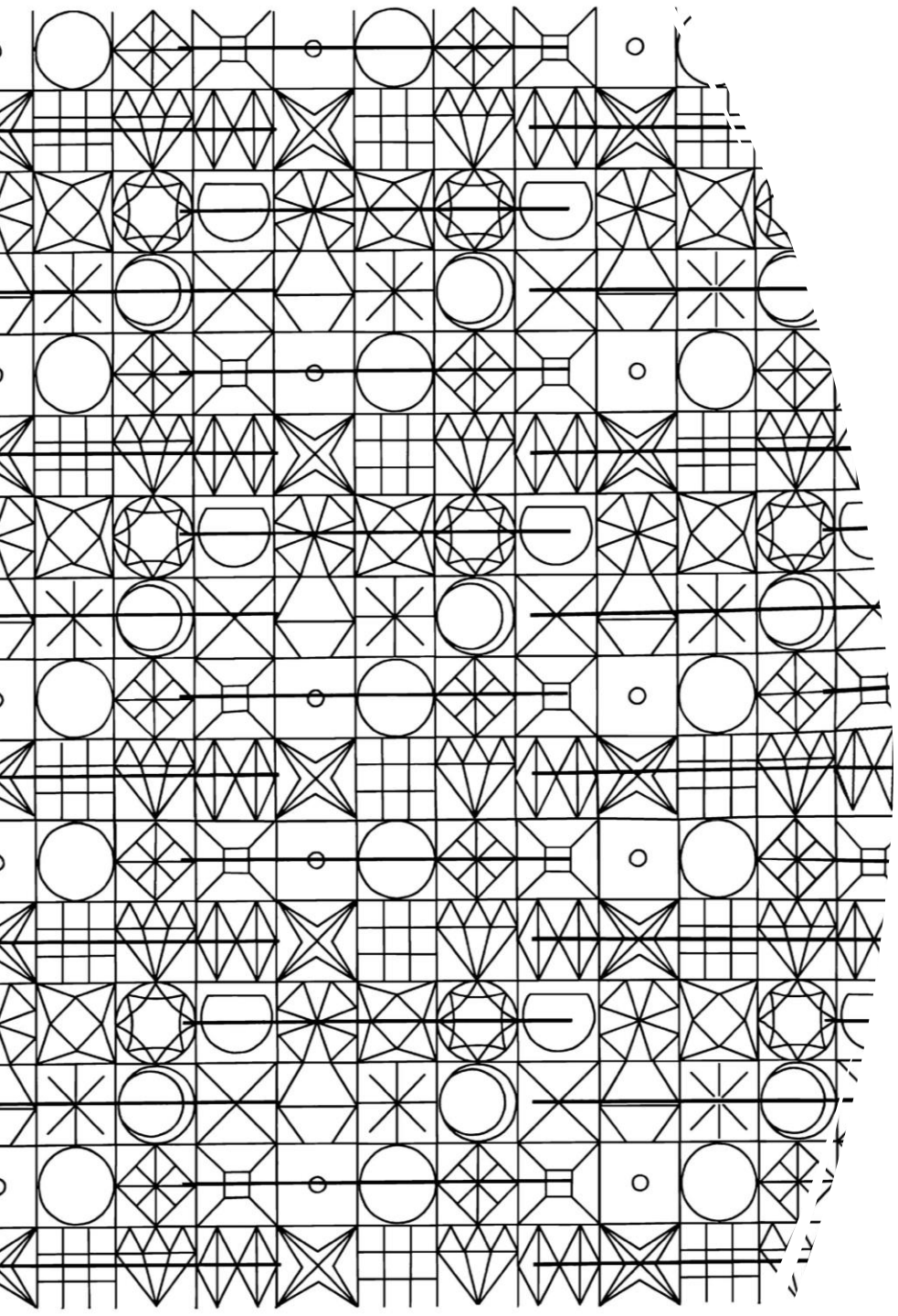


Padrões de criação

“Padrões de criação”

- Uma questão essencial na orientação a objetos é como criar um objeto a fim de utilizá-lo
- As melhores práticas em Programação Orientada a Objetos sinalizam na direção de deixar a aplicação o mais flexível possível
- Normalmente usam-se hierarquias de classes e classes abstratas de maneira a atingir esses objetivos
- De nada adianta, porém, trabalhar com abstrações, se os comandos de criação de objetos das linguagens de programação lidam apenas com classes concretas (é o caso do comando “new” de Java).
- Os padrões de criação visam definir formas mais flexíveis de se criarem objetos. Possuem dois enfoques principais:
  - Oferecer formas de criar objetos que deixem o código mais fácil de ler e entender (semântica mais explícita)
  - Oferecer formas mais flexíveis de criar objetos (que possam lidar facilmente com novas versões dos objetos)
  - Oferecer formas de criar objetos que simplifiquem o acesso ou o reuso dos objetos

Limitações dos métodos construtores

- O fato de não podermos especificar os tipos ou a quantidade de parâmetros do método construtor faz com que não se tenha informação sobre as diferentes possibilidades de ativação do construtor.
  - O código fica difícil de usar e de ler.
- Um construtor sempre cria um novo objeto
  - Sempre é necessário criar novos objetos?
- Um construtor só pode retornar objetos da mesma classe
  - Como seria possível instanciar objetos de uma hierarquia de herança sem depender de cada construtor em particular?

Static Factory Method: contexto

- Um “cliente” precisa criar um determinado tipo de objeto
- Este objeto possui diferentes alternativas de instanciação
- Exemplo: classe *Data*
  - Como saber quais as opções disponíveis?
  - Quanto mais opções mais complexa a lógica do construtor.

let d1 = new Data();
let d2 = new Data(10,3);
let d3 = new Data(10,3,2023);

```
public class Data{
    private int dia;
    private int mes;
    private int ano;

    public Data() {
        LocalDate hoje = LocalDate.now();
        this.dia = hoje.getDayOfMonth();
        this.mes = hoje.getMonthValue();
        this.ano = hoje.getYear();
    }

    public Data(int dia,int mes) {
        LocalDate hoje = LocalDate.now();
        this.dia = dia;
        this.mes = mes;
        this.ano = hoje.getYear();
    }

    public Data(int dia, int mes, int ano) {
        this.dia = dia;
        this.mes = mes;
        this.ano = ano;
    }

    @Override
    public String toString() {
        return "Data [dia=" + dia + ", mes=" + mes + ", ano=" + ano + "]";
    }
}
```

Static Factory Method: solução

- Defina um tipo **produto** que não exponha o construtor de forma pública
- Defina diferentes métodos de classe, os **factoryMethod,** no tipo **produto** que irão gerar objetos **produto**
- Objeto **cliente** utiliza os métodos de classe para obter instâncias
- Exemplos:

Data d1 = Data.hoje();

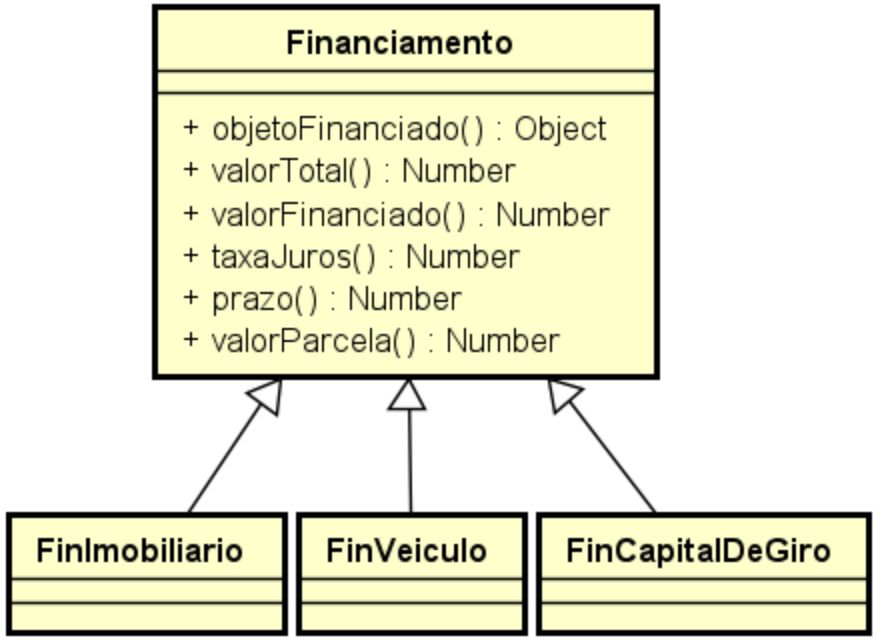
Data d2 = Data.doAno(3,6);

let d3 = Data.doMes(3);

```
public class Data{
   private int dia;
   private int mes;
   private int ano;

   private Data(int dia, int mes, int ano) {
      this.dia = dia; this.mes = mes; this.ano = ano;
   }

   Public static Data hoje() {
      LocalDate hoje = LocalDate.now();
      return new Data(hoje.getDayOfMonth(),
                      hoje.getMonthValue(),
                      hoje.getYear());
   }

   public static Data doMes(int dia) {
      LocalDate hoje = LocalDate.now();
      return new Data(dia, hoje.getMonthValue(), hoje.getYear());
   }

   public static Data doAno(int dia,int mes) {
      LocalDate hoje = LocalDate.now();
      return new Data(dia, mes, hoje.getYear());
   }

   @Override
   public String toString() {
      return "Data [dia=" + dia + ", mes=" + mes + ", ano=" + ano + "]";
   }
}
```

Factory: contexto

- Um **cliente** precisa utilizar objeto **produto**
- Objeto **produto** possui diferentes subclasses
- O **cliente** utiliza objetos **produto** sem conhecer diretamente suas subclasses



Factory: solução

- Defina uma classe “factory” que possui um método fabricante “criaProduto” que irá retornar instâncias concretas do **produto** baseado na informação recebida por parâmetro.
  - O exemplo ao lado usa o tipo do objeto recebido para decidir o tipo do financiamento
- O objeto **cliente** utiliza o objeto retornado sem precisar saber qual é o seu tipo específico.

```
class FinanciamentoFactory{
    public static Financiamento criaFinanciamento(
                                                Object aFinanciar){
        if (aFinanciar instanceof Veiculo){
            return new FinVeiculo(aFinanciar);
        }
        if (aFinanciar instanceof Casa ||
            aFinanciar instanceof Apartamento){
                return new FinImobiliario(aFinanciar);
        }
        if (aFinanciar instanceof CapitalDeGiro){
            return new FinCapitalGiro(aFinanciar);
        }
        return null;
    }
}
Veiculo carro = new Veiculo( ... );
Financiamento fCarro = FinanciamentoFactory.criaFinanciamento(carro);
```

Builder: contexto

- O construtor de um objeto alvo tem muitos parâmetros
- Nem sempre se deseja indicar todos os parâmetros (muitos tem valores "default")
- O fato de vários parâmetros serem do mesmo tipo complica para criar diferentes possiblidades no construtor
- Seria interessante dispor de uma interface fluente

```
public class Pizza{
   public enum Molho {TOMATE,GORGONZOLA,PARISIENCE,...}
   public enum Cobertura {CALABRESA,PORTUGUESA,...}

   private Molho molho;
   private Cobertura cobertura;
   private boolean massaIntegral;
   private boolean bordaRecheada;

   public Pizza(Cobertura cobertura){
      this.molho = Molho.TOMATE;
      this.cobertura = cobertura;
      this.massaIntegral = false;
      this.bordaRecheada = false;
   }

   public Pizza(Cobertura cobertura,boolean bordaRecheada){ ... }

   public Pizza(Cobertura cobertura, boolean bordaRecheada,
                boolean massaIntegral){ ... }

   public Pizza(Molho molho, Cobertura cobertura,
                boolean bordaRecheada,boolean massaIntegral){...}

   . . .
}
```

Builder: solução (P1)

- Criar uma classe construtora (builder) para o objeto alvo:
  - Assume valores “padrão” no construtor
  - Possui métodos que permitem alterar os valores padrão
  - Os métodos que permitem alterar os valores padrão retornam o próprio “builder” de maneira a permitir o encadeamento de chamadas: **interface fluente**
  - Possui um método chamado “build” que retorna uma instancia do objeto alvo

```
public class Pizza{
   public enum Molho { ...
   public enum Cobertura { ...

   private Molho molho;
   private Cobertura cobertura;
   private boolean massaIntegral;
   private boolean bordaRecheada;

   public static class Builder {
      private Molho molho = Molho.TOMATE;
      private Cobertura cobertura = PizzaB.Cobertura.MARGUERITA;
      private boolean massaIntegral = false;
      private boolean bordaRecheada = true;

      public Builder() {
      }

      public Builder molho(PizzaB.Molho molho) { this.molho = molho; return this; }

      public Builder cobertura(PizzaB.Cobertura cobertura) { this.cobertura = cobertura; return this; }

      public Builder comBorda() { this.bordaRecheada = true; return this; }

      public Builder semBorda() { this.bordaRecheada = true; return this; }

      public Builder massaNormal() { this.massaIntegral = false; return this; }

      public Builder massaIntegral() { this.massaIntegral = true; return this; }

      public PizzaB build() { return new PizzaB(this); }
   }

   private PizzaB(Builder builder) {
      this.molho = builder.molho;
      this.cobertura = builder.cobertura;
      this.massaIntegral = builder.massaIntegral;
      this.bordaRecheada = builder.bordaRecheada;
   }

   ...
}
```

Builder: solução (P2)

- A classe do objeto alvo tem construtor privado
- As instancias devem ser criadas usando-se uma instancia da classe “builder”
- Exemplos:

```
Pizza p1 = Pizza.builder()
                .molho(Molho.Gorgonzola)
                .comBorda()
                .massaNormal()
                .build();

Pizza p2 = Pizza.builder()
                .massaIntegral()
                .semBorda()
                .build();
```

```
public class Pizza {
   public enum Molho { ...
   public enum Cobertura { ...

   private Molho molho;
   private Cobertura cobertura;
   private boolean massaIntegral;
   private boolean bordaRecheada;

   public static class Builder {
      private Molho molho = Molho.TOMATE;
      private Cobertura cobertura = PizzaB.Cobertura.MARGUERITA;
      private boolean massaIntegral = false;
      private boolean bordaRecheada = true;

      public Builder() {
      }

      public Builder molho(PizzaB.Molho molho) { this.molho = molho; return this; }

      public Builder cobertura(PizzaB.Cobertura cobertura) { this.cobertura = cobertura; return this; }

      public Builder comBorda() { this.bordaRecheada = true; return this; }

      public Builder semBorda() { this.bordaRecheada = false; return this; }

      public Builder massaNormal() { this.massaIntegral = false; return this; }

      public Builder massaIntegral() { this.massaIntegral = true; return this; }

      public PizzaB build() { return new PizzaB(this); }
   }

   private Pizza(Builder builder) {
      this.molho = builder.molho;
      this.cobertura = builder.cobertura;
      this.massaIntegral = builder.massaIntegral;
      this.bordaRecheada = builder.bordaRecheada;
   }

   ...
}
```

Considerações sobre o padrão Builder

- Vantagens:
  - Legibilidade do código: código mais legível não é um “luxo” e sim uma necessidade que se impõe devido ao aumento na complexidade dos sistemas
  - A construção do “builder” ajuda na identificação do foco da classe, facilitando a criação de classes coesas
- Desvantagens:
  - O planejamento do “builder” toma tempo. Seu uso só será vantajoso se ele for bem pensado.
  - O nome dos métodos não é tão intuitivo quando analisado fora do contexto. O que é mais fácil de entender: “cobertura” ou “defineCobertura”? Esta desvantagem some, porém, quando do uso.

Singleton: contexto

- Todos os clientes necessitam acessar uma única instância compartilhada da classe.
- Deve-se garantir que nenhuma instância adicional será criada acidentalmente.
- Exemplo: deve-se conseguir acessar uma instancia única do “spooler” de impressão de qualquer ponto do programa.

Singleton: solução

- Manter o construtor privado.
- Criar um atributo privado estático para manter a instância.
- Criar um método estático para ter acesso “global” a instância.
- Exemplo de uso:

```
Spooler imp = Spooler.getInstance();
imp.imprime("Teste de impressao");
imp = null;
imp = Spooler.getInstance();
imp.imprime("Arquivo para imprimir");
```

```
class Spooler{
  static Spooler instance;

  private Spooler() {
    //... Conteúdo do construtor ...
  }

  static public Spooler getInstance(){
      if(instance == null)
          instance = new Spooler();
      return(instance);
  }
  public void imprime(String texto){
      ...
  }

  // Outros métodos da classe ...

}
```

Considerações sobre o padrão Singleton

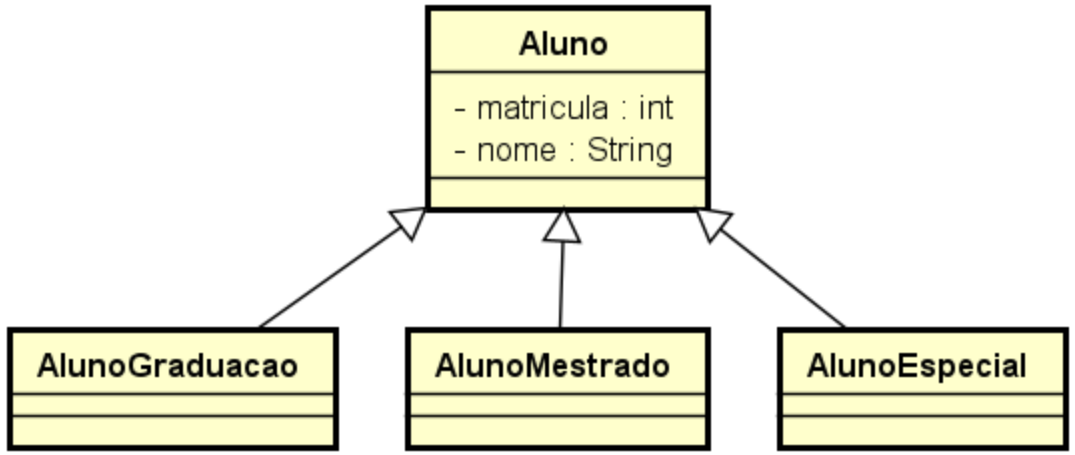
- Vantagens:
  - Permite acessar uma instância única de qualquer ponto do código
  - Aceita trabalhar com mais de uma instância se for o caso, mas controlando a quantidade
  - É útil quando as classes são “custosas” para criar: acesso a banco de dados
  - É útil para trabalhar com sistemas de “log”
  - É útil para acessar classes que guardam parâmetros de configuração de um sistema
- Desvantagens:
  - Esconde as dependências da classe
  - Dificulta o teste unitário

Dinâmica

D5) Em um jogo, a classe “Personagem” tem 3 atributos que devem ser informados na hora da criação da instância: visibilidade [0,10], poder [0;100] e vidas [0,5]. Aproximadamente 60% dos personagens criados são os ditos “normais” que são criados com visibilidade = 10, poder = 50 e vidas = 3. Uns 15% são chamados de “poderosos”. Nestes a visibilidade é 10, as vidas são 3, mas o poder é ajustado na hora da criação. Outros 15% são os conhecidos como “soturnos” onde o poder é 50, as vidas são 3 e a visibilidade é ajustada na hora da criação. Finalmente os últimos 10% têm os três parâmetros configurados na hora da criação. Explore o padrão “static factory method” na construção desta classe. Escreva a classe e exemplos de uso.

Dinâmica

D6) Uma universidade atende 3 tipos de estudantes, modelados conforme a hierarquia de herança ao lado. Sabe-se que os números de matrícula até 99000 correspondem a alunos especiais, entre 100000 e 199000 são alunos de mestrado e acima disso são alunos de graduação. Crie uma classe que implemente o padrão “Factory” capaz de criar instancias derivadas de aluno baseado no número de matrícula informado.



Dinâmica

D7) Em um determinado banco cada cliente possui dois saldos: o saldo livre para movimentações de depósitos e retiradas e o saldo em investimentos. Cada cliente tem uma taxa de remuneração para os investimentos e uma taxa de juros que é cobrada caso seu saldo livre fique negativo (o banco admite saques a descoberto até um certo limite). Um desenvolvedor criou uma classe para modelar esta conta bancária. O método construtor dessa classe possui os seguintes parâmetros:

```
numeroConta, nomeCorrentista, saldoLivreInicial,
salAplicacaoInicial,taxaRemuneracao, taxaSaldoNegativo
```

São muitos os parâmetros e na verdade apenas o número da conta e o nome do correntista são necessários para abertura da conta. Os demais podem definidos mais tarde visto que podem ser utilizados valores padrão (os saldos iniciais podem ser zero assim como as taxas e o limite para saques a descoberto). Demonstre como essa classe pode ser implementada usando o padrão “Builder”.