TÓPICO 08 - ESTRUTURAS DE DADOS

Clean Code - Professor Ramon Venson - SATC 2025

Dados

Um dado pode ser representado de diferentes formas.

`Structs`, `classes`, `arrays`, `dictionaries`, `records`, etc. são só alguns exemplos da infinidade de estruturas que usamos para moldar um dado.

Abstração de Dados

Considere o seguinte exemplo:

```
public class Ponto {
    public Double x;
    public Double y;
}
```

- Essa classe representa um ponto no plano cartesiano.
- Nem todos os pontos são representados dessa forma.
- A coordenada pode ser alterada de maneira independente.

Agora considere a seguinte interface:

```
public interface Ponto {
    private Double getX();
    private Double getY();
    private Double getR();
    private Double getTheta();
    void setCartesiano(Double x, Double y);
    void setPolar(Double r, Double theta);
}
```

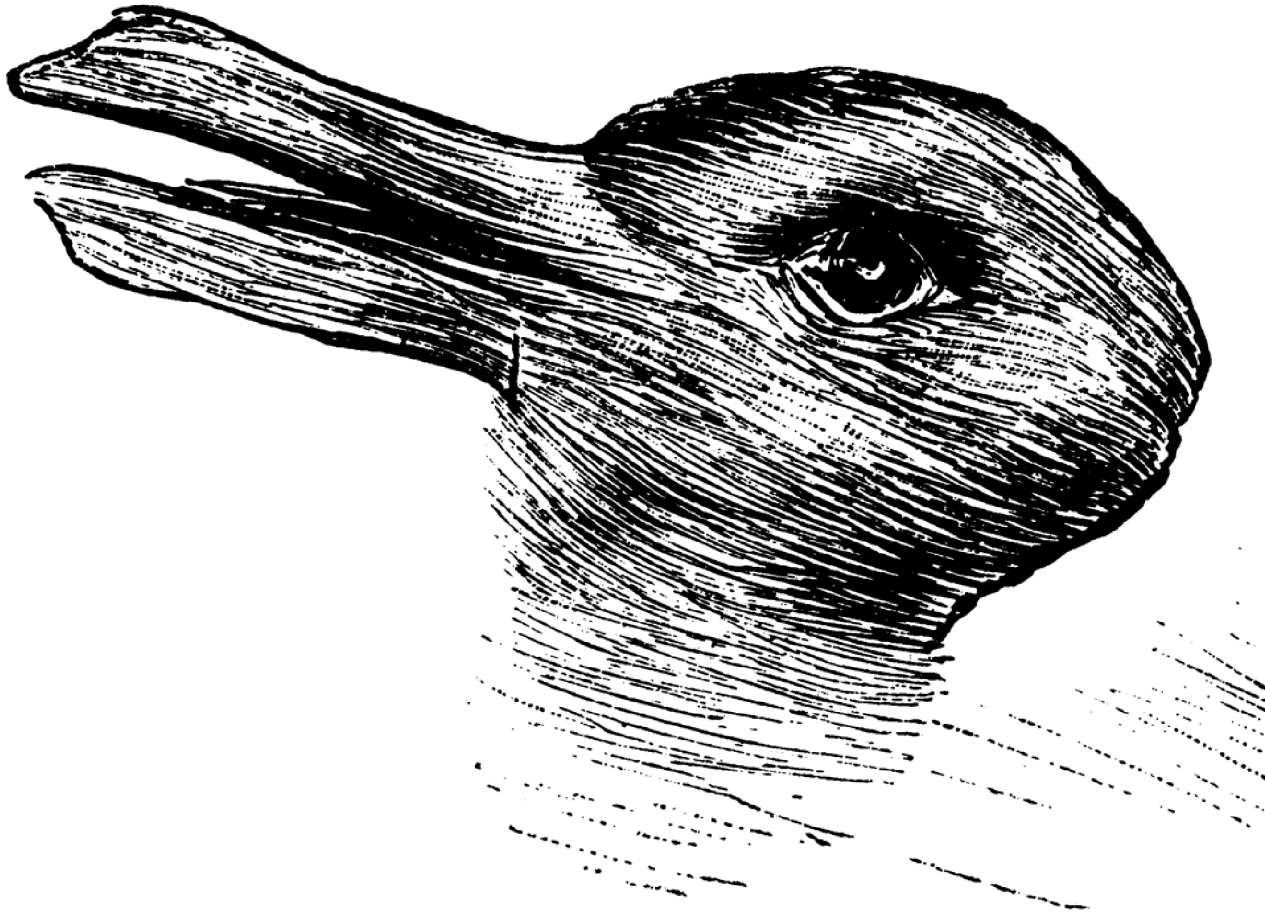
- O que mudou?
- Agora podemos representar pontos em diferentes sistemas de coordenadas.
- Essa estrutura garante que podemos ler as coordenadas de maneira independente, porém alterá-las de maneira consistente.

Anti-simetria dado/objeto

Objetos podem ser vistos de duas formas:

- Um objeto que manipula outro objeto
- Um objeto que é manipulado por outros objetos

Este último é o que chamamos de *objeto de valor* (ou POJO).



A lei de Demeter

> Um módulo não deve enxergar o interior dos objetos que ele manipula

Um método só deve ser capaz de chamar outros métodos em situações muito específicas

Método chama método da mesma Classe

Um método pode chamar outro método da mesma classe

```
public class RegistroPonto {
    private DateTime data;

    public void atualizarHorario(DateTime data) {
        setData(data);
    }

    private void setData(DateTime data) {
        this.data = data;
    }
}
```

Método chama método de parâmetro

```
public class RegistroPonto {
    private int dia;

    public void atualizarDia(DateTime data) {
        int dia = data.getDay();
        setDia(dia);
    }
}
```

Método chama método de atributo

```
public class RegistroPonto {
    private Funcionario funcionario;

    public void log() {
        String nome = funcionario.getNome();
        Console.WriteLine(nome + " realizou o registro.");
    }
}
```

Método chama método de objeto criado dentro do método

```
public class RegistroPonto {
    public void log() {
        Funcionario funcionario = new Funcionario("Ronaldinho");
        String nome = funcionario.getNome();
        Console.WriteLine(nome + " realizou o registro.");
    }
}
```

Transgressão de Demeter

```
public class RegistroPonto {
    public void log(Funcionario funcionario) {
        String nomeSetor = funcionario.getSetor().getNome();
        Console.WriteLine("O setor " + nomeSetor + " fez um registro de ponto");
    }
}
```

Carrinhos de trem

Métodos encadeados são chamados de carrinhos de trem e devem ser evitadas, como no exemplo:

```
public class MusicPlayer{
    public void play(MusicFile music) {
        music.getArtist().getAlbum().getSong().play();
    }
}
```

Carrinhos de Trem vs Interfaces Fluentes

As interfaces fluentes podem facilmente parecerem com o problema dos carrinhos de trem.

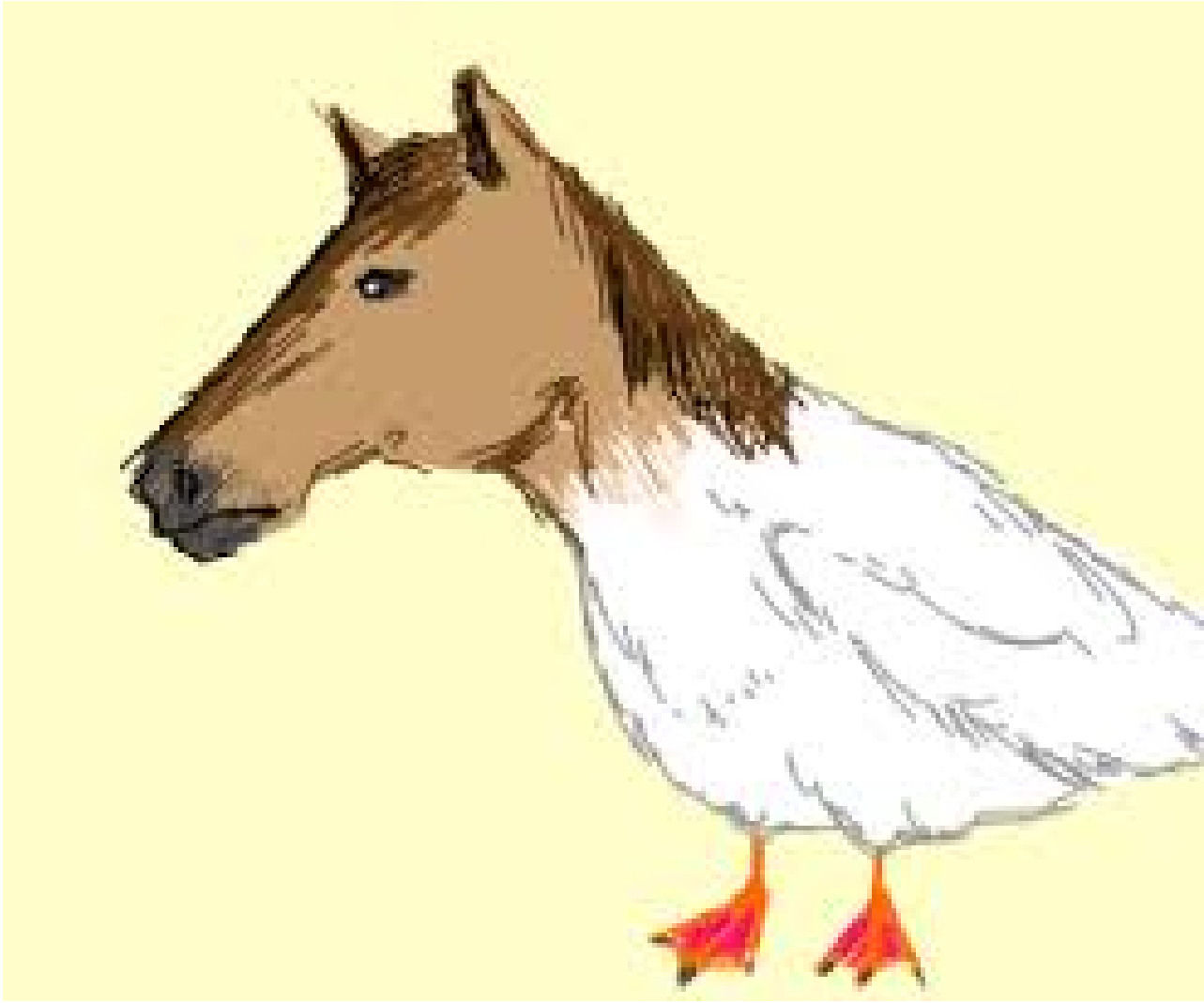
```
Order order = new Order()
    .setCustomer("John")
    .setShippingAddress("123 Main St")
    .setTotal(100.0);
```

No entanto, interfaces fluentes não expõe a estrutura interna do objeto, tornando-as *"Demeter-compatíveis"*.

Híbridos

Objetos que manipulam outros objetos e objetos que apenas possuem dados são categorizados de maneiras diferentes.

É importante que um objeto que apenas contem dados (POJO) não tenha métodos que manipulem outros objetos e vice-versa.



Objetos de Transferência de Dados (DTO)

A forma perfeita para uma estrutura com apenas dados é uma classe com variáveis públicas e nenhuma função.

Os DTOs (Data Transfer Objects) são classes com apenas dados, que são usados para transferir dados entre diferentes camadas de uma aplicação.

Ao invés de:

```
public class Pessoa {
    private String nome;
    private String cpf;

    public String getNome() {
        return this.nome;
    }

    public String getCpf() {
        return this.cpf;
    }

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

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

Podemos usar:

```
public record Pessoa(String nome, String cpf);
```

Active Record

Os Active Records são formas especiais de DTOs que possuem métodos que manipulam o próprio objeto, como `save` e `find`.

É importante que regras de negócios não sejam implementadas nesses objetos, assim como nos DTOs.

Outras Dicas

Maps vs Arrays

```
Map<String, String> mapaFuncionarios = new HashMap<>();
ArrayList<String> listaFuncionarios = new ArrayList<>();
```

Utilize vetores para armazenar coleções que precisam ser processadas em ordem e/ou como um todo.

Use mapas (associação par-valor) para coleções que precisam de buscas rápidas.

Lists vs Sets

```
List<String> listaFuncionarios = new ArrayList<>();
Set<String> grupoFuncionarios = new HashSet<>();
```

Utilize listas para quando a ordem dos elementos é importante.

Utilize conjuntos para quando a ordem não é importante e não podem haver elementos repetidos.

Evite Heranças Profundas

```
public class SerHumano extends SerVivo {}
public class Pessoa extends SerHumano {}
public class Funcionario extends Pessoa {}
public class FuncionarioGerente extends Funcionario {}
public class FuncionarioSupervisor extends FuncionarioGerente {}
```

Heranças profundas são heranças que estendem muitos níveis de uma hierarquia.

Elas são difíceis de entender e podem ser substituídas por composição.

Usando composição:

```
public class Funcionario {
    private Pessoa pessoa;
    private Cargo cargo;
    private Setor setor;
}
```

KISS (Keeep It Simple, Stupid)

Mantenha o código simples e fácil de entender.

Evite implementar métodos, camadas, abstrações e atributos desnecessários.

Polimorfismo vs Condicionais

Utilize polimorfismo para escolher entre implementações diferentes de um mesmo método para evitar estruturas de `switch/case` e `if/else`.

Essas estruturas reduzem a legibilidade e extensibilidade do código.

Maps vs Condicionais

```
// Condicionais
if (tipo == "Gerente") {
    return new Gerente();
}

// Maps
Map<String, Funcionario> funcionarios = new HashMap<>();
funcionarios.put("Gerente", new Gerente());
```

Algumas condicionais podem ser substituídas por mapas e a design pattern `Strategy` .

Identidade vs Igualdade

Entenda a diferença entre comparar um os valores de dois objetos e comparar se eles são o mesmo objeto.

Enquanto estruturas primitivas são geralmente comparadas por igualdade, objetos são comparados por identidade.

```
String nome1 = "Marta";
String nome2 = "Marta";
nome1 == nome2; // true
nome1.equals(nome2); // true

Pessoa pessoa1 = new Pessoa("Marta", "123456789");
Pessoa pessoa2 = new Pessoa("Marta", "123456789");
pessoa1.equals(pessoa2); // false
pessoa1 == pessoa2; // false
```

```
public void revisaFuncionarios(List<Funcionario> funcionarios) {
    for (Funcionario funcionario : funcionarios) {
        if (funcionario.getSalario() > 10000) {
            funcionario.setSalario(funcionario.getSalario() * 1.1);
        }
    }
}

List<Funcionario> funcionarios = new ArrayList<>();
funcionarios.add(new Funcionario("Ronaldinho", 10000));
funcionarios.add(new Funcionario("Marta", 15000));
revisaFuncionarios(funcionarios);
```

Estruturas primitivas e Strings são geralmente passadas por valor, enquanto outros objetos (como coleções) são passados por referência. Isso significa que alterações feitas em um objeto passado como parâmetro podem **afetar o objeto original**.

Material de Apoio

- Codex
- Mahmoud Ibrahim
- Clean Code Studio - Data Structures