Exercícios

Princípio da Responsabilidade Única (Single Responsibility Principle – SRP)

Exercício 1: Considere a interface **Modem** e a refatore para que ela obedeça ao Princípio da Responsabilidade Única (SRP). Para isso, você deverá criar novas interfaces que representem as responsabilidades únicas da interface **Modem**.

A interface original lida com duas tarefas distintas: o gerenciamento de uma conexão telefônica e a comunicação de dados. Sua tarefa é separar essas responsabilidades em novas interfaces com nomes claros e que reflitam sua função.

Ao final, explique por que essa nova organização é mais vantajosa para a manutenibilidade e flexibilidade do código.

```
// Código original com violação do SRP
interface Modem
{
   // Contexto: Gerenciamento de Conexão. O dial é responsável por iniciar uma chamada.
   public void dial(String pno);
   // Contexto: Gerenciamento de Conexão. O hangup é responsável por encerrar a chamada.
   public void hangup();
   // Contexto: Comunicação de Dados. O send é responsável por enviar informações.
   public void send(char c);
   // Contexto: Comunicação de Dados. O recv é responsável por receber informações.
   public char recv();
}
```

Exercícios

Princípio da Responsabilidade Única (Single Responsibility Principle – SRP)

Exercício 2: A Figura 8-4 mostra uma violação comum do SRP. A classe **Employee** contém regras de negócio e controle de persistência.

Essas duas responsabilidades **quase nunca devem ser misturadas**. As regras de negócio tendem a mudar com frequência e, embora a persistência possa não mudar tão frequentemente, ela o faz por razões completamente diferentes. **Vincular as regras de negócio ao subsistema de persistência é pedir por problemas.**

Para solucionar essa violação do SRP, proponha uma implementação em Java utilizando o padrão de projeto **Facade (**https://refactoring.guru/pt-br/design-patterns/facade **)**. Simule um subsistema de persistência criando um pacote à parte e teste a sua implementação em uma programa Java.

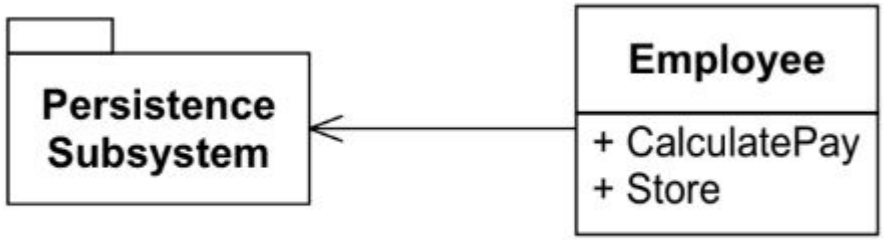

Figure 8-4 Coupled Persistence

Exercícios

Princípio do Aberto/Fechado (Open-Closed Principle – OCP)

Exercício 3: Crie um projeto Java com os códigos fornecidos (sem modificá-los).

```java
public class Main {
    public static void drawAllShapes(List<Object> shapes) {
        for (Object s : shapes) {
            if (s instanceof Circle) {
                ((Circle) s).drawCircle();
            } else if (s instanceof Square) {
                ((Square) s).drawSquare();
            }
        }
    }

    public static void main(String[] args) {
        List<Object> shapes = Arrays.asList(new Circle(), new Square());
        drawAllShapes(shapes);
    }
}
```

```java
public enum ShapeType {
    CIRCLE,
    SQUARE
}
```

```java
public class Square {
    private ShapeType type = ShapeType.SQUARE;

    public void drawSquare() {
        System.out.println("Desenha um quadrado");
    }

    public ShapeType getType() {
        return type;
    }
}
```

```java
public class Circle {
    private ShapeType type = ShapeType.CIRCLE;

    public void drawCircle() {
        System.out.println("Desenha um círculo");
    }

    public ShapeType getType() {
        return type;
    }
}
```

Exercícios

Princípio do Aberto/Fechado (Open-Closed Principle – OCP)

Exercício 3 (continuação): No projeto em que montou com os códigos do slide anterior, tente realizar a seguinte tarefa:

- Crie uma nova classe Triangle.
- Tente manter a função principal (main) funcionando sem modificações.

👉 **Você vai perceber que não é possível sem mexer na main ou no código existente, violando o princípio Aberto/Fechado.**

Exercícios

Princípio do Aberto/Fechado (Open-Closed Principle – OCP)

Exercício 3 (continuação): Agora modifique o código original para que o mesmo atenda ao princípio aberto/fechado:

- Crie uma interface Shape com o método draw().
- Faça Circle, Square e Triangle implementarem essa interface.
- Refatore a main para trabalhar apenas com Shape.

Por fim, agora responda (pode ser adicionado como comentário na Main no código):

- Por que agora conseguimos adicionar o Triangle sem modificar a main?
- Como isso se relaciona com o princípio Aberto/Fechado?

Exercícios

Princípio do Aberto/Fechado (Open-Closed Principle – OCP)

Exercício 4: O projeto que usa como base a classe **Shape**, precisa agora levar em consideração a ordem em que as formas geométricas são desenhadas. Por exemplo:

- Pode-se quer que a ordem de desenho seja **Circle, Rectangle ou Triangle.**
- Ou, pode-se querer a ordem **Rectangle, Triangle ou Circle.**
- Crie uma interface Shape com o método draw().

Agora verifique as indagações:

a) É possível adicionar ordem ao código do exercício anterior sem alterar o código-fonte pré-existente?
b) Refatore o código para que o código possa receber estratégias de ordenação sem ferir o OCP. Explique a sua solução que atende OCP (pode responder como comentário no código-fonte do Main)

Exercícios

Princípio do Aberto/Fechado (Open-Closed Principle – OCP)

Exercício 4: O projeto que usa como base a classe **Shape**, precisa agora levar em consideração a ordem em que as formas geométricas são

DICAS:

A responsabilidade de ordenar não deve estar na **Main** nem em cada **Shape**.

Use uma estratégia separada e explore a API Stream.sorted() para definir a ordem de desenho com um **Comparator**. Seguem sugestão de links sobre essa API:

- https://medium.com/@AlexanderObregon/javas-stream-sorted-method-explained-52b9b25e9f84
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#sorted-java.util.Comparator-

Princípio de Substituição de Liskov (Liskov's Substitution Principle - LSP)

Subtipos devem ser substituíveis pelo seus tipos base

Os mecanismos principais por trás do LSP são: **abstração** e **polimorfismo.**

Barbara Liskov escreveu esse princípio em, 1988

O que se quer aqui é algo como a seguinte propriedade de substituição: se para cada objeto o_1 do tipo S houver um objeto o_2 do tipo T tal que para todos os programas P definidos em termos de T, o comportamento de P permanece inalterado quando o_1 é substituído por o_2, então S é um subtipo de T.