DES-SIS-II - A4 Texto de apoio

Site: EAD Mackenzie
Tema: DESENVOLVIMENTO DE SISTEMAS II {TURMA 03B} 2023/1
Livro: DES-SIS-II - A4 Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .
Data: sexta, 28 abr 2023, 00:38

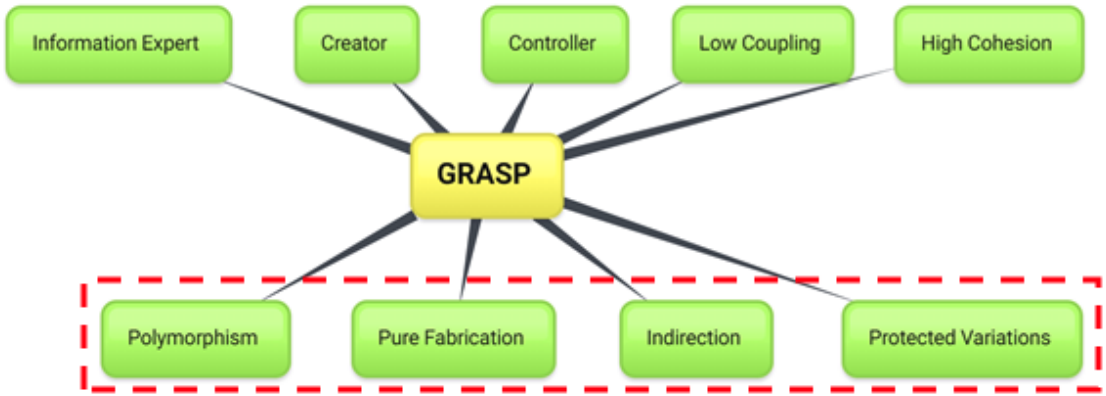
Descrição

Índice

1. APROFUNDANDO EM GRASP

Já sabemos que padrões representam boas práticas no projeto e desenvolvimento de software. Essas práticas já foram testadas em diferentes contextos e situações, de forma que o trabalho do projetista de software é adaptar a solução proposta pelo padrão para o contexto do problema que o software pretende resolver.

Na aula passada, estudamos cinco – de um total de nove – padrões GRASP (General Responsibility Assignment Software Patterns – Padrões de Software para Atribuição Geral de Responsabilidades): Information Expert, Creator, Controller, Low Coupling e High Cohesion. Na aula de hoje, continuaremos nosso estudo com os demais padrões GRASP: Polymorphism, Pure Fabrication, Indirection e Protected Variations.



Fonte: Elaborada pelo autor.

Nesta aula, prosseguiremos com os exemplos iniciados na aula anterior.

> Atenção! Se você não se lembra de onde paramos, dê uma olhadinha no conteúdo da Aula 3 antes de iniciar esta aula.

Sem mais delongas, vamos lá!

2. PADRÕES GRASP

Polimorfismo

Este termo é bastante importante em Programação Orientada a Objetos e você já deve ter ouvido falar dele. O termo vem do grego (“muitas formas”) e se refere a diversos aspectos do desenvolvimento com orientação a objetos.

De modo geral, há quatro maneiras de se obter polimorfismo, de acordo com a classificação clássica de dois pesquisadores da área de linguagens da programação (Cardelli e Wegner), que dividem o polimorfismo em dois grandes tipos:

- Polimorfismo universal: quando não há uma limitação prévia do número de tipos diferentes (classes) envolvidas. Dentro deste tipo de polimorfismo, incluímos o polimorfismo por inclusão e o polimorfismo universal paramétrico.

> No contexto deste componente curricular, o que nos interessa neste momento é estudar o polimorfismo por inclusão, que será abordado logo em seguida. Mas, para não aguçar sua curiosidade, o polimorfismo universal paramétrico é quando usamos generics em Java ou templates em C++, por exemplo.

- Polimorfismo ad-hoc: quando há uma limitação prévia da quantidade de tipos (classes) envolvidos. Nesta categoria, inclui-se a sobrecarga e a coerção.

> Na Aula 2, na qual falamos sobre refatoração, chegamos a mencionar a sobrecarga de métodos. A coerção é um tipo de polimorfismo que não depende do projetista de software, mas sim das particularidades de cada linguagem de programação. Grosso modo, a coerção é a conversão “automática” entre tipos, proporcionada por algumas linguagens.

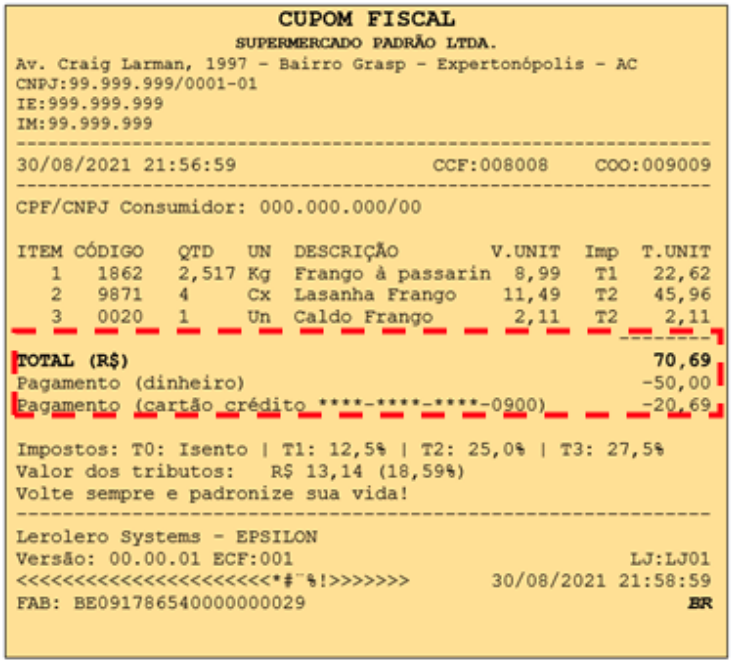
> Você deve estar sentindo falta de outro termo bastante comum quando aprendemos a programar em uma linguagem orientada a objetos

como a Java: onde entra a sobreposição? Veremos que a sobreposição é uma consequência do polimorfismo universal por inclusão.

Polymorphism – Exemplo

Partiremos para um exemplo, pois fica mais fácil para você compreender o padrão:

Lembra de nosso sistema do supermercado? Pois bem, se você se recorda do cupom fiscal, repare que o cliente fez parte do pagamento em dinheiro (R$ 50,00) e o restante no cartão de crédito com final 0900 (os R$ 20,69 que faltavam). Pelo jeito, além de aceitar distintos meios de pagamento, nosso supermercado aceita que o cliente realize um único pagamento usando mais de um meio (isso é uma prática comum no comércio em geral). Como nosso sistema dará conta deste requisito?



CUPOM FISCAL

SUPERMERCADO PADRÃO LTDA.

Av. Craig Larman, 1997 - Bairro Grasp - Expertonópolis - AC
CNPJ:99.999.999/0001-01
IE:999.999.999
IM:99.999.999

30/08/2021 21:56:59 CCF:008008 COO:009009

CPF/CNPJ Consumidor: 000.000.000/00

ITEM	CÓDIGO	QTD	UN	DESCRIÇÃO	V.UNIT	Imp	T.UNIT
1	1862	2,517	Kg	Frango à passarin	8,99	T1	22,62
2	9871	4	Cx	Lasanha Frango	11,49	T2	45,96
3	0020	1	Un	Caldo Frango	2,11	T2	2,11

TOTAL (R$) **70,69**
Pagamento (dinheiro) -50,00
Pagamento (cartão crédito ****-****-****-0900) -20,69

Impostos: T0: Isento | T1: 12,5% | T2: 25,0% | T3: 27,5%
Valor dos tributos: R$ 13,14 (18,59%)
Volte sempre e padronize sua vida!

Lerolero Systems - EPSILON
Versão: 00.00.01 ECF:001 LJ:LJ01
<<<<<<<<<<<<<<<<<<<<<<*#¨%!>>>>>>> 30/08/2021 21:58:59
FAB: BE091786540000000029 **BR**

Fonte: Elaborada pelo autor.

Uma implementação que pode vir imediatamente a sua mente é: crie uma classe chamada Pagamento e identifique os tipos de pagamento com um número para cada tipo (DINHEIRO=0, CRÉDITO=1 etc.):

```
class Pagamento
{
    private int tipo; //DINHEIRO=0, CRÉDITO=1, ...
    private float valor;
    public Pagamento (int tipo, float valor)
    {
        this.tipo = tipo;
        this.valor = valor;
    }
}
```

Parece uma boa solução? Bom, ao menos é simples. Mas ela tem várias fragilidades! Vamos acompanhar uma por uma, realizar algumas refatorações (lembra da Aula 2?) para, em seguida, apresentar o padrão GRASP que pode resolver o problema.

Problema 1: É muito fácil acontecer uma entrada de dados errada (por exemplo, entrar com um valor inexistente para tipo de pagamento, como 11 – é muito frequente digitar uma tecla em duplicidade).
Ainda que você trate isso com uma mensagem de erro, não é muito amigável.

Fonte: Elaborada pelo autor.

Algumas linguagens de programação têm formas de minimizar esse tipo de erro. A Java, por exemplo, possui uma construção chamada enum, que pode ser usada nesses casos, limitando os valores possíveis que podem ser atribuídos a uma variável desse tipo.

```
enum TiposPagamento
{
    DINHEIRO,
    CRÉDITO,
    DÉBITO,
    PIX
    //...
}


class Pagamento
{
    private TiposPagamento tipo;
    private float valor;
    public Pagamento (TiposPagamento tipo, float valor)
    {
        this.tipo = tipo; //TiposPagamento.DINHEIRO, ...
        this.valor = valor;
    }
}
```

Será que nossos problemas terminaram? Bom, não exatamente... Veja que cada tipo de pagamento pode ter dados extras: por exemplo, pagamentos com cartão de crédito precisam do número do cartão, nome impresso no cartão, data de validade e código verificador. Outros tipos de pagamento (débito, PIX, cheque etc.) possivelmente precisarão de outros dados. E como resolver isso? Com um código cheio de condicionais (*if – else*)?

```
class Pagamento
{
    private int tipo;
    private float valor;
    public Pagamento (int tipo, float valor)
    {
        this.tipo = tipo;
        if (tipo == TiposPagamento.CRÉDITO)
            // código para obter os dados do cartão de crédito
          else if (tipo == TiposPagamento.DÉBITO)
            // código para obter os dados do cartão de débito
            else if tipo == TiposPagamento.PIX)
            // código para obter os dados de PIX
            //...
        this.valor = valor;
    }
}
```

Problema 2: Um código com muitas condicionais (especialmente `if – else` aninhados) tende a não ser eficiente, além de ser de difícil manutenção. Imagine se o Banco Central cria um tipo de pagamento (como aconteceu com o PIX em 2021)? Todo o código da classe Pagamento teria que passar por alteração – e onde quer que exista essa lógica condicional!

Fonte: Elaborada pelo autor.

Qual a solução, então? Polimorfismo!

Sistematizando:

- **Problema**: Como lidar com alternativas baseadas no tipo? Como criar componentes de software “plugáveis”?
- **Solução**: Quando o comportamento do sistema varia de acordo com o tipo (classe), use herança e métodos polimórficos para implementar os comportamentos que variam em determinadas classes.

> **Regra de ouro**: não teste o tipo de um objeto e nunca use a lógica condicional para executar várias alternativas com base no tipo!

Veremos como fica este exemplo no código:

```
abstract class Pagamento
{
    private float valor;
    protected String msg;
    public Pagamento (float valor){
        this.valor = valor;
    }
    public abstract void registra();
}

class Dinheiro extends Pagamento{
    public void registra(){
        msg = "pagto em dinheiro"; }
    public Dinheiro (float valor){
        super(valor);
    }
}
```

```
class CartãoDeCrédito extends Pagamento{
    private String portador, num;
    private java.util.Date validade;
    private int cvc;
    public void registra(){
        msg = "pagto em cartão de crédito"; }
    public CartãoCrédito (float valor,
                          String portador,
                          String num,
                          java.util.Date validade,
                          int cvc){
        super(valor);
        this.portador = portador;
        this.num = num;
        this.validade = validade;
        this.cvc = cvc;
    }
}
```

Veja que, no código:

- A classe Pagamento foi marcada como abstract, ou seja, não faz sentido termos instâncias dessa classe, pois teremos instâncias de suas classes-filhas (Dinheiro, CartãoDeCrédito etc.). Note que toda classe abstrata precisa ter ao menos um método abstrato, ou sem implementação (afinal, quem implementará este método serão as classes-filhas).
- **Este tipo de polimorfismo é classificado como polimorfismo universal por inclusão.**
- As classes-filhas (Dinheiro, CartãoDeCrédito etc.) devem obrigatoriamente implementar o método abstrato herdado da classe-mãe (ou então teriam que ser abstratas também).
- **É aqui que entra a sobreposição, que é uma característica deste tipo de polimorfismo.**

E, como usar essas classes? Por exemplo, elas podem ser usadas na classe Compra da seguinte forma:

```
class Compra
{
    //...
    private java.util.ArrayList<Pagamento> pagamentos;
    public void registraPagamento(Pagamento p)
    {
        pagamentos.add(p);
    }
   //...
}
```

E, em algum ponto do código em que a classe Compra é instanciada, os produtos registrados, os dois pagamentos de nosso exemplo podem aparecer desta forma:

```
Compra c = new Compra();
//registra produtos
c.registraPagamento (new Dinheiro(50.0f));
c.registraPagamento (new CartãoDeCrédito(20.69f,
                                         "Zé das Couves",
                                         "0800070099990900",
                                         new java.util.Date(...),
                                         999);
```

Note que uma das “regras” do polimorfismo é que uma referência para a classe-mãe pode ser usada para referenciar instâncias das classes-filhas. Ou seja: o método registraPagamento da classe Compra, em sua declaração, espera como parâmetro uma referência para Pagamento. Porém, sabemos que Pagamento é abstrata e não pode ser instanciada. Quando o método registraPagamento é chamado, são passados dois objetos, um da classe Dinheiro e outro da classe CartãoDeCrédito. Faça, agora, a atividade Praticando, disponível no ambiente virtual.

Pure Fabrication

- **Problema**: A qual classe atribuir certas responsabilidades, quando não se quer violar os princípios de alta coesão e baixo acoplamento, ou outras boas práticas de projeto, mas as soluções oferecidas pelo padrão Information Expert, por exemplo, não são apropriadas?
- **Solução**: Atribua um conjunto altamente coeso de responsabilidades a uma classe que não representa um conceito de domínio de problema – algo inventado, para dar suporte de alta coesão, baixo acoplamento e reuso.

Esta classe é uma invenção da imaginação! Idealmente, as responsabilidades

atribuídas a esta classe "fabricada" suportam alta coesão e baixo acoplamento, de modo que seu design é muito limpo ou puro – daí o nome Pure Fabrication.

Veja o seguinte exemplo: sabemos que muitas compras são feitas em um supermercado todos os dias – ou seja, vários objetos da classe Compra acabam sendo instanciados. Porém, quem é responsável por armazenar esses objetos (para fechar o balanço do dia, por exemplo)?

> Sim, você pode responder: "Ah, o banco de dados!" – OK, mas lembre-se de que estamos modelando nosso sistema no nível lógico, e não podemos assumir compromissos com detalhes de implementação (como garantir que teremos um banco de dados) na fase de projeto de classes.

Neste ponto, o padrão Pure Fabrication pode ajudar: inventaremos uma classe cuja principal responsabilidade é armazenar as compras do dia. Vamos chamar essa classe de *ContêinerDeCompras* (em geral, essas classes com este tipo de responsabilidade – armazenar instâncias de outra classe – são chamadas de classes contêineres). Esta classe ContêinerDeCompras não "existe" em nosso domínio do supermercado (diferente de Compra, Produto etc.) – ela é um exemplo de pura fabricação.

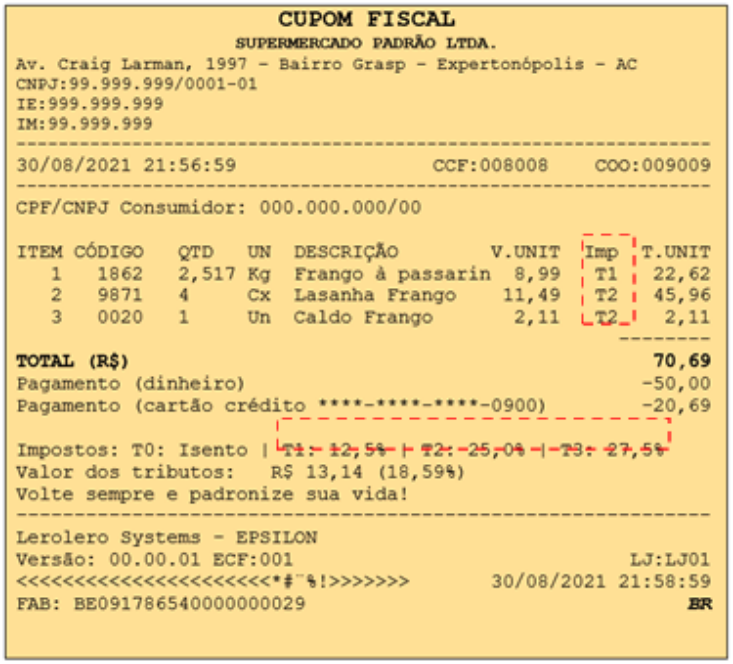
Fonte: Elaborada pelo autor.

box

Indirection

- **Problema**: Para qual classe atribuir uma responsabilidade, para evitar o acoplamento direto entre os objetos de duas ou mais classes? Como desacoplar objetos para que o baixo acoplamento seja suportado e o potencial de reutilização continue alto?
- **Solução**: Atribuir a responsabilidade a uma classe intermediária para mediar as demais classes para que não fiquem diretamente acopladas. Essa classe intermediária cria uma indireção entre as demais classes.

Vamos a um exemplo. Voltemos ao nosso cupom fiscal:

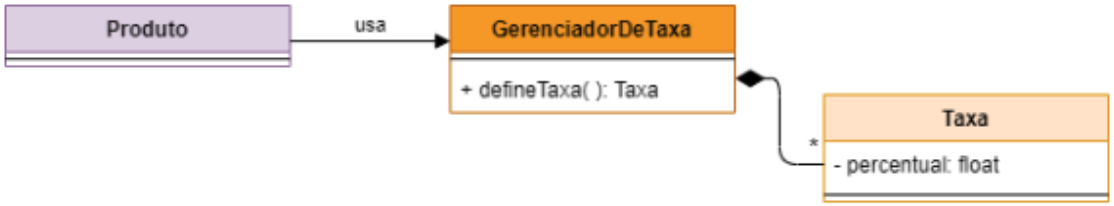


```
                    CUPOM FISCAL
                SUPERMERCADO PADRÃO LTDA.
Av. Craig Larman, 1997 - Bairro Grasp - Expertonópolis - AC
CNPJ:99.999.999/0001-01
IE:999.999.999
IM:99.999.999
------------------------------------------------------------
30/08/2021 21:56:59                CCF:008008    COO:009009
------------------------------------------------------------
CPF/CNPJ Consumidor: 000.000.000/00

ITEM CÓDIGO   QTD   UN  DESCRIÇÃO        V.UNIT  Imp  T.UNIT
   1   1862   2,517 Kg  Frango à passarin  8,99  T1   22,62
   2   9871   4     Cx  Lasanha Frango    11,49  T2   45,96
   3   0020   1     Un  Caldo Frango       2,11  T2    2,11
                                                    --------
TOTAL (R$)                                          70,69
Pagamento (dinheiro)                               -50,00
Pagamento (cartão crédito ****-****-****-0900)     -20,69

Impostos: T0: Isento | T1: 12,5% | T2: 25,0% | T3: 27,5%
Valor dos tributos:   R$ 13,14 (18,59%)
Volte sempre e padronize sua vida!
------------------------------------------------------------
Lerolero Systems - EPSILON
Versão: 00.00.01 ECF:001                           LJ:LJ01
<<<<<<<<<<<<<<<<<<<<<<*#%!>>>>>>>      30/08/2021 21:58:59
FAB: BE091786540000000029                             BR
```

Fonte: Elaborada pelo autor.

Note que cada produto está associado a um tipo de taxa. Porém, acontece que o tipo de taxa pode variar entre os estados (por exemplo, a Lasanha de Frango paga a taxa T2 no estado do Acre, onde fica nosso Supermercado Padrão. Mas e se a rede expandir e desejar abrir filiais em Tocantins ou São Paulo? Pode ser que, neste caso, o próprio tipo de taxa mude – por exemplo, pode ser que em Tocantins se aplique a taxa T3).

Dessa forma, o uso de Indirection pode desacoplar as classes Produto e Taxa, permitindo que a Taxa varie sem que isso afete o Produto. Veja o exemplo a seguir (note que nem todos os atributos e métodos são exibidos, por simplificação):



Fonte: Elaborada pelo autor.

Protected Variations

- **Problema**: Como projetar sistemas de maneira que as variações ou instabilidade em seus objetos não tenham um impacto indesejável sobre os outros?
- **Solução**: Identificar classes que representam pontos de variação ou instabilidade que podem ser previstos no projeto e atribuir responsabilidades a uma nova classe de acesso estável que se conecte a elas.

> Note que este padrão está bastante associado aos padrões Pure Fabrication e Indirection.

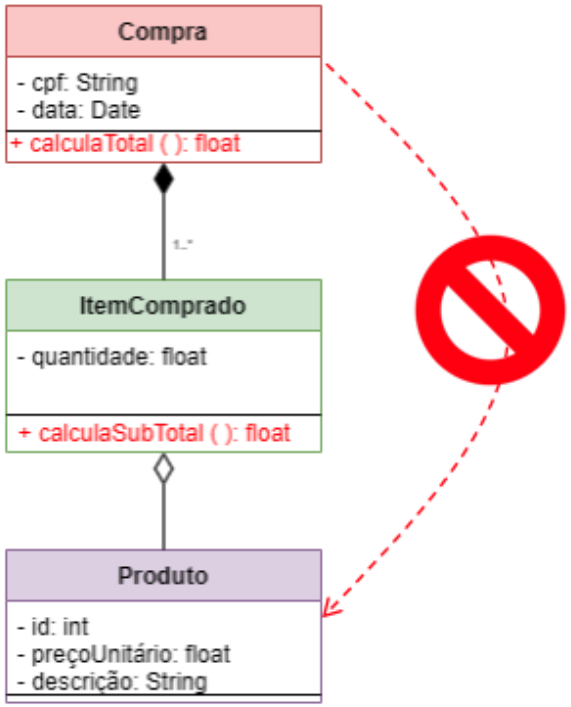
Uma das aplicações diretas do padrão Protected Variations é um importante princípio de desenvolvimento conhecido como Don't Talk to Strangers, ou Law of Demeter. Este princípio impõe restrições aos objetos para os quais um determinado objeto deve enviar mensagens dentro de um método. Ele afirma que, dentro de um método, as mensagens devem ser enviadas apenas para os seguintes objetos:

-

- O próprio objeto.
- Um objeto recebido como parâmetro do método.
- Um objeto que seja atributo deste objeto.
- Um elemento de uma coleção que é um atributo deste objeto.

- Um objeto criado dentro do método.

Por exemplo, no diagrama desenvolvido na aula anterior, um objeto da classe Compra nunca deveria enviar mensagens diretamente a objetos da classe Produto, pois essas classes são “estranhas” entre si (não há relacionamento direto entre elas previsto no diagrama).



Fonte: Elaborada pelo autor.