Padrão para atribuir responsabilidades: Baixo Acoplamento

Problema

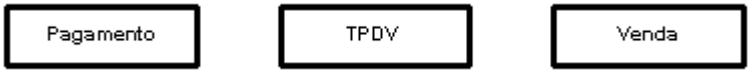
- Como minimizar dependências e maximizar o reuso?
- O acoplamento é uma medida de quão fortemente uma classe está conectada, possui conhecimento ou depende de outra classe
- Com fraco acoplamento, uma classe não é dependente de muitas outras classes
- Com uma classe possuindo forte acoplamento, temos os seguintes problemas:
  - Mudanças em uma classe relacionada força mudanças locais à classe
  - A classe é mais difícil de entender isoladamente
  - A classe é mais difícil de ser reusada, já que depende da presença de outras classes

Solução

- Atribuir responsabilidades de forma a minimizar o acoplamento

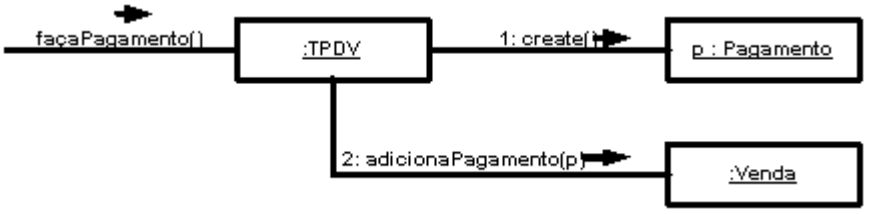
Exemplo

- Considere o seguinte diagrama parcial de classes no estudo de caso

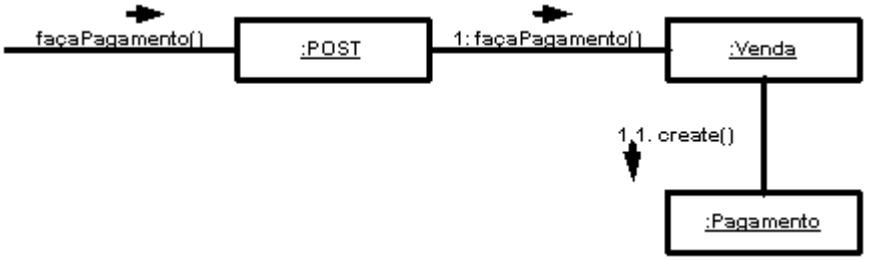


- Suponha que temos que criar um Pagamento e associá-lo a uma Venda
- Que classe deveria ter essa responsabilidade?
- Alternativa 1: No mundo real, um TPDV "registra" um pagamento e o padrão Creator sugere que TPDV poderia criar Pagamento

- TPDV deve então passar o pagamento para a Venda
- Veja o resultado abaixo



- Alternativa 2: Criar o Pagamento com Venda e associá-lo à Venda
  - Veja o resultado abaixo



- Supondo que a Venda deva ter conhecimento do pagamento (depois da criação) de qualquer jeito, a alternativa 2 tem menos acoplamento (TPDV não está acoplado a Pagamento)
  - Dois padrões (Creator e Low Coupling) sugeriram diferentes soluções
  - Minimizar acoplamento ganha

Discussão

- Minimizar acoplamento é um dos princípios de ouro do projeto OO
- Acoplamento se manifesta de várias formas:
  - X tem um atributo que referencia uma instância de Y
  - X tem um método que referencia uma instância de Y
    - Pode ser parâmetro, variável local, objeto retornado pelo método
  - X é uma subclasse direta ou indireta de Y
  - X implementa a interface Y

- A herança é um tipo de acoplamento particularmente forte
  - Uma seção futura aprofunda o assunto
- Não se deve minimizar acoplamento criando alguns poucos objetos monstruosos (God classes)
  - Exemplo: todo o comportamento numa classe e outras classes usadas como depósitos passivos de informação
- Tipos de acoplamentos (do menos ruim até o pior)
  - Acoplamento de dados
  - Acoplamento de controle
  - Acoplamento de dados globais
  - Acoplamento de dados internos

Acoplamento de dados

- Situações
  - Saída de um objeto é entrada de outro
  - Uso de parâmetros para passar itens entre métodos
- Ocorrência comum:
  - Objeto a passa objeto x para objeto b
  - Objeto x e b estão acoplados
    - Uma mudança na interface de x pode implicar em mudanças a b
- Exemplo:

```
class Servidor {
  public void mensagem(MeuTipo x) {
    // código aqui
    x.façaAlgo(Object dados); // dados e x estão acoplados
                              // (se interface de dados muda
    // mais código
  }
}
```

  - Exemplo pior:
    - Objeto a passa objeto x para objeto b
    - x é um objeto composto ou agregado (contém

outro(s) objeto(s))
- Objeto b deve extrair objeto y de dentro de x
- Há acoplamento entre b, x, representação interna de x, y
- Exemplo: ordenação de registros de alunos por matrícula

```
class Aluno {
  String nome;
  long   matrícula;

  public String getNome() { return nome; }
  public long   getMatrícula() { return matrícula; }

  // etc.
}

ListaOrdenada listaDeAlunos = new ListaOrdenada();
listaDeAlunos.add(new Aluno(...));
//etc.
```

- Agora, vamos ver os problemas:

```
class ListaOrdenada {
  Object[] elementosOrdenados = new Object[tamanhoAdequado];

  public void add(Aluno x) {
    // há código não mostrado aqui
    long matrícula1 = x.getMatrícula();
    long matrícula2 = elementosOrdenados[k].getMatrícula();
    if(matrícula1 < matrícula2) {
      // faça algo
    } else {
      // faça outra coisa
    }
  }
```

- O problema da solução anterior é que há forte acoplamento
  - ListaOrdenada sabe muita coisa de Aluno
    - O fato de que a comparação de alunos é feito com a matrícula
    - O fato de que a matrícula é obtida com getMatrícula()

- O fato de que matrículas são long (representação de dados)
- Como comparar matrículas (com <)

- O que ocorre se mudarmos qualquer uma dessas coisas?

- Solução 2: mande uma mensagem para o próprio objeto se comparar com outro

```
class ListaOrdenada {
  Object[] elementosOrdenados = new Object[tamanhoAdequado];

  public void add(Aluno x) {
    // código não mostrado
    if(x.compareTo(elementosOrdenados[K]) < 0) {
      // faça algo
    } else {
      // faça outra coisa
    }
  }
```

- Reduzimos o acoplamento escondendo informação atrás de um método
- Problema: ListaOrdenada só funciona com Aluno
- Solução 3: use interfaces para desacoplar mais ainda

```
interface Comparable {
  public int compareTo(Object outro);
}

class Aluno implements Comparable {
  public int compareTo(Object outro) {
    // compare registro de aluno com outro
    // retorna valor < 0, 0, ou > 0 dependendo da comparaçã
  }
}

class ListaOrdenada {
  Object[] elementosOrdenados = new Object[tamanhoAdequado];

  public void add(Comparable x) {
    // código não mostrado
    if(x.compareTo(elementosOrdenados[K]) < 0) {
      // faça algo
    } else {
```

```
      // faça outra coisa
    }
  }
```

- Em C++, teria outras soluções possíveis
  - Apontador de função
  - Apontador de função com tipos genéricos (templates)

Acoplamento de controle

- Passar flags de controle entre objetos de forma que um objeto controle as etapas de processamento de outro objeto
- Ocorrência comum:
  - Objeto a manda uma mensagem para objeto b
  - b usa um parâmetro da mensagem para decidir o que fazer

```
class Lampada {
  public final static ON = 0;

  public void setLampada(int valor) {
    if(valor == ON) {
      // liga lampada
    } else if(valor == 1) {
      // desliga lampada
    } else if(valor == 2) {
      // pisca
    }
  }
}

Lampada lampapa = new Lampada();
lampada.setLampada(Lampada.ON);
lampada.setLampada(2);
```

- Solução: decompor a operação em múltiplas operações primitivas

```
class Lampada {
  public void on() { // liga lampada }
  public void off() { // desliga lampada }
  public void pisca() { // pisca }
```

```
}

Lampada lampada = new Lampada();
lampada.on();
lampada.pisca();
```

- Ocorrência comum:
  - Objeto a manda mensagem para objeto b
  - b retorna informação de controle para a
  - Exemplo: retorno de código de erro

```
class Teste {
  public int printFile(File aImprimir) {
    if(aImprimir está corrompido ) {
      return CORRUPTFLAG;
    }
    // etc. etc.
  }
}

Teste umTeste = new Teste();
int resultado =  umTese.printFile(miniTeste);
if(resultado == CORRUPTFLAG) {
  // oh! oh!
} else if(resultado == -243) {
  // etc. etc.
```

- Solução: use exceções

```
class Teste {
  public int printFile(File aImprimir) throws PrintExceptio
    if(aImprimir está corrompido ) {
      throw new PrintExeception();
    }
    // etc. etc.
  }
}

try {
  Teste umTeste = new Teste();
  umTeste.printFile(miniTeste);
} catch(PrintException printError) {
  // mail para a turma: não tem miniteste amanhã!
}
```

Acoplamento de dados globais

- Dois ou mais objetos compartilham estruturas de dados globais
- É um acoplamento muito ruim pois está escondido
  - Uma chamada de método pode mudar um valor global e o código não deixa isso aparente
- Um tipo de acoplamento muito ruim

Acoplamento de dados internos

- Um objeto altera os dados locais de um outro objeto
- Ocorrência comum:
  - Friends em C++
  - Dados públicos, package visibility ou mesmo protected em java
- Use com cuidado!

Consequências

- Uma classe fracamente acoplada não é afetada (ou pouco afetada) por mudanças em outras classes
- Simples de entender isoladamente
- Reuso mais fácil

Exercício para casa

- Discuta os problemas de acoplamento (ou sua ausência) neste exemplo

programa