Acoplamento e Coesão

Acoplamento é uma medida “inter” componentes. Isto é, é uma medida entre componentes de um conjunto. No caso de um sistema é uma medida do relacionamento entre subsistemas. No caso de um programa composto de funções, é uma medida do relacionamento entre funções. Os dois espectros dessa medida: são alto acoplamento e baixo acoplamento. Componentes que têm baixo ou fraco acoplamento são considerados mais independentes um do outro. O inverso vale para alto ou forte acoplamento.

Coesão é uma medida “intra” componentes. Isto é, procura medir um componente individualmente. No caso de um sistema, mede-se a coesão de cada sub-sistema. No caso de um programa organizado por funções, mede-se a coesão de cada função. A coesão pode ser alta ou baixa. Obter coesão alta é imprescindível em todo sistema bem organizado. Entende-se coesão alta, quando os integrantes de um componente estão relacionados a um tema comum, isto é tem o mesmo objetivo, fazem uma única coisa. A coesão alta é também conhecida como coesão funcional. Um bom indicativo de componentes com baixa coesão é o título ou nome do componente: todo componente que se utiliza do conector e ou do conector ou é um forte candidato para ser classificado como tendo baixa coesão. Veja que o título dessa nota (propositalmente) apresenta uma baixa coesão.

O conceito de organização modular é fundamental em Teoria Geral de Sistemas, por um motivo óbvio: um sistema é sempre um subsistema de um sistema mais abstrato, ou de nível hierárquico mais alto, representando-se a idéia da decomposição como uma hierarquia. Usa-se o termo modularidade para refletir a propriedade de algo ter uma organização modular. Veja que usei o termo subsistema, mas poderia ter utilizado o termo sub-módulo.

Organização Modular

A organização modular tem como finalidade básica auxiliar o gerenciamento de sistemas complexos.

Creio que um bom resumo sobre organização modular é texto inicial do Capítulo 3 e do Capítulo 4 do livro de Abelson, Sussman and Sussman. Cito algumas partes desses dois textos a seguir: a) “A Síntese efetiva de programas também precisa de princípios organizacionais que nos guie na formulação do desenho de um programa. Em particular, precisamos de estratégias para nos ajudar a estruturar grandes sistemas de tal maneira que eles tenham uma organização modular, isto é que eles possam ser divididos “naturalmente” em partes coerentes que possam ser separadamente desenvolvidas e mantidas” e b) “Em nosso estudo de desenho de programas, nos vimos que especialistas controlam a complexidade de seus desenhos usando técnicas gerais usadas por desenhistas de todos os sistemas complexos. Eles

combinam elementos primitivos para formar objetos compostos, eles abstraem objetos compostos para formar blocos de construção de alto-nível, e eles preservam a modularidade através da adoção de visões de larga escala para a estrutura do sistema."

No tratado de Teoria Geral de Sistemas, Bertallanfy escreve na página 55 que o ditado "O todo é maior que a soma das partes" é na verdade o fato de que as características constitutivas não são explicadas a partir das características das partes isoladas, mas a partir dessas partes e das suas relações.

No entanto, fica no ar a questão de como identificar módulos, como fracionar um todo?

A idéia de que forma segue função é uma das mais importantes heurísticas para organizar de forma modular. O livro de Abelson, Sussman e Sussman, no início do Capitulo 3 menciona explicitamente essa estratégia.

No entanto, como saber se nosso todo está bem organizado modularmente. Ou seja, como qualificar o resultado de termos organizado o sistema de maneira modular. Qual a modularidade desse sistema?

Em sistemas que precisam ter independência entre suas partes para facilitar sua evolução é comum adotar-se a estratégia de módulos com forte independência de outros módulos e com forte unicidade. Isso caracteriza sistemas onde o acoplamento entre sub-sistemas é fraco e onde cada sub-sistema é fortemente coeso.

Em engenharia de software, principalmente nos escritos de Constantine e Yourdon as noções de acoplamento e coesão foram definidas de maneira clara, mas sem um formalismo adjacente. Essas noções foram herdadas dos conceitos de organização da teoria geral de sistemas e da cibernética.

Além do trabalho de Constantine e Yourdon é importantíssimo o trabalho do Prof. Parnas, que também tem relação com a teoria geral de sistemas. Além de ressaltar a idéia de independência de módulos no sentido tanto de coesão e acoplamento, coube a Parnas criar um conceito sobre o compartilhamento de informação que se tornou um dos conceitos gerais mais citados na literatura de software. Refiro-me ao conceito de "information hiding" ou de proteção da informação. Esse conceito prega que cada módulo ou sub-sistema deve guardar para si as informações que só a ele interessa. É claro, que o conceito, ao ser aplicado a sistemas, gera como efeito um acoplamento fraco, em função da heurística de minimizar o conhecimento compartilhado.

No mundo de orientação a objetos o conceito de acoplamento ficou um tanto confuso. Vários autores definem métricas de acoplamento como sendo uma enumeração de quantas relações existem entre as partes e não observam que o fundamental não é o número de relações que um objeto possui, mas sim a qualidade dessa relação. Aqui, qualidade entende-se pelo quanto cada objeto se deixa conhecer pelo outro. No entanto, o fato de que a orientação a objetos mistura taxonomia com mereologia[1] dificulta a aplicação do conceito de acoplamento fraco.

[1] (Lógica) teoria ou estudo lógico-matemático das relações entre as partes e o todo e das relações entre as partes no interior de um todo.

Existem métricas que procuram medir acoplamento e coesão, mas métricas sobre proteção de informação são raras e menos utilizadas.

É importante ressaltar que o acoplamento forte pode algumas vezes ser necessário. Um exemplo onde o acoplamento forte é positivo é no emprego em engenharia de produção do conceito de produção “just in time”. Em alguns casos essa forte dependência entre produtores e fornecedores é justificada pela redução de custos. Em software, muitas vezes um compartilhamento de memória comum é plenamente justificável em algumas situações, caracterizando um acoplamento forte entre os componentes que dividem esse recurso.

Uma heurística fundamental: o uso do conector “e” ou “ou” em um título de um módulo (ou subsistema) é um forte indicativo da falta de coesão!

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:

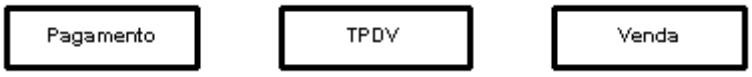
- Mudanças em uma classe relacionada força mudanças locais à classe
- A classe é mais difícil de entender isoladamente
- A classe é mais difícil de ser reusada, já que depende da presença de outras classes

Solução

Atribuir responsabilidades de forma a minimizar o acoplamento

Exemplo

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

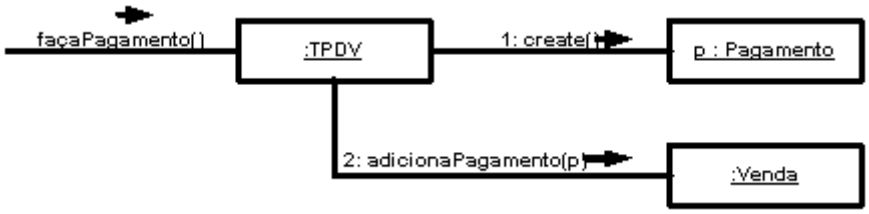


Suponha que temos que criar um Pagamento e associá-lo a uma Venda

Que classe deveria ter essa responsabilidade?

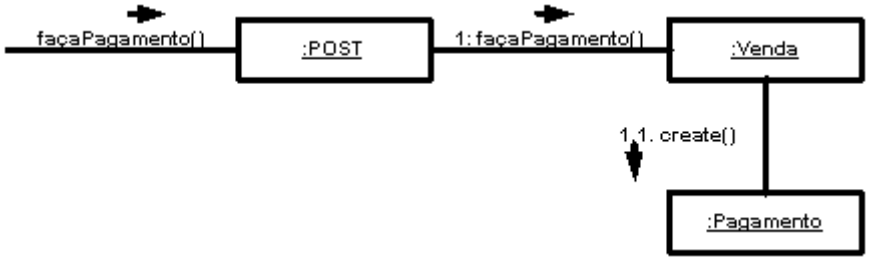
Alternativa 1: No mundo real, um TPDV "registra" um pagamento e o padrão Creator sugere que TPDV poderia criar Pagamento. TPDV deve então passar o pagamento para a Venda.

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.

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 mudar x terá que mudar)
    // 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 comparação
  }
}

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 int 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 PrintExeception {
    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 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.

Alta Coesão

Problema

Como gerenciar a complexidade?

A coesão mede quão relacionadas ou focadas estão as responsabilidades da classe

Também chamada de "coesão funcional" (ver à frente)

Uma classe com baixa coesão faz muitas coisas não relacionadas e leva aos seguintes problemas:

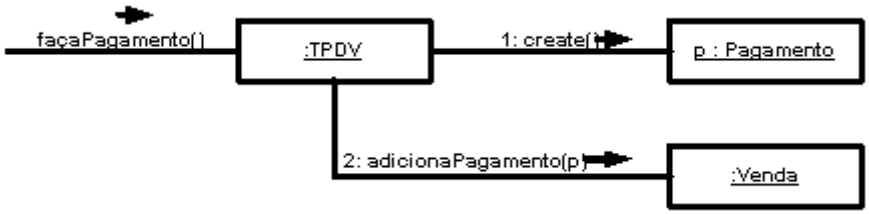
- Difícil de entender
- Difícil de reusar
- Difícil de manter
- "Delicada": constantemente sendo afetada por outras mudanças

Uma classe com baixa coesão assumiu responsabilidades que pertencem a outras classes

Solução
Atribuir responsabilidades que mantenham alta coesão

Exemplo
Mesmo exemplo usado para Low Coupling

Na primeira alternativa, TPDV assumiu uma responsabilidade de efetuar um pagamento (método façaPagamento())



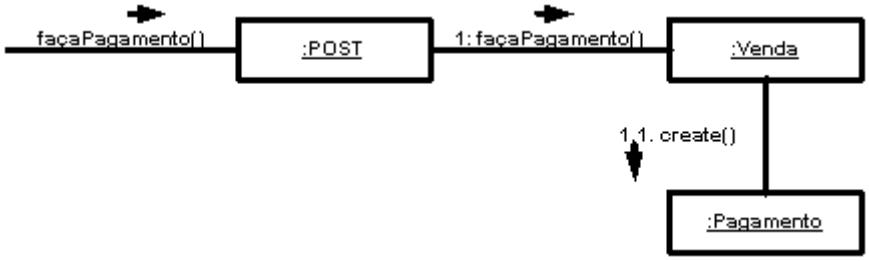
Até agora, não há problema

Mas suponha que o mesmo ocorra com várias outras operações de sistema:

- TPDV vai acumular um monte de métodos não muito focados
- Resultado: baixa coesão

A segunda alternativa delega façaPagamento() para a classe Venda

- Mantém maior coesão em TPDV



Discussão
Alta coesão é outro princípio de ouro que deve ser sempre mantido em mente durante o projeto

Tipos de coesão entre módulos:

- Coincidente (pior)
- Lógico
- Temporal
- Procedural
- De comunicação
- Sequencial
- Funcional (melhor)

Coesão coincidental

Há nenhuma (ou pouca) relação construtiva entre os elementos de um módulo

No linguajar OO:

- Um objeto não representa nenhum conceito OO
- Uma coleção de código comumente usado e herdado através de herança (provavelmente múltipla)

```
class Angu {
  public static int acharPadrão(String texto, String padrão) {
    // ...
  }
  public static int média(Vector números) {
    // ...
  }
  public static outputStream abreArquivo(string nomeArquivo) {
    // ...
  }
}

class Xpto extends Angu { // quer aproveitar código de Angu
  ...
}
```

Coesão lógica

Um módulo faz um conjunto de funções relacionadas, uma das quais é escolhida através de um parâmetro ao chamar o módulo

Semelhante a acoplamento de controle

Cura: quebrar em métodos diferentes

```
public void faça(int flag) {
  switch(flag) {
  case ON:
    // coisas para tratar de ON
    break;
  case OFF:
    // coisas para tratar de OFF
    break;
  case FECHAR:
    // coisas para tratar de FECHAR
    break;
  case COR:
    // coisas para tratar de COR
    break;
  }
}
```

Coesão temporal

Elementos estão agrupados no mesmo módulo porque são processados no mesmo intervalo de tempo

Exemplos comuns:

- Método de inicialização que provê valores defaults para um monte de coisas diferentes
- Método de finalização que limpa as coisas antes de terminar

```
procedure inicializaDados() {
  font = "times";
  windowSize = "200,400";
  xpto.nome = "desligado";
  xpto.tamanho = 12;
  xpto.localização = "/usr/local/lib/java";
}
```

Cura: usar construtores e destrutores

```
class Xpto {
  public Xpto() {
    this.nome = "desligado";
    this.tamanho = 12;
    this.localização = "/usr/local/lib/java";
  }
}
```

Outro exemplo: arquivo de configuração típico

```
[Macintosh]
EquationWindow=146,171,406,661
SpacingWindow=0,0,0,0

[Spacing]
LineSpacing=150%
MatrixRowSpacing=150%
MatrixColSpacing=100%
SuperscriptHeight=45%
SubscriptDepth=25%
LimHeight=25%
LimDepth=100%
LimLineSpacing=100%
NumerHeight=35%
DenomDepth=100%
FractBarOver=1pt
FractBarThick=0.5pt
SubFractBarThick=0.25pt
FenceOver=1pt
```

```
SpacingFactor=100%
MinGap=8%
RadicalGap=2pt
EmbellGap=1.5pt
PrimeHeight=45%

[General]
Zoom=200
CustomZoom=150
ShowAll=0
Version=2.01
OptimalPrinter=1
MinRect=0
ForceOpen=0
ToolbarDocked=1
ToolbarShown=1
ToolbarDockPos=1

[Fonts]
Text=Times
Function=Times
Variable=Times,I
LCGreek=Symbol,I
UCGreek=Symbol
Symbol=Symbol
Vector=Times,B
Number=Times

[Sizes]
Full=12pt
Script=7pt
ScriptScript=5pt
Symbol=18pt
SubSymbol=12pt
```

Coesão procedural

Associa elementos de acordo com seus relacionamentos procedurais ou algorítmicos

Um módulo procedural depende muito da aplicação sendo tratada

- Junto com a aplicação, o módulo parece razoável
- Sem este contexto, o módulo parece estranho e muito difícil de entender

Não pode entender o módulo sem entender o programa e as condições que existem quando o módulo é chamado

Cura: reprojete o sistema!

Coesão de comunicação

Todas as operações de um módulo operam no mesmo conjunto de dados e/ou produzem o mesmo tipo de dado de saída

Cura: isole cada elemento num módulo separado

"Não deveria" ocorrer em sistemas OO usando polimorfismo (classes diferentes para fazer tratamentos diferentes nos dados)

Coesão sequencial

A saída de um elemento de um módulo serve de entrada para o próximo elemento

Cura: decompor em módulos menores

Coesão funcional (a melhor)

Um módulo tem coesão funcional se as operações do módulo puderem ser descritas numa única frase de forma coerente

Num sistema OO:

- Cada operação na interface pública do objeto deve ser funcionalmente coesa
- Cada objeto deve representar um único conceito coeso

Exemplo: um objeto que esconde algum conceito ou estrutura de dados ou recurso e onde todos os métodos são relacionados por um conceito ou estrutura de dados ou recurso

- Meyer chama isso de "information-strength module"

Consequências

- Melhor claridade e facilidade de compreensão do projeto
- Simplificação da manutenção
- Frequentemente vai mão na mão com acoplamento fraco
- Com granularidade baixa e funcionalidade bem focada, aumenta o reuso