Coesão e Acoplamento em Sistemas OO

Article · March 2010

CITATIONS
0

READS
5,591

1 author:

Leandro Luque
Centro Paula Souza
32 PUBLICATIONS **84** CITATIONS

SEE PROFILE

Coesão e Acoplamento em Sistemas Orientados a Objetos

Melhorando a qualidade de seus projetos.

Conheça os conceitos de coesão e acoplamento e como eles podem influenciar no desenvolvimento de sistemas com maior qualidade.

De que se trata o artigo:

Neste artigo, é apresentada uma visão geral sobre os conceitos de coesão e acoplamento e sua importância no desenvolvimento de sistemas orientados a objetos de qualidade.

Para que serve:

Apresentar conceitos que podem auxiliar desenvolvedores na produção de sistemas orientados a objetos com maior qualidade, principalmente no que diz respeito à facilidade de manutenção e ao potencial de reuso.

Em que situação o tema útil:

O desenvolvimento de software de qualidade, dentro do prazo e com o custo estabelecido é essencial para a sobrevivência de qualquer organização desenvolvedora de software. A facilidade com que se dá a manutenção e o potencial de reuso de um software desempenham papel de destaque nesse contexto, e os conceitos de coesão e acoplamento podem auxiliar muito neste sentido.

Coesão e acoplamento em sistemas orientados a objetos:

Os conceitos de coesão e acoplamento, surgidos no contexto da análise e projeto estruturados, embora tenham um grande impacto na qualidade de sistemas, são geralmente desconhecidos ou negligenciados por desenvolvedores iniciantes de sistemas orientados a objetos. O desenvolvimento de softwares com alta coesão e fraco acoplamento facilita, entre outras coisas, a manutenção e o reuso. Assim sendo, o estudo desses conceitos e seus desdobramentos torna-se importante para melhorar a qualidade de projetos de software.

Muitos desenvolvedores de software percorreram caminhos que passaram pelo desenvolvimento procedural antes de chegarem à orientação a objetos.

Desenvolver um programa procedural significa entender e conceber o programa como um conjunto de procedimentos (também conhecidos como rotinas, sub-rotinas ou funções) que manipulam dados. Neste modelo de desenvolvimento, os procedimentos são geralmente as unidades de subdivisão ou modularidade de sistemas.

Diferentemente do paradigma orientado a objetos, no qual a subdivisão de sistemas é baseada no mapeamento de objetos do domínio do problema para o domínio da solução, o desenvolvimento procedural não possui uma semântica forte que oriente a subdivisão de sistemas. Por isso, os procedimentos ou conjuntos relacionados de procedimentos de sistemas procedurais muitas vezes tratam de aspectos distintos do sistema e, consequentemente, apresentam certas características, como um alto grau de interdependência, que dificultam a manutenção e o reuso.

Neste contexto, muitos conceitos e métricas foram definidos para avaliar e auxiliar a subdivisão de sistemas. Dois destes conceitos são especialmente importantes por terem grande influência na qualidade dos sistemas desenvolvidos: **coesão** e **acoplamento**.

Os conceitos de coesão e acoplamento surgiram em meados da década de 1960, a partir de um estudo conduzido por Larry Constantine sobre o motivo pelo qual certos tipos de módulos de sistemas eram definidos e da análise dos pontos positivos e negativos relativos a estes tipos. Estes conceitos foram bastante estudados e utilizados no contexto da análise e projeto estruturado de sistemas.

A falta de conhecimento ou o negligenciamento destes conceitos e métricas contribui para que muitos desenvolvedores de sistemas orientados a objetos, principalmente iniciantes, desenvolvam sistemas com características indesejáveis.

O objetivo desse artigo é apresentar uma introdução aos conceitos de **coesão** e **acoplamento**, buscando demonstrar o efeito de desprezá-los na modelagem de sistemas orientados a objetos através de alguns exemplos simples. Os exemplos são apresentados na forma de diagramas de classes da Linguagem de Modelagem Unificada-UML e código Java.

Coesão

Estendendo a definição clássica de coesão para o paradigma orientado a objetos, pode-se definir coesão de um sistema de software como a relação existente entre as responsabilidades de uma determinada classe e de cada um de seus métodos. Uma classe altamente coesa tem responsabilidades e propósitos claros e bem definidos, enquanto uma classe com baixa coesão tem muitas responsabilidades diferentes e pouco relacionadas.

Para se ter uma ideia de classes com níveis diferentes de coesão, vamos analisar dois exemplos de classes retiradas de um sistema bancário e de um sistema de locadora de vídeos, representadas nas **Figuras 1** e **2**, respectivamente.

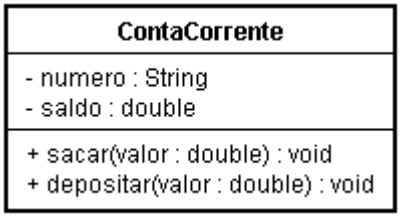


Figura 1. Exemplo de classe com alta coesão (Sistema bancário).

No exemplo do sistema bancário, a classe **ContaCorrente** é responsável por operações relacionadas apenas a contas correntes: **sacar** e **depositar**. Nenhum outro "assunto" não relacionado a contas correntes é tratado por esta classe. Assim sendo, pode-se classificar esta classe como altamente coesa.

Figura 2. Exemplo de classe com baixa coesão (Sistema de locadora de vídeos).

Por outro lado, a classe **Cliente**, do sistema de locadora (**Figura 2**), além de possuir atributos referentes a clientes, possui um método para a realização de aluguel de filmes. Essa modelagem é muito comum entre iniciantes e surge, na maioria das vezes, de um entendimento incorreto do mapeamento do diagrama de casos de uso para o diagrama de classes. Como no diagrama de casos de uso de uma locadora geralmente há uma associação entre o ator **Cliente** e o caso de uso "Alugar Filme", pode-se imaginar que seja razoável definir um método **alugarFilme()** na classe **Cliente**.

No entanto, esse tipo de definição faz com que a classe **Cliente** tenha baixa coesão (por tratar de assuntos diferentes: clientes e aluguel de filmes). O problema dessa modelagem é que o reuso e o entendimento das classes, quando observadas individualmente, ficam prejudicados.

Como exemplo ilustrativo do porquê de um baixo grau de coesão causar estes problemas, vamos imaginar um cenário bastante provável de reuso da classe **Cliente** em outros sistemas, como: Oficina Mecânica, Sistema Bancário, entre outros.

Nestes sistemas, não faz sentido que um cliente alugue filmes (**Figura 3**). O método **alugarFilme()**, embora pudesse simplesmente não ser chamado nesses sistemas, dificultaria o entendimento da classe **Cliente**, exigiria que a classe **Filme** também estivesse disponível e seria carregado como "lixo", podendo inclusive causar algum tipo de comportamento inesperado ou exceção, caso fosse executado.

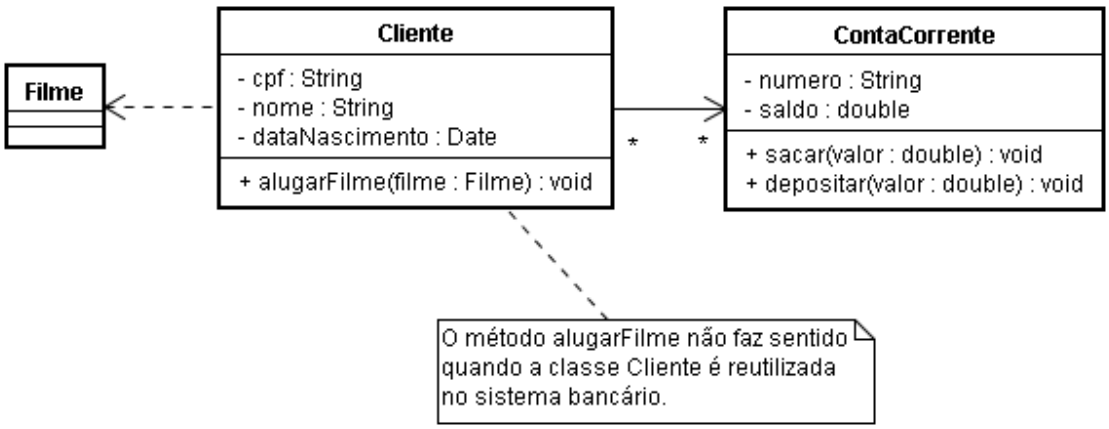


Figura 3. Exemplo de reuso da classe Cliente, da locadora de vídeos, em um sistema bancário.

Dos estudos de coesão e posteriores aprimoramentos, Stevens, Myers, Constantine e Yourdon propuseram uma classificação de coesão voltada para sistemas estruturados

baseada em sete níveis: coesão funcional, sequencial, comunicacional, procedural, temporal, lógica e coincidental.

Estes níveis variavam do melhor tipo de coesão (funcional), que significava que um módulo qualquer continha elementos que contribuíam para a execução de uma e apenas uma tarefa, ao pior tipo de coesão (coincidental), que significava que um módulo continha elementos que contribuíam para atividades sem relação significativa entre si.

Alguns autores procuraram fazer uma analogia direta destes níveis para sistemas orientados a objetos, principalmente no que se refere à coesão de métodos quando individualmente analisados. Outros autores definiram níveis de coesão de classes considerando o uso de atributos pelos métodos da classe. Quanto mais os métodos usavam os mesmos atributos de uma classe, maior era a coesão. Por fim, outros autores procuraram uma classificação diferenciada.

Uma destas classificações diferenciadas, proposta por Meilir Page-Jones, define três tipos de coesão para classes e três tipos para métodos.

Coesão de classes:

- **Coesão de instância mista:** ocorre quando uma classe tem características que são indefinidas para alguns dos objetos da classe.
  Este tipo de coesão pode ser observado na classe **Pessoa**, representada na **Figura 5**. Esta classe foi criada para representar tanto pessoas físicas quanto jurídicas. No caso de um objeto que represente uma pessoa física, os atributos **cnpj** e **nomeFantasia** serão indefinidos e não farão sentido. Caso outra aplicação necessite apenas trabalhar com pessoas físicas, por exemplo, o reuso fica prejudicado. Classes com esse tipo de coesão, além de carregarem atributos e métodos que não fazem sentido em determinados casos, geralmente exigem um código costurado para verificar com que tipo de objeto estamos lidando;

Figura 5. Exemplo de classe com coesão de instância mista.

- **Coesão de domínio misto:** ocorre quando classes de domínios diferentes (domínio de arquitetura, domínio do negócio, dentre outros), que podem ser definidas de forma independente, mantêm alguma dependência.
  Este tipo de coesão pode ser observado na **Figura 6**, na qual a classe **ContaPoupanca** (domínio do negócio), de um sistema bancário, possui um método para imprimir, na impressora do caixa eletrônico, o seu saldo, através do uso da classe **Impressora** (domínio da arquitetura). Caso o banco desejasse reutilizar essa classe em uma aplicação de dispositivo móvel, por exemplo, na qual os clientes pudessem consultar saldo, o método **imprimirSaldoNaImpressora()** não faria sentido;

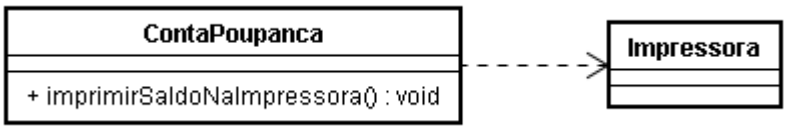


Figura 6. Exemplo de classe com coesão de domínio misto.

- **Coesão de papel misto:** ocorre quando classes do mesmo domínio, que podem ser definidas de forma independente, mantêm alguma dependência. Este tipo de coesão foi o mesmo existente no exemplo da **Figura 2**, no qual a classe **Cliente** (domínio do negócio) apresentava uma dependência desnecessária em relação à classe **Filme** (também do domínio do negócio). Os efeitos desse tipo de coesão já foram citados no exemplo dado anteriormente.

Coesão de métodos:

- **Coesão alternada:** ocorre quando um método agrega diversas etapas do comportamento da classe, mas só realiza uma delas mediante especificação por parâmetro. Um exemplo deste tipo de coesão pode ser observado no método **calcular()** da classe **Triangulo** (**Figura 7**), que realiza o cálculo da área (quando o parâmetro **area** é **true**) ou o perímetro do triângulo (quando o parâmetro **area** é **false**). Este tipo de coesão resulta em métodos extensos, com muitas linhas de código, e de difícil manutenção. Este tipo de coesão pode ser percebido pela existência do conectivo "OU" no nome do método;
- **Coesão múltipla:** é similar à coesão alternada. A diferença é que na coesão múltipla, um mesmo método realiza não só uma, mas diversas etapas do comportamento da classe. O método **moverERedimensionarERotacionar()** da classe **Triangulo** (**Figura 7**) é um exemplo de método com esse tipo de coesão. Este método move um triângulo, o redimensiona e o rotaciona ao mesmo tempo e, para isso, recebe como parâmetros as novas posições **x** e **y**, as proporções para redimensionamento na horizontal (**propH**) e vertical (**propV**), e o ângulo para rotação (**ang**). As mesmas observações feitas sobre os métodos com coesão alternada são extensíveis a esse tipo de coesão. Este tipo de coesão pode ser percebido pela existência do conectivo "E" no nome do método;
- **Coesão funcional:** ocorre quando um método é dedicado a uma única etapa de comportamento. Essa é a coesão ideal para métodos. Um exemplo desse tipo de coesão pode ser observado no método **calcularArea()** da classe **Triangulo**, que realiza apenas o cálculo da área do triângulo e nada mais.

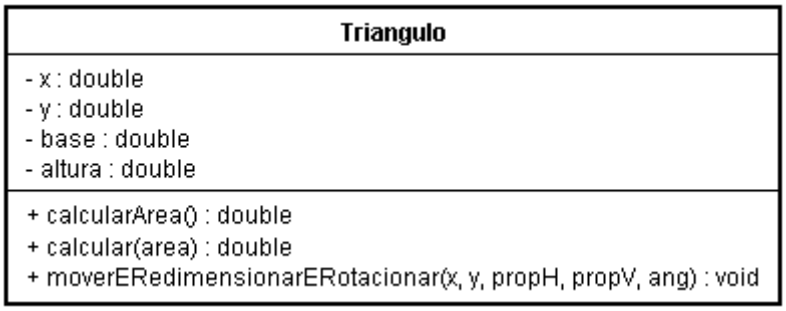


Figura 7. Exemplo de classe com métodos com coesão alternada, múltipla e funcional.

Independente da forma ou métrica de classificação usada para definir a coesão de uma classe, é importante atentar para algumas questões que podem ajudar na verificação e validação da coesão de uma classe:

- Se esta classe fosse reutilizada em outro contexto, todos os seus atributos e sua interface pública seriam úteis?
    - Em caso negativo, é interessante questionar se não seria o caso de dividir a responsabilidade desta classe entre outras classes ou mesmo definir subclasses que assumam parte desta responsabilidade;

- Caso esta classe seja isolada do sistema para o qual ela está sendo implementada, seria possível entendê-la?
    - Em caso negativo, pode ser que as responsabilidades da classe não estejam bem definidas.

Embora seja válida a lição tirada de metodologias ágeis que diz que não é produtivo dispensar muito tempo procurando prever todos os cenários de uso e reuso de uma classe, a cautela na modelagem, sua verificação e validação, pelo menos em um nível superficial, podem ser de grande valia para a definição de um projeto coeso que contribuirá para a redução do trabalho futuro com manutenção e facilitará o reuso.

Acoplamento

O outro dos conceitos abordados nesse artigo é o acoplamento, que pode ser entendido como a interdependência existente entre diferentes classes. Quanto mais uma classe conhece ou depende de outras classes, maior é o grau de acoplamento entre elas.

Os conceitos de coesão e acoplamento são geralmente abordados juntos porque estão correlacionados. Um baixo grau de coesão geralmente acarreta em um forte acoplamento e vice-versa. Por outro lado, um alto grau de coesão geralmente resulta em um fraco acoplamento e vice-versa.

Isto pode ser entendido pelo fato de que uma classe com baixa coesão possui responsabilidades que deveriam ser de outras classes e, assim sendo, deverá ser acessada por essas outras classes para que essas possam cumprir suas tarefas, fortalecendo assim o acoplamento.

Em sistemas com forte acoplamento, mudanças em uma classe forçam mudanças em classes relacionadas, fica mais difícil entender as classes isoladamente e é mais difícil reusar as classes, já que elas dependem da presença umas das outras.

Como exemplo de um problema causado pela existência de um forte acoplamento entre classes, vamos considerar um conhecido conceito de sistemas orientados a objetos: o encapsulamento.

Encapsulamento (também conhecido como ocultamento de informações) consiste em manter uma interface pública para um objeto sem expor os seus detalhes internos de implementação, que permanecem escondidos dos outros objetos.

O encapsulamento evita que um programa torne-se tão interdependente (tão fortemente acoplado) que uma pequena mudança tenha grandes efeitos colaterais. Motivos para modificar a implementação de um objeto podem ser, por exemplo, aprendizado de novas características e funcionalidades da linguagem de programação, melhoria de desempenho, mudanças legais ou culturais, correção de erros ou mudança de plataforma de execução.

Para entender como um forte acoplamento dificulta modificações em um sistema, suponha que foi implementada uma classe em Java para um sistema de agenda de compromissos que representa um evento (**Listagem 1**). Esta classe possui atributos para armazenar o nome, a data, uma breve descrição e o status do evento. Os métodos dessa classe permitem que eventos sejam criados, cancelados e consultados.

Como você não conhecia bem a linguagem Java no momento da implementação desta classe, optou por representar a data do evento através de três atributos (linha 4): **dia**, **mes** e **ano**, todos do tipo **int**. Sem se preocupar com o conceito de encapsulamento, não foi proibido o acesso direto aos atributos da classe (atributos públicos).

Listagem 1. Código da classe **Evento** cujos atributos não estão encapsulados.

```
1  public class Evento {
2
```

```
3    public String nome;
4    public int dia, mes, ano;
5    public String descrição;
6    public boolean cancelado;
7
8    public void agendar(int dia, int mes, int ano) {
9      this.dia = dia;
10     this.mes = mes;
11     this.ano = ano;
12   }
13
14   public void cancelar() {
15     // código para cancelar um evento.
16   }
17
18   public void consultar() {
19     // código para consultar um evento.
20   }
21  }
```

Esta classe foi utilizada em diversas aplicações (ex.: sistema acadêmico, locadora, entre outras). Em cada uma delas, os valores dos atributos foram definidos diretamente, conforme **Listagem 2**.

Listagem 2. Código do sistema acadêmico e de locadora que utilizam a classe **Evento** definida na Listagem 1.

```
1  //... código do sistema acadêmico ...
2  Evento e1 = new Evento();
3  e1.nome = "Reunião com os professores";
4  e1.dia = 28;
5  e1.mes = 06;
6  e1.ano = 2010;
7  e1.descricao = "Reunião entre pais e professores para a discussão" + " dos resultados
    do semestre";
8  //...
9
10  //... código do sistema da locadora ...
11  Evento e2 = new Evento();
12  e2.nome = "Sessão de autógrafos com a atriz Y";
13  e2.dia = 12;
14  e2.mes = 03;
15  e2.ano = 2010;
16  e2.descricao = "A atriz Y estará na locadora para uma sessão de autógrafos";
17  //...
```

Após algum tempo, você descobriu que a linguagem Java possui uma classe que permite o armazenamento e manipulação de datas (**java.util.Date**) e deseja alterar a estrutura da classe **Evento** para que a data passe a ser armazenada através de um objeto da classe **Date**. Você decidiu então alterar a classe **Evento**, conforme **Listagem 3**.

Listagem 3. Código da classe **Evento** após a alteração dos atributos referentes à data do evento.

```
1  import java.util.Date;
2
3  public class Evento {
4    public String nome;
5    public Date data;
6    public String descricao;
7    public boolean cancelado;
8    //...
```

Esta alteração acarretará em alterações em todas as classes que usam a classe **Evento**, como é o caso dos exemplos anteriores apresentados para o sistema acadêmico e da locadora. Isto ocorre porque as linhas de código que alteravam os atributos **dia**, **mes** e **ano** (linhas 4, 5, 6, 13, 14 e 15 da **Listagem 2**) não funcionarão mais, já que estes atributos não existem na nova versão da classe.

Isto ocorre porque a classe **Evento** e as classes que a usam estão fortemente acopladas em relação à estrutura de dados. Este acoplamento seria reduzido se a classe **Evento** inicialmente tivesse encapsulado seus atributos. Neste cenário, o código da classe **Evento** teria sido escrito conforme a **Listagem 4**.

Listagem 4. Código da classe **Evento** com os atributos encapsulados.

```
1  public class Evento {
2
3    private String nome;
4    private int dia, mes, ano;
5    private String descricao;
6    private boolean cancelado;
7
8    public void agendar (int dia, int mes, int ano) {
9      this.dia = dia;
10       this.mes = mes;
11       this.ano = ano;
12     }
13
14     //... outros métodos
15   }
```

As classes que usam este código teriam sido escritas conforme o código da **Listagem 5**.

Listagem 5. Código do sistema acadêmico e de locadora que utilizam a versão encapsulada da classe **Evento**.

```
1  //... código do sistema acadêmico ...
2  Evento e1 = new Evento();
3  e1.setNome("Reunião com os professores");
4  e1.agendar(28, 06, 2010);
5  e1.setDescricao("Reunião entre pais e professores");
6  //...
7
8  //... código do sistema da locadora ...
9  Evento e2 = new Evento();
10  e2.setNome("Sessão de autógrafos com a atriz X");
11  e2.agendar(12, 03, 2010);
12  e2.setDescricao("A atriz X estará na locadora para uma sessão de autógrafos");
13  //...
```

Como as classes que utilizam a classe **Evento**, na **Listagem 5**, não conhecem sua estrutura interna, há um enfraquecimento do acoplamento. Isso permitiria que a alteração dos atributos referentes à data fosse feita sem que as classes que a usam tivessem que ser alteradas. Veja como a alteração poderia ser feita no código da **Listagem 6**.

Listagem 6. Código da versão encapsulada da classe **Evento** após a alteração dos atributos referentes à data do evento.

```
1  import java.util.Date;
2
3  public class Evento {
4    private String nome;
5    private Date data;
6    private String descricao;
7    private boolean cancelado;
8
9    public void agendar(int dia, int mes, int ano) {
10       Calendar calendario = new GregorianCalendar(ano, mes, dia);
11       data = calendário.getTime();
12     }
13
14     //...
15   }
```

Assim sendo, nenhuma outra classe teria que ser alterada. Este é um exemplo claro de benefício de enfraquecer o acoplamento através da aplicação do conceito de encapsulamento.

No exemplo apresentado, seria possível fazer algumas alterações na classe **Evento** não encapsulada (**Listagem 1**) para que as alterações nos atributos referentes a data não tivessem ocasionado o efeito "em cascata" de alteração nas aplicações do sistema acadêmico e da locadora.

No entanto, essas alterações acarretariam em um código remendado com diversas informações repetidas (data representada tanto pelos atributos **dia**, **mes** e **ano**, quanto pelo atributo **data**) e mecanismos de controle de alteração (como algumas aplicações antigas usariam os atributos **dia**, **mes** e **ano** e outras, mais novas, o atributo **data**, deveriam existir mecanismos para mantê-los com o mesmo valor).

Este mesmo efeito de um forte acoplamento poderia ser observado em relacionamentos de herança, que estabelecem um alto grau de acoplamento entre superclasse e subclasse, em classes internas, entre outros.

Diversos padrões e estratégias de projeto estão relacionados à redução do acoplamento, como é o caso do padrão *Iterator*, *Model-View-Controller* (MVC), programação para interfaces, injeção de dependência, entre outros.

Embora eliminar o acoplamento seja impossível, visto que, quando uma classe conhece outra classe, já existe algum acoplamento entre elas, devemos procurar reduzi-lo através da eliminação de relações desnecessárias, redução do número de relações necessárias e do enfraquecimento da dependência das relações necessárias.

O encapsulamento é um exemplo de enfraquecimento da dependência das relações necessárias. Como mais um exemplo de redução do acoplamento, vamos analisar o código de um sistema acadêmico no qual existem relações desnecessárias.

Neste sistema, foram utilizadas duas bibliotecas matemáticas (biblioteca **estatistica** e **matematica**) para a realização do cálculo das médias dos alunos e da produção de outros dados estatísticos. Uma das classes do sistema utiliza essas duas bibliotecas, conforme código da **Listagem 7**.

Listagem 7. Código de um sistema acadêmico no qual são utilizadas duas bibliotecas matemáticas.

```
1  import estatistica.*;
2  import matematica.*;
3  //... início da classe
4
5    public void calcularMedia() {
6      // calcula a média ponderada do aluno.
7      MediaPonderada media = new MediaPonderada();
8      //... restante do método.
9    }
10
11  //... fim da classe
```

Na linha 7 do código da **Listagem 7**, a classe **MediaPonderada**, disponível apenas na biblioteca **estatistica**, é utilizada para o cálculo da média ponderada do aluno. Perceba que, neste código, embora desnecessariamente, todas as classes dos pacotes **estatistica** e **matematica** são importadas, o que cria um acoplamento relacionado à referência entre a classe em questão e as classes desses pacotes.

Agora suponha que na próxima versão da biblioteca **matematica** também seja disponibilizada uma classe chamada **MediaPonderada** e que você atualize essa biblioteca no seu sistema. Da próxima vez que sua classe for compilada, um erro anteriormente inexistente causado pela existência de duas classes com o mesmo nome ocorrerá. Caso apenas as classes necessárias tivessem sido importadas na nossa classe, o acoplamento relacionado à referência teria sido enfraquecido e o tipo de erro citado seria evitado.

Assim como para o conceito de coesão, foram definidas diferentes classificações de acoplamento. A primeira destas classificações definiu cinco tipos diferentes de acoplamento: acoplamento de conteúdo, acoplamento comum, acoplamento de controle, acoplamento de imagem e acoplamento de dados.

Estes níveis variavam do melhor tipo de acoplamento (dados), que significava que dois módulos comunicavam-se através da passagem de dados por parâmetros simples, ao pior tipo de acoplamento (conteúdo), que ocorria quando um módulo fazia referência ao interior de outro módulo. Isso era possível em linguagens de montagem e algumas outras linguagens, como COBOL, nas quais existiam instruções para alterar um comando de outro módulo ou mesmo desviar a sequência de instruções para um ponto qualquer de outro módulo. Esse era considerado o pior tipo de acoplamento porque os módulos acoplados dessa forma conheciam muito sobre a estrutura interna do módulo respectivo.

Essa classificação foi adotada por muitos autores para sistemas orientados a objetos, fazendo-se algumas adaptações. Outros autores, no entanto, optaram por adotar classificações diferentes, propostas especificamente para sistemas orientados a objetos.

Além dessas classificações de coesão e acoplamento em tipos ou níveis, outras formas de medir estas características em sistemas orientados a objetos são as diversas métricas propostas com este fim: LCOM (*Lack of Cohesion in Methods*) e suas variantes, CBO (*Coupling Between Objects*), DAC (*Data Abstraction Coupling*), LORM (*Logical Relatedness of Methods*), entre outras.

Estas métricas tratam de diferentes aspectos relacionados à coesão e ao acoplamento que, quando analisados em conjunto, podem contribuir para a identificação de problemas na modelagem e implementação de um sistema orientado a objetos.

Existem algumas ferramentas que auxiliam na obtenção e análise de métricas de acoplamento para códigos escritos em Java, tais como o JDepend e JHawk.

O JDepend é uma ferramenta de código aberto que permite a análise de diretórios com classes e calcula diversas métricas, algumas das quais estão descritas na **Tabela 1**.

Tabela 1. Algumas métricas calculadas pela ferramenta JDepend.

Métrica	Descrição
Acoplamentos aferentes **(Ca)**	Número de pacotes que dependem das classes do pacote analisado. É um indicador que pode, em conjunto com outros, ser utilizado na análise do acoplamento.
Acoplamentos eferentes **(Ce)**	Número de pacotes dos quais as classes do pacote analisado dependem. Assim como o acoplamento aferente, pode ser utilizado na análise do acoplamento.
Nível de Abstração **(A)**	A proporção entre o número de classes abstratas (e interfaces) do pacote analisado e o número total de classes dentro do pacote. Por se tratar de uma proporção, esta métrica assume valores entre 0 e 1. Um valor igual a 0 indica que um pacote é completamente concreto e um valor igual a 1 indica que um pacote é completamente abstrato. Pacotes com mais classes abstratas podem indicar que há um menor acoplamento em relação à implementação.
Instabilidade **(I)**	A proporção entre o acoplamento eferente (Ce) e o total de acoplamento (Ca + Ce) do pacote analisado: Ce/(Ce+Ca). Assim como para o nível de abstração, esta métrica assume valores entre 0 e 1. Um valor igual a 1 indica que nenhum outro pacote depende das classes do pacote analisado, mas o pacote analisado depende de classes de outros pacotes (instabilidade). Um valor igual a 0 indica que o pacote analisado não depende de classes de outros pacotes (estabilidade).

O JHawk, por sua vez, é uma ferramenta paga que calcula métricas como número de métodos externos a uma classe chamados por ela, número de classes referenciadas por um código e o número de variáveis referenciadas por métodos.

Essas ferramentas possuem plug-ins para IDEs e interfaces gráficas que facilitam o processo de análise e cálculo de métricas.

Conclusão

A coesão e o acoplamento são conceitos que devem ser observados em projetos de sistemas orientados a objetos, já que a produção de software com baixa coesão e forte acoplamento dificulta seu entendimento, manutenção e reuso.

A literatura sobre coesão e acoplamento é bastante extensa e não existe muito consenso em relação às classificações, nem em relação às melhores métricas para a avaliação destas características em projetos de software. No entanto, o estudo destes conceitos e seus desdobramentos pode contribuir para a produção de sistemas de software com maior qualidade.

Daqui para frente, nos seus projetos de sistemas orientados a objetos, reserve um pouco de atenção para analisar a coesão e o acoplamento das suas classes e verifique se alguma modificação pode ser feita para melhorar a qualidade do seu projeto.

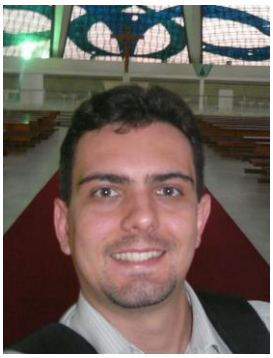
Referências

http://www.eli.sdsu.edu/courses/spring98/cs635/notes/index.html
Site com vários materiais da disciplina de "Programação e Projeto Orientados a Objetos" da Universidade do Estado de San Diego.

http://clarkware.com/software/JDepend.html
Site da ferramenta JDepend.

http://www.virtualmachinery.com/jhawkprod.htm
Site da ferramenta JHawk.

Leandro Luque (*leandro.luque@gmail.com*) É gestor acadêmico e professor da Universidade de Mogi das Cruzes (UMC) e professor da FATEC-Mogi. Tem formação em Ciência da Computação e mestrado em Computação Aplicada pelo Instituto Nacional de Pesquisas Espaciais (INPE). Trabalha com Java há 10 anos, tendo atuado no desenvolvimento de aplicações de grande porte tanto no segmento empresarial quanto governamental.



Rodrigo Rocha Silva (*rrochas@gmail.com*) Formado em Ciência da Computação pela Universidade de Mogi das Cruzes (UMC), mestre em Computação Aplicada pelo INPE e Doutorando pelo ITA. Trabalha com Java há mais de 6 anos, atuando como desenvolvedor, analista e arquiteto em empresas de pequeno e grande porte, nos segmentos

de gestão empresarial, saúde e governo. Atua também como professor na UMC e na Veris Faculdades. Possui certificação SCJP 1.4.