Modularidade em Sistemas Orientados por Objetos

Kécia Aline Marques Ferreira
Mariza Andrade da Silva Bigonha
Roberto Silva Bigonha

Sumário

Prefácio

Este livro foi escrito para ser utilizado principalmente por alunos de cursos de graduação na área de computação, como ciência da computação, matemática computacional, engenharia de computação, sistemas de informação e outros. Também pode ser utilizado por alunos de pós-graduação, eventualmente complementado com outras referências que abordem alguns assuntos com maior profundidade ou que apresentem alguns tópicos não cobertos aqui. Além disso, pode ser útil para profissionais da área de computação em geral, tanto para aqueles que desejem fazer uma revisão quanto para aqueles que queiram ter um primeiro contato com a área.

Agradecimentos

noindent Este livro foi escrito para ser utilizado principalmente por alunos de cursos de graduação na área de computação, como ciência da computação, matemática computacional, engenharia de computação, sistemas de informação e outros. Também pode ser utilizado por alunos de pós-graduação, eventualmente complementado com outras referências que abordem alguns assuntos com maior profundidade ou que apresentem alguns tópicos não cobertos aqui. Além disso, pode ser útil para profissionais da área de computação em geral, tanto para aqueles que desejem fazer uma revisão quanto para aqueles que queiram ter um primeiro contato com a área.

Capítulo 1

Introdução

Desenvolver software de alta qualidade e baixo custo não é tarefa simples. Nas últimas décadas, muito se estudou e investigou sobre o processo de desenvolvimento de software [14, 36, 37, 41, 55]. Os sistemas de informação tornaram-se mais complexos e mais presentes na vida das pessoas. Eles estão presentes na medicina, na educação, nas telecomunicações, nos processos e serviços prestados pelo governo à população, nas relações comerciais, enfim, em uma série de situações no dia a dia das pessoas. Algumas destas situações envolvem risco para a vida humana, como sistemas que controlam aviões, lançamentos de foguetes e linhas ferroviárias; outras envolvem manipulação de grande quantidade de valor monetário, como a automação bancária, etc. Pressman [44] ressalta que à medida que a importância dos softwares na sociedade cresce, aumenta também a preocupação da comunidade produtora de software em obter recursos para desenvolver software de forma mais fácil, mais rápida e com custos menores. A difusão de modelos de capacitação em processos de software hoje confirma isso. Segundo Paula [42], este tipo de mo- delo visa avaliar a maturidade de uma organização para produzir software de boa qualidade, com custos razoáveis e cumprir prazos. Dentre eles, destaca-se o CMM (Capability Maturity Model) [14] , modelo que o Departamento de Defesa Americano utiliza para avaliar as organizações que lhe fornecem softwares e que tem sido mundialmente reconhecido [42].

Embora exista hoje à disposição dos desenvolvedores de software uma gama de conceitos, tecnologias, metodologias e linguagens de programação poderosas, desenvolver software é ainda muito difícil e caro, um campo fértil a ser explorado e melhorado. Um bom exemplo disso é a Orientação por Objetos, uma das grandes contribuições dos últimos tempos para a produção de software de alta qualidade e baixo custo [36]. Este paradigma de projeto e programação representa a possibilidade de transposição do contexto caótico muitas vezes vivenciado por grande parte da comunidade desenvolvedora de software para um ambiente mais harmonioso no qual softwares possam ser produzidos com alta qualidade e com custo consideravelmente amenizado. Isso porque uma das características principais da orientação por objetos é a modularidade, a independência entre os

componentes de software, o que viabiliza a obtenção de reusabilidade, de sistemas mais flexíveis, mais fáceis de testar e manter. Apesar de todos os recursos da orientação por objetos, ainda é difícil desenvolver software que seja reutilizável e flexível, como apontam Gamma et al [24].

A qualidade de software é um dos fatores de maior importância na produção de software. Meyer [36] sintetiza este fato definindo a engenharia de software como a produção de software de qualidade. Paula [42] define qualidade como a conformidade do software com os seus requisitos e ressalta que ela depende do processo utilizado para produzir o software. De acordo com Meyer [36], a qualidade de um software pode ser observada sob dois pontos de vista: a forma como ele foi construído - fatores internos - e a forma como ele se apresenta ao usuário - fatores externos. Por meio dos fatores externos é possível avaliar se o software está adequado aos requisitos do sistema. Os fatores internos são relacionados à estrutura do software. Estes são determinantes para se alcançar os fatores externos.

Outro fator crítico na produção de software é o custo. Como Paula [42] destaca, o custo de um software é um dos fatores que define a sua viabilidade. Segundo Pressman [44], a manutenção é a fase que mais demanda esforço no ciclo de vida de um sistema. Confirmado isso, Meyer [36] aponta que mais de 70% do custo total de um sistema é referente a custo de manutenção. Segundo Pfleeger [43], este percentual ultrapassa 80%. De uma forma geral, sistemas têm vida longa e não é possível dizer que um sistema está livre de erros, ou que não sofrerá alterações ou que não necessitará de novas funcionalidades. O estudo de Lientz e Swanson [1] (1980 apud Meyer [36]) revelou que 41.8% do custo de manutenção é decorrente de alterações em requisitos de usuário. Meyer [36] conclui, então, que a manutenção de um software é penalizada pela dificuldade de realizar alterações no mesmo. Esses dados mostram que reduzir os custos da manutenção de software é imperativo e que tal redução pode ser obtida principalmente quando for possível realizar alterações no software de forma mais fácil. Myers [38] aponta a modularidade como a resposta para tal problema, visto que a independência entre os módulos de um sistema permite que modificações em um módulo afetem um número pequeno de módulos. A modularidade contribui para a redução da complexidade do software e para a sua manutenibilidade [44].

Segundo Myers [38], a modularidade de um sistema é determinada basicamente por duas práticas fundamentais: a minimização do relacionamento entre módulos e a maximização do relacionamento entre elementos de um mesmo módulo. A medida do grau de relacionamento entre módulos é chamada acoplamento; a medida do grau de relacionamento entre os elementos internos de um módulo é chamada coesão. Uma situação ideal seria aquela na qual uma mo-

[1] LIENTZ, Bennet P. e SWANSON, E. Burton. em Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley, 1980.

dificação realizada em determinado módulo afetasse o menor número possível de outros módulos. Um sistema com esta característica possui alto grau de *estabilidade*, que é a capacidade de um software se manter inalterado diante de uma alteração em seu ambiente [26]. Para ilustrar suas idéias, Myers [38] modela um sistema como um conjunto de lâmpadas conectadas entre si, no qual o fato de uma lâmpada estar acesa afeta o estado das demais lâmpadas conectadas a ela. Cada lâmpada representa um módulo. A mudança de estado de uma lâmpada representa atividade de manutenção do módulo correspondente. Neste modelo, descrito em detalhes no Capítulo 4, fica evidente o impacto da conectividade no custo de manutenção.

A importância da conectividade entre módulos na determinação da manutenibilidade de um software encontra apoio na literatura sobre projeto e construção de software. Em um artigo clássico da área, Parnas [41] defende que um dos benefícios da modularização é a flexibilidade do software, o que torna possível realizar alterações drásticas em um módulo sem afetar os demais. Pfleeger [43] ressalta a importância da qualidade do projeto do software para a manutenibilidade do software, apontando que o software estruturado com componentes coesos e independentes é mais inteligível e sua manutenção é facilitada. Já Pressman [43] alerta que sistemas nos quais existe forte dependência entre seus módulos podem se tornar pesadelos no caso de depuração. Os sistemas mais fáceis de alterar são considerados por Xavier [61] como aqueles constituídos por módulos fáceis de entender e o mais independente possível uns dos outros. Uma das mais importantes referências sobre construção de software orientado por objetos [36] define duas regras, dentre outras, para a obtenção de software modular: todo módulo deve se comunicar com um menor número de módulos possível; se dois módulos se comunicam, eles devem trocar o mínimo de informação possível.

Conforme Pfleeger [43], outros fatores têm impacto na manutenibilidade de um software, dentre eles destacam-se: qualidade da documentação, complexidade do tipo de aplicação e a rotatividade de pessoal nas equipes. De acordo com isso, não se pode tomar a avaliação da conectividade de um software como a solução de todos os problemas de manutenção de software, pois as demais questões não devem ser desprezadas. Todavia, a redução da conectividade deve receber atenção especial na construção de software, visto que tem impacto determinante na qualidade e no custo do sistema. Ainda que os demais aspectos favoreçam a manutenibilidade de um sistema, se o aspecto conectividade não for adequado, a qualidade e o esforço de manutenção do software são seriamente comprometidos.

Para saber se a conectividade em um determinado software é satisfatória, é preciso conhecê-la e, para isso, faz-se necessário uma forma de medi-la. A partir do momento em que se toma conhecimento que a conectividade de um software não é satisfatória, deve-se, então, analisar os fatores adjacentes que contribuem para tal situação. Neste caso também é imprescindível uma forma de medir tais fatores. Assim, surgem as seguintes necessidades principais: identificar os fatores que têm impacto na conectividade de um software e identificar as métricas que

permitam que os fatores determinantes da conectividade sejam avaliados.

A área de métricas de software é um assunto que tem sido estudado há cerca de três décadas [44]. A literatura sobre métricas de software é ampla, por exemplo, [2, 5, 12, 13, 17, 22, 62]. Apresentamos, aqui, métricas que possam auxiliar no processo de avaliação da conectividade em sistemas orientados por objetos. Muitos pesquisadores e empresas têm oferecido contribuições valiosas para a medição de softwares orientados por objetos [1, 2, 13, 17, 32] . Destacam-se, dentre esses trabalhos, o conjunto de métricas proposto por Chidamber e Kemerer, conhecido na literatura como métricas CK [13], e o conjunto de métricas proposto por Abreu e Carapuça, conhecido como MOOD [2]. Encontram-se na literatura propostas de ferramentas de coleta dessas métricas para várias linguagens, como exemplo, a proposta do INESC/Portugal denominada MOODKIT [3], que permite a coleta das métricas MOOD [2] em códigos como C++, Java, Smalltalk ou Eiffel. Contudo, conforme Fenton e Neil [17], há uma carência de formas efetivas de aplicação de métricas de software como suporte à decisão na produção de softwares. Faz-se necessário a existência de instrumentos de apoio à decisão gerencial no ciclo de vida do sistema, que auxiliem gerentes e técnicos a realizarem acompanhamento e atuação na produção de software, no sentido de decidirem em que ponto agir para corrigir, evitar ou reverter situações indesejadas.

A maior parte do conteúdo deste livro é fruto do trabalho realizado na dissertação de mestrado de Ferreira [20]. O livro está estruturado da seguinte forma:

Capítulo 2 apresenta e analisa os principais conceitos relacionados à qualidade de software. Identificamos, nesse capítulo, os principais os fatores externos de qualidade de software, que são as características que podem ser avaliadas pelos usuários de um software. Destacamos o aspecto modularidade como o principal fator interno de software para atingir alto grau de qualidade de software, analisando a relação deste fator com dois conceitos: coesão interna de módulos e acoplamento entre módulos, para os quais apresentamos e exemplificamos as respectivas classificações com base na descrição inicial de Glenford Myers [38]. Apresentamos os critérios, regras e princípios propostos por Bertrand Meyer [36] para a construção de software modular. Discutimos a questão da manutenibilidade, identificando seus principais fatores determinantes, com destaque para a qualidade do código do software. Analisamos os benefícios da orientação por objetos para a obtenção de software de qualidade.

Capítulo 3 , dada a importância dos conceitos de acoplamento entre módulos e de coesão interna de módulos para a modularidade de software, apresenta uma adaptação de tais aspectos à luz da Orientação por Objetos. Definimos uma *classe* como um *módulos* na orientação por objetos. Apresentamos e exemplificamos uma classificação para: coesão interna de classes, coesão in-

terna de métodos, coesão interna de interface e acoplamento, para software orientado por objetos.

Capítulo 4 aborda a conectividade de software como fator preponderante na sua estabilidade e manutenibilidade. Identificamos, neste capítulo, os fatores de impacto na conectividade de software orientado por objetos.

Capítulo 5 descreve a estratégia de estruturação de software em camadas. Analisamos os benefícios desta abordagem para a obtenção de software modular e com baixo grau de conectividade.

Capítulo 6 , diante da importância da medição de software para alcançar qualidade, trata sobre métricas de software, abordando sua importância e as principais métricas propostas na literatura, em particular aquelas destinadas à medição de software orientado por objetos. Neste capítulo, analisamos tais métricas, com o objetivo de identificar aquelas que auxiliam na avaliação da conectividade de softwares OO. São relatadas também ferramentas disponíveis para a coleta de métricas em softwares orientados por objetos.

Capítulo 7 apresenta as conclusões sobre o assunto tratado neste livro.

Capítulo 2

Qualidade de Software

Antes de se falar em avaliação de qualidade de software, há de se definir o que é qualidade. O dicionário Aurélio da língua portuguesa [18] define qualidade como:

1. Propriedade, atributo ou condição das coisas ou das pessoas que as distingue das outras e lhes determina a natureza.

2. Superioridade, excelência de alguém ou algo.

Qualidade é palavra de ordem em quase todos os segmentos da produção humana nos tempos atuais, talvez pelo caráter competitivo que a maior parte das relações possui. A qualidade do trabalho pode significar o sucesso de um profissional, assim como a qualidade dos produtos e serviços pode significar o sucesso de uma organização. Não é diferente no caso da produção de software. Em um contexto de mercado competitivo e forte demanda por sistemas cada vez mais complexos, a qualidade de um software é o fator determinante para o seu sucesso. Assim, é importante a existência de formas de se avaliar a qualidade de software.

Pressman [44] define qualidade de software como "conformidade a requisitos funcionais e de desempenho explicitamente declarados, padrões de desenvolvimento explicitamente documentados e características implícitas, que são esperadas em todo software desenvolvido profissionalmente".

A abordagem de Bertand Meyer [36] sobre qualidade de software considera dois aspectos: as características internas e externas de um software. Quando se avalia a qualidade de um produto qualquer, pode-se considerar basicamente dois pontos de vista: o de quem utiliza o produto e o de quem o produz. Quem utiliza o produto avalia a sua qualidade baseando-se em aspectos externos, tais como boas condições de uso e atendimento às necessidades do usuário. Os aspectos externos constituem o objetivo fim de quem se propõe a construir um produto, pois são determinantes na aceitação ou não por parte de quem irá utilizá-lo. Para atingir o objetivo de construir um produto que seja aceito, investe-se na qualidade da forma utilizada para se construir tal produto, por exemplo: utilização de matéria-prima

de qualidade e boas técnicas de produção. Pode-se dizer, então, que os aspectos internos, aqueles relativos à produção, ajudam a atingir os aspectos externos.

Software é um produto e, como todo produto, tem também a sua qualidade avaliada sob dois aspectos: fatores internos e fatores externos. Este assunto será tratado em detalhes na Seção 2.1. Os requisitos do software são referenciais para se avaliar um software do ponto de vista externo; a estrutura do software corresponde aos seus aspectos internos de qualidade. A estrutura de um software é determinada pelo seu projeto, também denominado desenho, tradução do termo inglês *design*. O projeto de software define uma estrutura que possa ser implementada e que atenda aos requisitos especificados para determinado software [42]. A forma como o software é estruturado é a chave para a obtenção de software de alta qualidade.

Projetar software é tarefa árdua e tem demandado atenção de pesquisadores e profissionais desde os primórdios da produção de software. Glenford Myers, em 1975 [38], escreveu que talvez o maior problema enfrentado naquela época fosse a extrema dificuldade e custo de criar e manter sistemas de programação de grande porte. Hoje, trinta anos depois, a situação não é diferente. É claro que muito se evoluiu nesta área: ferramentas, métodos, técnicas, linguagens, etc., estão disponíveis para quem se propõe a criar um software. Mas a produção de software ainda conta com o mesmo problema: criar e manter softwares complexos a um custo que seja ao menos razoável.

Do ponto de vista do projeto de softwares, um dos fatores mais importantes é a modularidade. A modularidade é um mecanismo para melhorar a flexibilidade e a compreensão de um sistema, além de possibilitar a diminuição do tempo de seu desenvolvimento. Um programa modular possui partes que podem ser entendidas, implementadas e alteradas de forma independente, o que torna o software flexível, mais fácil de manter, propício à reutilização, mais fácil de testar e, conseqüentemente, impacta na redução de custo de produção de software.

O caminho para obter software com baixo grau de conectividade, construído de forma que os seus módulos sejam o mais independentes possível, é investir em aspectos de sua construção que potencializam este objetivo. Este capítulo apresenta uma revisão da literatura sobre qualidade de software, identificando tais aspectos. Para isto, ele está estruturado da seguinte forma:

Seção 2.1 descreve os principais fatores externos de qualidade de software, identificando a influência que a conectividade tem sobre eles.

Seção 2.2 descreve o papel da modularidade na construção de software, apresenta as definições dos conceitos *coesão* e *acoplamento*, apresenta os critérios, regras e princípios para construção de software modular abordados por Meyer [36], destacando a importância desses aspectos para a conectividade.

Seção 2.3 discorre sobre a manutenibilidade de sistemas e os fatores que a determinam, enfatizando o papel da conectividade sobre este aspecto.

Seção 2.4 aborda o código como artefato principal na produção de software, destacando a importância de se investir na sua qualidade.

Seção 2.5 aponta o paradigma OO como detentor de recursos úteis na obtenção de software de qualidade, justificando o porquê da escolha deste paradigma como objeto de estudo desta dissertação.

Seção 2.6 apresenta as conclusões deste capítulo.

2.1 Fatores Externos de Qualidade de Software

Fatores externos de qualidade de software são aquelas características que podem ser avaliadas por quem utiliza o software. Bertrand Meyer [36], destaca como fatores externos de qualidade de software:

- ***Correção***: é a capacidade do software realizar as tarefas como foram definidas em sua especificação de requisitos. Certamente, este é o primeiro aspecto a ser observado em um software.

- ***Robustez***: um software é robusto se realiza as suas tarefas de forma correta mesmo quando submetido a condições anormais.

- ***Extensibilidade***: é a característica de um software poder ser facilmente adaptado a inclusões e alterações de requisitos. A importância desta característica dá-se pelo fato de que, em geral, a vida de um software é longa, e, ao longo de sua existência, alterações ou novos requisitos são inevitáveis.

- ***Reusabilidade***: é a característica de um software poder ser reutilizado ao todo ou em parte por outros softwares. Na produção de software a possibilidade de se reutilizar elementos já construídos facilita o projeto e o seu desenvolvimento, torna o produto mais confiável e impacta principalmente no seu custo. Do ponto de vista de projeto e desenvolvimento de software, a reutilização é melhorada em função da independência dos constituintes do software. Quanto menor o nível de dependência de um módulo ou de um conjunto de módulos, maior a facilidade de reutilizá-lo, por exemplo, na construção de um outro software.

- ***Compatibilidade***: é a facilidade de se combinar o software com outros softwares. Essa característica é importante porque raramente um software é construído sem interação com outros softwares.

- ***Eficiência***: refere-se ao bom uso que o software faz dos recursos de hardware, tais como memória e processadores.

- ***Portabilidade***: é a facilidade de se utilizar o software em diferentes ambientes de hardware e software.

- ***Verificabilidade***: é a facilidade de se preparar rotinas para se verificar a conformidade do software com os seus requisitos.

- ***Integridade***: é uma característica relacionada à segurança de dados, programas e documentos. Integridade é a habilidade de proteger tais componentes contra acessos não autorizados.

- ***Usabilidade***: é a facilidade de uso de um software. Engenharia de Usabilidade [11, 39, 54] é uma área ampla do conhecimento que visa a estudar a interação humano-computador. O projeto de interfaces de usuário é uma questão fundamental na produção de software e deve ser realizado de forma a garantir questões como produtividade, facilidade e uso e aprendizado do software pelos usuários.

Um dos objetivos principais na produção de software é satisfazer esses fatores, pois eles determinam o sucesso do produto. A forma como um software é construído, ou seja, os fatores internos, determinam os seus fatores externos, dentre os quais destacam-se a extensibilidade e a reusabilidade, que são fatores fortemente influenciados pela independência dos módulos do software. Esta característica, denominada *modularidade*, é detalhada na próxima seção.

2.2 Modularidade

Modularidade é a característica de um software construído a partir de unidades básicas, denominadas módulos. É o aspecto chave para a construção de software de arquitetura flexível e, conseqüentemente, para a obtenção de softwares mais fáceis de se manter. Um bom projeto de software deve definir módulos o mais independente possível, que possam ser entendidos, implementados e alterados sem a necessidade de conhecer o conteúdo dos demais módulos e com o menor impacto possível sobre eles.

Muitas definições são dadas para módulo na literatura. Myers [38] define um módulo como um conjunto de instruções de programas com as seguintes características: fisicamente juntas na listagem do programa, limitadas por fronteiras identificáveis, coletivamente referenciadas por um nome, podem ser referenciadas pelo nome do módulo por qualquer outra parte do programa.

Para Staa [55], um módulo é um conjunto de um ou mais arquivos que podem ser compilados com sucesso. Para Bertand Meyer [37], um módulo é a menor unidade na qual o sistema pode ser decomposto. De acordo com Bigonha [8], um módulo é um componente de software que possa ser compilado separadamente. Todas essas definições convergem para a idéia de que um módulo é a unidade

básica de estruturação de um sistema, são peças que se articulam para dar forma ao software. A forma como esta articulação é realizada determina a flexibilidade do sistema.

Segundo Myers [38], um projeto modular ótimo é aquele no qual o relacionamento entre elementos que não estejam no mesmo módulo são minimizados. Ghezzi e Jazayeri [25] consideram um bom módulo aquele que representa uma abstração utilizável, que interage com outros módulos de uma maneira bem definida e regular.

Myers [38] define dois caminhos principais para se obter modularidade:

1. Minimizar o relacionamento entre módulos, o que ele denominou acoplamento (*coupling*).

2. Maximizar o relacionamento entre elementos no mesmo módulo, o que ele denominou coesão (*strength*).

A seguir, são descritos estes dois conceitos importantes, tal como foram definidos por Myers [38], para a obtenção de software modular: coesão interna de módulos e acoplamento entre módulos.

2.2.1 Acoplamento

Acoplamento é uma medida do nível de relacionamento existente entre dois módulos. Dois módulos estão acoplados se existe algum tipo de comunicação entre eles. A forma de comunicação entre dois módulos é importante porque determina quão dependentes os módulos são entre si. Quanto mais forte for o elo entre dois módulos, mais dependentes eles são. Myers [38] define uma escala para nível de acoplamento de módulos. Esta escala é apresentada a seguir, do pior para o melhor nível de acoplamento entre módulos.

1. **Acoplamento por Conteúdo**

   É o tipo mais forte de acoplamento entre dois módulos. Este tipo de acoplamento existe quando um módulo faz referência direta ao conteúdo do outro módulo. O acesso é feito a elementos não exportados do módulo, ou seja, o módulo detentor do conteúdo acessado não autoriza tal acesso, porém, de forma sub-reptícia, outro módulo obtém o acesso. O exemplo típico do acoplamento por conteúdo se dá devido ao uso de desvios, em linguagens mais antigas, a partir de um módulo para outro módulo com o comando GO TO.

2. Acoplamento por Dado Comum

Existe este tipo de acoplamento entre um grupo de módulos quando eles compartilham uma área de estrutura de dados comum e é facultado a cada um deles uma interpretação específica da área de dados.
Como exemplo, seja um software que mantenha o cadastro de alunos. Neste software é definido um vetor de caracteres que representa a estrutura global para os dados do aluno. Os caracteres no vetor são organizado como mostra a Figura 2.1.

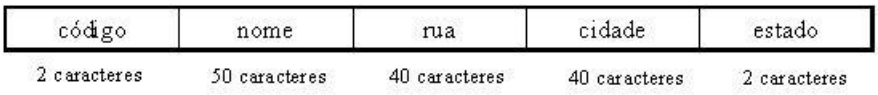


Figura 2.1 Vetor de caracteres representando dados dos alunos

Suponha que o software possui vários módulos que usam esta estrutura, com as funcionalidades de incluir, excluir, alterar e listar alunos. Neste caso, se, por exemplo, o tamanho do campo *código* for alterado, todos os módulos sofrerão impacto desta alteração e necessitarão ser alterados também, mesmo se não utilizarem especificamente o campo alterado.

3. Acoplamento Externo

Existe este tipo de acoplamento em grupo de módulos que referenciam um mesmo termo declarado externamente. Este tipo de acoplamento é muito próximo do acoplamento por dado comum. A diferença básica está na unidade compartilhada entre os módulos: no acoplamento externo os módulos usam itens de dados individuais declarados em uma área de dados comum e não a área comum completa, como ocorre no acoplamento por dado comum.
Como exemplo, seja o mesmo software descrito no item anterior, porém define-se a estrutura global para os dados do aluno mostrada no Código 2.2.1.

```
struct {
  char codigo[2];
  char nome[50];
  char rua[40];
  char cidade[40];
  char estado[2];
}Aluno;
```

Código 2.2.1: *Acoplamento Externo - estrutura global para dados de Aluno*

Neste caso, se, por exemplo, o tamanho do campo *código* for alterado, somente os módulos que utilizam este elemento da estrutura sofrerão impacto desta alteração.
O dano causado por este tipo de acoplamento é menor do que aquele causado pelo acoplamento por dado comum porque uma alteração na área de dados comum ou em algum dos módulos que a utilizam, não afeta necessariamente todos os módulos envolvidos.

4. Acoplamento de Controle

O acoplamento entre dois módulos é desse tipo quando um módulo passa ao outro um parâmetro que determina diretamente o fluxo de execução do outro módulo.
Por exemplo, seja a função ExibeMensagem escrita em C, mostrada no Código 2.2.2.

```
void ExibeMensagem (int controle){
   switch (controle){
    case 1:  printf ("Erro na leitura do arquivo. \n";)
             break;
    case 2:  printf ("Erro na gravação do arquivo. \n";)
             break;
    case 3:  printf ("Dado inválido. \n";)
             break;
    default: printf ("Erro. \n";)
             break;
   }
}
```

Código 2.2.2: *Acoplamento de Controle*

Uma função que chame a função ExibeMensagem está acoplada a ela por controle, pois passa-lhe um parâmetro que define a sua execução.
O problema com este tipo de acoplamento é o fato de um módulo precisar conhecer os detalhes de implementação do outro para poder chamá-lo corretamente.

5. Acoplamento de Referência ou *Stamp*

Ocorre quando dois módulos compartilham uma área de dados não declarada globalmente. Isso se dá quando a comunicação entre dois módulos é feita por meio de chamada de rotinas com passagem de parâmetro por referência. Este tipo de acoplamento é menos grave do que os acoplamentos por dado comum e externo, porém ainda apresenta o problema de efeito colateral de alterações sobre os dados compartilhados.

6. **Acoplamento por Informação**

   Dois módulos estão acoplados por informação se a comunicação entre eles é feita por chamada de rotina com passagem de parâmetros por valor, desde que tais parâmetros não sejam elementos de controle.
   Se dois módulos precisam se comunicar, este é o melhor nível de acoplamento que pode existir entre eles, pois reflete uma situação ideal em que:

   - um módulo não conhece detalhes do outro;
   - um módulo recebe apenas valores enviados pelo outro e pode manipular tais informações na sua execução sem efeito colateral em outros módulos.

7. **Desacoplado**

   Dois módulos estão desacoplados se não existe tipo algum de comunicação entre eles.

2.2.2 Coesão

Coesão é a medida do relacionamento entre elementos internos de um módulo. Um bom projeto visa à maximização da coesão de módulos. Glenford Myers [38] define a seguinte classificação para a coesão interna de módulos, considerando do pior para o melhor nível:

1. **Coesão Coincidental**

   Um módulo tem coesão coincidental se não há relacionamento com significado relevante entre os seus elementos. Os elementos parecem ter sido reunidos ao acaso, portanto não é possível descrever a função do módulo. É o pior tipo de coesão interna de módulo porque módulos com essa característica são prejudiciais a todos os fatores internos de qualidade de software.
   Como exemplo, seja um programa no qual a seqüência de instruções sem relacionamento entre si, mostrada no Código 2.2.3, apareça várias vezes no código do programa.

```
x = 3;
scanf ("%d",y);
if (a > 5) then b = 100;
printf("%d", k);
```

Código 2.2.3: *Coesão Coincidental*

O desenvolvedor, então, para economizar linhas de código resolve criar um módulo M, que contenha tais instruções, cuja chamada substitui

o conjunto de instruções em questão. O módulo M resultante possui coesão coincidental.

2. Coesão Lógica

Um módulo com este tipo de coesão contém elementos que apresentam uma relação lógica entre si. É um nível de coesão melhor do que a coincidental porque aqui os elementos internos do módulo apresentam alguma relação entre si. Porém, módulos com coesão lógica executam mais de uma função, mas com uma única interface com os demais módulos e com passagem de parâmetros desnecessários.
Um exemplo de módulo com esse tipo de coesão é aquele que se responsabiliza por abrir todos os arquivos em um programa e recebe como parâmetro um indicador que determina qual conjunto de arquivos deve ser aberto a cada chamada do módulo.
Este tipo de coesão está associado ao acoplamento de controle e é indesejável, pois cria forte dependência com os demais módulos já que o seu conteúdo precisa ser conhecido externamente para que o mesmo possa ser utilizado.

3. Coesão Clássica

Tal como no caso da coesão lógica, um módulo com este tipo de coesão possui uma série de funções logicamente relacionadas, porém são temporalmente dependentes, isto é, precisam ser executadas em conjunto e em determinada seqüência. Como não há parâmetros para determinar o fluxo de execução do módulo, este tipo de coesão introduz uma melhoria em relação à coesão lógica.
Exemplos típicos desta coesão são módulos de inicialização e finalização em um programa. Um módulo que possua a seqüência de instruções mostrada no Código 2.2.4 constitui um exemplo de ocorrência deste tipo de coesão. Esse módulo tem coesão clássica porque as funções estão logicamente relacionadas, visto que se destinam a iniciar o programa, e estão relacionadas no tempo, pois devem ser executadas juntas em um momento distinto no programa.

```
Abre conexão com servidor
...
Captura data e hora do sistema
...
Inicializa variáveis
```

Código 2.2.4: *Coesão Coincidental*

Myers aponta que a coesão clássica pode ser inevitável em algumas situações, por exemplo, naqueles módulos que se destinam à recuperação de erros em sistema cujas seqüências de passos sejam do tipo:

identificar o erro ocorrido, recuperar da falha e continuar a execução.

4. Coesão Procedural

É similar à coesão clássica, porém, além de as funções executadas pelo módulo precisarem ser executadas em determinada seqüência, elas pertencem ao domínio do problema a ser solucionado. Ainda não é um nível ideal de coesão porque o módulo não executa uma única função. Como exemplo, seja um software que se destina a reproduzir músicas a partir de um CD de áudio. Suponhamos que seja um requisito funcional do software: antes de executar uma música, deve ser exibida uma imagem selecionada aleatoriamente no fundo da janela da aplicação. Um módulo deste software possui a seqüência de instruções que implementam tal requisito mostrada no Código 2.2.5. Esse módulo tem coesão procedural porque o relacionamento temporal do conjunto de instruções se dá em função da aplicação, do problema a ser solucionado e não em função dos procedimentos do programa.

```
Exiba uma imagem de fundo selecionada aleatoriamente.
...
Reproduza a música.
...
```

Código 2.2.5: *Coesão Procedural*

5. Coesão Comunicacional

Um módulo com este tipo de coesão é um módulo com coesão procedural com uma característica a mais: os elementos do módulo comunicam-se entre si por meio de dados. Os elementos do módulo são dependentes de dados comuns quando referenciam um mesmo conjunto de dados ou quando trocam dados entre si. Neste caso, isso pode ser feito quando um elemento do módulo tem como entrada a saída gerada pelo elemento anterior.

Como exemplo, seja um programa que se destina a gerenciar a matrícula de alunos de uma escola. Suponhamos que seja requisito deste software um relatório que exiba, para determinada turma, a lista de alunos que a compõe. O módulo para atender tal requisito foi implementado com a seqüência de passos mostrada no Código 2.2.6. Esse módulo tem coesão comunicacional porque os dados obtidos no primeiro passo são entrada para o segundo passo.

```
Obter alunos da turma
...
Exibir lista de alunos
...
```

Código 2.2.6: *Coesão Comunicacional*

6. Coesão Funcional

É o mais alto grau de coesão interna de módulo. Um módulo com este tipo de coesão é caracterizado por desempenhar uma única função bem definida.
Um módulo cujo objetivo seja calcular a raiz quadrada de um número é um exemplo de módulo com coesão funcional.

7. Coesão Informacional

Um módulo com coesão informacional implementa um tipo abstrato de dados; ele armazena informações e operações, sendo que cada operação realiza um função específica e manipula as informações.
Um módulo que implemente o tipo abstrato de dados Fila possui coesão informacional. Seja, por exemplo, um módulo que implemente uma Fila de Tarefas. A *tarefa* constitui o elemento de informação armazenado; as operações *enfileira*, *desenfileira*, *vazia* e *fazFilaVazia* realizam, cada uma delas, um função específica sobre a fila.

Deve-se buscar obter módulos com alto grau de coesão, pois módulos fortemente coesos tendem a ser mais independentes [38], o que contribui para a construção de software com baixa conectividade, mais reutilizáveis, estáveis e mais fáceis de manter.

Myers propôs as classificações para acoplamento e coesão em uma época em que o paradigma estruturado era o mais difundido. Assim, os conceitos têm base nas possibilidades de projeto e implementação decorrentes dos recursos disponíveis pelas linguagens de programação utilizadas na época, como Fortran e PL/I. Embora estas escalas tenham sido propostas há muito tempo, as observações a respeito dos impactos de cada nível de acoplamento entre módulos e coesão interna de módulos no custo e qualidade na produção de software são válidas para o paradigma da orientação por objetos. Porém, um novo olhar sobre essa classificação à luz da orientação por objetos se faz necessário, para que se possa detalhar como esses tipos de acoplamento e coesão são personificados pelos conceitos e recursos desse paradigma.

O Capítulo 3 apresenta uma análise sobre as formas de acoplamento entre módulos e coesão interna de módulos no contexto da orientação por objetos.

2.2.3 Construção de Software Modular

A procura por técnicas que auxiliem a construção de software modular tem sido motivação para vários trabalhos na área de produção de software [8, 36, 38, 41, 55]. Tamanha importância dada ao assunto não é casual. Como Pressman aponta em [44], a modularidade é a chave para um bom projeto, que por sua vez é chave para a qualidade de software.

O projeto de software deve ser orientado a obter o maior grau de modularidade possível. Para isso, o projetista deve utilizar um método que viabilize atingir tal objetivo. Bertrand Meyer [36] define critérios, regras e princípios para se alcançar modularidade. Os critérios são os requisitos que o método utilizado para criar o software deve atender para ser dito modular. Eles são independentes entre si, isto é, um método pode atender um critério e ferir outro. As regras sucedem os critérios, e são normas a serem seguidas para se atingi-los. Os princípios decorrem das regras, e são características necessárias para que as regras sejam aplicadas corretamente.

Os critérios para que um método seja modular são:

- Decomposibilidade: um método atende a esse critério se permite a construção de software a partir da decomposição do problema a ser resolvido, dividindo-se o problema em subproblemas.

- Composibilidade: um método atende a esse critério se permite a construção de um software a partir de elementos já existentes. Esse critério está relacionado à reusabilidade.

- Inteligibilidade: em um método que segue esse critério, é possível entender um módulo sem a necessidade de recorrer aos demais módulos.

- Continuidade: o método que segue esse critério gera sistemas com estrutura tal que uma alteração em um módulo resulta em poucas alterações nos demais módulos.

- Proteção: esse critério determina que a ocorrência de um erro dentro de um módulo deve gerar pouca propagação de erros para os demais módulos.

Verifica-se em todos os critérios a preocupação com a minimização de dependência entre os módulos do sistema, visando uma estrutura mais flexível.

As regras a serem seguidas para se alcançar os critérios expostos são:

- Mapeamento direto: essa regra determina que deve ser possível definir um modelo para o domínio do problema a ser resolvido. Então, deve haver uma correspondência direta entre tal modelo e a solução dada para o problema.

- Poucas interfaces: a estrutura do sistema deve conter poucas interfaces entre os módulos que o compõe. Essa regra relaciona-se diretamente com o conceito de conectividade e é a chave para atingir os critérios de modularidade.

- Interface pequena: essa regra está relacionada ao conceito de acoplamento de módulos. Ela determina que se dois módulos se comunicam, eles devem trocar o menor número de informação possível.

- Interface explícita: se dois módulos se comunicam, tal comunicação deve estar clara no texto de ambos.

- Ocultação de informação: as informações que são de conhecimento relevante para os demais módulos devem ser publicadas, as demais devem estar ocultas, sendo de conhecimento somente do módulo proprietário.

Os princípios decorrentes dessas regras são os seguintes:

- Unidade modular lingüística: um módulo deve corresponder a uma unidade sintática na linguagem utilizada para representar o sistema, seja uma linguagem de especificação, de projeto ou de implementação.

- Auto-documentação: um módulo deve ser auto-documentado. A documentação de um módulo deve estar fisicamente próxima dele, para garantir a sua legibilidade.

- Acesso uniforme: esse princípio relaciona-se com a regra de ocultação de informação. Os serviços oferecidos por um módulo devem ser disponíveis por meio de uma notação definida e conhecida externamente, não sendo necessário que os clientes de tais serviços tenham conhecimento de seus detalhes.

- Aberto-fechado: este princípio determina que um módulo deve estar aberto para extensão e fechado para modificação. Isto significa que deve ser possível estender o comportamento do módulo, contudo sem modificar o seu código fonte. Módulos que seguem este princípio propiciam a reusabilidade e a manutenibilidade.

- Escolha única: se um sistema possui um conjunto de alternativas de execução, somente um módulo deve conhecer essa lista de opções.

A maior parte dessas orientações de Meyer refletem o cuidado que se deve tomar em relação às comunicações entre os módulos constituintes de um sistema, e leva a uma direção em que se obtém um software constituído por módulos que possuem o mínimo de conexões possíveis entre si. Isso é fundamental para a manutenibilidade, questão abordada na próxima seção.

2.3 Manutenibilidade

Manter um software significa garantir a sua existência. Quando a implementação de um software é finalizada, não se pode dizer que ele está pronto e acabado. Provavelmente, ao longo de sua existência, alguma modificação ele terá que sofrer, seja por uma alteração de requisito, pela identificação de um requisito novo ou para a correção de um erro. A manutenibilidade é a medida da facilidade de se manter um software.

A manutenção de software é ponto crítico para a Engenharia de Software devido ao peso que ela tem no custo total de um sistema. Estatísticas relatadas na literatura mostram que os custos de manutenção de um software são os maiores de todo o seu ciclo de vida. Pfleeger [43] alerta que atualmente o custo de manutenção de software é acima de 80% do custo do sistema. Para Pfleeger, estão entre os fatores que contribuem para o aumento do custo de manutenção de software:

- complexidade do tipo de aplicação: a dificuldade de manutenção é um problema inerente a aplicações de caráter complexo;
- remanejamento de pessoal (*turnover*): a rotatividade de pessoal nas equipes dos sistemas impacta no tempo e no custo de manutenção porque um novo membro na equipe demandará tempo de aprendizado para estar apto a manter o software;
- qualidade da documentação: a inexistência de documentação de especificação de requisitos, de projeto e implementação torna a atividade de manutenção impossível. As informações contidas na documentação devem estar corretas e atualizadas, caso contrário a manutenção também será inviabilizada;
- qualidade do projeto: a qualidade do projeto é definida pela existência de componentes coesos e independentes. Quando tais características não estão presentes no projeto, o seu entendimento é prejudicado e a facilidade de manutenção é seriamente comprometida;
- qualidade do código: se o código não segue as determinações do projeto, a dificuldade de dar manutenção é muito grande.

A chave para minimizar o custo de manutenção está na arquitetura do sistema. Uma arquitetura de sistema flexível, que atenda aos preceitos de um bom projeto modular, contribui de forma determinante para amenizar os problemas acima citados.

Dentre os fatores enumerados por Pfleeger [43], destacamos, na próxima seção, a importância da *qualidade do código* como ponto crítico na manutenibilidade,

tendo em vista que a manutenção de fato se dá sobre o produto. Se o produto for mal construído, a sua manutenção é gravemente prejudicada, ainda que problemas nos demais fatores, como remanejamento de pessoal e qualidade da documentação, sejam superados.

2.4 A Importância da Qualidade do Código de um Software

A gestão de qualidade deve ser feita durante todo o processo de desenvolvimento do software [14]. No processo de software, entende-se como artefato qualquer resultado tangível que seja fruto de alguma etapa no processo ou sirva como insumo para alguma delas [42]. Por exemplo: a proposta de software, o documento de especificação de requisitos, o modelo de dados e o código do sistema. É importante a verificação da qualidade dos artefatos durante todo o processo, pois quanto mais cedo se identificar um problema, menor será o custo de seu ajuste [44] [42].

Um ponto importante na obtenção de software de qualidade é a especificação dos requisitos. Uma especificação de requisitos deve ser completa, correta, precisa e verificável [42], pois nela deve estar definido o objetivo do software, os requisitos que ele deve atender. Contudo, a fase determinante da qualidade do software é o projeto, pois nesta fase define-se como o software será construído. O projeto deve ser criterioso, atender aos requisitos especificados e modelar o sistema de forma a obter uma estrutura flexível. Mas, de nada adianta uma boa especificação de requisitos, um projeto cuidadoso se o produto final - o código - não refletir todos os cuidados que precederam sua implementação. O código é o software propriamente dito, é o artefato mais importante de todo o processo, o objeto de maior valor, pois é com ele que o usuário interage diretamente e é ele quem deve estar preparado para sofrer possíveis alterações ao longo de sua vida. A especificação de requisitos, o projeto, os testes, a manutenção, tudo tem como objeto central o código.

Não se pode correr o risco de confiar demasiadamente nas avaliações realizadas nos artefatos que precedem a implementação do código e dispensar a sua própria avaliação. Questões como legibilidade, eficiência das rotinas utilizadas, tratamento de exceções etc, [42] são relevantes na avaliação do código de um software. Porém, assim como no caso do projeto do software, a estrutura do software implementado é determinante para a sua manutenção, reutilização e para o seu custo. A forma utilizada para implementar deve guardar com rigor os cuidados para se obter software modular. Assim, é necessária a existência de modelos de avaliação de qualidade de código de software do ponto de vista de sua estrutura. E mais ainda, é necessária a existência de ferramentas que viabilizem tal avaliação, tendo em vista o porte que os sistemas reais em geral assumem.

2.5 A Orientação por Objetos e a Qualidade de Software

A orientação por objetos é um paradigma amplamente difundido [2, 36]. Tamanha aceitação desta forma de se conceber, projetar e implementar um software deve-se ao fato de a orientação por objetos possuir características que possibilitam a obtenção de software de alta qualidade, mais fáceis de manter e de reutilizar.

A base da orientação por objetos é a diminuição da distância ou lacuna semântica entre o mundo real e a modelagem do sistema [36]. O software é construído a partir dos objetos que o sistema manipula e da troca de mensagens que ocorre entre os objetos. Classes de objetos são abstrações utilizadas para representar um grupo de objetos com as mesmas características e comportamentos, isto é, representa objetos que possuem em comum um conjunto de atributos e métodos. O recurso de herança possibilita que se modelem eventos do mundo real nos quais uma classe de objetos herda as características e o comportamento de uma ou mais classes, formando o que se chama hierarquia de classes.

As características e os recursos da orientação por objetos - classes, herança, ligação dinâmica e polimorfismo - permitem atingir fatores de qualidade de software com mais facilidade porque viabilizam a obtenção de sistema de estrutura muito flexível. Tal flexibilidade estrutural impacta em maior extensibilidade, maior reusabilidade e maior facilidade de manutenção. As classes de objetos encapsulam dados e serviços, o que permite a construção de software modular. Quando bem empregado, esse recurso permite o mínimo de acoplamento entre as classes, o que contribui significativamente para a redução de custos de manutenção e para a reusabilidade. Somado a isso, o polimorfismo, que é a habi- lidade de se esconder implementações distintas atrás de interfaces comuns, torna os softwares orientados por objetos menos sensíveis a alterações, o que os tornam mais extensíveis [16].

2.6 Conclusão

Software de boa qualidade é sinônimo de software que atende às necessidades dos usuários e é construído de forma que seja fácil de entender, estender, reutilizar e manter. Esses fatores dependem primordialmente do nível de independência entre os módulos do software.

O caminho principal para a obtenção de software com essa característica é construí-lo a partir de módulos fortemente coesos e com o menor grau de acoplamento entre si [38]. Conforme descrito neste capítulo, a literatura sobre este assunto aborda métodos, regras e princípios que levam a obtenção de software constituído por módulos o mais independentes possível uns dos outros.

A conectividade, a medida de interconexões entre módulos de um software, fornece uma avaliação primária da qualidade de sua estrutura e, conseqüente-

mente, da facilidade de sua manutenção. A alta conectividade é sinal de que o software não foi construído de forma a manter a independência entre os seus módulos, o que compromete a sua qualidade e a sua manutenibilidade. Esta idéia é abordada em maiores detalhes no Capítulo 4.

Muitas vezes, a qualidade de um software é avaliada na base do senso comum, o que não propicia uma avaliação objetiva do software. No processo de desenvolvimento de software, é importante a existência de métodos de controle de qualidade objetivos, o que garante uma gerência no processo de software mais segura. É preciso quantificar, comparar, avaliar para que se possa controlar. As métricas de software assumem papel de extrema importância, pois permitem a avaliação quantitativa dos produtos, das pessoas e do próprio processo. O próximo capítulo apresenta uma revisão bibliográfica sobre métricas de software orientado por objetos.

Capítulo 3

Coesão e Acoplamento na Orientação por Objetos

Os conceitos de coesão interna de módulos e acoplamento entre módulos são instrumentos úteis na avaliação do grau de modularidade de um sistema. As definições realizadas para as escalas de coesão e acoplamento definidas por Myers [38], apresentadas nas Seções 2.2.2 e 2.2.1 , respectivamente, desta dissertação, foram realizadas em uma época na qual a orientação por objetos ainda não era um paradigma consolidado. Como aquelas definições não abordam as características específicas da OO, elas necessitam ser revisitadas para que possam se adequar a esse paradigma. É importante conhecer como os diversos graus de coesão e acoplamento apresentam-se na construções da OO, pois isso é um conhecimento útil na avaliação do grau de modularidade em sistemas neste paradigma. Em particular, a avaliação desses dois fatores auxilia na obtenção de sistemas com menor nível de conectividade, pois atuar na estrutura dos módulos de um sistema e na forma como eles interagem entre si é o ponto essencial para aumentar a independência entre eles.

A avaliação do grau de coesão interna de módulos e do grau de acoplamento entre módulos tem como pré-requisito a definição do que será considerado *módulo*. Conforme descrito na Seção 2.2, as seguintes definições genéricas para módulos são encontradas na literatura, dentre outras:

- conjunto de um ou mais arquivos que possa ser compilado separadamente [55];
- menor unidade na qual o sistema pode ser decomposto [36].

Booch [10] aponta que a *classe* é a unidade primária de decomposição em sistemas orientados por objetos, e não o algoritmo, como ocorre em sistema desenvolvidos no paradigma estruturado. Para Bertrand Meyer [36], a classe desempenha o papel central na OO, representando além de um tipo de objetos, uma unidade modular. Ele enfatiza esta idéia apontando que em ambientes orientados por objetos as classes devem ser os únicos módulos.

Neste capítulo propomos uma adaptação dos critérios de coesão interna de módulos e acoplamento entre módulos em sistemas orientados para objetos, que é uma releitura das escalas propostas por Myers [38] à luz da OO. Para isso, consideraremos *classe* como módulos em sistemas orientados por objetos, em concordância como Booch[10] e Meyer [36].

3.1 Coesão Interna de Classes

A coesão interna de um módulo é definida pelo grau de relacionamento existente entre seus elementos. Desta forma, a coesão interna de uma classe é determinada pelo grau de relacionamento existente entre seus constituintes: seus membros de dados e seus métodos. A avaliação da coesão de um módulo não é tarefa trivial, pois, como ressaltado em [51], essa tarefa requer conhecimento técnico do domínio da aplicação do sistema em questão, dentre outros requisitos.

Em [51], destacam-se os seguintes aspectos como indicadores do bom nível de coesão de uma classe:

- a classe deve representar um conceito completo e coerente, e não uma coleção aleatória de informações;

- cada método de sua interface pública deve realizar uma única função bem definida.

Este último ponto ressalta que, além da coesão entre os elementos da classe, é necessário avaliar a coesão interna de seus métodos, sobretudo daqueles que constituem a sua interface.

Em sistemas orientados por objetos, a principal via de comunicação entre classes se dá por meio de suas interfaces. Conforme Meyer [36] define, a interface de uma classe corresponde ao seu contrato, que inclui os contratos individuais dos métodos exportados pela classe bem como os seus membros de dados públicos. Staa [55] define o conceito de *conector* como a via de comunicação entre dois módulos e define a *coesão de um conector* como o grau de interdependência semântica entre os seus elementos. Desta forma, para sistemas orientados por objetos, definiremos, aqui, *coesão da interface da classe* como o grau de interdependência semântica entre os elementos que compõem a interface de uma classe.

Assim, identificamos três níveis de avaliação de coesão em sistemas orientados por objetos:

- coesão interna de método: grau de relacionamento entre os elementos internos do método: seus dados locais e suas instruções;

- coesão interna da classe: grau de relacionamento entre os elementos internos da classe;

- coesão da interface da classe: refere-se ao grau de relacionamento entre os elementos da interface da classe.

A seguir, são apresentadas uma classificação de grau de coesão para esses três níveis.

3.1.1 Coesão Interna de Métodos

Métodos são estruturalmente próximos a procedimentos e funções do paradigma estruturado. Assim, a maior parte da classificação de coesão interna de módulos proposta por Glenford Myers [39] pode ser utilizada ou adaptada à avaliação de coesão no caso de métodos, resultando na seguinte escala, do pior para o melhor caso:

- **Coesão Coincidental**

  É o pior tipo de coesão, pois não há relacionamento com significado relevante entre os elementos do método.

- **Coesão Lógica**

  O método realiza mais de uma função logicamente relacionadas, com uma única interface com o usuário do método. A função a ser executada é determinada por meio de parâmetros.

- **Coesão Temporal**

  Corresponde à coesão clássica defina por Myers. As funções realizadas pelo método, além de serem logicamente relacionadas, são temporalmente dependentes, pois precisam ser executadas em conjunto e em determinada seqüência.

- **Coesão Procedimental**

  É similar à coesão temporal, somado-se o fato de que as funções desempenhadas pelo método são relacionadas ao domínio do problema a ser solucionado.

- **Coesão Comunicacional**

  Um método com esse tipo de coesão possui coesão procedimental além de suas funções se comunicarem por meio de dados comuns.

- **Coesão Funcional**

  Um método com este tipo de coesão desempenha uma única função bem definida.

Como destaca Pressman [44], um módulo coeso realiza uma única função, ocasionando pouca interação com outras partes do software. Trazendo esta leitura para o universo de *métodos* na orientação por objetos, podemos dizer que métodos coesos, por realizarem uma função bem definida, ocasionam poucas conexões, seja com outros métodos de sua própria classe, seja com outras classes.

Cabe, aqui, uma análise particular dos menores níveis de coesão: coincidental e lógica. Um método com coesão coincidental pode realizar várias tarefas sem relacionamento entre si. Desta forma, tende a ser mais usado por outras partes do software, ou ser cliente de várias outras partes do software. Um método com esta característica é fonte para o estabelecimento de alta conectividade da classe. Da mesma forma, no caso da coesão lógica, como o método realiza mais de uma função, ele é determinante para a conectividade.

Além disso, métodos com coesão baixa podem denotar um problema estrutural da classe, como o baixo grau de coesão interna da classe à qual pertencem. Se um método realiza mais de uma função, é preciso investigar se isso decorre do fato de a classe implementar mais de um contrato, por exemplo. Como a coesão interna de uma classe impacta na sua conectividade, e é influenciada pela coesão interna de seus métodos, em última análise, concluimos que a coesão interna dos métodos de uma classe tem impacto na conectividade dessa classe.

Desta forma, a criação de métodos altamente coesos é indispensável para a obtenção de software orientado por objetos de baixa conectividade.

3.1.2 Coesão Interna de Classes

O grau de coesão interna de uma classe é definido a partir da observação do relacionamento existente entre seus elementos - seus membros de dados e seus métodos. Um conceito importante na avaliação de coesão interna de classes é o *contrato* [36]. Em um sistema orientado por objetos, classes podem ser vistas como *fornecedoras* ou *clientes* de serviços. Esta relação de cliente-fornecedor é regida por um *contrato*, que define regras a serem seguidas tanto por quem usa quanto por quem fornece o serviço. A classe fornecedora compromete-se a fornecer um determinado serviço a partir de uma especificação do problema e publica a interface a ser utilizada por aquelas que precisarem utilizar o serviço. Para benefício da modularidade, é importante a criação de classes que implementam um único contrato.

Os critérios de organização dos elementos internos de uma classe são apresentados a seguir, do pior para o melhor caso:

- **Coesão Coincidental**

  Uma classe tem coesão coincidental se não se observa relacionamento relevante entre os seus elementos. A ausência de relacionamento relevante entre os elementos de uma classe ocorre nos seguinte casos:

- A classe implementa mais de um contrato. Um exemplo é uma classe na qual são agrupadas as implementações dos tipos abstratos de dados pilha e lista.
- Não há relacionamento entre os métodos da classe ou não há relacionamento entre os dados da classe. Um exemplo deste caso ocorre em classes que contêm definições de tipos, dados e métodos sem qualquer relacionamento entre si. Em [58], este caso é referenciado como *utility cohesion*, pois agrupam-se utilitários que não pertencem logicamente a nenhuma outra parte do software. O exemplo em Java do Código 3.1.1 ilustra este caso.

```
public class Geral {
  int ponto;
  int usuario;
  int codigoLivro;

  public void LimpaTela() { ... }

  public float CalculaImpostoRenda() { ... }

  public int CalculaNotaGlobal() { ... }
}
```

Código 3.1.1: *Coesão Coincidental na Orientação por Objetos*

- **Coesão Lógica**

  A coesão lógica, tal como é definida por Myers [38], ocorre quando um módulo desempenha um conjunto de funções relacionadas logicamente e uma delas é selecionada por meio de um indicador passado como parâmetro para o módulo. Essa situação não ocorre no caso de classes, pois o serviço a ser realizado é chamado diretamente pelo cliente da classe. Uma classe tem coesão lógica quando implementa mais de um contrato relacionados logicamente. A classe *Modem*, cuja estrutura é mostrada no Código 3.1.2, exemplifica este caso. No exemplo, a classe implementa dois contratos distintos: a gerência da conexão, constituída pelos métodos *discar()* e *desligar()*, e a comunicação de dados, constituída pelos métodos *enviar(char c)* e *receber()*. Há uma relação lógica entre os contratos, pois ambos referem-se à interface de um modem.

  Um melhoria em coesão interna de classes seria obtida com a construção de duas classes, em substituição a *Modem*, cada uma delas realizando um contrato. As classes *ModemConexao* e *ModemComunicacao*, do Código 3.1.3, mostram o resultado dessa separação de responsabilidades.

```
public class Modem {
  public void Discar() { ... }
  public void Desligar() { ... }
  public void enviar(char c) { ... }
  public void receber() { ... }
}
```

Código 3.1.2: *Coesão Lógica na Orientação por Objetos*

```
public class ModemConexao {
  public void Discar() { ... }
  public void Desligar() { ... }
}
public class ModemComunicacao {
  public void enviar(char c) { ... }
  public void receber() { ... }
}
```

Código 3.1.3: *Coesão Lógica na Orientação por Objetos - separação de responsabilidades*

- **Coesão Temporal**

  A coesão temporal corresponde à coesão clássica, definida por Myers [38], caracterizada por um conjunto de funções relacionadas logicamente e que necessitam ser executadas em conjunto e em seqüência determinada. Um exemplo desta situação é uma classe cujos métodos destinam-se a iniciar o ambiente de execução da aplicação. A classe *AmbienteInicial*, do Código 3.1.4, exemplifica este caso. No exemplo, o método *AbreArquivoParametros(Strin nomeArq)* abre um arquivo que contém parâmetros a serem utilizados na aplicação, entre eles, o nome e a localização de um arquivo de *log* da aplicação; o método *AbreArquivoLog(String nomeArq)* abre o arquivo de *log*. Há um relacionamento lógico entre os métodos, pois ambos destinam-se à inicialização do ambiente da aplicação. Além disso, *AbreArquivoLog* precisa ser executado após *AbreArquivoParametros*.

- **Coesão Procedimental**

  Uma classe tem coesão procedimental se entre os seus elementos internos observa-se relacionamento procedimental, isto é, os métodos da classe precisam ser executados em uma seqüência determinada pelo domínio do problema.

```
public class Ambiente{
  ...
  public void AbreArquivoParametros(String nomeArq) { ... }
  public void AbreArquivoLog() { ... }
}
```

Código 3.1.4: *Coesão Temporal na Orientação por Objetos*

Um exemplo desta situação ocorre quando utiliza-se métodos de inicialização para determinar o estado inicial do objeto em vez de utilizar um método construtor. Nesta situação, os métodos de inicialização devem ser executados antes dos demais.

- **Coesão Comunicacional**

  A coesão comunicacional ocorre quando os elementos do módulo são dependentes de membros de dados comuns da classe. Em uma classe, essa é uma característica essencial, pois é importante que os métodos operem de maneira relevante sobre os dados da classe.

- **Coesão Contratual**

  É o melhor nível de coesão interna de uma classe. Uma classe tem coesão contratual se implementa um único contrato. Um exemplo é uma classe que implementa o tipo abstrato de dados Pilha.

3.1.3 Coesão de Interface de Classe

A avaliação do grau coesão da interface de uma classe tem os mesmos preceitos daquela realizada para a classe como um todo. A diferença é que no caso da coesão de interface os elementos avaliados são exclusivamente aqueles exportados pela classe. Então, a coesão de interface de classe pode ser classificada de acordo com os mesmos critérios da coesão interna de classes, descrita na seção 3.1.2.

3.2 Acoplamento

Como, na orientação por objetos, uma classe corresponde a um módulo, o acoplamento entre módulos neste paradigma é definido pelo relacionamento entre as classes de um sistema. A forma como duas classes se relacionam é definida pela forma como seus elementos se relacionam. O grau de acoplamento entre classes pode ser definido de acordo com os seguintes critérios, do nível mais forte de acoplamento para o mais fraco:

- **Acoplamento por Conteúdo**

  O acoplamento por conteúdo ocorre quando uma classe modifica sub-repticiamente dados de objetos de outra classe. Isso ocorre principalmente em decorrência do uso de atributos públicos. Identificar este tipo de acoplamento em sistemas orientados por objetos pode ser difícil, pois envolve saber em que situação o acesso ao dado se dá de forma não autorizada, isto é, se o dado acessado é público em decorrência da aplicação ou por um eventual descuido do projetista.
  Como exemplo, seja uma classe A que define o tipo abstrato de dados pilha via suas operações. Entretanto, o projetista, por descuido, deixa um de seus atributos públicos. Se uma classe B modificar o valor deste atributo, o fará de forma não prevista pelo projetista e as classes A e B estarão acopladas por conteúdo. A ocorrência deste tipo de acoplamento pode ser evitada por meio do uso exclusivo de atributos privados.

- **Acoplamento por Dado Comum**

  Este tipo de acoplamento ocorre entre duas ou mais classes que têm acesso à estrutura de um objeto comum declarado globalmente. Neste caso, o acesso é previsto pelo projetista.

  Alterações na estrutura do objeto comum impacta em todos os módulos que a utilizam, assim como uma alteração em algum desses módulos pode ter impacto em todos os demais. Como exemplo deste tipo de acoplamento, sejam as classes *Protocolo*, *Remetente* e *Receptor* do Código 3.2.1, definidas em Java.

  Neste exemplo, a classe *Protocolo* define um vetor que representa um protocolo de comunicação entre as classes *Remetente* e *Receptor*. Se a estrutura do objeto comum for modificada na classe *Protocolo*, as demais classe sofrerão impacto desta alteração. Da mesma forma, se houver uma alteração na estrutura da classe *Remetente* ou *Receptor* que esteja relacionada ao objeto compartilhado, as demais classes também sofrerão impacto.

  Como no caso do acoplamento por conteúdo, esse tipo de acoplamento pode ser contornado evitando-se o uso de dados públicos, utilizando-se métodos públicos para acesso a tais estruturas.

- **Acoplamento por Inclusão**

  Em [58] e [51] é descrito um tipo de acoplamento distinto daqueles inicialmente identificados por Myers [38]: o acoplamento por inclusão[58] ou por conteúdo léxico[51]. Este tipo de acoplamento ocorre quando um módulo inclui o código de outro módulo. Todos os módulos que incluiram um determinado módulo estão acoplados por inclusão entre si e ao módulo incluído.

```
public class Protocolo {
  public static int[]inf = new int[3];
}

public class Remetente{
...
  public void GravaInformacao(){
    Protocolo p;
    ...
    p.inf[1] = x;
    ...
  }
...
}

public class Receptor{
...
  public void LeInformacao(){
    Protocolo p;
    ...
    y = p.inf[2];
    ...
  }
...
}
```

Código 3.2.1: *Acoplamento por Dado Comum na Orientação por Objetos*

Na orientação por objetos, é representado pelo uso da diretiva *include* do C/C++. Este tipo de acoplamento foge do conceito estabelecido aqui para módulo na orientação por objeto - uma classe - e pode ser visto como um tipo particular de acoplamento por dado comum. Porém merece destaque por dois motivos: diferencia-se do acoplamento por dado comum porque o que é compartilhado neste caso é código e não dado; embora não esteja diretamente relacionado ao conceito de classe, é um tipo de acoplamento que impacta na manutenção do sistema, visto que gera forte dependência entre os módulos envolvidos.

- **Acoplamento por Elemento Externo**

  Este tipo de acoplamento ocorre entre duas ou mais classes que têm acesso a elementos individuais da estrutura de um objeto comum. Neste caso também, o acesso é previsto pelo projetista. Como o acesso é feito a elementos individuais e não à totalidade da estrutura, como ocorre no acoplamento por dado comum, as conseqüências desse tipo de acoplamento são menores do que o por dados comum. Uma alteração na estrutura comum afeta somente aqueles que utilizam os elementos alterados, assim como uma alteração em uma classe que usa parte da estrutura afeta somente as classes que utilizam os mesmos elementos da estrutura. Porém, ainda é um nível inadequado de acoplamento.

  Um exemplo da ocorrência deste tipo de acoplamento, são classes que possuem definições de dados e suas clientes. As classes D, A e B do Código

3.2.2, definidas em Java, ilustram este exemplo.

```
public class D{
  public int x;
  public float z;
  public int y;
}

public class A{
  ...
  D d;
  ...
  d.x = 10;
  ...
  a = d.y;
  ...
}

public class B{
  ...
  D d;
  ...
  b = d.x;
  ...
  d.z = 50;
  ...
}
```

Código 3.2.2: *Acoplamento por Elemento Externo na Orientação por Objetos*

No exemplo, a classe D contém a definição do objeto comum e as classe A e B fazem acesso a componentes individuais do objeto comum.

Conforme apontado em [58], o uso do padrão *façade* [24] contribui para a redução deste tipo de acoplamento.

- **Acoplamento por Controle**

  O acoplamento de controle entre classes ocorre quando o método de uma classe invoca o método de outra classe e conhece o funcionamento deste, utilizando parâmetro que controla o seu fluxo de execução. No exemplo mostrado no Código 3.2.3, a classe B invoca o método m da classe A, e controla a execução deste método por meio do parâmetro p.

  Este tipo de acoplamento é indesejável porque implica que uma classe conheça os detalhes de implementação da outra. Ele pode ser contornado realizando-se chamadas diretas às funções desejadas. No exemplo anterior, na classe B, a chamada *a.m(2)* poderia ser substituída por uma chamada direta ao método *desenhaRetangulo*.

- **Acoplamento por Referência**

  Duas classes estão acopladas por referência se um método de uma delas invoca um método da outra passando-lhe parâmetros por referência.

  Em Java, a passagem de parâmetros de tipos primitivos ocorre por valor. Varejão [59] destaca que a passagem de parâmetros do tipo referência ocorre

```
public class A{
  ...
  public void m(int p){
    if (p == 1)
       desenhaCirculo();
    else
       desenhaRetangulo();
  }
  ...
}

public class B{
  ...
    A a;
    ...
    a.m(2);
  ...
}
```

Código 3.2.3: *Acoplamento por Controle na Orientação por Objetos*

também por valor; entretanto, neste caso, passa-se uma cópia da referência do parâmetro real. Assim, alterações em componentes do parâmetro formal têm efeito no objeto apontado pelo parâmetro real. Portanto, a passagem de um tipo referência como parâmetro em Java tem efeitos de passagem de parâmetro por referência. As classes A e B mostradas no Código 3.2.4, definidas em Java, exemplificam este tipo de acoplamento:

```
public class A{
  B b;
  ...
  public void p (){
     ClasseQualquer obj = new ClasseQualquer();
     ...
     b.metodo(obj);
     ...
  }
  ...
}

public class B{
   private int s = 1000;
   ...
   public void metodo(ClasseQualquer x){
      ...
      x.atualizaCampo(s);
      ...
   }
   ...
}
```

Código 3.2.4: *Acoplamento por Referência na Orientação por Objetos*

No exemplo, a alteração no valor de um campo de x, realizada pela classe B, é percebida na classe A e ambas estão acopladas por referência.

- **Acoplamento por Informação**

  O acoplamento entre duas classes é por informação se não é nenhum dos

descritos anteriormente e se o método de uma delas invoca o método da outra passando-lhe parâmetros por valor.

As classes *A* e *B* do Código 3.2.5, definidas em Java, exemplificam este tipo de acoplamento:

```
public class A{
  private int x;

  public void fazAlgo(int s){
     x = s + 200;
     ...
  }
  ...
}

public class B{
   private A a;
   private int q = 1000;
   ...
   public void m(){
      ...
      a.fazAlgo(q);
      ...
   }
   ...
}
```

Código 3.2.5: *Acoplamento por Informação na Orientação por Objetos*

No exemplo, a alteração no valor de *s* realizada pela classe *A* não é percebida na classe *B*. A única dependência entre ambas está no significado do dado passado como parâmetro.

- **Desacoplado**

  Duas classes estão desacopladas se não há tipo algum de comunicação entre elas.

3.3 Conclusão

Sistemas orientados por objetos são construídos basicamente por classes que se conectam entre si, seja por colaboração ou por herança [10]. A forma como essas conexões ocorrem determinam o grau de acoplamento entre as classes. Por exemplo, se uma classe B é herdeira de A, e uma classe C usa um serviço de A, a dependência de B em relação a A é mais forte do que aquela de C em relação a A.

O grau de dependência entre duas classes decorre do critério utilizado para o estabelecimento da conexão entre elas. No exemplo supracitado, a construção da classe B como herdeira de A seguiu o critério de forte acoplamento; e a decisão de que C utilizasse um serviço de A, seguiu o critério de estabelecer baixo grau

de acoplamento entre C e A. O ideal é que o estabelecimento de conexões entre classes seja orientado pelo critério de menor grau de acoplamento possível.

Um fator importante na construção de uma classe é a determinação de sua coesão interna, a forma como os seus elementos internos se relacionam. Por exemplo, sejam duas classe X e Y. No caso de X, o projetista decidiu implementá-la como um TAD; já Y, fora implementada com uma classe depositária. Como resultado, o grau de interrelacionamento entre os elementos de X é maior do que aquele entre os elementos de Y. Como a coesão interna de uma classe é determinante para a modularidade do sistema, é de fundamental importância a construção de classes com alta coesão interna.

Como as estruturas responsáveis pela realização das operações em sistemas orientados por objetos são os *métodos*, deve-se observar cuidado ao defini-los. Assim como as classes, os métodos devem ser definidos com o maior grau de coesão possível, para que sua legibilidade e manutenibilidade sejam melhores. Método pouco coeso é fonte para o estabelecimento de alta conectividade da classe à qual pertence com outras classes do sistema, o que impacta na conectividade geral do sistema.

Neste contexto, faz-se necessário conhecer os diversos critérios de acoplamento entre classes, de coesão interna de classes e de coesão interna de métodos.

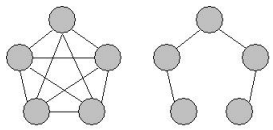
Capítulo 4

A Conectividade e a Estabilidade de Sistemas

Uma situação ideal na produção de um software seria aquela em que o projetista ou o implementador pudesse realizar uma alteração em determinada parte do software e tal alteração não afetasse as demais partes deste software. Entretanto, ao contrário desta situação ideal, é comum ocorrer o que Martin [35] chama de apodrecimento de projeto (*rotting design*), quando o software se transforma em um conjunto de código de difícil manutenção. Martin identifica a rigidez como um dos sintomas principais deste processo. Em um software de estrutura rígida, uma alteração em um módulo causa uma cascata de outras alterações nos demais. De acordo com Martin[35], a rigidez de um software determina a sua degradação e se estabelece devido à existência de alto grau de interdependência entre os módulos do software.

Gilb [26] denomina a característica de um sistema que se mantém relativamente inalterado diante de uma alteração em seu ambiente como *estabilidade*. Para Gilb [26], a medida da estabilidade de um sistema S diante de uma alteração A é dada pelo percentual de alterações realizadas em S em decorrência de A. A estabilidade é, então, o fator que indica a amplitude do impacto que uma alteração tem em um sistema. Poder conhecer a estabilidade de um sistema é um recurso poderoso no processo de software, pois isso significa, dentre outros benefícios, um forte dado para a predição do esforço real necessário para a realização de alterações em sistemas.

Conforme Meyer [36] aponta, o caminho para se obter software constituído por módulos o mais independentes entre si possível é que a comunicação entre eles ocorra de forma disciplinada. Para alcançar isso, ele define, dentre outras regras, a regra de *poucas interfaces*, a qual determina que todo módulo deve se comunicar com o menor número possível de outros módulos. Com isso, uma alteração em determinado módulo propaga-se para poucos módulos. Um software cuja estrutura é planejada segundo essa regra tem estrutura mais flexível e possui alta estabilidade. A Figura 4.1*a* representa de forma simplificada um sistema no



(a) (b)

Figura 4.1 (a) Alto grau de conectividade (b) Menor grau de conectividade

qual cada módulo do sistema comunica-se com um número alto de outros módulos e a Figura 4.1*b*, um sistema com menor grau de comunicação entre os módulos.

Staa [55] diz que os módulos de um sistema comunicam-se entre si por meio de *conectores*, que são vistos como canais por meio dos quais os módulos trocam dados ou sinais de controle. São exemplos de conectores: variáveis globais, membros de dados e métodos públicos de uma classe.

Definimos, aqui, o conceito de *conectividade* como o grau de intercomunicação entre os módulos de um sistema. A conectividade é o aspecto principal a ser analisado para obter a resposta de questões importantes no processo de software, como: dificuldade de manter o sistema e amplitude do impacto de determinada modificação no sistema.

Myers [38] modela a importância da conectividade para a estabilidade de um sistema por meio de um conjunto de 100 lâmpadas, no qual uma lâmpada representa um módulo, e uma ligação entre duas lâmpadas representa a existência de comunicação entre dois módulos. Uma alteração em um módulo corresponde a ligar a lâmpada que representa o módulo. No esquema de conexões das lâmpadas, a probabilidade de uma lâmpada passar do estado de ligada para desligada no próximo segundo é de 50%; se está desligada, a probabilidade de ser ligada no próximo segundo é de 50% se pelo menos outra lâmpada que esteja a ela conectada for ligada. Por outro lado, se uma lâmpada estiver desligada, permanecerá assim enquanto todas as lâmpadas conectadas a ela estiverem desligadas. Diz-se que o circuito atingiu o equilíbrio quando todas as lâmpadas estiverem desligadas. Nesse Modelo das Lâmpadas, Myers analisa três situações possíveis de arranjo das conexões:

- Inexistência de conexões entre as lâmpadas: neste caso, o tempo de equilíbrio do circuito é de 7 segundos.
- Circuito constituído por agrupamentos de 10 lâmpadas: os agrupamentos são independentes entre si e cada um deles é totalmente conectado. Neste caso, o tempo de equilíbrio do circuito é de 20 minutos.
- Circuito totalmente conectado: nesta configuração, cada lâmpada possui uma conexão com todas as demais lâmpadas do circuito. Esse é o pior caso, pois tempo para atingir o equilíbrio do circuito é de 10^{22} anos.

As situações mostradas ilustram quão importante a conectividade é para a estabilidade e manutenção de sistemas. O efeito de uma alteração em um módulo de um sistema com alto grau de conectividade é explosivo, o que torna a sua manutenção um problema intratável.

Diante da necessidade de uma modificação qualquer no sistema, por exemplo em decorrência de uma alteração de um requisito ou de uma correção de um erro identificado, em primeiro lugar, deve ser possível a análise completa do impacto desta modificação. Deve ser possível que o projetista ou o implementador possa identificar quais outros módulos do sistema sofrerão impacto em conseqüência da modificação a ser realizada. Deixar de analisar esse impacto favorece o surgimento de problemas em outros pontos do sistema e, na pior das hipóteses, o sistema pode passar a operar de forma incorreta. A conectividade é o aspecto principal nesta análise, pois o fato de um módulo estar conectado a outro implica que uma alteração em um dos módulos pode gerar impacto no outro.

Diante de um nível de conectividade alto, deve ser possível realizar uma análise sobre a estrutura do sistema, a estrutura de seus módulos e a forma como as conexões se estabelecem nele e atuar nos fatores que elevam a conectividade com o objetivo de reduzi-la.

A seção seguinte identifica os principais fatores que têm impacto na conectividade em sistemas orientados por objetos.

4.1 Fatores de Impacto na Conectividade

Quatro importantes fatores, dentre outros, relacionados à estrutura de software orientado por objetos contribuem para a sua conectividade: profundidade da árvore de herança, ocultação de informação, coesão interna de classes e acoplamento entre classes. O diagrama da Figura 4.2 representa o relacionamento existente entre esses fatores. Os retângulos representam os fatores e as arestas direcionadas entre os retângulos representam uma relação de causa e efeito, onde a causa é o retângulo na origem da aresta e o efeito é o retângulo no final da aresta. As relações entre esses fatores e o aspecto *conectividade* são descritas a seguir.

- **Ocultação de Informação**

  Este é um fator muito importante a ser observado para se manter baixo grau de conectividade em sistemas orientados por objetos. Manter as informações de determinada classe o mais ocultas possível dentro dela contribui para a diminuição da conectividade do sistema, uma vez que minimiza o número de portas abertas para conexão com uma classe. Além disso, a ocultação de informação contribui para a diminuição do grau de acoplamento, pois evita o surgimento de acoplamentos de alto grau decorrentes do acesso direto a

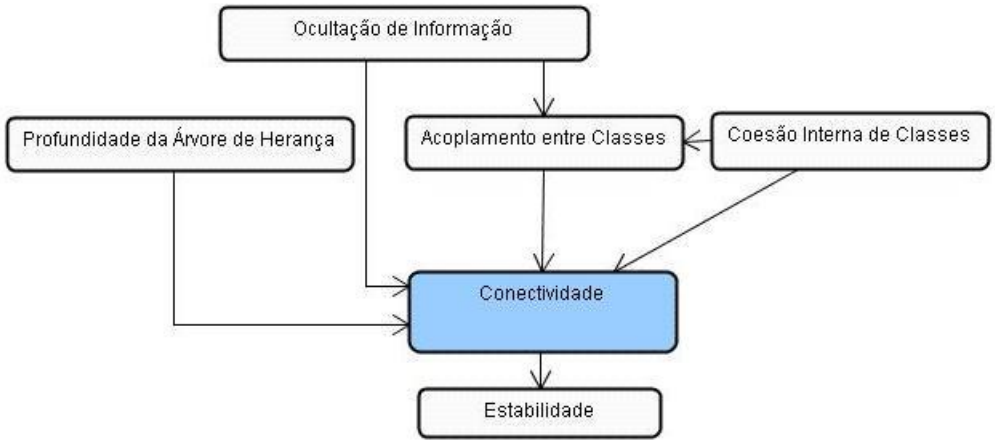


Figura 4.2 Fatores que contribuem para a conectividade em sistemas OO

dados de um módulo por outros módulo.

- **Acoplamento entre Classes**

  O principal caminho para obter diminuição do grau de conectividade do software é a avaliação das conexões estabelecidas nele. Em ambientes orientados por objetos, duas formas de conexões diretas básicas podem ser estabelecidas entre duas classes: por herança, quando uma classe herda características de outra, ou por uso, quando uma classe usa serviços ou dados de outra, o que caracteriza uma relação do tipo cliente-servidor [55].

  Nas conexões estabelecidas por uso, as classes envolvidas são estruturadas de tal forma que objetos de uma fazem referência a objetos da outra. Gamma et al. [24] denominam este tipo de relacionamento como composição de objetos. Idealmente, neste tipo de conexão, um objeto deve utilizar o outro apenas por meio de suas interfaces, o que garante o respeito à ocultação de informação e ao encapsulamento.

  Para conhecer o impacto real das conexões estabelecidas em um sistema, é preciso conhecer as características de tais conexões. Faz-se necessário conhecer a largura de cada conexão, que corresponde ao grau de acoplamento existente nelas. Uma conexão fina é aquela que resulta em baixo grau de acoplamento, por exemplo, quando um método de uma classe invoca um método da outra com passagem de parâmetro por valor; uma conexão grossa resulta na existência de acoplamento forte entre duas classes, por exemplo, quando um atualiza diretamente um dado da outra. Uma conexão fina não representa para o sistema o mesmo grau de dependência de uma conexão

grossa. Os efeitos de alterações em um sistema cujas conexões sejam finas são mais controláveis do que as alterações em um segundo sistema com o mesmo nível de conectividade do primeiro, porém com conexões mais grossas.

A espessura das conexões – o grau de acoplamento – é um fator de importante impacto na conectividade de sistemas, porque atuando-se sobre o grau de acoplamento das conexões do tipo cliente-servidor do sistema, obtém-se um ganho quando se consegue diminuir a largura das conexões ou quando, na melhor das hipóteses, consegue-se eliminar conexões.

- **Coesão Interna de Classes**

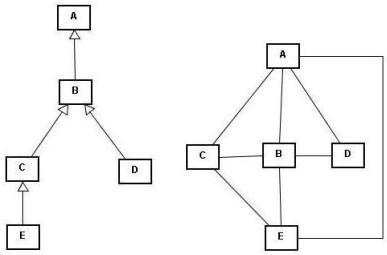
  Módulos pouco coesos tendem a realizar tarefas diversas e a quantidade de conexões com os demais módulos tende a ser maior. A reestruturação de uma classe de baixa coesão interna pode resultar na criação de duas ou mais classes de alto grau de coesão interna. A conectividade das classes resultantes tende a ser menor do que a da classe original, o que impacta na diminuição da conectividade do sistema.

- **Profundidade da Árvore de Herança**

  A conexão de herança surge quando uma classe B é herdeira de uma classe A. Neste caso, existe entre A e B uma relação estreita que determina que uma alteração em A impacta em alteração em B. Isso porque a relação de herança é uma relação do tipo "é um" [36]. Se B é herdeira de A, então B é uma classe A, ou seja, B possui as características definidas para A. Se uma característica de A sofre alteração, seja um membro de dado ou um membro função, esta alteração é percebida por seus herdeiros.

  Na hierarquia de classes representada na Figura 4.3a, qualquer alteração em A é percebida imediatamente em B, mesmo que a característica alterada tenha sido redefinida em B, pois, pelo emprego de polimorfismo, B pode se comportar de acordo com a definição estabelecida na superclasse. Por exemplo, se um método for incluído, excluído ou alterado na superclasse, a subclasse é afetada. As classes C e D, sofrem impacto tanto das alterações ocorridas em A quanto daquelas ocorridas em B. A classe E, que está no nível mais profundo da hierarquia, sofre impacto das alterações ocorridas em A, B e C. A Figura 4.3b representa as conexões das classes em decorrência da hierarquia existente entre elas mostrada na Figura 4.3a.

  Embora seja um consenso na literatura que a herança é um recurso poderoso da orientação por objetos para a obtenção de reusabilidade [36] [16] [44], estudos de alguns pesquisadores revelam que seu uso deve ser cauteloso. Gamma et al. [24] denominam a reutilização obtida por herança como reutilização caixa-branca, pois os detalhes de implementação de uma classe



(a) (b)

Figura 4.3 (a) Hierarquia de classes (b) Representação das conexões existentes na hierarquia de classes

ficam expostos às suas descendentes e uma alteração na superclasse impacta em alterações nas suas descendentes. Baseado em uma comparação das vantagens e desvantagens da reutilização obtidas por herança e por composição de objetos, eles definem o princípio de projeto orientado por objetos "Favoreça a composição de objetos em relação à herança". Este princípio não defende que todo uso de herança deve ser abandonado, mas que este é um recurso que deve ser usado com cuidado.

Um estudo de Daly et al. [15] mostra que árvores de herança profundas provocam perda de manutenibilidade de software, pois quanto mais profunda a árvore, maior o tempo e maior a dificuldade para compreendê-la. Sommerville [53] também aponta que a herança introduz dificuldades para análise e entendimento do comportamento dos objetos, o que torna mais difícil solucionar erros nos sistemas.

Tendo em vista os efeitos da herança em sistemas orientados por objetos, Beyer et al. [6] apontam que a herança deve ser considerada na avaliação de métricas referentes a tamanho, acoplamento e coesão em sistemas orientados por objetos. Ao avaliar uma classe, devem ser considerados não só as características definidas nela, mas também aquelas que são herdadas de seus ascendentes.

Conclui-se, então, que a profundidade da árvore de herança impacta na conectividade de um sistema. Se a conectividade de um sistema está alta, as relações de herança devem ser avaliadas. O ponto a ser avaliado neste caso é a profundidade da árvore de herança, pois este é um fator que contribui diretamente para a quantidade de conexões existentes no sistema. Quanto mais profunda a árvore, maior o número de conexões envolvidas na hierarquia.

4.2 Conclusão

Neste capítulo descrevemos a importância da conectividade na estabilidade de um software. O ideal é que busquemos construir software no qual uma alteração em determinado módulo afete o menor número de outros módulos possível. Um software com esta característica tem alta estabilidade, e o custo de sua manutenção é consideravelmente amenizado.

o caminho principal para obtenção de software estável é construi-lo de forma a garantir que o seu grau de conectividade seja o menor possível. Para isso, as suas partes constituintes - seus módulos - devem manter baixo número de comunicação com os demais módulos do software.

Capítulo 5

Estruturação de Software em Camadas

O ponto fundamental para a criação de software com alto padrão de qualidade é a forma como ele é estruturado. Para que um software possa atender a aspectos como manutenibilidade, extensibilidade e reusabilidade, é necessário que a organização modular do programa seja bem definida. O processo de se organizar um programa em termos de seus módulos é chamada arquitetura. A arquitetura de um sistema define uma estrutura de relacionamento entre os módulos que compõem o sistema [19]. Pressman [44] define a arquitetura de um software como "a estrutura hierárquica dos componentes de programa (módulo), o modo pelo qual esses componentes interagem e a estrutura dos dados que são usados pelos componentes".

A questão é como produzir boas arquiteturas. A arquitetura de um software deve ser orientada a produzir módulos que favoreçam a facilidade de manuntenção e que sejam reutilizáveis por outros softwares. A orientação por objetos oferece uma gama de recursos para a criação de softwares modulares, de fácil manutenção, que favoreçam a reusabilidade. Contudo, é necessário fazer bom uso desses recursos para produzir softwares que realmente alcancem esses propósitos. Scott Ambler [4] apresenta uma Arquitetura Tipo-classe, também denominada Modelo de Camadas, como uma solução para a organização de software orientado por objetos. Esta estratégia é detalhadamente discutida a seguir.

5.1 Arquitetura Tipo-Classe ou Modelo de Camadas

A Arquitetura Tipo-classe ou Modelo de Camadas é uma forma de estruturação de software orientado por objetos que tem por objetivo aumentar a modularidade, a manutenibilidade, a extensibilidade e a portabilidade do sistema. Uma arquitetura de tipo-classe provê uma estratégia de como distribuir as funcionali-

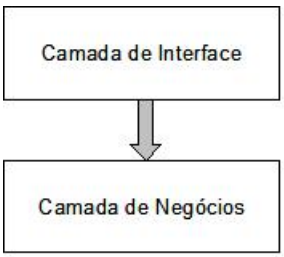


Figura 5.1 Modelo de Duas Camadas

dades de uma aplicação entre classes [4]. A organização das classes deve ser feita de tal forma a garantir o menor acoplamento possível entre elas, para aumentar a modularidade e tornar possível realizar alterações que causem pouco ou nenhum impacto nas demais classes.

Neste tipo de arquitetura, classes que têm propósito comum são agrupadas na mesma camada. Desta forma, uma camada encapsula as funcionalidades de um conjunto de classes que exibem comportamento similar.

Para traçar uma estratégia de organização de software em camadas, é preciso identificar o conjunto de funcionalidades a serem desempenhadas pelo software. Embora tais funcionalidades dependam do problema a ser resolvido, verificam-se funcionalidades que são comuns à maioria dos softwares. Ambler[4] apresenta quatro visões de arquiteturas tipo-classe: Modelo de Duas Camadas, Modelo de Três Camandas, Modelo de Quatro Camadas e Modelo de Cinco Camadas. Tais modelos são descritos a seguir:

5.1.1 Modelo de Duas Camandas

Em um software qualquer, inicialmente podem-se identificar dois tipos de funcionalidades: uma que representa a interface com o usuário e outra que representa as regras de negócios da aplicação. Desta forma, então, pode-se organizar um software em duas camadas: a Camada de Interface e a Camada de Negócio, como mostra a Figura 5.1.

A Camada de Negócio envolve as classes de negócio que modelam o domínio do problema, que é identificado durante o processo de análise. A Camada de Interface envolve as classes de interface que modelam a interação da aplicação com o mundo exterior, principalmente com os usuários. A questão chave na Arquitetura Tipo-Classe é a forma como se dá o fluxo de mensagens entre as classes. Nesta forma de estruturação de software, dentro de uma camada, as mensagens podem fluir livremente entre as classes. Porém, a comunicação entre classes de camadas diferentes devem seguir a diretiva de que as mensagens devem fluir da Camada de Interface para a Camada de Negócio, e o contrário não é permitido. Isto aumenta a modularidade, na medida em que diminui o grau de

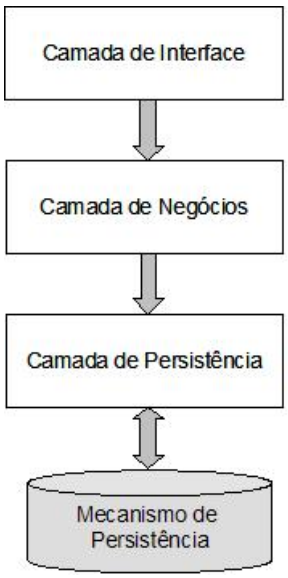


Figura 5.2 Modelo de Três Camadas

conectividade entre as classes de camadas diferentes. Embora este modelo de duas camadas forneça um nível razoável de modularidade, ele não é satisfatório, pois não aborda a maneira de armazenamento de dados persistentes. Uma classe que trata a manipulação de uma tabela de um banco de dados relacional, por exemplo, não parece adequada para ser tratada como classe de negócio, tampouco como uma classe de interface. O Modelo de Três Camadas, descrito na Seção reftresCamadas, visa solucionar esta questão.

5.1.2 Modelo de Três Camadas

O Modelo de Três Camadas adiciona outra camada de classes àquelas tratadas pelo Modelo de Duas Camadas. Trata-se da camada que agrupa as classes que têm como propósito o armazenamento de dados persistentes. O Modelo de Três Camadas, então, constitui-se pelas seguintes camadas: Camada de Interface, Camada de Negócio e Camada de Persistência, como mostra a Figura 5.2.

A Camada de Persistência envolve as classes de persistência, que têm a funcionalidade de tratar criação, recuperação, atualização e remoção de objetos persistentes. Não interessa aqui onde os objetos serão armazenados, se em um arquivo, se em um banco de dados ou em outro meio qualquer. É importante salientar que a Camada de Persistência provê o acesso ao mecanismo de persistência, mas não é o próprio mecanismo de persistência. A Camada de Persistência encapsula o armazenamento e a recuperação de objetos para que seja possível utilizar várias tecnologias sem impactos para as aplicações. Assim é possível realizar alterações no mecanismo de persistência de dados sem a necessidade de alteração

no restante do sistema. Quanto ao fluxo de mensagens, as classes da Camada de Persistência, tal como as das outras camadas, podem trocar mensagens entre si livremente. Entre as camadas, as mensagens devem fluir da Camada de Interface para a Camada de Negócio e da Camada de Negócio para a Camada de Persistência. Não deve existir troca de mensagens direta entre a Camada de Interface e a Camada de Persistência, pois tal prática introduz prejuízos para a modularidade e, por conseqüência, para a manutenibilidade do software. Isto porque quando se permite que a Camada de Interface se comunique diretamente com a Camada de Persistência, criam-se conexões entre as classes destas duas camadas. O Modelo de Três Camadas apresenta uma melhoria em relação Modelo de Duas Camadas, mas ainda não é suficiente para atender, de maneira altamente modular, às necessidades gerais da grande parte dos sistemas. Ela não apresenta o empacotamento de um conjunto de funcionalidades que atendam, por exemplo, a chamadas ao sistemas operacional, acesso a rede, etc. Um modelo mais abrangente, que considera essa questão, é apresentado na Seção 5.1.3.

5.1.3 Modelo de Quatro Camadas

Para suprir a deficiência do Modelo de Três Camadas, adiciona-se a ele um quarto conjunto de classes: a Camada de Sistema. As classes de sistema fornecem acesso aos recursos de sistema, tais como o próprio sistema operacional, e aos componentes de hardware, ou podem empacotar funcionalidades de outras aplicações. Nesta arquitetura, destaca-se o empacotamento, pois as classes de sistema encapsulam e oferecem funcionalidades que não são tipicamente orientadas por objetos, mas que precisam ser utilizadas pelas outras classes. Empacotando as funcionalidades de sistema, a aplicação torna-se mais portável entre ambientes diferentes, pois ao ocorrer mudança de ambiente, as alterações a serem realizadas ficam confinadas dentro da Camada de Sistema, diminuindo ou eliminando os impactos no software como um todo.

No Modelo de Quatro Camadas, o fluxo de mensagens entre as classes das camadas de Interface, de Negócio e de Persistência conserva-se igual à forma adotada no Modelo de Três Camadas. A Camada de Sistema é incluída no modelo e mantém comunicação como todas as demais camadas, como mostra a Figura 5.3. Observam-se características peculiares para cada a comuncicação da Camada de Sistema como cada uma das outras três camadas.

O fluxo de mensagens da Camada de Interface para a Camada de Sistema deve ser rara, para reduzir o acoplamento entre estas duas camadas e garantir a maior portabilidade possível. O fluxo de mensagens da Camada de Negócio para a Camada de Sistema é ocasional. Já o fluxo de mensagens da Camada de Persistência para a Camada de Sistema é muito freqüente, pois a primeira necessita, para a maioria de suas funções, solicitar serviços para a segunda, tais como funcionalidades de manipulação de arquivos e acesso à rede. Nota-se que, nesta arquitetura em particular, existe fluxo de mensagens bi-direcional entre a

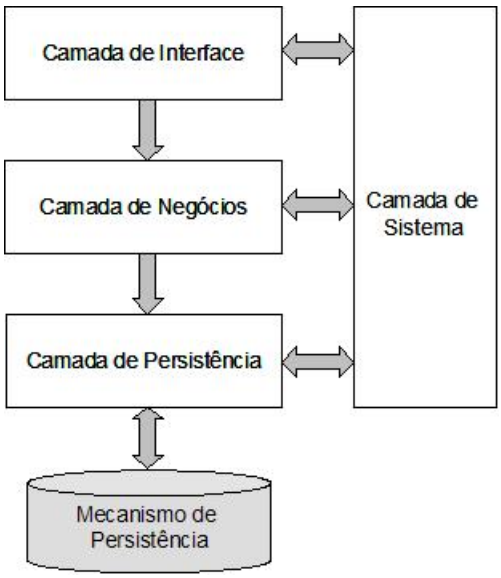


Figura 5.3 Modelo de Quatro Camadas

Camada de Sistema e as demais. Este fluxo é ocasional e ocorre, por exemplo, quando existe método de rechamada ou despachante de mensagem. Quando um método `A` estabelece comunicação com um outro método `B` e solicita que no futuro `B` envie uma mensagem `M` sempre que um certo evento ocorrer, o método `B` é dito método de rechamada. Um despachante de mensagem é um objeto que existe apenas para passar mensagens para outros objetos. Deve-se ter em mente que esta arquitetura não é uma solução para todas as aplicações. Adaptações podem ser feitas a ela de acordo com as necessidades identificadas na aplicação. No entanto, é um modelo de propósito geral que pode ser aplicado à maioria dos sistemas. Esta arquitetura apresenta alto grau de modularidade, pois favorece o mínimo de conectividade entre as classes de camadas distintas, o que impacta favoravelmente manutenibilidade e extensibilidade de software.

5.1.4 Modelo de Cinco Camadas

Scott Ambler [4] propõe adicionar ao Modelo de Quatro Camadas uma quinta camada: os usuários, posto que as pessoas que utilizam os sistemas são peças fundamentais no processo como um todo. A Figura 5.4 ilustra o Modelo de Cinco Camadas.

O argumento para se incluir os usuários como uma camada particular na arquitetura é que esta inclusão denota que foram considerados aspectos referentes à usabilidade na estruturação do sistema.

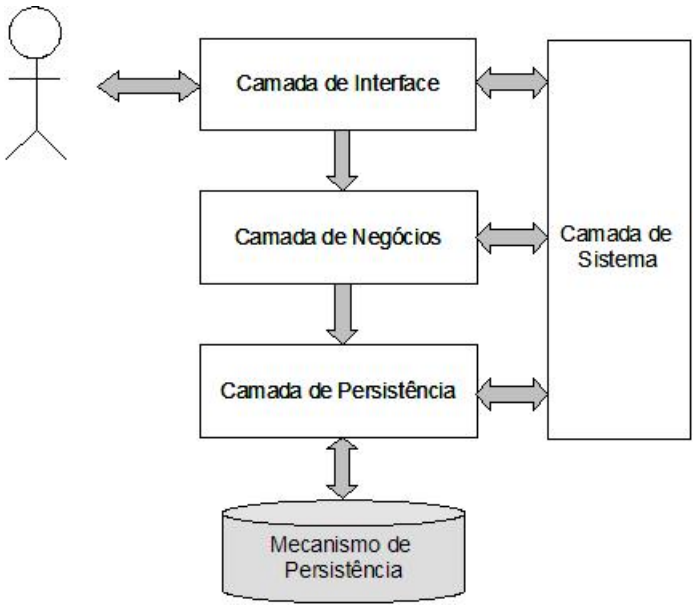


Figura 5.4 Modelo de Cinco Camadas

5.1.5 Identificação de Uma Camada Para Uma Classe

Cada camada na arquitetura tipo-classe reúne classes que possuem funcionalidades similares. Uma questão a ser considerada na estruturação de software em camadas é a definição de critérios a serem utilizados como diretrizes na identificação de uma camada para determinada classe. Ambler propõe os seguintes critérios para esse aspecto:

- Camada de Interface: devem ser incluídas nesta camada as classes que representam ou controlam uma janela, um formulário ou qualquer outra interface com usuário, as classe que representam um relatório e as classes que têm interação com o mundo exterior.

- Camada de Negócio: fazem parte desta camada, as classes que implementam um conceito do negócio da aplicação, foram descritas pelos usuários do sistema e não seria afetadas se fosses portadas para outro sistema.

- Camada de Persistência: as classes incluídas nesta camada devem prover a criação, a busca ou a atualização de dados em um mecanismo de persistência, como um arquivo ou banco de dados.

- Camada de Sistema: deve-se verificar se a classe a ser incluída nesta camada: esquematiza funcionalidade específica do sistema operacional; empacota acesso a outro sistema ou aplicação; pode ser desenvolvida sem in-

formações provenientes dos usuários do sistema; necessitará alteração caso o sistema operacional mude.

5.2 Benefícios do Modelo de Camadas

Em relação ao processo de software, o Modelo de Camadas possibilita a organização dos esforços de desenvolvimento baseados na arquitetura; ele permite que equipes de profissionais sejam alocadas para trabalhos nas diferentes camadas: um grupo pode trabalhar na Camada de Negócio, enquanto os demais são distribuídas entre as Camadas de Sistema, de Persistência e de Interface. Desta forma, elimina-se a necessidade de se ter uma camada completa, já implementada e disponibilizada, para trabalhar em outra. Os esforços de desenvolvimentos são diferenciados e característicos de cada camada. As classes da Camada de Sistema são conceitualmente muito diretas, pois empacotam características técnicas que geralmente são muito bem definidas. Assim, gasta-se, nesta camada, pouco esforço na fase de análise, algum esforço na fase de projeto e grande parte do esforço concentra-se na fase de codificação. A divisão de esforços na Camada de Persistência difere um pouco da Camada de Sistema. Há pouco esforço na fase de análise, a maior parte do esforço concentra-se no projeto e há alguma codificação. Isso porque, na fase de análise, basta listar as exigências técnicas para a camada. Nas classes da Camada de Negócio, grande parte dos esforços são gastos na fase de análise. Gasta-se esforço considerável na fase de projeto e algum esforço na codificação. A Camada de Interface é intimamente ligada aos usuários. Assim, grande esforço é dispensado na fase de análise, sendo que a maior parte trata-se de prototipação. Há algum esforço na fase de projeto e pouca esforço de codificação.

O Modelo de Camadas, também chamada Arquitetura Tipo-classe, provê uma estratégia de se construir software orientado a objetos com alto grau de modularidade, uma vez que consiste em distribuir as classes do sistema em camadas com um mínimo de conexões entre elas, restringindo-se dramaticamente o fluxo de mensagens entre estas camadas. Isso favorece, dentre outros aspectos, a manutenibilidade de software.

Capítulo 6

Métricas de Software

Avaliar é uma necessidade em vários segmentos da produção humana. A avaliação quantitativa fornece ao homem uma representação matemática do mundo real, possibilita-lhe aferir, comparar, acompanhar e saber quando é necessário interferir para melhorar, de maneira a garantir maior qualidade na sua produção. Essa necessidade faz-se presente também na produção de software e tem-se evidenciado com o aumento da demanda por sistemas cada vez mais complexos.

Alguns termos são comumente utilizados na literatura que trata sobre métricas de software, dentre eles: *medição (measurement), métrica (metric), medida (measure)*. Staa [55] define esses termos da seguinte maneira: *métrica* é um padrão de medição, a unidade de grandeza a ser medida; *medição* é o ato de medir algo de acordo com uma métrica; *medida* é o efeito, o resultado da medição.

No contexto da Engenharia de Software, métrica é um padrão de medição para avaliar determinado atributo de algo que esteja relacionado ao software. De acordo com Berard [7], métricas de software são utilizadas para avaliar produtos, processos e pessoas. Assim, métricas de software têm papel fundamental, pois a avaliação de tais elementos possibilita, dentre outros aspectos: a definição quantitativa do sucesso ou a falha de determinado atributo; a identificação da necessidade de melhorias do atributo avaliado; a tomada de decisão gerencial e técnica; a realização de estimativas. Para que uma métrica seja realmente útil é preciso ter em mente o que exatamente pretende-se medir, que tipo de informação tal medida fornecerá e a que papel ela se prestará, como a métrica será coletada e como será avaliada [7] [37].

Como Meyer avalia em [37], há uma extensa literatura sobre métricas de software [1, 2, 5, 6, 7, 12, 13, 17, 21, 22, 26, 27, 32, 37, 50]. O interesse pela área vem de algumas décadas. O primeiro livro sobre assunto [26], como o próprio autor destaca na obra, data de meados da década de 70. Um estudo realizado por Xenos et al. [62] mostra que o universo de métricas que têm sido propostas chega a algumas centenas. Como destaca Berard [7], neste universo, embora as métricas tradicionais sejam úteis no processo de desenvolvimento de software, elas não são suficientes, algumas inclusive são inadequadas, para se avaliar software OO devido

às características peculiares deste paradigma tais como herança, polimorfismo, ocultação de informação e encapsulamento. Assim, existe um grupo específico de métricas propostas para avaliar software orientado por objetos, dentre as quais destacam-se os conjuntos CK [13] e MOOD [2].

Neste capítulo são descritos os conjuntos de métricas CK e MOOD, e são relatadas ferramentas para coleta de métricas em softwares orientados por objetos.

6.1 Métricas de Software Orientado por Objetos

Dois conjuntos de métricas para orientação por objetos são amplamente citados na literatura: o conjunto de métricas proposto por Chidamber e Kemerer [13], conhecido como CK, e o proposto por Abreu e Carapuça [2], conhecido como MOOD. Dada a importância de tais conjuntos de métricas, a seguir eles são descritos em maiores detalhes.

6.1.1 Métricas CK

Chidamber e Kemerer [13] propõem um conjunto de métricas para projeto orientado por objetos, referenciadas na literatura como métricas CK. Nesse trabalho, os autores apresentam seis métricas, validam-nas usando os critérios de avaliação de métricas propostos por Weyukers [60] e relatam os resultados de experimentos realizados com as métricas propostas. Para a realização dos experimentos, foram utilizados dois sistemas, um escrito em C++ e outro em Smalltalk. O conjunto CK é constituído pelas seguintes métricas: Métodos Ponderados por Classe (WMC - *Weighted Methods per Class*), Profundidade de Árvore de Herança (DIT - *Depth of Inheritance Tree*), Número de Filhos (NOC - *Number of Children*), Acoplamento entre Classes de Objetos (CBO - *Coupling between Object Class*), Resposta de Classe (RFC - *Response for a Class*) e Ausência de Coesão em Métodos (LCOM - *Lack of Cohesion in Methods*).

- **WMC (Métodos Ponderados por Classe):** é uma métrica que representa a complexidade da classe por meio de seus métodos. O cálculo da métrica é dado pelo somatório das complexidades dos métodos que constituem a classe. Fica em aberto a definição para complexidade. Os autores não determinam um cálculo específico da complexidade dos métodos visando tornar a aplicação desta métrica mais geral. Segundo Chidamber e Kemerer, esta métrica é um indicador de custo de desenvolvimento e manutenção de uma classe, assim como do grau de reúso da classe. A quantidade de métodos de uma classe e a complexidade de tais métodos constituem um indicador do esforço de manutenção da classe. Além disso,

classe com um grande número de métodos têm potencial de reúso limitado, pois tendem a ter um uso específico da aplicação da qual fazem parte.

- **DIT (Profundidade de Árvore de Herança):** indica a posição de uma classe na árvore de herança de um software, que é dada pela distância máxima da classe até a raiz da árvore. Essa métrica é considerada um indicador da complexidade de desenho e de predição do comportamento de uma classe, visto que quanto maior a profundidade da classe na árvore de herança, mais classes, e portanto mais métodos e atributos, estarão envolvidos na análise.

- **NOC (Número de filhos):** indica a quantidade de sub-classes imediatas de uma classe. É um indicador da importância que uma classe tem no sistema, pois quanto mais sub-classes possuir uma classe, maior a importância de seu teste no sistema.

- **CBO (Acoplamento entre Classes de Objetos):** é um totalizador do número de classes às quais uma determinada classe está acoplada. Para Chidamber e Kemerer, o acoplamento entre duas classes existe quando métodos de uma delas usa métodos ou variáveis de instância da outra. A razão da existência desta métrica é justificada pelos autores pela necessidade de redução de acoplamento entre classes de objetos para atender fatores como melhoria de modularidade e aumento de reusabilidade.

- **RFC (Resposta de Classe):** apresenta o resultado do número de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto da classe. Este resultado é dado pela quantidade de métodos da classe somada à quantidade de métodos invocados por cada método da classe. Visto que RFC considera a ativação de métodos de outras classes, ela é, como CBO, um indicador de conectividade de uma classe. Enquanto CBO mostra a quantas outras classes uma classe está conectada, RFC é um detalhamento desta informação, pois apresenta por quantos caminhos uma classe está conectada a outras classes.

- **LCOM (Ausência de Coesão em Métodos):** é uma métrica da ausência de coesão entre os métodos de uma classe. Chidamber e Kemerer consi- deram que a coesão entre os métodos de uma classe é definida pela similaridade entre eles. A avaliação da similaridade entre dois métodos é determinada pelo uso de variáveis de instância da classe por eles. Seja P o conjunto formado pelos pares de métodos que não possuem variáveis de instância em comum e Q o conjunto formado pelos pares de métodos que possuem variáveis de instância em comum. Por definição, se nenhum método da classe utiliza variáveis de instância da classe, P = ϕ. O cálculo de LCOM é dado por:

$LCOM =\mid P \mid - \mid Q \mid$, se $\mid P \mid > \mid Q \mid$
$LCOM = 0$, caso contrário

Por exemplo, seja uma classe que possua dez pares de métodos sem variáveis de instância em comum e dois pares de métodos com variáveis de instância em comum. Assim, P = 10 e Q = 2. Então, LCOM = P - Q = 10 - 2 = 8.

Consideremos, agora, uma classe que possua dez pares de métodos sem variáveis de instância em comum e quatro pares de métodos com variáveis de instância em comum. Assim, P = 10 e Q = 4. Então, LCOM = P - Q = 10 - 4 = 6. Em comparação com a classe do exemplo anterior, essa classe apresenta melhor grau de coesão entre os seus métodos, o que é indicado pelos valores da métrica LCOM obtidos para ambas.

LCOM indica a diferença entre a quantidade de pares de métodos sem similaridade, isto é, pares de métodos que não possuem variáveis de instância em comum, e a quantidade de pares de métodos com similaridade. Baixos valores para essa métrica indicam bom nível de similaridade, portanto de coesão, entre os métodos da classe avaliada. Um valor alto para LCOM indica que a classe não provê uma funcionalidade bem específica.

Entretanto, vale destacar o seguinte aspecto desta métrica. Seja uma classe cujos métodos não fazem uso de variáveis de instância. Conforme a definição da métrica, P = ϕ. Assim, $\mid P \mid = 0$ e, como, neste caso, não há possibilidade de $\mid P \mid$ ser maior do que $\mid Q \mid$, LCOM = 0. Esse valor, porém, não deve ser entendido como coesão máxima na classe. Uma classe com esta característica pode ser, por exemplo, uma classe depositária de funções não relacionadas entre si, o que resulta em baixo grau de coesão da classe.

Outro aspecto ressaltado pelos próprios autores da métrica é que quando $\mid P \mid = \mid Q \mid$, LCOM é igual a zero. Entretanto, isso não implica grau máximo de coesão, uma vez que no universo de classes com esta característica, umas são mais coesas do que as outras.

6.1.2 Métricas MOOD

O conjunto de métricas MOOD (Metrics for Object Oriented Design) foi proposto por Abreu e Carapuça [2]. Nesse trabalho, além de serem definidas as métricas que compõem o conjunto MOOD, os autores apresentam sete critérios, a seguir, que os orientaram na definição de tais métricas.

- Definição formal de métricas: o significado de uma métrica deve ser formalmente definido. A definição formal de uma métrica elimina possíveis subjetividades, o que garante maior confiabilidade na avaliação de métricas de software.

- Independência de tamanho de software: métricas que não são relacionadas a tamanho de software devem ser independentes desse fator. O valor resultante de uma métrica deve proporcionar uma avaliação do software sem a necessidade de consideração do seu tamanho.

- Unidade de medida consistente: o resultado de uma métrica deve ser representado por uma unidade de medida que possibilite a sua análise de forma objetiva.

- Obtenção precoce: quanto mais cedo puder obter-se métricas no processo de software, melhor, pois problemas identificados nas fases iniciais custam menos do que aqueles identificados em fases tardias do processo de software. Cabe ressaltar que, embora seja um fato que problemas identificados em fases tardias do processo de software tenham custo muito elevado, com demonstra Boehm [9], isso não quer dizer que somente as métricas coletadas nas fases iniciais sejam importantes. Métricas são instrumentos úteis em todo o processo de software.

- Escalabilidade: uma métrica deve servir para avaliar tanto um sistema inteiro como parte dele.

- Facilidade de computação: a computação de uma métrica deve ser fácil para que a sua utilização seja viável.

- Independência de linguagem: uma métrica não deve ser definida em termos de uma linguagem específica.

As métricas MOOD avaliam os aspectos de herança, ocultação de informação, acoplamento, polimorfismo e reusabilidade em um software orientado por objetos. Compõem o conjunto MOOD as seguintes métricas: Fator Herança de Método (MIF - *Method Inheritance Factor*), Fator Herança de Atributo (AIF - *Attribute Inheritance Factor*), Fator Acoplamento (COF - *Coupling Factor*), Fator Agrupamento (CLF - *Clustering Factor*), Fator Polimorfismo (PF - *Polymorphism Factor*), Fator Ocultação de Método (MHF - *Method Hiding Factor*) , Fator Ocultação de Atributo (AHF - *Attribute Hiding Factor*), Fator Reúso (RF - *Reuse Factor*).

O cálculo de uma métrica MOOD é dado por uma razão, onde o numerador é o número de ocorrências encontradas no sistema para o aspecto avaliado e o denominador é o maior número possível de ocorrências no sistema para tal aspecto. Desta forma, o resultado de qualquer métrica MOOD é sempre um valor entre 0 e 1, o que representa o nível de ocorrência do aspecto avaliado no sistema. Esse tipo de resultado é apropriado porque fornece uma dimensão para a métrica independente do tamanho do sistema avaliado, o que torna possível comparar sistemas que possuam tamanhos e características distintos.

A seguir, cada métrica é apresentada e exemplificada. O diagrama de classes apresentado na Figura 6.1 será utilizado para exemplificar o cálculo de algumas dessas métricas. Ele foi construído utilizando-se notação UML [49] e mostra a estrutura simplificada de classes de objetos encontrados em um sistema acadêmico: pessoas, alunos, professores, disciplinas, matrícula e turmas. Neste diagrama, as classes são representadas por retângulos divididos em três compartimentos: o primeiro contém o nome da classe; o segundo, os atributos da classe e o terceiro contém a assinatura dos métodos no formato `nome do método (lista de parâmetros): tipo de retorno`. O símbolo "+" que precede o nome de um membro da classe indica que ele é público e o símbolo "-" indica que o membro é privado.

1. **Métricas Para Avaliação de Herança**

   Herança é o recurso da orientação por objetos que permite criar classes a partir de classes já existentes. Por meio da herança obtém-se a reutilização de estruturas que já estão definidas e possivelmente depuradas e testadas, o que é um ponto considerável na redução do custo da produção do software. Porém, outro aspecto deve ser observado em relação a herança, como apontam Chidamber e Kemerer [13]: árvores de herança muito profundas conferem maior complexidade ao software, o que é um fator negativo para a sua manutenção.

   Um indicador que permita a avaliação da herança em um sistema é, então, um instrumento útil para a predição do esforço de sua manutenção. As seguintes métricas para este aspecto fazem parte de MOOD: MIF (Fator Herança de Métodos) e AHF(Fator Herança de Atributos). Elas são descritas a seguir.

   - **MIF (Fator Herança de Método):** essa métrica indica o percentual de métodos herdados no sistema. Seja C_i uma classe do sistema a ser avaliado. Para a definição da métrica MIF, são considerados as seguintes métricas básicas:
     - Métodos herdados: $M_h(C_i)$. São os métodos que uma classe possui em decorrência de herança e que não foram redefinidos na classe.
     - Métodos novos: $M_n(C_i)$. São métodos criados na classe, que não foram herdados nem redefinidos.
     - Métodos redefinidos: $M_r(C_i)$. São métodos herdados que têm uma redefinição na classe.
     - Métodos definidos: $M_d(C_i)$. Englobam os métodos novos e os métodos redefinidos na classe.
     - Métodos disponíveis: $M_{dis}(C_i)$. É a totalidade de métodos que uma classe possui, o que engloba métodos definidos nela e os métodos herdados por ela.

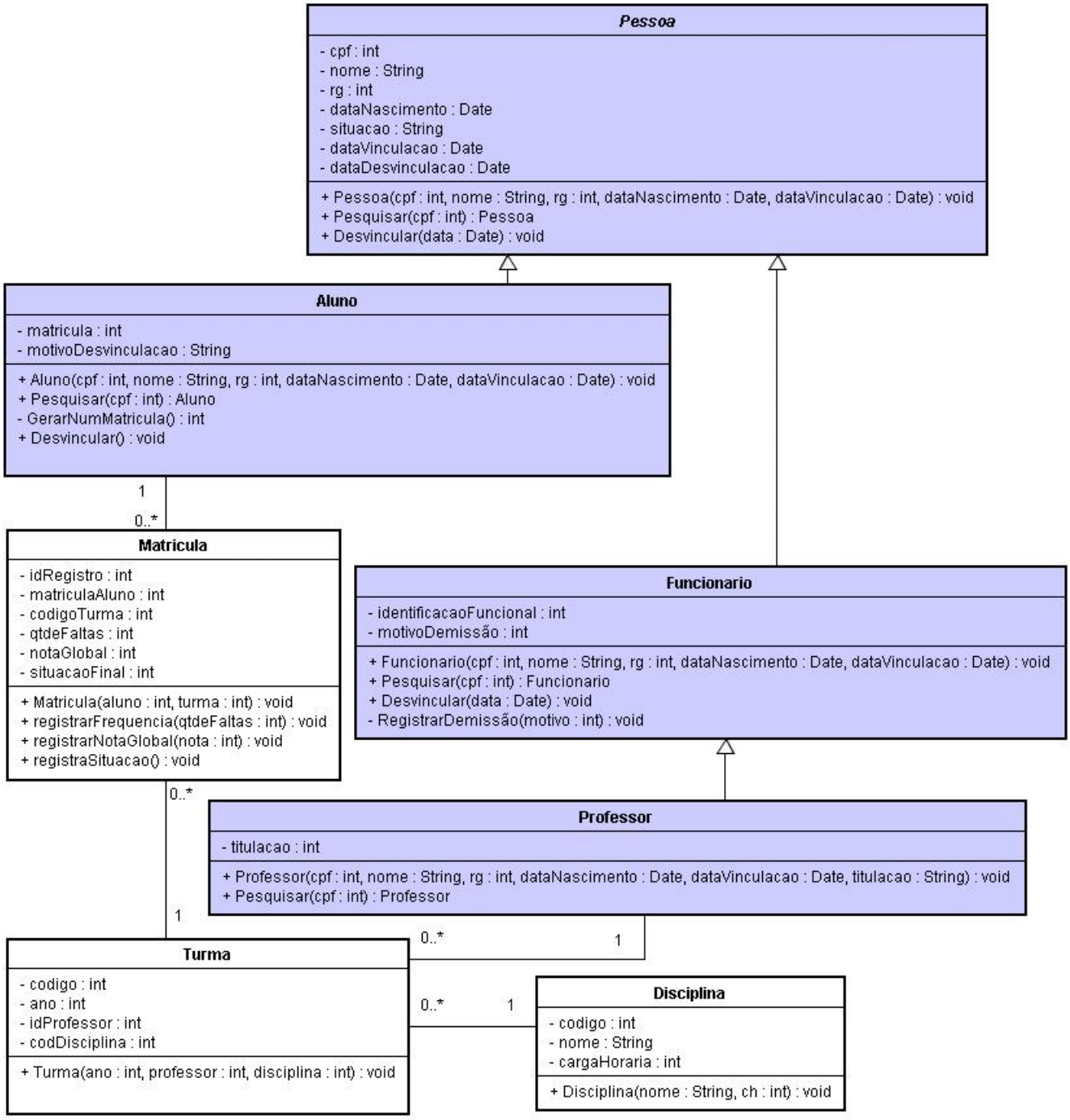


Figura 6.1 Exemplo de estrutura de classes em um sistema acadêmico

- Total de classes do sistema: TC.
- Total de métodos definidos no sistema: é o somatório do número de métodos definidos em cada classe do sistema, que englobam tanto os métodos novos das classes quanto os redefinidos nelas. Esse total é dado pela Equação 6.1.

$$TM_d = TM_n + TM_r = \sum_{k=1}^{TC} M_d(C_k) \tag{6.1}$$

- Total de métodos novos definidos no sistema: é o somatório do número de métodos novos de cada classe do sistema, dado pela Equação 6.2.

$$TM_n = \sum_{k=1}^{TC} M_n(C_k) \tag{6.2}$$

- Total de métodos novos redefinidos no sistema: é o somatório do número de métodos redefinidos em cada classe do sistema, dado pela Equação 6.3.

$$TM_r = \sum_{k=1}^{TC} M_r(C_k) \tag{6.3}$$

- Total de métodos herdados no sistema: é o somatório do número de métodos herdados em cada classe do sistema, dado pela Equação 6.4.

$$TM_h = \sum_{k=1}^{TC} M_h(C_k) \tag{6.4}$$

- Total de métodos disponíveis no sistema: é o somatório do número de métodos disponíveis em cada classe do sistema, dado pela Equação 6.5.

$$TM_{dis} = \sum_{k=1}^{TC} M_{dis}(C_k) \tag{6.5}$$

O cálculo de MIF é realizado da seguinte forma: para cada classe do sistema, verifica-se a quantidade de métodos herdados e a quantidade de métodos disponíveis. O valor de MIF é dado pela razão entre o somatório do número de métodos herdados de cada classe do sistema e o somatório do número de métodos disponíveis de cada classe do sistema. Assim, o cálculo de MIF é dado pela Equação 6.6

$$MIF = \frac{TM_h}{TM_{dis}} \tag{6.6}$$

A Tabela 6.1 detalha o cálculo de MIF para o sistema representado no diagrama de classes da Figura 6.1.

Classe	Métodos				
	Herdados	*Novos*	*Redefinidos*	*Definidos*	*Disponíveis*
Pessoa	0	3	0	3	3
Aluno	1	2	2	4	5
Funcionario	1	1	2	4	5
Professor	4	1	1	2	6
Matricula	0	4	0	4	4
Turma	0	1	0	1	1
Disciplina	0	1	0	1	1
Total do sistema	6	-	-	-	25
MIF = Total de métodos herdados / Total de métodos disponíveis = 6/25 = 0.24					

Tabela 6.1 Exemplo de cálculo da métrica MIF

MIF com valor igual a 0 indica que no sistema em questão não houve utilização efetiva do recurso de herança de métodos, o que significa que não existe relacionamento algum de herança entre as classes do sistema ou se existe, todos os métodos herdados foram redefinidos. Valores de MIF próximos de 1 indicam alta utilização do recurso de herança de métodos no sistema. Um valor igual a 1 para MIF indica que todos os métodos disponíveis em todas as classes do sistema são herdados. Esta situação parece estranha, mas sua ocorrência é possível, visto que uma métrica pode ser utilizada para avaliar um conjunto de classes particular de um sistema.

MIF indica, portanto, se o recurso de herança de métodos foi explorado amplamente no sistema, o que é um instrumento na predição no custo de manutenção do sistema. Quanto menor o valor de MIF, maior o custo de manutenção do sistema, pois valores baixos para esta métrica indicam que há pouca reutilização de métodos já definidos e possivelmente depurados e testados.

- **AIF (Fator Herança de Atributo):** indica o percentual de atributos herdados no sistema. Um raciocínio similar ao realizado no cálculo do fator herança de métodos é realizado para o fator herança de atributos AIF. O valor de AIF é dado pela razão entre o somatório do número de atributos herdados de cada classe do sistema e o somatório do número de atributos disponíveis de cada classe do sistema. Assim, o cálculo de AIF é dado pela Equação 6.7, onde TA_h é o total de atributos herdados no sistema e TA_{dis} é o total de atributos disponíveis

no sistema. O cálculo de TA_h e TA_{dis} são similares aos de TM_h e que são dados pelas Equações 6.4 e 6.5 respectivamente.

$$AIF = \frac{TA_h}{TA_{dis}} \tag{6.7}$$

A Tabela 6.2 detalha o cálculo de AIF para o sistema representado no diagrama de classes da Figura 6.1.

Classe	Atributos				
	Herdados	*Novos*	*Redefinidos*	*Definidos*	*Disponíveis*
Pessoa	0	7	0	7	7
Aluno	7	2	0	2	9
Funcionario	7	2	0	2	9
Professor	9	1	0	1	10
Matricula	0	6	0	6	6
Turma	0	4	0	4	4
Disciplina	0	3	0	3	3
Total do sistema	23	-	-	-	48
AIF = Total de atributos herdados / Total de atributos disponíveis = 23/48 = 0.48					

Tabela 6.2 Exemplo de cálculo da métrica AIF

As conclusões a cerca dos valores obtidos para esta métrica são similares às referentes à métrica MIF: valores próximos de 0 indicam pouca utilização do recurso de herança de atributos e o oposto, valores próximos de 1, indicam boa utilização de tal recurso. Entretanto, a importância da métrica AIF é menos relevante do que a da métrica MIF, pois, como apontam Abreu e Carapuça [2], o custo de manutenção dos métodos das classes que compõem o sistema tem peso muito maior no custo de manutenção do sistema do que os custos de manutenção decorrentes dos atributos das classes.

2. Métricas Para Avaliação de Ocultação de Informação

A ocultação de informação é um conceito importante relacionado à modularidade, pois a sua aplicação potencializa a independência de módulos. Quanto mais as informações e os serviços de uma classe estiverem confinados dentro dela, menor é a necessidade de as demais classes conhecerem sua organização interna e mais fraco é o nível de interdependência entre elas. Uma classe deve ser conhecida somente pelos serviços que ela disponibiliza. Na orientação por objetos, a ocultação de informação é obtida pelo uso de atributos e métodos privados nas classes.

Uma métrica de ocultação de informação em um sistema é um indicador que influencia a avaliação da modularidade do sistema porque reflete quão restritas estão as informações pertencentes aos módulos do software. As seguintes métricas para avaliação de ocultação de informação em sistemas orientados por objetos fazem parte de MOOD: MHF (Fator Ocultação de Métodos) e AHF (Fator Ocultação de Atributos). Elas são descritas a seguir.

- **MHF (Fator Ocultação de Método):** esta métrica representa o percentual de métodos ocultos no sistema. Para o seu cálculo, as seguintes métricas básicas são definidas, considerando-se C_i uma classe qualquer do sistema a ser avaliado.
  - Métodos visíveis: $M_v(C_i)$. São os métodos que constituem a interface da classe.
  - Métodos ocultos: $M_o(C_i)$. São os métodos privados da classe.
  - Métodos definidos: $M_d(C_i)$. São os métodos visíveis mais os métodos ocultos da classe. Essa métrica é dada pela Equação 6.8.

$$M_d(C_i) = M_v(C_i) + M_o(C_i) \tag{6.8}$$

  MHF é a razão entre o número de métodos ocultos em todas as classes e o número de métodos definidos em todas as classes, dado pela Equação 6.9

$$MHF = \frac{\sum_{k=1}^{TC} M_o(C_k)}{\sum_{k=1}^{TC} M_d(C_k)} \tag{6.9}$$

  A Tabela 6.3 detalha o cálculo de MHF para o sistema representado no diagrama de classes da Figura 6.1.

  Valores próximos de 1 para a métrica MHF indicam um alto nível de ocultação de métodos das classes do sistema. Esse tipo de resultado reflete que, de uma forma geral, as classes do sistema exportam poucos serviços, o que deve propiciar baixa conectividade entre as classes do sistema. O contrário ocorre quando se obtém valores próximos a 0 para essa métrica, o que indica que as classes do sistema exportam muitos serviços, portanto favorecem alto grau de conectividade entre as classes do sistema.

- **AHF (Fator Ocultação de Atributo):** essa métrica é o percentual de atributos ocultos no sistema. Similarmente a MHF, o cálculo de

Classe	Métodos		
	Visíveis	*Ocultos*	*Definidos*
Pessoa	3	0	3
Aluno	3	1	4
Funcionario	3	1	4
Professor	2	0	2
Matricula	4	0	4
Turma	1	0	1
Disciplina	1	0	1
Total do sistema	-	2	19
MHF = Total de métodos ocultos / Total de métodos definidos = 2/19 = 0.11			

Tabela 6.3 Exemplo de cálculo da métrica MHF

AHF é dado pela razão entre o número de atributos ocultos em todas as classes e o número de atributos definidos em todas as classes, conforme a Equação 6.10. Nesta equação, A_o corresponde ao número de atributos ocultos na classe e A_d, ao número de atributos disponíveis na classe; o cálculo de A_d é similar ao de M_d, que é dado pela Equação 6.8.

$$AHF = \frac{\sum_{k=1}^{TC} A_o(C_k)}{\sum_{k=1}^{TC} A_d(C_k)} \tag{6.10}$$

A Tabela 6.4 detalha o cálculo de AHF para o sistema representado no diagrama de classes da Figura 6.1.

A ocultação de atributos é característica de extrema importância para garantir a independência entre módulos, pois impossibilita a ocorrência dos tipos mais graves de acoplamento que podem existir entre duas classes. Quando uma classe torna público um atributo, outras classes do sistema podem alterar o valor desse dado e, então, perde-se a garantia da sua integridade e estabelece-se uma forte dependência entre todas as classes que fazem uso de tal atributo. Conhecer o grau de ocultação de informação de atributos de um sistema é saber quão propenso é o surgimento de acoplamentos desse tipo no sistema.

Valores de AHF próximos a 1 indicam que poucos atributos no sistema em questão são públicos. A situação ideal é que nenhum atributo seja público, o que resulta em AHF igual a 1. O pior caso é um valor igual a 0 para esta métrica, que indica que todos os atributos de todas as classes do sistema são públicos.

Classe	Atributos		
	Visíveis	*Ocultos*	*Definidos*
Pessoa	0	7	7
Aluno	0	2	2
Funcionario	0	2	2
Professor	0	1	1
Matricula	0	6	6
Turma	0	4	4
Disciplina	0	3	3
Total do sistema	0	25	25
AHF = Total de atributos ocultos / Total de atributos definidos = 25/25 = 1			

Tabela 6.4 Exemplo de cálculo da métrica AHF

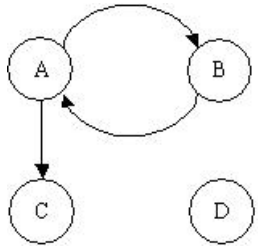


Figura 6.2 Exemplo de conexões em um sistema

3. Métricas Para Acoplamento

Métricas que possibilitam análise sobre os acoplamentos existentes em um sistema são úteis na predição do custo da manutenção do mesmo. MOOD contém as seguintes métricas para acoplamento: COF (Fator Acoplamento) e CLF(Fator Agrupamento). Elas são descritas a seguir.

- **COF (Fator Acoplamento):** para a avaliação de acoplamento, Abreu e Carapuça [2] consideram o conceito de relação *cliente-servidor* entre as classes constituintes de um software. Segundo esse conceito, uma classe `A` é cliente de uma classe servidora `B` quando `A` referencia pelo menos um membro de `B`, seja este membro um atributo ou um método. Uma relação cliente-servidor entre duas classes corresponde à existência de uma conexão entre elas. A Figura 6.2 é um exemplo de representação de conexões existentes entre classes de um software. Nesse exemplo, `A` é cliente de `B`, `B` é cliente de `A` e `A` é cliente de `C`.

  Em um software com n classes, o maior número possível de conexões é $n^2 - n$. A métrica COF é dada pela razão entre o número total de conexões existentes entre as classes do software e o maior número

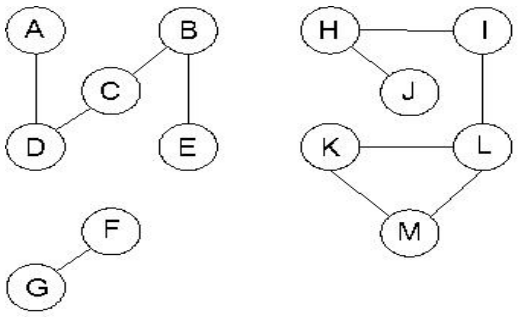


Figura 6.3 Agrupamentos de classes

possível de conexões para o software. Assim, o cálculo de COF para o software representado na Figura 6.2 é dado por $3/(4^2 - 4) = 0.25$. Um software totalmente conectado possui COF = 1.

COF é uma métrica importante, pois indica quão conectado é um software. Um software fortemente conectado possui estrutura rígida, baixo grau de independência entre os módulos e, conseqüentemente, o custo na sua manutenção é explosivo.

- **CLF (Fator Agrupamento):** esta métrica é um indicador do nível agrupamento de classes – *clustering* - em um software. Abreu e Carapuça consideram a representação de softwares por meio de um grafo para definir essa métrica, onde as classes são representadas pelos nodos do grafo e as relações entre as classes - relações cliente-servidor e de herança - são representadas pelas arestas. Os grafos disjuntos obtidos representam agrupamentos de classes (*clustering*). Um agrupamento de classes é, então, um conjunto de classes independente dos demais agrupamentos de classes do software. Esse grau de independência do agrupamento de classes propicia seu reúso. A Figura 6.3 mostra um exemplo simples de um software no qual se observam agrupamentos de classes.

  A métrica CLF representa a proporção entre a quantidade de agrupamentos de classes e o total de classes do software. No software da Figura 6.3, o cálculo para essa métrica é CLF = 3/13 = 0,23.

  O fator CLF é, então, um indicador de reusabilidade de agrupamentos de classes de um software. Um CLF baixo indica que há baixo potencial de reúso de agrupamentos de classes do software. O pior caso para essa métrica é quando não se observam agrupamentos de classes no software, o que resulta em valor 0 para CLF. O outro extremo se dá quando o número de agrupamentos é igual ao número de classes,

o que corresponde a um software constituído por classes totalmente independentes entre si, resultando em valor 1 para CLF.

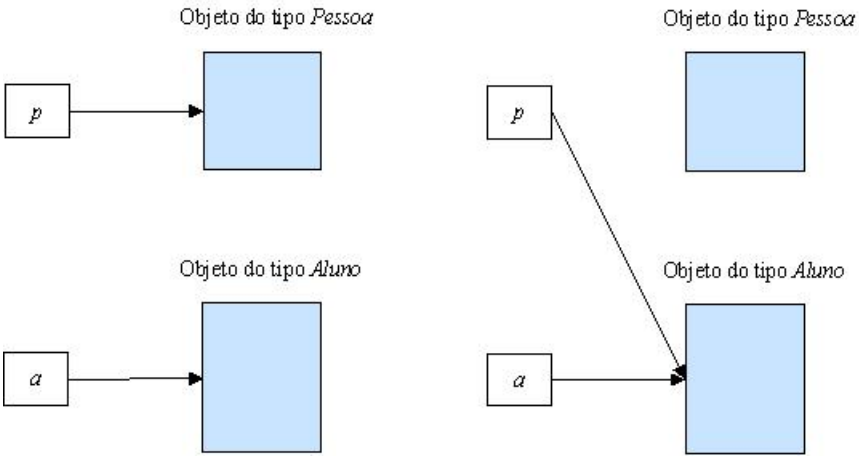


Figura 6.4 Exemplo de situação polimórfica

4. Métricas Para Polimorfismo

Na orientação por objetos, a herança é um dos recursos centrais de reusabilidade, pois possibilita a criação de novas classes a partir de classes já existentes. Em conjunto com a herança, a orientação por objetos conta com dois recursos principais para a construção de softwares extensíveis: a redefinição de características (métodos e atributos) e o polimorfismo.

Uma classe herdeira pode redefinir características herdadas de suas ascendentes. Por exemplo, no diagrama de classes da Figura 6.1, a classe *Funcionário* herda da classe *Pessoa* o método *Desvincular* e dá a esse método uma implementação diferente daquela definida na classe *Funcionário*. Assim, outras classes podem ser adicionadas à hierarquia de classes e, caso seja necessário, as características herdadas por elas podem ser redefinidas.

A redefinição de características aumenta a extensibilidade de um software. Mas, o grande ganho no uso de herança e de redefinição é conseqüência do polimorfismo. Como exemplo da ocorrência de uma situação polimórfica, consideremos a estrutura de classes da Figura 6.4. Seja p uma referência para o tipo *Pessoa* e a uma referência para o tipo *Aluno*. A atribuição $p = a$ é válida, pois a é um objeto da classe *Aluno*, que é herdeira da classe *Pessoa*. A Figura 6.4 esquematiza o que ocorre antes e depois desta atribuição polimórfica. Primeiro, p é criado e referencia um objeto do tipo *Pessoa*, e a é criado e referencia um objeto do tipo *Aluno*. Após a execução de $p = a$, p passa a referenciar um objeto do tipo *Aluno*. *Pessoa* é o tipo estático de p e *Aluno* é o seu tipo dinâmico.

Uma mensagem *Desvincular* enviada ao objeto p antes da atribuição polimórfica, resulta na execução conforme definição do método na classe *Pessoa*.

A mesma mensagem enviada a p após a atribuição polimórfica, resulta em uma execução conforme definição do método na classe *Aluno*. Assim, uma mesma mensagem pode assumir diferentes formas em tempo de execução, pois o método apropriado para processar a mensagem é definido de acordo com o objeto receptor da mensagem.

O polimorfismo, na orientação por objetos, confere grande flexibilidade e extensibilidade ao software. A possibilidade de se agregar novas funcionalidades a um software tem impacto no seu custo total. Assim, métricas de avaliação sobre polimorfismo em um software são importantes na avaliação de seu custo.

MOOD conta com a métrica para polimorfismo em software orientado por objetos, referenciada aqui como PF (Fator Polimorfismo), descrita a seguir.

- **PF (Fator Polimorfismo):** o cálculo de PF é baseado na redefinição de métodos, porque se não há redefinição de um método para processar determinada mensagem, então não há possibilidade de ocorrência de poliformismo no comportamento desta mensagem. PF é dada pela razão entre a quantidade de métodos redefinidos no sistema e o total de possibilidades de redefinição de métodos. O cálculo desses dois itens é realizado das seguintes formas:

  (a) *Total de possibilidades de redefinição de métodos* : é dado pelo somatório dos produtos entre *quantidade de métodos definidos na classe* e *total de descendentes da classe* de cada classe do sistema. Esse valor corresponde ao que os autores de MOOD chamam de *número máximo de situações polimórficas diferentes possíveis.*

  (b) *Quantidade de métodos redefinidos no sistema*: para cada classe, com- puta-se o somatório da quantidade de métodos redefinidos em suas descendentes. Somam-se, então, os valores obtidos para cada classe. Esse valor corresponde ao que os autores de MOOD chamam de *número de situações polimórficas diferentes possíveis.*

  A Tabela 6.5 detalha o cálculo de PF para o sistema representado no diagrama de classes da Figura 6.1. A métrica PF é um indicador do potencial de emprego de polimorfismo no software. O valor 0 para esta métrica indica que nenhum método foi redefinido e, conseqüentemente, não há possibilidade de comportamento polimórfico no envio de mensagens no software. O valor 1 para esta métrica indica que uma mensagem enviada a um objeto de uma classe c que possua n descendentes pode se comportar de n formas distintas daquela definida na classe c, isto é, a possibilidade de ocorrência de comportamento polimórfico no processamento de uma mensagem é máxima.

Classe	*Métodos definidos*	*Número de descentes*	*Redefinições de métodos possíveis*
Pessoa	3	3	9
Aluno	4	0	0
Funcionario	4	1	4
Professor	2	0	0
Matricula	4	0	0
Turma	1	0	0
Disciplina	1	0	0
Total de possibilidades de redefinição de métodos			13
Classe	*Descendente*	*Qtde. de métodos redefinidos no descendente*	*Total métodos redefinidos nos descendentes*
Pessoa	Aluno Funcionário Professor	2 2 1	5
Aluno	-	-	0
Funcionário	Professor	1	1
Professor	-	-	0
Matricula	-	-	0
Turma	-	-	0
Disciplina	-	-	0
Quantidade de métodos redefinidos no sistema			6
PF = Quantidade de métodos redefinidos no sistema / Total de possibilidades de redefinição de métodos = 6 / 13 = 0.46			

Tabela 6.5 Exemplo de cálculo da métrica PF

5. Métricas para Reusabilidade

- **RF (Fator Reúso):** o projeto de software direcionado a maximizar reusabilidade é essencial na redução de custo de produção de softwares. MOOD contém uma métrica para avaliar o reúso em software orientado por objetos. No cálculo desta métrica, duas formas de reúso são consideradas: o reúso de classes já existentes em uma biblioteca de classes e o reúso por meio de herança de métodos. O fator herança de atributos não entra no cálculo desta métrica porque os seus autores consideram que a herança de métodos impacta em maior custo de construção e manutenção do que a herança de atributos.

  O cálculo de RF para um software é dado pela soma de dois fatores:

  (a) *Reúso de classes na biblioteca*: razão entre o número de classes

reutilizadas de bibliotecas de classes e o número total de classes do software.

(b) *Reúso por herança de métodos*: calcula-se o produto dado por MIF e a quantidade de classes do software que não estão na biblioteca. O *Reúso por herança de métodos* é dado pela razão entre esse produto e o número total de classes do software.

A Tabela 6.6 detalha o cálculo de RF para o sistema representado no diagrama de classes da Figura 6.1. Como RF é um indicador da reusabilidade aplicada no software em questão e não uma avaliação do potencial de reusabilidade do mesmo, ele é uma variável a ser considerada na avaliação da facilidade de teste e do custo de manutenção do software.

Reúso de classes na biblioteca = Número de classes reutilizadas de bibliotecas / Número total de classes do software = 0 / 7 = 0
Reúso por herança de métodos = (MIF x Quantidade de classes que não estão na biblioteca) / Número total de classes do software = (0.24 x 7) / 7 = 0.24
RF = Reúso de classes na biblioteca + Reúso por herança de métodos = 0 + 0.24 = 0.24

Tabela 6.6 Exemplo de cálculo da métrica RF

6.2 Ferramentas para Coleta de Métricas de Software

A coleta de métricas em software é uma tarefa que demanda ferramentas. Iniciativas neste segmento tiveram início marcante no ínicio dos anos 90 [1] [23]. Existem hoje no mercado uma série de ferramentas para coleta automática de métricas [33]. Buscamos identificar aquelas que destinam-se a coletar métricas de software orientados por objetos.

Understand [28] é um conjunto de ferramentas comerciais que coletam métricas em softwares escritos em Ada, Delphi, Fortran, Java e C++. Entre as métricas coletadas pela ferramenta estão: número de linhas, número de linhas de código, número de linhas de comentários; para um sistema, coleta métricas como número de classes, número de linhas de código, número de linhas de comando e número de linhas de declarações. Embora esta ferramenta analise códigos escritos em linguagens orientadas por objetos como C++ e Java, ela não coleta métricas relevantes para a avaliação de software neste paradigma.

Krakatau Essencial Metrics [30] é uma ferramenta que coleta métricas em programas escritos em Java e C/C++. A sua principal funcionalidade é prover

meios de comparações de versões de software. Baseia-se na coleta das seguintes métricas: linhas de código alteradas, adicionadas e excluídas. Apesar de ser uma ferramenta que vise a análise de programas escritos em linguagens orientadas por objetos, não provê métricas específicas a esse paradigma.

A ferramenta *ObjectDetail* [40] coleta métricas específicas do paradigma orientado por objetos em softwares desenvolvidos em C++. Dentre as métricas coletadas por esta ferramenta, destacam-se: percentual de atributos públicos, percentual de atributos privados, percentual de métodos públicos, percentual de métodos privados, profundidade da árvore de herança e acoplamento de classe. Esta métrica é dada pelo número de classes que uma classe particular usa. De acordo com [40], um valor alto para essa métrica indica dificuldade de manutenção da classe em questão.

MOODKIT [3] é uma ferramenta desenvolvida pelo grupo de estudos relacionado à proposta do conjunto de métricas MOOD. Esta ferramenta coleta as métricas MOOD em softwares escritos em linguagens como C++, Eiffel e Java. A principal característica desta ferramenta é o uso de uma espécie de linguagem intermediária denominada GOODLY, proposta pelo mesmo grupo. A idéia basica é converter o código fonte a ser analisado em um código equivalente GOODLY, sobre o qual ocorre a coleta das métricas.

Together [57] é uma ferramenta que possui, dentre outros recursos, a coleta de métricas em códigos escritos em Java, de forma integrada à ferramenta de desenvolvimento.

Q.Metrics [46] extrai métricas de códigos escritos em C# e ASP.NET. A ferramenta é integrada ao ambiente de desenvolvimento e permite que o usuário selecione as métricas que deseja coletar. Os resultados podem ser visualizados em HTML ou PDF. A versão desta ferramenta para análise de código em C# coleta cerca de 20 métricas, dentre elas, métricas de orientação por objetos como *acoplamento aferente* e *acoplamento eferente*. Estas duas métricas são propostas por Robert Martin [34], que as considera como indicadores da independência entre agrupamentos de classes. De acordo com Martin, *acoplamento aferente* é dado pelo número de classes externas ao agrupamento em questão que dependem das classes constituintes do agrupamento; *acoplamento eferente* é dado pelo número de classes constituintes do agrupamento que dependem de classes externas a ele.

Essas ferramentas possuem em comum o fato de que realizam análise em código de linguagens orientadas por objetos. Algumas são simples relatórios de métricas coletadas e outras possuem recursos gráficos de visualização.

Ferreira [20] apresenta a ferramenta *Connecta*, que realiza coleta de métricas em software implementados na linguagem Java. Esta ferramenta faz parte da pesquisa realizada por Ferreira, na qual foi proposto um Modelo de Avaliação de Conectividade de Sistemas Orientados por Objetos - MACSOO. Este modelo é um método que visa a diminuição de conectividade em software orientado por objetos e conta com a indicação de um conjunto de métricas para avaliar este fator. *Connecta* foi construída para permitir a aplicação de tal modelo em sistemas

desenvolvidos em Java.

6.3 Conclusão

Neste capítulo, realizamos um estudo sobre métricas de software orientado por objetos. O estudo deste assunto faz-se importante no contexto da Engenharia de Software porque métricas são utilizadas na avaliação de atributos de algo que esteja relacionado ao software, por exemplo, sua modularidade e o grau de coesão interna de seus módulos. Apresentamos, aqui, as principais métricas propostas na literatura para a avaliação de software orientado por objetos.

Os dois conjuntos de métricas estudados, CK e MOOD, são amplamente citados na literatura sobre métricas de software orientado por objetos. De acordo com Pressman [44], embora essas métricas sejam debatidas na literatura e alguns considerem que elas não possuem um grau de formalismo adequado, elas fornecem informações úteis para a avaliação de software.

O conjunto CK caracteriza-se por possuir métricas para classes. Já o conjunto MOOD fornece métricas para o sistema como um todo ou para um conjunto de classes. CK contém métricas para avaliar os fatores: complexidade de métodos de uma classe, profundidade da classe na sua árvore de herança, número de filhos de uma classe, quantidade de classes a qual determinada classe está acoplada, o conjunto resposta de uma classe e ausência de coesão em métodos de uma classe. MOOD conta com métricas para avaliar os seguintes fatores de um sistema: herança, acoplamento, agrupamento, polimorfismo, ocultação de informação e reúso. Desta forma, CK e MOOD mostram-se complementares, pois um fornece uma avaliação em um grão mais fino e o outro, métricas para avaliação geral do sistema. Além disso, um aborda aspectos que não são abordados no outro.

Para que o uso de métricas de software seja viável, é necessário o uso de um ferramenta. Elencamos algumas das ferramentas disponíveis no mercado e no meio acadêmico para realização desta tarefa em softwares orientados por objetos.

Capítulo 7

Conclusão

A atividade de manutenção de software é ponto crítico na Engenharia de Software, pois tem impacto em questões chave desta disciplina, como custo e prazo. Essa atividade comumente é difícil de ser realizada e é responsável pela maior parcela do custo total de um sistema. Desta forma, é imperativo investir em recursos que proporcionem alcançar softwares com maior nível de manuntenibilidade.

Obter software de fácil manutenção depende de investimento em qualidade da sua produção, sobretudo no que diz respeito à sua estrutura. Um software com bom nível de manutenibilidade é aquele que diante da necessidade de alteração, tem poucos módulos alterados. Esta característica, denominada *estabilidade*, depende da forma como o software é estruturado. A facilidade de manutenção de um software é determinada primordialmente em função do nível de independência entre os seus módulos. Softwares com essa característica podem ser obtidos quando são contruídos a partir de módulos fortemente coesos e com o menor grau de acoplamento entre si [38].

A conectividade, a medida de interconexões entre módulos de um software, fornece uma avaliação primária da qualidade de sua estrutura e, conseqüentemente, da facilidade de sua manutenção. A alta conectividade é sinal de que o software não foi construído de forma a manter a independência entre os seus módulos, o que compromete a sua qualidade e a sua manutenibilidade.

Tendo como objeto de estudo os sistemas orientados por objetos, este livro apresentou:

- os principais conceitos relacionados à qualidade de software, tais como modularidade, coesão, acoplamento, manutenibilidade, extensibilidade e reusabilidade;

- uma adaptação dos aspectos coesão e acoplamento à luz da Orientação por Objetos;

- a conectividade como fator preponderante na manutenibilidade de software;

- a descrição da estratégia de estruturação de sofware orientado por objetos em camadas;
- as principais métricas de software orientado por objetos propostas na literatura.

Bibliografia

[1] ABREU, Fernando Brito e. *As Métricas na Gestão de Projetos de Desenvolvimento de Sistemas de Informação*. In: Actas das Sextas Jornadas para a Qualidade de Software, APQ, Lisboa,Dezembro de 1992.

[2] ABREU, Fernando Brito e CARAPUÇA, Rogério. *Object-Oriented Software Engineering: Measuring and Controlling the Development Process.*. In: Proceedings of 4th Int. Conf. of Software Quality, McLean, VA, USA, 3-5 October 1994.

[3] ABREU, Fernando Brito e; OCHOA, Luis; GOULO, Miguel. *The GOODLY Design Language for MOOD Metrics Collection*. Portugal: ISEG/INESC. ECOOP Workshops, 1997.

[4] AMBLER, Scott W., *Análise e Projeto Orientado a Objeto*. Volume 2, IBPI Press, Livraria e Editora Infobook AS, 1988.

[5] BELLIN, David;Tyagi, Manish; Tyler, Maurice, *Object-Oriented Metrics: an Overview*. Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, p.4, Canada, October 31-November 03, 1994.

[6] BEYER, D.; Lewerentz, C.; Simon, F. *Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems*. In: Dumke/Abran: New Approaches in Software Measurement, LNCS 2006, Springer Publ., 2001, pp. 1-17.

[7] BERARD, Edward V. *Metrics for Object-Oriented Software Engineering*. Disponível em [http://www.toa.com/pub/moose.htm], em Setembro de 2004.

[8] BIGONHA, Mariza A. S.; Bigonha, Roberto S. *Programação Modular*. Belo Horizonte: Apostila, DCC-UFMG, 2001.

[9] BOEHM, Barry W. *Software Engineering Economics*. Estados Unidos: Prentice-Hall, 1981.

[10] BOOCH, Grady. *Object Solutions - Managing the object-oriented project*. Estados Unidos: Addison-Wesley , 1996. ISBN: 0805305947.

[11] CONSTANTINE, Larry L.; Lockwood, Lucy A. D. *Software for Use: A Practical Guide to The Models and Methods of Usage-Centered Design*. Addison-Wesley, Reading - MA, 1999.

[12] CHIDAMBER, Shyam. R. e Kemerer, Chris F. *Towards a metrics suite for object oriented design*. Conference on Object Oriented Programming Systems Languages and Applications, 1991, pp. 197-211.

[13] CHIDAMBER, Shyam R.; Kemerer, C.F. *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, 20(1994)6, pp. 476-493.

[14] CMM - *Capability Maturity Model*. Carnegie Mellon University, Software Engineering Institute. Disponível em http://www.sei.cmu.edu/cmm/. Último acesso em Maio de 2006.

[15] DALY, John; BROOKS, Andrew; MILLER, James; ROPER, Marc; WOOD, Murray. *An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software*. ISERN, Scotland, 1996.

[16] DEITEL, H. M.; DEITEL, P. J. *Java - Como Programar*. 3. Ed. Porto Alegre: Bookman, 2001. 1201 p. ISBN 0-13-012507-5.

[17] FENTON, Norman; NEIL, Martin. *Software Metrics: Roadmap*. In: Proceedings of the Conference on the Future of Software Engineering, Maio de 2000.

[18] FERREIRA, Aurélio Buarque de Holanda. *Miniaurélio Século XXI Escolar: O minidicionário de alíngua portuguesa*. Rio de Janeiro: Nova Fronteira, 2000.

[19] FERREIRA, Kecia Aline Marque; Bigonha, Mariza Andrade da Silva. *Estruturação de Sistemas Orientados por Objetos*. Monografia. Belo Horizonte: DCC/UFMG, 2001.

[20] FERREIRA, Kecia Aline Marques; Bigonha, Mariza Andrade da Silva; Bigonha, Roberto da Silva. *Avaliação de Conectividade em Sistemas Orientados por Objetos*. Dissertação de Mestrado. Belo Horizonte: DCC/UFMG, 2006.

[21] FRAKES, Willian; TERRY, Carol. *Software Reuse: Metrics and Models*. In: ACM Computing Surveys, Vol. 28, No. 2, Junho de 1996.

[22] FRANCA, Luiz Paulo Alves; Staa, Arndt von; Fonte II, Hamilton José Sales. *Um modelo de Classes para um Ambiente de Geração de Programas de Medição de Software Baseados na Web*. In: XIII SBES. Florianópolis, 1999.

[23] FRANCA, Luiz Paulo Alves; Staa, Arndt von; Lucena, Carlos José Pereira de. *Medição de Software para Pequenas Empresas: Uma Solução Baseada na Web*. In: Anais XII SBES. Rio de Janeiro: SBC, 1998. pp. 71-86.

[24] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLOSSIDES, John. *Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos*. Porto Alegre: Bookman, 2000. 364 p. ISBN 0-201-63361-2.

[25] GHEZZI, Carlo e JAZAYERI, Medhi. *Programing Language Concepts. Estados Unidos*. Ed John Wiley & Sons, 1998. ISBN: 0-471-10426-4

[26] GILB, Tom. *Software Metrics*. Estados Unidos: Winthrop Publishers, 1977. 282 p. ISBN 0-87626-855-6.

[27] HASSOUN, Youssef; JOHNSON, Roger; COINSELL, Steve. *A Dynamic Runtime Coupling Metric for Meta-Level Architetures*. In: Proceedings of Eighth European Conference on Software Maintenance and Reengineering, 2004.

[28] JavaCount . Disponível em http://csdl.ics.hawaii.edu/Tools/JavaCount/JavaCount.html. Último acesso em Maio de 2006.

[29] JUDE - *UML Modelling Tool*. Disponível em http://jude.change-vision.com/jude-web/index.html. Acesso em Maio de 2006.

[30] Krakatau Essencial Metrics. Disponível em http://www.powersoftware.com/. Último acesso em Maio de 2006.

[31] LINDHOLM, Tim e Yellin, Frank. *The Java Virtual Machine Specification*. Second Edition. Disponível em http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html. Acesso em Janeiro de 2005.

[32] MORRIS, Kenneth L. *Metrics for Object-Oriented Software Development Environments*. Master's Thesis, M.I.T. Sloan School of Management, 1989.

[33] Metrics Tools. Disponível em http://www.laatuk.com/tools/metric-tools.html. Último acesso em Maio de 2006.

[34] MARTIN,Robert; *OO Design Metrics - An analysis of dependencies*. Outubro de 1994.

[35] MARTIN, Robert. *Design Principles and Design Patterns*. Disponível em: www.objectmentor.com. Acessso em Agosto de 2004.

[36] MEYER, Bertrand. *Object-oriented software construction*. 2. Ed. Estados Unidos: Prentice Hall International Series in Computer Science, 1997. 1254 p. ISBN 0-13-629155-4.

[37] MEYER, Bertrand. *The role of object oriented metrics*. Disponível em http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer /metrics/page.html. Acesso em Agosto de 2004

[38] MYERS, Glenford J. *Reliable software through composite design*. Nova York: Petrocelli/Charter, 1975. 159 p. ISBN 0-88405-284-2.

[39] NIELSEN, Jakob. *Usability Engineering*. Academic Press. 1994.

[40] ObjectDetail. Disponível em http://www.obsoft.com/Product/ObjDet.html e http://www.obsoft.com/Product/DetailPaper.html. Último acesso em Maio de 2006.

[41] PARNAS, D. L. *On the criteria to be used in decomposing systems into modules* Communications of ACM 15, 12, Dezembro, 1972. pp. 1053-1058.

[42] PAULA FILHO, Wilson de Pádua. *Engenharia de Software - Fundamentos, Métodos e Padrões*. Rio de Janeiro: LTC, 2001. 584 p. ISBN 85-216-1260-5.

[43] PFLEEGER, Shari Lawrence. *Software engineering theory and practice*. Upper Saddle River: Prentice-Hall, 1998. 576p. ISBN 013624842X

[44] PRESSMAN, Roger S. *Engenharia de Software*. Rio de Janeiro: MacGraw Hill, 2002. 843 p. ISBN 85-86804-25-8.

[45] PURAO, Sandeep; Vaishnavi, Vijay. *Produt Metrics for Object-Oriented Systems*. ACM Computing Surveys, Vol. 35, June 2003, pp. 191-221

[46] QUALITI METRICS. Disponível em: http://www.qualiti.com.br/home.html. Acesso em Janeiro de 2005

[47] QUASAR - Quantitative Approaches on Software Engineering And Reengineering. Disponpivel em http://www-ctp.di.fct.unl.pt/QUASAR/index.html. Acesso em Maio de 2006.

[48] ROCHE, J.M. *Software Metrics and Measurement Principles*. In: Software Engineering Notes, ACM, Vol. 19, No. 1, January 1994, pp. 76-85.

[49] RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. *The unified modeling language reference manual*. Reading, Mass.: Addison-Wesley, c1999. 550 p. ISBN 020130998X

[50] SANT'ANNA, Cláudio Nogueira; Garcia, Alessandro Fabrício; Chaves, Christina Von Flach Garcia; Lucena, Carlos José Pereira; Staa, Arndt Von Staa. *On the Reuse and Maintenance of Aspect-Oriented Software: an Assessment Framework*. In: 17º SBES. Anais. Porto Alegre: SBC, 2003.

[51] SAN Diego State University. *Advanced Object-Oriented Design and Programming*. Disponível em http://www.eli.sdsu.edu/courses/spring98/cs635/notes/. Acesso em Fevereiro de 2005.

[52] SHILDT, Robert. *C - Completo e Total*. São Paulo: Mc GraW Hill, 1990.

[53] SOMMERVILLE, Ian. *Engenharia de software*. 6. ed. São Paulo: Addison Wesley, 2003. 592p. ISBN 8588639076.

[54] SHNEIDERMAN, Ben. *Designing the user interface: strategies for effective human-computer interaction.* Addison-Wesley, 2.ed., 1992.

[55] STAA, Arndt von. *Programação Modular - Desenvolvendo programas complexos de forma organizada e segura*. Rio de Janeiro: Editora Campus, 2000. 690p. ISBN: 85-352-0608-6

[56] TEGARDEN, D.P.; Sheetz, S.D.; Monarchi, D.E. *A Software Complexity Model of Object-Oriented Systems*. In: Journal of Decision Support Systems, 1994. p.241-262.

[57] Together. Disponível em http://www.borland.com.br/together. Último acesso em Fevereiro de 2006.

[58] UNIVERSITY of Ottawa. *Object Oriented Software Engineering*. Disponível em http://www.site.uottawa.ca:4321/oose/index.html. Acesso em Fevereiro de 2005.

[59] VAREJÃO, Flávio Miguel. *Linguagens de Programação - Conceitos e Técnicas*. Rio de Janeiro: Elsevier, 2004.

[60] WEYUKER, E. *Evaluating software complexity measures*. IEEE Transactions Software Engineering, Vol. 14, pp. 1357-1365, 1988.

[61] XAVIER, Carlos Magno Da S.; PORTILHO, Carla. *Projetando com Qualidade a Tecnologia em Sistemas de Informação*. Rio de Janeiro: LTC, 1995. 117 p. ISBN:85-216-1047-5.

[62] XENOS, M.; Stavrinoudis, D.; Zikouli, K.; Christodoulakis, D. *Object-Oriented Metrics - A Survey*. Proceedings of the FESMA 2000, Madrid, Spain, 2000.