RODRIGO PEREIRA DOS SANTOS

CRITÉRIOS DE MANUTENIBILIDADE PARA CONSTRUÇÃO E AVALIAÇÃO DE PRODUTOS DE *SOFTWARE* ORIENTADOS A ASPECTOS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

LAVRAS
MINAS GERAIS – BRASIL
2007

RODRIGO PEREIRA DOS SANTOS

CRITÉRIOS DE MANUTENIBILIDADE PARA CONSTRUÇÃO E AVALIAÇÃO DE PRODUTOS DE *SOFTWARE* ORIENTADOS A ASPECTOS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Área de Concentração:

Engenharia e Qualidade de *Software*

Orientador:

Heitor Augustus Xavier Costa

LAVRAS
MINAS GERAIS – BRASIL
2007

Ficha Catalográfica preparada pela Divisão de Processo Técnico da Biblioteca Central da UFLA

Santos, Rodrigo Pereira dos

Critérios de Manutenibilidade para Construção e Avaliação de Produtos de Software Orientados a Aspectos / Rodrigo Pereira dos Santos. Lavras – Minas Gerais, 2007. 92.

Monografia de Graduação – Universidade Federal de Lavras. Departamento de Ciência da Computação.

1. Qualidade de Software. 2. Manutenibilidade. 3. Orientação a Aspectos. I. SANTOS, R. P. II. Universidade Federal de Lavras. III. Critérios de Manutenibilidade para Construção e Avaliação de Produtos de Software Orientados a Aspectos.

RODRIGO PEREIRA DOS SANTOS

CRITÉRIOS DE MANUTENIBILIDADE PARA CONSTRUÇÃO E AVALIAÇÃO DE PRODUTOS DE *SOFTWARE* ORIENTADOS A ASPECTOS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Aprovada em 22 de Março de 2007.

MSc. Ana Rubélia Mendes de Lima Resende

Prof. MSc. Antônio Maria Pereira de Resende

Prof. Dr. Plínio de Sá Leitão Júnior

Prof. Dr. Heitor Augustus Xavier Costa
(Orientador)

LAVRAS
MINAS GERAIS – BRASIL

Viver novas emoções a cada dia, pensando que o infinito representa o centro da existência...
Dedico ao meu pai Cláudio, à minha vovó e madrinha Francisca e ao meu tio Aroldo, que acompanham e iluminam minhas decisões do céu...
Atribuo à minha mãe Zilma e irmã Ana Luiza, que me entenderam e me ensinaram a crescer,
A Eduardo e Marli, irmãos e amigos de laços eternos...

AGRADECIMENTOS

Agradeço,

A Deus e a Nossa Senhora.

A minha mãe Zilma, uma pessoa fundamental, que me ensinou como seguir e vencer. Amo muito você e serei eternamente grato por todo amor e dedicação.

Ao meu pai Cláudio (*in memorian*), que mostrou sua fortaleza, sagacidade e persistência. Amo você.

A minha irmã Ana Luiza, companheira de todas as horas, amiga e grande presença.

Aos parentes que estiveram presentes durante minha trajetória, especialmente a vovó Francisca (*in memorian*) e vovó Sofia, ao tio Aroldo (*in memorian*), a tia Hélia, a tia Maria do Carmo, que me deram forças e ensinaram o significado de família.

Aos meus amigos, em especial:

A Eduardo e Marli, pela cumplicidade e presença constante em vários momentos;

A Letícia, Álvaro, Wanderson, Ana Paula, Lílian, Mônica e Cristina, pelas risadas, festas e conversas descontraídas.

Aos meus alunos, muito importantes para meu desenvolvimento. Foram marcantes e, sem querer, companheiros e amigos, transformando momentos difíceis e penosos em ocasiões de ensinamento e aprendizagem contínuos.

Aos professores dos Departamentos de Ciência da Computação e de Ciências Exatas, que me acompanharam e incentivaram a pensar alto e a ultrapassar os limites do conhecimento.

Por fim, agradeço ao meu orientador, o professor Heitor Augustus Xavier Costa, que me orientou e ajudou em diversas atividades de pesquisa e na conclusão deste trabalho.

RESUMO

Critérios de Manutenibilidade para Construção e Avaliação de Produtos de *Software* Orientados a Aspectos

Tradicionalmente, a atividade de manutenção se inicia após a entrega do produto de *software* ao usuário, quando está em operação e surge um novo requisito a ser contemplado. Na verdade, a manutenção em produtos de *software* deve ser realizada para que esteja atual e prolongue a sua vida útil. Dessa forma, a característica de manutenibilidade, que é um requisito não funcional, deve ser incorporada ao produto de *software* desde o início de seu desenvolvimento. O objetivo deste trabalho é a definição de critérios de manutenibilidade para a construção e eventual verificação do Modelo de Implementação. Para isto, foram analisados produtos de *software* manuteníveis orientados a aspectos para detectar informações que os faziam ter esta característica. Reunindo essas informações, as convenções e características da linguagem de programação *AspectJ* e a experiência do autor na implementação de produtos de *software*, obtiveram-se subsídios que permitiram identificar o que é necessário à construção de produtos de *software* manuteníveis orientados a aspectos. Com isso, tais critérios foram estabelecidos e são utilizados nos artefatos de *software*, sendo construídos sob a luz da norma ISO/IEC 9126, que é usada para qualidade de produtos de *software*. Esta norma define manutenibilidade e as suas quatro subcaracterísticas analisabilidade, estabilidade, modificabilidade e testabilidade.

Palavras-chave: Qualidade de *Software*, Manutenibilidade, Orientação a Aspectos.

ABSTRACT

Criteria of Maintainability for Building and Evaluation of Aspect-Oriented Software

Traditionally, the maintenance phase of life cycle of software begins after the delivery of software to the user, when it is in operation and there is a new requirement to be met. Therefore, software maintenance is very important because it increases the lifetime of software. Thus, maintainability, a non-functional requirement, must be incorporated into the software since the beginning of its development. The objective of this research is to define criteria of maintainability to build and verify the Implementation Model to aspect-oriented software. With this in mind, some aspect-oriented software with easy maintenance were analyzed and maintainability information was acquired. Gathering this important information, some conventions and characteristics of the programming language AspectJ, and the author's experience in implementing software supplied subsidies to construct aspect-oriented software with maintainability. The ISO/IEC 9126 standard contributed to build these criteria for providing a definition of maintainability and its four sub characteristics: analyzability, stability, changeability, and testability.

Keywords: Software Quality, Maintainability, Aspect Orientation.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

A atividade de manutenção de produtos de *software* está assumindo um papel cada vez mais importante dentro do chamado ciclo de vida de produtos de *software*. Ela é caracterizada pela modificação de um produto de *software* entregue ao cliente, para corrigir eventuais erros, melhorar seu desempenho, ou qualquer outro atributo, ou ainda adaptar o produto a um ambiente modificado.

No início dos anos noventa, estatísticas revelaram que muitas organizações alocavam, no mínimo, 50% de seus recursos financeiros em manutenção de produtos de *software*. Na década atual, a manutenção de produtos de *software* tem chegado a 70% de seus custos, tornando-se um dos grandes desafios da Engenharia de *Software*. Isto evidencia a importância das pesquisas sobre o processo de manutenção, através das quais técnicas e métodos são criados ou metodologias existentes são adaptadas, para contemplar as atividades inerentes a este processo. Entretanto, ainda são escassos na literatura trabalhos que visam a documentar as principais áreas de problemas durante a atividade de manutenção.

Nesse contexto, devido à necessidade de corrigir erros e às mudanças constantes das necessidades dos usuários, que acarretam alterações na especificação dos requisitos dos produtos de *software* durante sua vida útil, as subcaracterísticas de manutenibilidade apresentadas na norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)], analisabilidade, estabilidade, modificabilidade e testabilidade, devem ser consideradas durante o seu processo de desenvolvimento.

Em decorrência da complexidade na criação e da implementação de produtos de *software* não-triviais, novas tecnologias são estudadas e aplicadas ao seu processo de desenvolvimento para facilitar essa tarefa. Isto é realizado desde os primórdios da programação, podendo-se observar a evolução destas tecnologias partindo da programação utilizando linguagens em nível de máquina e passando pelos paradigmas procedural, orientado a objetos e orientado a aspectos.

O paradigma de orientação a aspectos vem suprir a fragilidade encontrada na orientação a objetos, que é notada quando interesses distintos, requisitos funcionais e não funcionais, encontram-se entrelaçados e/ou espalhados em outras partes de códigos, tornando-os difíceis de compreender, desenvolver e manter. Segundo Sant'anna (2004),

tais interesses são denominados como multidimensionais (*crosscutting concerns*) porque atravessam várias dimensões (classes) do sistema.

O paradigma de orientação a aspectos encapsula interesses que afetam múltiplas classes; por isso, este paradigma permite aos programadores modularizar melhor o seu produto de *software*. Assim, o código, que antes se tornara confuso em decorrência do entrelaçamento de interesses distintos, torna-se mais claro por estar separado em um aspecto.

Atualmente, pesquisas no desenvolvimento de produtos de *software* utilizando a orientação a aspectos têm crescido na comunidade acadêmica na área de Engenharia de *Software* [Garcia *et al.* (2004)]. Assim sendo, tornar-se-á imprescindível a necessidade de realizar manutenção nestes produtos de *software*.

Este trabalho é relativo à manutenibilidade dos produtos de *software* orientados a aspectos da categoria de Sistemas de Informação e é direcionado à fase de implementação dentro do processo de desenvolvimento de produtos de *software*.

1.1. Motivação

Como motivação para a realização deste trabalho, é importante salientar as seguintes necessidades:

- redução do tempo de compreensão do produto de *software*, durante a manutenção. Como conseqüência, pode-se aumentar a eficiência da manutenção;
- diminuição dos custos de manutenção em função de menor tempo para a sua realização. Ao investir esforço extra em fazer um produto de *software* manutenível, é provável obter retorno significativo na redução do custo da manutenção;
- redução do nível de abstração do artefato de *software* utilizado para a programação e para a manutenção. A suposição sobre os elementos de modelagem pode ser reduzida quando a implementação é mais robusta e mais clara;
- menor equipe de manutenção. Isso pode ser alcançado quando a manutenção é feita com maior eficiência;
- aumento da força de trabalho disponível para direcionar esforços para a realização de novos projetos na organização;
- critérios que auxiliem a identificação das características relevantes para a manutenção no código-fonte de produtos de *software* orientados a aspectos, fornecendo um

entendimento mais amplo. Tais critérios devem mostrar informações importantes para o entendimento do produto de *software* e a localização das alterações a serem feitas na manutenção.

1.2. Objetivos

O objetivo deste trabalho consiste em definir critérios de manutenibilidade aplicáveis na avaliação das características de manutenibilidade do Modelo de Implementação de um produto de *software* orientado a aspectos.

Um produto de *software* corresponde à entidade de *software* disponível para liberação a um usuário [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)]. Entende-se por artefato de *software* todo documento gerado ao final de cada fase do ciclo de vida útil do produto de *software* e utilizado na fase seguinte.

Um critério de manutenibilidade caracteriza-se por uma exigência, que deve ser atendida pelos artefatos de *software* que compõem o Modelo de Implementação, para que os produtos de *software* orientados a aspectos possam ser manuteníveis, segundo a definição da característica de manutenibilidade, presente na norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)].

Enfim, este trabalho refere-se à manutenibilidade de produtos de *software* orientados a aspectos da categoria de Sistemas de Informação.

1.3. Metodologia de Desenvolvimento

1.3.1. Tipo da Pesquisa

Conforme Jung (2004) e Marconi; Lakatos (2003) e observando o método científico, tem-se que a presente pesquisa é de natureza tecnológica, com objetivos de caráter exploratório, utilizando procedimentos experimentais fundamentados em um trabalho de campo.

1.3.2. Procedimentos Metodológicos

O presente trabalho foi realizado no período de junho de 2006 a março de 2007, iniciando-se por um levantamento bibliográfico, na *internet* e em bibliotecas digitais e impressas, de artigos científicos relacionados ao tema.

A seguir, as melhores práticas de construção dos artefatos de *software* do Modelo de Implementação[1] foram estudadas, obtendo-se um produto de *software* orientado a aspectos manutenível. Além disso, estudou-se o paradigma de orientação a aspectos, de forma a verificar suas características, técnicas e contribuições para a Computação. Durante este processo, formou-se um núcleo de discussões em Engenharia de *Software*, coordenado pelo autor e seu orientador, a partir do qual dois alunos do curso de Ciência da Computação da UFLA começaram a acompanhar o trabalho. Um deles procurou contribuir com conceitos e técnicas de manutenção e o outro se envolveu com a programação orientada a aspectos.

A etapa seguinte consistiu em identificar os aspectos de manutenibilidade a serem incorporados nos artefatos de *software* do Modelo de Implementação, sob a luz da norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)], definindo-se assim os critérios de manutenibilidade. Um diagrama sintético desse processo pode ser visto na Figura 1.1. Os critérios de manutenibilidade foram elaborados segundo os conceitos de manutenibilidade presentes na norma ISO/IEC 9126 e nas técnicas de boa programação disponíveis na literatura para conduzir a construção do Modelo de Implementação. Tal modelo deve atender às exigências dos critérios para que produtos de *software* orientados a aspectos manuteníveis sejam construídos e atendam aos requisitos de manutenibilidade, mantendo-se um ciclo.

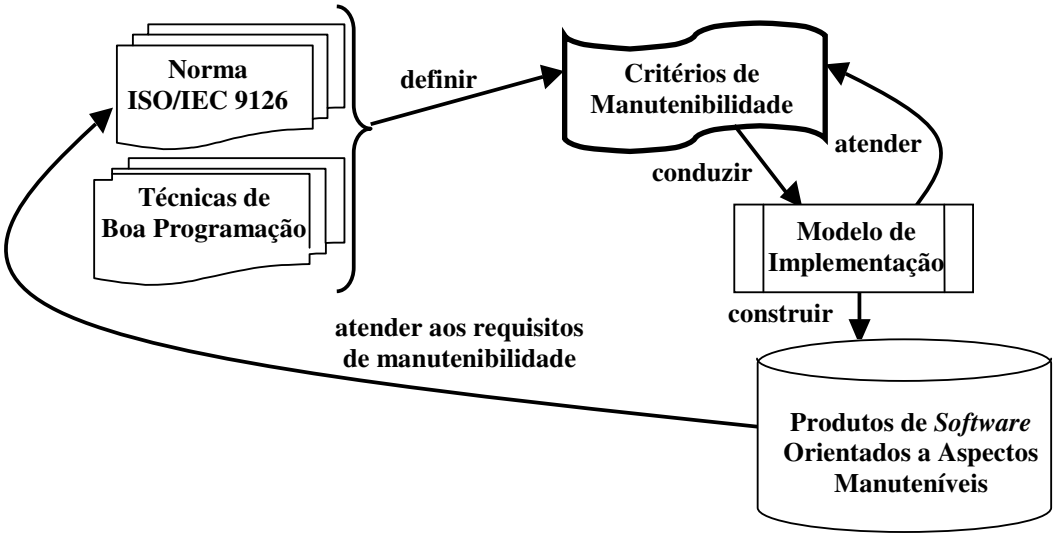

Figura 1.1 – Obtenção de produtos de *software* manuteníveis

[1] Modelo do *Software* é uma representação do *software* em vários níveis de abstração, dependendo do passo do processo de desenvolvimento (análise/projeto/implementação/teste – manutenção). Neste caso, o Modelo de Implementação está relacionado à codificação dos sistemas de informação em desenvolvimento [Filman *et al.* (2005)].

Reuniões semanais ocorreram entre o autor, o orientador e os dois alunos visando a manter a consistência e bom andamento do trabalho. Assim sendo, os critérios de manutenibilidade foram verificados quanto à sua importância para o Modelo de Implementação.

A partir da base teórica construída até então, realizou-se um estudo da linguagem *AspectJ*, a fim de entender melhor o paradigma de orientação a aspectos, identificar as características de manutenibilidade nos produtos de *software* e sedimentar o conhecimento desta linguagem.

A seguir, os resultados obtidos foram aplicados em um estudo de caso implementado nas linguagens de programação *Java* e *AspectJ*, possibilitando a verificação da confiabilidade dos critérios propostos.

1.3.3. Trabalhos Relacionados

Além dos pontos expostos até o momento, percebe-se que há poucas referências sobre a manutenibilidade de produtos de *software* [Costa (2005)]. A seguir, são apresentados alguns trabalhos relacionados investigados pelo autor.

Costa (2005) realizou um trabalho relativo à manutenibilidade de produtos de *software* orientados a objetos, utilizando a *Unified Modeling Language* (UML), como representação dos modelos dos artefatos de *software*, e o *Unified Process* (Processo Unificado), como processo de desenvolvimento. Seu objetivo foi a definição de critérios e a elaboração de diretrizes de manutenibilidade para a verificação e/ou adaptação do Modelo de Análise e para a construção e eventual verificação do Modelo de Projeto. Através da análise de produtos de *software* manuteníveis orientados a objetos, com o uso de dois padrões de projetos (*design patterns*) *Singleton* e *State*, das convenções e características das linguagens de programação *Eiffel*, *Java* e *Smalltalk* e de sua experiência, Costa (2005) identificou o que é necessário à construção de produtos de *software* manuteníveis orientados a objetos. Tais critérios e diretrizes foram construídos sob a luz da norma ISO/IEC 9126, sendo o resultado obtido através de testes empíricos considerado satisfatório por Costa (2005).

Sant'Anna (2004) apresentou alguns comentários sobre o paradigma de orientação a aspectos e relatou que, como este paradigma apresenta-se em sua fase inicial, não se sabe o quanto melhora a manutenibilidade e a reusabilidade dos produtos de *software*. Além

disso, listou alguns problemas que ocorrem durante a implementação e frisou que sem meios apropriados para sua separação e modularização, alguns interesses no código-fonte tendem a ficar espalhados e misturados a outros. As conseqüências naturais consistem na redução da facilidade de entendimento, no aumento da dificuldade de evolução e na diminuição da reusabilidade dos artefatos de *software*. Aspectos podem ser eficazes para modularizar interesses espalhados e/ou entrelaçados no código, minimizar código replicado e reduzir o tamanho do sistema. Contudo, o uso inapropriado de aspectos pode afetar negativamente esses atributos e aumentar a complexidade do produto de *software*. Dessa forma, Sant'Anna (2004) propôs um *framework* cujo objetivo principal é avaliar a manutenibilidade e a reusabilidade de produtos de *software* orientado a aspectos. O *framework* é composto por um conjunto de métricas e um modelo de qualidade. No sentido de aproveitar os benefícios de soluções bem testadas, os elementos do *framework* baseiam-se em princípios bem conhecidos da Engenharia de *Software* e em métricas existentes. As métricas são especializadas para aplicação no projeto e no código de produtos de *software* orientados a aspectos. Elas medem quatro atributos de qualidade: separação de *concerns*, acoplamento, coesão e tamanho. O modelo de qualidade relaciona a manutenibilidade e a reusabilidade às métricas e aos atributos medidos. Engenheiros de *software* podem usar o *framework* para comparar soluções orientadas a aspectos com soluções orientadas a objetos ou para avaliar decisões de projeto orientadas a aspectos. Em sua conclusão, Sant'Anna (2004) ressalta que atributos de qualidade importantes, como a manutenibilidade e a reusabilidade, podem ser afetados negativamente pelo uso inadequado de linguagens orientadas a aspectos e suas respectivas abstrações.

Lieberherr; Holland (1989) apontaram dois tipos de regras de estilo de programação: i) regras que limitam a estrutura de classes; e ii) regras que limitam a implementação de métodos. Em seguida, eles enunciaram a Lei de Demeter, na qual um objeto recebe uma mensagem de um método pertencente a um conjunto restrito de objetos. Tal conjunto restrito é composto por objetos: i) que são argumentos dos métodos; ii) ele próprio; e iii) os seus objetos componentes diretamente. Este artigo ressalta ainda que a Lei de Demeter tem o objetivo de organizar e diminuir as dependências comportamentais, troca de mensagens, entre os objetos. A melhoria de produtos de *software* orientados a objetos escritos sem estilo pode ser obtida através da utilização da Lei de Demeter, visto que ela não influencia o programador, no sentido de dizer o que deve ser solucionado, mas como deve ser solucionado. Neste artigo, os autores apresentaram uma notação estendida

para o sistema de Demeter e, posteriormente, definem a Lei de Demeter formalmente utilizando exemplos e examinam questões práticas e teóricas.

Medina (1984) apresentou alguns aspectos da documentação de produtos de *software*, ressaltando sua importância e seu propósito como facilitadora do entendimento e da usabilidade do código-fonte. Medina (1984) explicita dois tipos de documentação, a documentação intracódigo, como comentários, e o manual de usuário, argumentando que ambos devem apresentar informações essenciais para o entendimento do programa, operação do produto de *software* e interpretação dos resultados. A documentação interna deve ser um comentário em frente a cada procedimento, completamente inteligível perante as circunstâncias. Os tópicos comumente cobertos pelos comentários são: i) nome do procedimento; ii) nome do autor; iii) data de criação; iv) para que e como deve ser usado; v) que algoritmo e/ou estratégia é usada; vi) lista de argumentos; vii) qualquer restrição ou limitação durante sua execução; viii) quais os erros verificados; ix) subprogamas requeridos; x) um exemplo de uso; e xi) referências. As técnicas são preferíveis a produtos de *software* mais visualmente legíveis e os comentários aumentam a modularidade e a reusabilidade. O manual de usuário precisa ser cuidadosamente elaborado e devem conter: i) sumário; ii) introdução; iii) estratégias, que descrevem algoritmos e procedimentos usados; iv) descrição dos arquivos de entrada; v) instruções das operações; vi) descrição das saídas; vii) referências; e viii) apêndices. Como conclusão, o autor mostrou que o sucesso de um programa está baseado na sua usabilidade e que uma boa documentação e um bom programa estão relacionados. Ele descreveu os aspectos importantes para uma boa documentação e a organização interna dos programas, por exemplo, uso de comentários, identação dos comandos, padronização de nomes de variáveis e espaçamento para enfatizar a estrutura lógica, no sentido de auxiliar o entendimento do programa.

1.4. Estrutura do Trabalho

Este trabalho encontra-se organizado em 5 (cinco) capítulos, sendo os próximos descritos a seguir:

O Capítulo 2 discorre sobre qualidade, qualidade de *software* e manutenção em produtos de *software*, e sintetiza a norma ISO/IEC 9126, mostrando as seis características de qualidade para produtos de *software* e suas respectivas subcaracterísticas.

O Capítulo 3 apresenta a definição e os conceitos importantes do paradigma de orientação a aspectos, listando linguagens de programação disponíveis atualmente, além de seus benefícios e desafios.

O Capítulo 4 apresenta os critérios de manutenibilidade para a construção de produtos de *software* manuteníveis orientados a aspectos e um estudo de caso que ilustra a utilização e avalia os critérios de manutenibilidade.

O Capítulo 5 conclui o trabalho fazendo algumas considerações finais sobre a pesquisa realizada. Neste capítulo, além das contribuições, são avaliados os resultados alcançados e sugeridos trabalhos futuros.

2. MANUTENIBILIDADE DE *SOFTWARE*

2.1. Considerações Iniciais

Este capítulo apresenta a definição de qualidade e de qualidade de produtos de *software*, sob a visão dos autores mais relevantes da literatura. Além disso, apresenta uma descrição sucinta da norma ISO/IEC 9126 e uma discussão sintética de manutenção e de manutenibilidade de produtos de *software*, apontando os tipos de manutenção que podem ocorrer. Também são exibidos os fatores técnicos e não-técnicos que afetam a manutenção de produtos de *software*.

A seção 2 discorre sobre a etapa de manutenção dentro da Engenharia de *Software*, ressaltando a sua importância durante o processo de desenvolvimento de produtos de *software*. A seção 3 apresenta uma breve situação da manutenção nos últimos sessenta anos, de forma a listar suas causas, o cenário evolutivo do desenvolvimento de produtos de *software* que levou à ampliação desta necessidade, livros, conferências, organizações, periódicos, padrões e acontecimentos envolvidos. A seção 4 define qualidade e qualidade de produtos de *software*. A seção 5 sintetiza a norma ISO/IEC 9126, ressaltando a característica de manutenibilidade, foco deste trabalho. A seção 6 aborda o tema manutenibilidade e manutenção em produtos de *software*, exibindo seus conceitos básicos, tipos, problemas, desafios e práticas relacionadas.

2.2. Contextualização da Manutenção na Engenharia e Qualidade de *Software*

Apesar de muitos de autores terem desenvolvido definições pessoais de Engenharia de *Software*, uma definição proposta por Fritz Bauer *apud* Pressman (2006) em uma conferência pioneira sobre o assunto, ainda serve como base para discussão. Eles apontam que Engenharia de *Software* é a criação e a utilização de sólidos princípios de engenharia a fim de obter produtos de *software* de maneira mais econômica, que seja confiável e que trabalhe eficientemente em máquinas reais.

Entretanto, esta definição i) diz pouco sobre os aspectos técnicos da qualidade de produtos de *software*; ii) não inclui diretamente a necessidade de satisfação do cliente ou a pontualidade; iii) não faz menção à importância de medidas e unidades; e iv) não fala sobre

a importância de um processo amadurecido. Ainda assim, esta definição fornece uma linha básica.

IEEE (1993) desenvolveu uma definição mais abrangente quando estabelece que Engenharia de *Software* é: (1) aplicação de uma abordagem sistemática, disciplinada e quantificável para o desenvolvimento, operação e manutenção de produtos de *software*; isto é, a aplicação de engenharia ao produto de *software* e (2) o estudo de abordagens como as de (1).

De um modo geral, pode-se organizar o processo de desenvolvimento de um produto de *software* a partir de três grandes fases [Sommerville (2000), Pressman (2006), Pfleeger (2001) e Eisenmann *et al.* (2005)]:

- Fase de Definição. Esta fase está associada à determinação do **que** será feito. Assim, o profissional encarregado do desenvolvimento de produtos de *software* deve identificar as informações a serem manipuladas, as funções a serem processadas, o nível de desempenho desejado, as interfaces a serem oferecidas, as restrições do projeto e os critérios de validação. Isto é feito não importando o modelo de desenvolvimento adotado e independente da técnica utilizada para fazê-lo. Esta fase é caracterizada pela realização de três etapas específicas:
    - ⇒ análise (ou definição) do sistema. Permite determinar o papel de cada elemento (*hardware*, *software*, equipamentos, pessoas), cujo objetivo é determinar, como resultado principal, as funções atribuídas ao produto de *software*;
    - ⇒ planejamento do projeto de *software*. A partir da definição do escopo do produto de *software*, são feitas uma análise de riscos e a definição dos recursos, dos custos e da programação do processo de desenvolvimento;
    - ⇒ análise de requisitos. Permite determinar o conjunto das funções a serem realizadas e as principais estruturas de informação a serem processadas.
- Fase de Desenvolvimento. Nesta fase, é determinado **como** realizar as funções do produto de *software*. Nela, definem-se aspectos como a arquitetura do produto de *software*, as estruturas de dados, os procedimentos a serem implementados, a forma como o projeto será transformado utilizando linguagens de programação, a geração de código e os procedimentos de teste. Normalmente, esta fase é organizada em três principais etapas:

⇒ projeto de *software*. Traduz, em um conjunto de representações gráficas, tabulares ou textuais, os requisitos do produto de *software* definidos na fase anterior. Estas representações, que podem adotar diversas técnicas de representação em um mesmo projeto, permitem definir, com alto grau de abstração, aspectos do produto de *software* como a arquitetura, os dados, a lógica de comportamento (algoritmos) e as características da interface;

⇒ codificação. Faz o mapeamento das representações realizadas na etapa de projeto em uma ou várias linguagens de programação, a qual será caracterizada por um conjunto de instruções executáveis no computador. Nesta etapa, considera-se a geração de código obtido a partir do uso de ferramentas (compiladores, *linkers*, etc) que será executado pelo *hardware* do sistema;

⇒ testes de *software*. Realiza uma bateria de testes no programa obtido para verificar e corrigir defeitos relativos às funções, à lógica de execução, às interfaces, etc.

- Fase de Manutenção. Esta fase, que se inicia a partir da entrega do produto de *software*, é caracterizada pela realização de alterações das mais diversas naturezas, seja para corrigir erros residuais da fase anterior, para incluir novas funções exigidas pelo cliente, seja para adaptar o produto de *software* a novas configurações de *hardware*. Sendo assim, pode-se caracterizar esta fase pelas seguintes atividades:

  ⇒ correção. Consiste da atividade de correção de erros observados durante a operação do sistema;

  ⇒ adaptação. Realiza alterações no produto de *software* para que este possa ser executado sobre um novo ambiente (CPU, arquitetura, novos dispositivos de *hardware*, novo sistema operacional, etc);

  ⇒ melhoramento funcional. Realiza alterações para melhorar alguns aspectos do produto de *software*, como por exemplo, o seu desempenho, a sua interface, a introdução de novas funções, etc.

Por outro lado, a norma NBR ISO/IEC 12207 – Processos do Ciclo de Vida do *Software* – estabelece um ciclo de vida padrão composto pelos seguintes processos fundamentais: i) aquisição; ii) fornecimento; iii) desenvolvimento; iv) operação; e v) manutenção [NBR ISO/IEC 12207 (1998)]. Dessa forma, em quaisquer das duas

definições de etapas de processo de desenvolvimento de produtos de *software*, a fase de manutenção é ativada quando o produto de *software* é submetido a modificações no código-fonte e na documentação associada, devido a um problema ou à necessidade de melhoria ou adaptação. O objetivo é modificar o produto de *software* existente, preservando a sua integridade [Brusamolin (2004)].

Nesse contexto, Pigoski (1996) nota que o ciclo de vida do produto de *software* leva a uma interpretação equivocada do processo de manutenção, pois tem início após ele estar pronto e em produção. Não se concebe a participação do pessoal de manutenção nos processos iniciais do ciclo. O autor sugere, dessa forma, que o processo de manutenção inicie junto com o desenvolvimento. Assim, especialistas em manutenção poderiam atuar de forma a obter um produto de *software* com arquitetura propícia a reparos mais rápidos e baratos.

De posse destas informações, como a Engenharia de *Software* busca qualidade, existem vários modelos de qualidade sugeridos por alguns autores (modelo da *Hewlett Packard*, o modelo de *Boehm*, o modelo de *McCall* e o modelo de *Dromey*, a serem referenciados na seção 2.5). Além desses, a norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)] contém um modelo de qualidade que é vastamente adotado, apresentando seis características de qualidade e subcaracterísticas relacionadas. Entretanto, no contexto deste trabalho, a característica de manutenibilidade é a mais relevante, a qual relaciona-se com a terceira grande fase de desenvolvimento de produtos de *software* apresentada sinteticamente nesta seção.

Tais modelos de qualidade foram propostos devido à Crise do *Software*, porque os gastos demandados pela manutenção de produtos de *software* eram e continuam sendo bem maiores que os gastos necessários para o desenvolvimento de um novo.

Dessa forma, deve-se ter a constante preocupação com a manutenibilidade de produtos de *software* ao longo do seu processo de desenvolvimento, desde o início. Contudo, essa preocupação não existe por diversos fatores, ressaltando, principalmente, a urgência que as organizações que desenvolvem produtos de *software* querem ou precisam para colocá-lo disponível no mercado; assim, elas não correm o risco de perder clientes para os seus concorrentes.

2.3. Breve Situação da Manutenção de *Software*: História, Evolução e Lições Aprendidas

Desde que começaram a ser construídos produtos de *software*, existe a necessidade de realizar a tarefa de manutenção. Se analisadas as demais engenharias, mais antigas que a Engenharia de *Software*, percebe-se que, até então, não há fatores, na teoria ou na prática, indicando que esta realidade sofrerá alteração [Paduelli; Sanches (2006)].

De acordo com Zvegintzov; Parikh (2005), observações muito antigas de produtos de *software* pressentiram seu crescimento e sua mudança ininterruptos devido a desenvolvimento, encarecimento, correção, integração e redesenvolvimento. Desde o final da década de 40, estas dinâmicas têm permanecido em vigor e o produto de *software* continua crescendo sob a influência delas.

Se traçada uma linha de tempo do produto de *software*, visando a verificar as lições aprendidas, pode-se analisar a sua história e construir uma perspectiva de como e por que ele cresceu [Zvegintzov; Parikh (2005) e Paduelli; Sanches (2006)]:

- as funções do produto de *software* – aplicações, domínios e capacidades (suas funções estavam cada vez mais atreladas ao processamento em tempo real com o passar do tempo);
- as plataformas de produtos de *software* – dispositivos, redes e sistemas embarcados (rápida e constante evolução das plataformas);
- a tecnologia de produtos de *software* – linguagens, sistemas operacionais, ambientes de programação e tecnologia de apoio (a tecnologia evoluiu muito);
- o volume de produtos de *software* – por programa, por dispositivo, por sistema e por organização (o tamanho do produto de *software* aumentou);
- os profissionais de produtos de *software* – programadores, analistas, documentadores, testadores, pessoal de apoio, pessoal de operações e gerentes (a variedade de profissionais ampliou);
- o esforço do produto de *software* – horas, papéis, times e interações (observações de esforço eram cada vez mais constantes).

Se for examinado o por quê de produtos de *software* necessitarem de manutenção, alguns pontos serão observáveis: i) requisições de mudança funcional (encarecimento); ii) fracassos e erros (correção); iii) mudanças de tecnologia (adaptação); iv) ajuda e suporte ao usuário (apoio); e v) novos desenvolvimentos.

Segundo Zvegintzov; Parikh (2005), os pioneiros da manutenção de produtos de *software* são: Ned Chapin, Robert Glass, Carma McClure, J. Cris Miller, Girish Parikh, Tom Pigoski, Paul C. Tinnirello, Jean Dominique Warnier, Gerald M. Weinberg, Nicholas Zvegintzov, dentre outros. Alguns livros podem ser citados como referência na área:

- Jean-Dominique Warnier. **La Transformation des Programmes**, 1975 (primeiro livro de manutenção de produtos de *software*). **Program Modification**, 1978 (edição em inglês);
- Richard C. Linger, Harlan D. Mills, Bernard I. Witt. **Structured Programming: Theory and Practice**, 1979. Clássico em leitura, escrita e estabelecimento da função de programas estruturados;
- Girish Parikh, editor. **Techniques of Program and System Maintenance**, 1980, 1ª edição. 1ª edição reimpressa, 1982. Primeiro livro americano em manutenção de produtos de *software*. 2ª edição, revisada e ampliada, 1988;
- Bennet P. Lientz, E. Burton Swanson. **Software Maintenance Management – A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations**, 1980. Dados originais sobre práticas de manutenção e atitudes de gerência validados por muitos estudos mais recentes;
- Girish Parikh e Nicholas Zvegintzov, editores. **Tutorial on Software Maintenance**, 1982. Leituras e reimpressões, com contexto histórico e biografias;
- Nachum Dershowitz. **The Evolution of Programs**, 1983. Primeiro livro a enfrentar o problema central da manutenção de produtos de *software*: "o que significa modificar um produto de *software*?";
- James Martin, Carma L. McClure. **Software Maintenance: The Problem and its Solutions**, 1983. Um livro de co-autoria do "guru" da indústria, James Martin, o qual, com "manutenção" no título, soma legitimidade para o tema;
- M. M. Lehman, Laszlo A. Belady. **Program Evolution - Processes of Software Change**, 1985. Documentos colecionados pelos dois criadores das leis observáveis das dinâmicas de evolução do programa;
- Thomas M. Pigoski. **Practical Software Maintenance: Best Practices for Managing your Software Investment**, 1997. O apoio e encarecimento de produtos de *software* a partir do projeto e desenvolvimento inicial através da sua vida útil;
- Penny Grubb, Armstrong A. Takang. **Software Maintenance: Concepts and Practice**, 2003.

- Michael C. Feathers. **Working Effectively with Legacy Code**, 2005. Pouco a pouco, os livros começam a aparecer, dada a necessidade de se explorar o tema.

Algumas conferências como a *IEEE International Conference on Software Maintenance* (ICSM) e *Software Maintenance Association* (*SMA) Conference* acontecem em nível mundial. No Brasil, existe o *Workshop* de Manutenção de *Software* Moderna (WMSWM), realizado durante o Simpósio Brasileiro de Qualidade de *Software* (SBQS). Este *workshop* terá sua quarta edição em 2007 em Porto de Galinhas, Pernambuco, e demonstra a importância dada à manutenção na academia e na indústria, uma vez que ele conta com uma seção destinada a relatos de experiência. Neste contexto, a SMA consiste em uma organização que trabalha com este tema. Além disso, existem dois periódicos relacionados: *Software Maintenance News* e *Journal of Software Maintenance: Research and Practice*. Alguns padrões também existem para a área de manutenção: Military Handbook MIL-HDBK-347. **Mission-Critical Computer Resources Software Support** (1990) e IEEE Standard 1219-1993. **IEEE Standard for Software Maintenance** (1993).

Com relação às pesquisas em manutenção de produtos de *software*, elas realizam-se em universidades e divulgadas através de artigos no ICSM e em *workshops* em entendibilidade e modificabilidade de produtos de *software* e têm o apoio de vários projetos acadêmico-industriais multinacionais. Os tópicos envolvidos relacionam-se a: i) observação do produto de *software* atual e suas mudanças; ii) *frameworks* e arquiteturas para modificação e evolução; iii) estudos empíricos de métodos usados em análise e mudança; iv) base e condições prévias para modificação; e v) testes utilizados como base para mudança, como um componente de mudança e como validação de mudança.

Analisando as crises e as controvérsias na manutenção de produtos de *software*, sabe-se que a manutenção é freqüentemente não vista e não relatada, mas alguns assuntos e eventos se tornaram notórios: i) os desastres de produtos de *software* e como eles aconteceram; ii) o *bug* do ano 2000; iii) o teste e a exatidão provenientes de controvérsias; e iv) o crescimento do suporte e gerência de serviços. Ainda assim, anuncia-se e promete-se o fim da manutenção constantemente, com as novas tecnologias e os novos desenvolvimentos. Entretanto, eles sempre oferecem maiores problemas de complexidade e vulnerabilidade e maiores oportunidades para o encarecimento proveniente dos incrementos em tais tecnologias. Dessa forma, a manutenção representa o futuro dos produtos de *software* [Zvegintzov; Parikh (2005)].

2.4. Qualidade de Produtos de *Software*

Conforme Costa (2005), a literatura oferece várias definições de qualidade. A seguir, apresenta-se aquelas consideradas mais relevantes, seja pelo valor histórico, seja pela importância da referência bibliográfica.

Crosby (1979) diz que qualidade significa conformidade do produto com os seus requisitos, que devem estar bem definidos, a fim de não serem mal-interpretados. Deming (1982) diz que qualidade trata-se de um grau previsível de uniformidade e dependência, baixo custo e satisfação no mercado, ou seja, qualidade visa sempre aquilo que o cliente necessita e quer. Controle estatístico da qualidade, participação do trabalhador no processo de decisão e limitação das fontes de fornecimento são os passos para a sua abordagem [Côrtes; Chiossi (2001)].

Feigenbaum (1994) diz que qualidade representa uma estratégia que requer a percepção das pessoas na empresa. Além disso, ele diz que a qualidade compromete-se com a excelência, um modo de vida corporativa e um modo de gerenciamento. A liderança para a qualidade, a tecnologia moderna da qualidade e o compromisso organizacional se mostram como os passos da sua abordagem [Côrtes; Chiossi (2001)]. Por outro lado, a norma ISO Std. 8402 (1990) caracteriza o termo qualidade como a capacidade de satisfazer as necessidades implícitas e explícitas dos seus usuários.

Juran e Gryna Jr. (1970) dizem que qualidade significa conveniência para o uso, ou seja, os requisitos e as expectativas dos clientes devem ser considerados, uma vez que cada cliente pode usar o mesmo produto de maneiras diferentes. Rothery (1993) diz que qualidade significa adequação ao uso conforme as exigências, ou seja, o produto em questão deve realizar a sua função de maneira adequada.

Segundo Rocha (2002), uma organização com bom desempenho gasta 80% de seu esforço na prevenção de problemas, enquanto uma organização de baixo desempenho gasta 90% de seu tempo corrigindo sintomas em vez de causas de problemas. Dessa forma, sua definição de qualidade refere-se a um conjunto de características a serem satisfeitas em um determinado grau, de modo que o produto de *software* satisfaça às necessidades de seus usuários. Além disso, ela refina o conceito ao elucidar que a qualidade do processo[2] de

[2] Processo de *software* é um conjunto de atividades, métodos, práticas e tecnologias utilizadas para desenvolver e manter *software* e produtos relacionados [Rocha (2002)].

software e a do produto implicam no uso de um ambiente de Engenharia de *Software* de boa qualidade e adequado ao projeto [Rocha (2001); Rocha *et al.* (2001)]. Como observação, vale ressaltar que o Ministério da Ciência e Tecnologia (MCT) vem acompanhando a evolução da gestão da qualidade de *software*, direcionando as ações dos agentes responsáveis pela formulação e execução da Política de *Software* no Brasil, através de levantamento e análise de indicadores do mercado de produtos de *software* brasileiros [Nascimento (2004) e Nascimento (2005)].

Considerando o contexto dos produtos de *software*, o termo qualidade recebeu uma definição especial na norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)], na qual a qualidade é a totalidade das características de um produto de *software*, que lhe confere a capacidade de satisfazer às necessidades explícitas e implícitas. As necessidades explícitas são requisitos para o desenvolvimento de um produto de *software* que são explicitamente sugeridos pelo cliente ou obtidos na modelagem do problema. São, portanto, fatores relativos à qualidade do processo de desenvolvimento do produto e percebidos somente pelas pessoas que trabalharam no seu desenvolvimento. Por outro lado, as necessidades implícitas podem não ser explicitadas nem documentadas, mas devem estar presentes quando o produto de *software* é utilizado em condições particulares. São necessidades subjetivas dos usuários, inclusive operadores, destinatários dos resultados do produto de *software* e os mantenedores. Estas necessidades também chamadas de fatores externos, podem ser percebidas pelos desenvolvedores e usuários. Outra denominação utilizada trata da qualidade em uso e deve permitir a usuários atingir metas com efetividade, produtividade, segurança e satisfação em um contexto de uso especificado [Gomes (2001)].

Segundo Gomes (2001), em respeito às características e às necessidades desse usuário (cliente), algumas empresas desenvolvedoras de produtos de *software* introduziram modificações no desenvolvimento e no teste. Muitas delas colocam equipes para observar os usuários trabalharem em seu ambiente rotineiro. Outras trazem os usuários para seus laboratórios de teste, visando a melhorar a qualidade do produto de *software* antes de sua disponibilização para o mercado. Isso porque o impacto da usabilidade fica mais claro quando pessoas sem conhecimentos técnicos especiais e sem treinamento tentam usar o produto de *software*. A *Microsoft* inaugurou em 1989 o seu primeiro laboratório de usabilidade (*Usability Lab*), para que produtos de *software* ainda não liberados sejam

usados por usuários leigos e experientes, enquanto são observados por engenheiros de usabilidade que registram e analisam o que acontece. As informações coletadas são utilizadas por diversas áreas na empresa e têm permitido avanços significativos na melhoria da usabilidade. A *Microsoft*, antes de colocar seus produtos de *software* nas prateleiras dos revendedores, libera versões preliminares para grupo de usuários cadastrados testá-los em seus próprios computadores e reportar os resultados. Esta prática tem permitido à empresa diminuir custos, ampliar a equipe de teste, gerar conhecimento prévio do produto, realizar propaganda gratuita e garantir melhor qualidade e maiores lucros.

No entanto, os problemas relacionados com a Engenharia de *Software* e com o gerenciamento de qualidade decorrem da situação do termo qualidade possuir diferentes definições, conduzindo a mal-entendidos [Toledo (1987)]. Conforme Kan (1995), isto pode ocorrer devido aos seguintes fatores:

- o termo qualidade possui um único sentido, mas com conceitos multidimensionais – quem se interessa, seus pontos de vista e os atributos de qualidade relevantes para cada um deles;
- para cada conceito, há níveis de abstrações diferentes, ou seja, um grupo de pessoas pode se referir à qualidade com sentido geral e um outro grupo se referir com sentido específico;
- o termo qualidade está presente na linguagem diária das pessoas. Assim, a visão popular pode entrar em conflito com a visão profissional – visão da Engenharia de *Software* e visão do gerenciamento da qualidade. Na visão popular, o termo qualidade relaciona-se à classe e à elegância, enquanto, na visão profissional, o termo qualidade caracteriza funcionamento simples e barato.

No escopo deste trabalho, a definição de qualidade de produtos de *software* contida na norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)] foi utilizada como referência, pois apresenta caráter geral e altamente conceitual, o que a habilita a ser aplicada a uma grande variedade de ambientes e condições.

2.5. Conceituação e Objetivos da Norma ISO/IEC 9126

Conforme Gomes (2001), o principal problema com que se defronta a Engenharia de *Software* é a dificuldade de medir a qualidade de produtos de *software*. A qualidade de

um dispositivo mecânico é freqüentemente medida em termos de tempo médio entre suas falhas, que é uma medida da capacidade do dispositivo suportar desgaste. O produto de *software* não se desgasta, portanto tal método de medição de qualidade não pode ser aproveitado.

Dessa forma, foi publicada, em 1991, a norma ISO/IEC 9126 – Tecnologia da Informação – Características e Métricas de Qualidade de *Software* (*Information Technology – Software Quality Characteristics and Metrics*). O modelo proposto pela norma ISO/IEC 9126 tem por objetivo servir de referência básica na avaliação de produto de *software*. Além de ter força de norma internacional, ela cobre os aspectos mais importantes para qualquer produto de *software*. Ainda assim, existe uma versão traduzida pela Associação Brasileira de Normas Técnicas (ABNT) para o seu uso no Brasil [ABNT NBR 13596 (1996)], a qual foi publicada em abril de 1996, propiciando a difusão dos conceitos de qualidade de produtos de *software* e ampliando a terminologia específica e a própria cultura da área entre os profissionais. Essa norma define as características de qualidade de produtos de *software* e fornece uma metodologia genérica para a sua avaliação. São seis as características de qualidade de produtos de *software*: i) confiabilidade; ii) eficiência; iii) funcionalidade; iv) manutenibilidade; v) portabilidade; e vi) usabilidade.

De acordo com a norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)] *apud* Cordenonzi (2000), modelo de qualidade é o conjunto de características e seus relacionamentos, os quais fornecem a base para a especialização dos requisitos de qualidade e para a avaliação da qualidade. Existem, na literatura, vários modelos de qualidade, conforme Costa (2005): i) o modelo de *McCall* [Endo (1996), McCall *et al.* (1977) e Michilini (1997)]; ii) o modelo da *Hewlett Packard* (baseado no modelo de *McCall*) [Boaventura (2001)]; iii) o modelo de *Boehm* (semelhante ao modelo de *McCall*, uma vez que apresenta uma hierarquia de características e cada uma contribui para a qualidade total do produto de *software*) [Silva (1995)]; e iv) o modelo de *Dromey* (sugere uma técnica genérica para construir um modelo de qualidade) [Dromey (1996) e Pfleeger (2001)].

Classificando a qualidade, pode-se dizer que existem três tipos [Costa (2005)]:

- qualidade interna. Corresponde à totalidade de características do produto de *software* de uma visão interna. Detalhes de qualidade de produto de *software* podem ser

melhorados durante a implementação, a revisão e o teste do código, mas a natureza básica da qualidade de produto de *software* representada pela qualidade interna se mantém inalterada, a menos que este seja re-projetado (*redesigned*);

- qualidade externa. Corresponde à totalidade de características do produto de *software* de uma visão externa, obtida quando o produto de *software* é medido e avaliado em um ambiente com dados simulados, usando métricas externas. Durante o teste, a maioria das falhas deveria ser encontrada e eliminada, entretanto algumas ainda podem permanecer depois do teste. Devido à dificuldade de se corrigir a arquitetura de *software* ou outros aspectos de projeto (*design*) básicos do produto de *software*, o seu projeto básico geralmente se mantém inalterado;
- qualidade em uso. Corresponde à visão da qualidade do produto de *software*, do ponto de vista do usuário, quando utilizado em um determinado ambiente e contexto. Ela mede quanto os usuários podem atingir seus objetivos em um dado ambiente, ao invés de medir as propriedades do produto de *software* propriamente ditas.

A nova versão da ISO Std. 9126 (2001) constitui-se de quatro partes:

- ISO/IEC 9126-1: define as características e as subcaracterísticas de qualidade, introduzindo os conceitos de características internas, externas e em uso e incluindo, em seu corpo, as subcaracterísticas antes apresentadas apenas como sugestão;
- ISO/IEC 9126-2: define as métricas externas para avaliar e garantir a qualidade externa de produtos de *software*;
- ISO/IEC 9126-3: define as métricas internas para avaliar e garantir a qualidade interna de produtos de *software*;
- ISO/IEC 9126-4: define as métricas para avaliar e garantir a qualidade em uso de produtos de *software*.

A norma ISO/IEC 9126-1 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)] define seis características de qualidade, as quais descrevem a qualidade de produto de *software* com um mínimo de sobreposição, de forma a serem utilizadas na especificação ou na avaliação de requisitos de um produto de *software*. Um conjunto de atributos de um produto de *software*, através do qual sua qualidade é descrita e avaliada, define característica de qualidade de produto de *software*. Cada uma das características de qualidade pode ser melhor entendida por meio de um conjunto de subcaracterísticas. A Tabela 2.1 apresenta esta organização, com perguntas chaves para melhor entendimento.

Tabela 2.1 – Subcaracterísticas do Modelo de Qualidade da Norma ISO/IEC 9126 (Fonte: Gomes (2001))

CARACTERÍSTICAS	SUBCARACTERÍSTICAS	SIGNIFICADO
Funcionalidade O conjunto de funções satisfaz às necessidades explícitas e implícitas para a finalidade a qual se destina o produto?	Adequação	Propõe-se a fazer o que é apropriado?
	Acurácia	Gera resultados corretos ou conforme acordados?
	Interoperabilidade	É capaz de interagir com os sistemas especificados?
	Segurança de Acesso	Evita acesso não autorizado, acidental ou deliberado a programas e dados?
	Conformidade	Está de acordo com normas e convenções previstas em leis e descrições similares?
Confiabilidade O desempenho se mantém ao longo do tempo e em condições estabelecidas?	Maturidade	Com que freqüência apresenta falhas?
	Tolerância a Falhas	Ocorrendo falhas, como ele reage?
	Recuperabilidade	É capaz de recuperar dados após uma falha?
Usabilidade É fácil utilizar o produto de *software*?	Inteligibilidade	É fácil entender os conceitos utilizados?
	Apreensibilidade	É fácil aprender a usar?
	Operacionalidade	É fácil operar e controlar a operação?
Eficiência Os recursos e os tempos utilizados são compatíveis com o nível de desempenho requerido para o produto?	Comportamento em relação ao tempo	Qual é o tempo de resposta e de processamento?
	Comportamento em relação aos recursos	Quanto recurso utiliza?
Manutenibilidade Há facilidade para correções, atualizações e alterações?	Analisabilidade	É fácil encontrar uma falha quando ocorre?
	Modificabilidade	É fácil modificar e remover defeitos?
	Estabilidade	Há grandes riscos de *bugs* quando se faz alterações?
	Testabilidade	É fácil testar quando se faz alterações?
Portabilidade É possível utilizar o produto em diversas plataformas com pequeno esforço de adaptação?	Adaptabilidade	É fácil adaptar a outros ambientes sem aplicar outras ações ou meios além dos fornecidos para esta finalidade no produto de *software* considerado?
	Capacidade para ser instalado	É fácil instalar em outros ambientes?
	Capacidade para substituir	É fácil substituir por outro produto de *software*?
	Conformidade	Está de acordo com padrões ou convenções de portabilidade?

Eis as características e as subcaracterísticas de qualidade ora definidas [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)]:

- Funcionalidade: é o conjunto de atributos que evidencia a existência de um conjunto de funções e suas propriedades que satisfazem às necessidades implícitas e explícitas. São suas subcaracterísticas:

⇒ Adequação: consiste em atributos do produto de *software* que evidenciam a presença de um conjunto de funções e a sua adequação para as tarefas especificadas;

⇒ Acurácia: consiste em atributos do produto de *software* que evidenciam a geração de resultados ou efeitos corretos ou conforme acordados;

⇒ Interoperabilidade: consiste em atributos do produto de *software* que evidenciam a sua capacidade de interagir com outros produtos de *software*;

⇒ Segurança de Acesso: consiste em atributos do produto de *software* que evidenciam a sua capacidade de evitar acessos não autorizados, acidentais ou deliberados aos dados e ao produto de *software*;

⇒ Conformidade: consiste em atributos do produto de *software* que evidenciam o atendimento de normas, padrões, convenções e/ou regulamentações previstos em leis e descrições similares, relacionadas à aplicação.

- Confiabilidade: é o conjunto de atributos que evidencia a capacidade do produto de *software* em manter um bom nível de desempenho sob determinadas condições e em um determinado período de tempo. São suas subcaracterísticas:

  ⇒ Maturidade: consiste em atributos do produto de *software* que evidenciam a freqüência de falhas por defeitos no produto de *software*;

  ⇒ Tolerância a Falhas: consiste em atributos do produto de *software* que evidenciam a sua capacidade em continuar o nível de desempenho especificado nos casos de falhas no produto de *software* ou de violação nas interfaces especificadas;

  ⇒ Recuperabilidade: consiste em atributos do produto de *software* que evidenciam a sua capacidade de retornar ao funcionamento normal, através da recuperação dos dados após as falhas e o tempo e o esforço necessários para isso.

- Usabilidade: é o conjunto de atributos que evidencia o esforço necessário para poder utilizar o produto de *software*, bem como o julgamento individual desse uso, por um conjunto implícito ou explícito de usuários. São suas subcaracterísticas:

  ⇒ Apreensibilidade: consiste em atributos do produto de *software* que evidenciam o esforço do usuário para aprender a sua aplicação;

⇒ Inteligibilidade: consiste em atributos do produto de *software* que evidenciam o esforço do usuário para reconhecer o conceito lógico e a sua aplicabilidade;

⇒ Operacionalidade: consiste em atributos do produto de *software* que evidenciam o esforço do usuário para a sua operação e o controle desta operação.

- Eficiência: é o conjunto de atributos que evidencia o relacionamento entre o nível de desempenho do produto de *software* e a quantidade de recursos que utiliza sob determinadas condições. São suas subcaracterísticas:

  ⇒ Comportamento em Relação ao Tempo: consiste em atributos do produto de *software* que evidenciam o seu tempo de processamento e de resposta e a velocidade na execução de suas funções;

  ⇒ Comportamento em Relação a Recursos: consiste em atributos do produto de *software* que evidenciam a quantidade de recursos utilizados e a duração de seu uso na execução de suas funções.

- Manutenibilidade: é o conjunto de atributos que evidencia o esforço necessário para realizar modificações específicas no produto de *software*.

  ⇒ Analisabilidade: consiste em atributos do produto de *software* que evidenciam o esforço necessário para diagnosticar deficiências ou causas de falhas, ou para identificar partes a serem modificadas;

  ⇒ Modificabilidade: consiste em atributos do produto de *software* que evidenciam o esforço necessário para modificá-lo, remover seus defeitos ou adaptá-lo a mudanças ambientais;

  ⇒ Estabilidade: consiste em atributos do produto de *software* que evidenciam o risco de efeitos inesperados, ocasionados por modificações;

  ⇒ Testabilidade: consiste em atributos do produto de *software* que evidenciam o esforço necessário para validar o produto de *software* modificado.

- Portabilidade: é o conjunto de atributos que evidencia a capacidade do produto de *software* de ser transferido de um ambiente para outro.

  ⇒ Adaptabilidade: consiste em atributos do produto de *software* que evidenciam a sua capacidade de ser adaptado a ambientes diferentes do especificado, sem a necessidade de aplicação de outras ações ou meios além

daqueles fornecidos para essa finalidade pelo produto de *software* considerado;

⇒ Capacidade para ser Instalado: consiste em atributos do produto de *software* que evidenciam o esforço necessário para a sua instalação em um determinado ambiente;

⇒ Capacidade para Substituir: consiste em atributos do produto de *software* que evidenciam o esforço necessário para substituir um outro produto de *software*, no ambiente estabelecido para este segundo;

⇒ Conformidade: consiste em atributos do produto de *software* que o tornam de acordo com os padrões ou com as convenções relacionadas à portabilidade.

2.6. Manutenibilidade e Manutenção de Produtos de *Software*

A característica de manutenibilidade de um produto de *software*, conforme a norma ISO/IEC 9126, é o conjunto de atributos que evidencia o esforço necessário para realizar modificações [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)]. A norma ainda apresenta as subcaracterísticas de manutenibilidade, conforme descrito na seção 2.5.

Dessa forma, para que o produto de *software* seja manutenível, as subcaracterísticas de manutenibilidade devem ser incorporadas durante o seu processo de desenvolvimento. A seguir, alguns autores definem a manutenibilidade de produtos de *software*:

- Ambler (1998) a define como uma medida da facilidade de adicionar, remover ou modificar as características existentes de um produto de *software*, isto é, quanto mais fácil for modificar o produto de *software*, maior é a sua manutenibilidade;
- Maffeo (1992) a apresenta como um fator de qualidade associado à possibilidade de alterações serem introduzidas, a custos reduzidos, visando a corrigir, adaptar ou aperfeiçoar o produto de *software*;
- Pfleeger (2001) a entende como a probabilidade que, para uma dada condição de uso, uma atividade de manutenção pode ser realizada em um determinado intervalo de tempo e usar determinados procedimentos e recursos;

- Capretz; Capretz (1996) definem que a manutenibilidade englobaria todas as modificações do produto de *software* depois de entregue, de modo a corrigir falhas, melhorar o desempenho ou outros atributos ou adaptá-lo a um novo ambiente;
- Pressman (2006) a conceitua qualitativamente como a facilidade que um produto de *software* pode ser entendido, corrigido, adaptado e/ou aumentado.

Visando a realizar alterações de forma correta e segura, é importante que os artefatos de *software* sejam bem compreendidos e que sua estrutura apresente características que possibilitem e facilitem estas alterações. Para tal, a documentação dos produtos de *software* (os artefatos que o descrevem) deve ser elaborada e mantida adequadamente, pois é um importante subsídio de auxílio na tarefa de manutenção.

A característica de manutenibilidade deve ser incorporada ao produto de *software* desde o início do seu processo de desenvolvimento, como qualquer requisito. Entretanto, os produtos de *software*, normalmente, são desenvolvidos sem a preocupação com o seu tempo de vida útil, não sendo projetados visando a facilitar a sua manutenção [Pressman (2006)]. Dessa forma, o custo de tempo e o financeiro tendem a ser bem maior para se realizar a manutenção de um produto de *software* do que para desenvolver um novo. Isso é agravado pelo fato de que, de acordo com pesquisas realizadas na literatura, percebe-se que há poucas referências sobre a manutenibilidade de produtos de *software*, principalmente quanto à geração dos artefatos adequados desde o início de seu desenvolvimento [Costa (2005)].

Brusamolin (2004) afirma que existem quatro fatores que impactam na manutenibilidade e expõe os resultados de suas pesquisas:

- arquitetura: o investimento em arquitetura de *software* aprimora, entre outras características de qualidade, a manutenibilidade. A divisão da aplicação em componentes e dos componentes em classes tornou cada unidade de código-fonte bem menor do que em uma abordagem monolítica[3], de modo que o esforço para modificar o código apresenta-se pequeno e o potencial de inclusão de erros nessa modificação torna-se reduzido;
- tecnologia: o uso de tecnologias mais robustas na implementação contribui para a redução do pesadelo da manutenção, quando da proliferação de programas ao longo da vida da empresa de *software*. Na pesquisa, ele cita a orientação a objetos como o

[3] A abordagem monolítica procura descrever completamente um problema num único modelo, sem divisões.

paradigma que leva à menor degradação do código-fonte. A degradação ocorre em função do número de linhas acrescidas ou modificadas, o que implica que manutenção feita com menos código significa menos entropia[4];

- documentação: quando especificações de *design* do produto de *software* não estão disponíveis, o mantenedor tem que investir muito tempo para compreendê-lo antes de modificá-lo;
- compreensibilidade do programa: conforme estimativas pesquisadas por Brusamolin (2004), os programadores ficam entre 47% e 62% do tempo de trabalho tentando entender documentação e lógica dos produtos de *software*.

Tradicionalmente, completa-se o processo de desenvolvimento de produtos de *software* ao serem entregues aos usuários para uso. Considera-se manutenção qualquer mudança, após o produto de *software* estar operacional. Como mudanças representam uma verdade inevitável ao longo da sua vida, deve-se fornecer mecanismos previstos para avaliar, controlar e fazer essas modificações [Liverpool (2000)].

A atividade de manutenção de produtos de *software* está assumindo um papel cada vez mais importante dentro do chamado ciclo de vida [Paduelli; Sanches (2006)]. A manutenção é definida pela IEEE [IEEE std 610.12 (1994)] como a fase do ciclo de vida do produto de *software* em que são realizadas as atividades relativas às modificações em um produto de *software*, após a sua entrega aos usuários/clientes, a fim de corrigir falhas, melhorar desempenho (ou outros atributos) e/ou adaptá-lo a mudanças no ambiente.

Outras definições encontradas na literatura são:

- a manutenção corresponde às mudanças que devem ser feitas em programas de computador depois de sua entrega ao cliente ou usuário [Martin; MacClure (1983)];
- a manutenção cobre a vida do produto de *software* a partir de sua instalação até seu descarte [Mayrhauser (1990)].

A manutenção se apresenta como parte integrante do ciclo de vida do produto de *software*; contudo não recebeu a mesma atenção que outras fases [SWEBOK (2004)]. Dessa forma, existe a necessidade de concentrar esforços para a manutenção de produtos de *software*, pois muitos daqueles existentes são da década de 90. Mesmo com o uso das melhores práticas de projeto e implementação na época, os critérios para um bom produto de *software* eram um pouco diferentes dos atuais, porque a preocupação estava no tamanho

[4] Entropia pode ser entendida como a diferença entre o todo e suas partes [Brusamolin (2004)].

do código-fonte e no espaço para armazenamento. Muitos desses produtos de *software* vêm sendo migrados para novas plataformas, adaptados às mudanças de equipamentos e de tecnologia e incrementados para atender aos novos desejos dos usuários. Nessa situação, depara-se com estruturas e lógicas projetadas de forma fraca e pobre, com documentação deficiente e com códigos-fonte nem sempre organizados de produtos de *software* que devem continuar funcionando [Pressman (2006)].

Lehman; Belady (1985), preocupados com a manutenção, propuseram um conjunto de leis de evolução de produtos de *software*, chamadas Leis de Lehman. Dentre elas, a primeira[5] apresenta maior relevância por ora. Ela explicita a manutenção de produtos de *software* como inevitável e a eliminação de falhas como apenas uma parte das atividades de manutenção, pois, para o produto de *software* se tornar útil, a mudança dos seus requisitos deve ser incorporada a ele. A razão disso se mostra no ambiente onde ele está inserido poder mudar com o tempo, de forma que os requisitos (necessidades) dos usuários também mudem.

Os produtos de *software*, em geral, são desenvolvidos sem a preocupação com o seu tempo de vida útil, não sendo projetados objetivando facilitar a sua manutenção [Pressman (2006)]. Dessa forma, observa-se que o problema não está no produto de *software*, mas na forma como ele tem sido desenvolvido.

A manutenibilidade de produtos de *software* tende a ser negligenciada, devido a diversos fatores, entre eles [Paduelli; Sanches (2006), Liverpool (2000), Dekleva (1992) e Lientz; Swanson (1980)]:

- anseios das organizações responsáveis pelo seu desenvolvimento em atender rapidamente o mercado;
- cultura de que a manutenção não é uma atividade nobre;
- caracterização da atividade de manutenção como um problema a ser tratado posteriormente ao desenvolvimento dos produtos de *software*.

Outro problema encontrado é a existência de uma idéia equivocada: a manutenção de produtos de *software* é semelhante à manutenção de *hardware*. Sabe-se que esta última consiste em substituir peças gastas e/ou usar técnicas para prolongar a sua vida útil.

[5] Primeira Lei de Lehman: um produto de *software* que é usado em um ambiente do mundo real deve mudar ou torna-se progressivamente menos útil no ambiente [Lehman; Belady (1985)].

Entretanto, os produtos de *software* não degradam com o tempo e a manutenção de *software* está associada com alteração de produtos de *software*.

Deve-se considerar que não são os próprios projetistas que realizam a manutenção, na maior parte das vezes e, em geral, a equipe alocada pode não contar com os autores, seja pelo fato deles estarem alocados em outros projetos, seja por não estarem mais na organização [Page-Jones (1990)]. Dessa forma, em um desenvolvimento de produtos de *software*, devem ser adotadas medidas que facilitem a sua manutenção.

Enfim, construir e manter produtos de *software* não é uma tarefa trivial, pois eles [Costa (2005)]:

- são mais complexos do que o *hardware* onde são executados;
- não têm produção em série, ou seja, o seu custo está no projeto e no desenvolvimento;
- não se desgastam e nem se modificam com o uso;
- são invisíveis, isto é, a sua representação em grafos e diagramas não é precisa.

Por outro lado, para Kajko-Mattsson *et al.* (2001), a manutenção de produtos de *software* é uma atividade difícil pelos seguintes motivos:

- problemas de diagnóstico – meses podem ser gastos em processos investigativos para detectar a real causa do problema antes de corrigi-lo;
- falta de documentação – inexistência ou desatualização de documentos sobre os requisitos, as especificações e os processos;
- falta de conhecimento sobre o produto – familiarização com a estrutura e a composição interna dos produtos a serem alterados;
- negligência na condução de mudanças no produto de *software* – falta gerência, isto é, métricas, análises de riscos, acompanhamento, controle e documentação;
- pouca habilidade de escrita – manter um produto de *software* não é somente programar, precisa-se saber documentar o que está programando;
- baixa estima dos mantenedores – os profissionais geralmente dão preferência para atuar em novos projetos (desafios); manter um produto de *software* reduz a moral do *staff* e compromete a qualidade e o rendimento da atividade como um todo.

Vários são os problemas técnicos e gerenciais enfrentados pela atividade de manutenção de produtos de *software*, cujas principais causas [Booch (1994)] são: i) a complexidade do domínio do problema; ii) a dificuldade em gerenciar o processo de desenvolvimento e manutenção; iii) a eventual necessidade de flexibilização do produto de

software; e iv) a tecnologia ultrapassada presente nos sistemas legados, os quais resistem de maneira significativa a modificações ou evoluções.

Dessa forma, por causa dessas características, pouca pesquisa foi realizada em relação à manutenção, existindo poucas referências sobre o assunto, se comparado às outras etapas do ciclo de vida dos produtos de *software* [Pressman (2006)].

O tempo utilizado para a realização da manutenção de produtos de *software*, em comparação com o desenvolvimento, é muito maior e se faz como uma verdade da Engenharia de *Software*. Apesar disso, a maioria dos cursos, das disciplinas e das referências bibliográficas preferem abordar com maior ênfase o desenvolvimento de produtos de *software* em detrimento da manutenção [Dias (2004)]. Dessa forma, conforme Pfleeger (2001), o desenvolvimento típico de um produto de *software*, que leva entre um e dois anos, requer cinco a seis anos de manutenção. Pensando em esforços, mais da metade dos recursos acaba por ser alocada para manutenção ao longo do seu ciclo de vida.

A forte demanda pelo esforço e pelo tempo é gerada, principalmente, pela falta de disciplina e controle nas atividades iniciais do processo de desenvolvimento de produtos de *software*. Decisões inadequadas de projeto podem interferir negativamente no produto de *software* e, como conseqüência, ampliar a complexidade de sua manutenção [Araújo (1993)]. Isto ocorre devido à necessidade de conhecer a funcionalidade do produto de *software* e saber como ela foi idealizada [Capretz; Capretz (1996), Mayrhauser; Vans (1996) e Sharon (1996)].

A negligência nas tarefas de manutenção pode ser crítica, contribuindo para a deterioração da estrutura e da manutenibilidade do produto de *software*. O problema de deterioração, conhecido como erosão de projeto, significa que enquanto uma parte do produto de *software* envelhece e está sendo mantida, torna-se mais difícil manter a sua integridade. Assim, sua integridade conceitual foi violada, deixando-a remendada e emendada além do reparo [Land (2003)].

Embora não exista um consenso sobre o valor exato do custo atrelado à atividade de manutenção [Paduelli; Sanches (2006)], as pesquisas na área apontam, na totalidade dos casos, sempre mais de 50% dos investimentos. De fato, os custos de manutenção dos produtos de *software* podem girar em torno de 60% do esforço gasto por uma organização e este percentual pode crescer, pois os produtos de *software* continuam sendo

desenvolvidos [Ramaswamy (2000), Schach; Tomer (2000), Sharon (1996) e Veldwijk (1994)].

Outras porcentagens são ditas pelos seguintes autores: Pfleeger (2001) estima estes custos em cerca de 80% do orçamento total do ciclo de vida de um produto de *software*; para Bhatt *et al.* (2004), esse percentual corresponde a algo entre 67% a 75% do investimento total; e para Polo *et al.* (1999), corresponde a um valor entre 67% e 90%. Ainda de acordo com Polo *et al.* (1999), a razão desse custo elevado deve-se, em parte, à própria natureza da atividade de manutenção, caracterizada, principalmente, pela imprevisibilidade. Além dos altos custos financeiros, essa é a atividade que exige maior esforço dentre as atividades de Engenharia de *Software* [Sneed (2003)].

O maior agravante dessa problemática é a manutenção de um produto de *software*, na maioria das vezes, possuir um fluxo contínuo de requisitos seja por falhas, por solicitações do cliente através do SAC (Serviço de Atendimento a Clientes), por mudanças na Legislação ou, simplesmente, pela evolução voltada para o mercado [Costa; Rouiller (2005)]. Dessa forma, a demanda do mercado por profissionais mantenedores de produtos de *software* é alta (embora não sejam alocados de forma correta para a tarefa que deveriam cumprir), por ser uma atividade que consome a maior parte dos custos da indústria de produtos de *software* [Dias *et al.* (2003)].

A Figura 2.1 mostra a distribuição do orçamento total gasto no desenvolvimento de um novo produto de *software* e na manutenção de um produto de *software* existente. Conforme estas estimativas, a atividade de manutenção gasta em torno de 60% a 80% do orçamento destinado à construção de um produto de *software* [Alves (1993), Guedes (1993), Lucena (1991), Meyer (1997), Moura (1992), Perin (1992), Pressman (2006), Schach; Tomer (2000), Sommerville (2000) e Yourdon (1990)].

Os custos para ampliar a funcionalidade de um produto de *software*, após sua operação, por exemplo, são muito maiores do que fornecer funcionalidade similar quando ele ainda está em desenvolvimento. De acordo com Sommerville (2000), há algumas razões para isso:

- a equipe de manutenção relativamente inexperiente e não familiarizada com o domínio da aplicação. A manutenção é considerada uma atividade que necessita de pouca habilidade se comparada com o desenvolvimento e alocada ao pessoal novo;

- produtos de *software* em manutenção desenvolvidos há anos, desconsiderando as técnicas de Engenharia de *Software*. Podem estar desestruturados e eficientes, ao invés de apresentar características para facilitar o entendimento;
- alterações realizadas em produtos de *software* podem adicionar novos erros, devido à complexidade do produto de *software*, conduzindo a dificuldades na verificação dos efeitos de tais mudanças;
- devido às constantes mudanças no produto de *software*, a sua estrutura tende a degradar, fazendo com que ele se torne mais difícil de ser entendido e dificultando mudanças futuras, por gerar um produto de *software* menos coeso;
- a correspondência entre o código-fonte e a documentação do produto de *software* pode se perder ao longo do processo de manutenção, tornando a documentação não confiável para o seu entendimento.

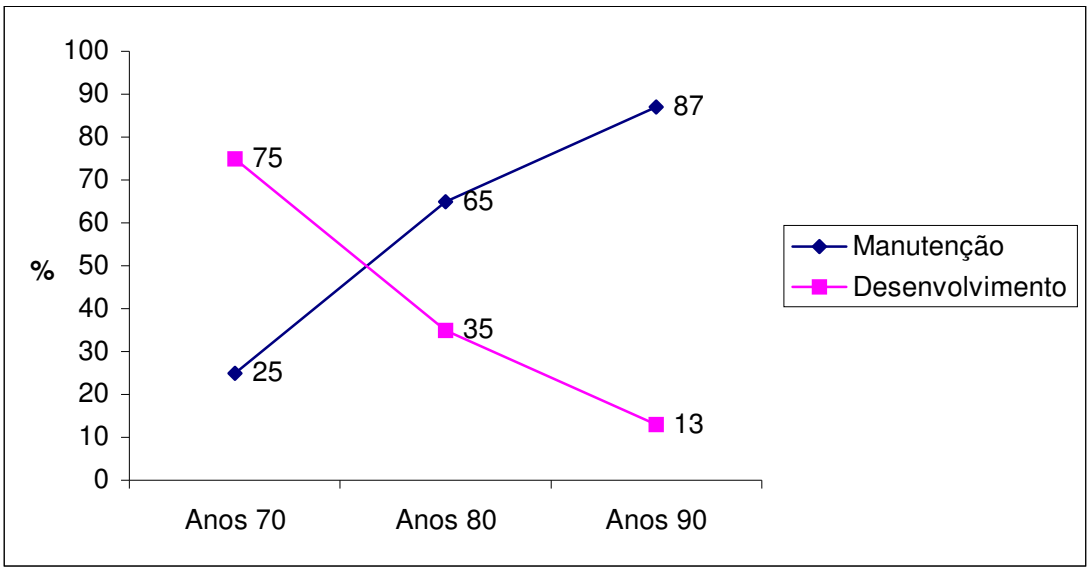

Figura 2.1 - Custos de Desenvolvimento – Manutenção (Fonte: Pfleeger (2001) e Pressman (2006))

A manutenção de produtos de *software* pode-se apresentar sob quatro tipos: corretiva, adaptativa, perfectiva e preventiva [Capretz; Capretz (1996), Pfleeger (2001), Pressman (2006), Sharon (1996), Sommerville (2000) e Liverpool (2000)]. Alguns autores chamam a manutenção perfectiva de evolutiva [Swanson (1976) e Arthur (1988)]. A Tabela 2.2 exibe uma breve descrição de cada um dos quatro tipos de manutenção mais comuns e a Figura 2.2 apresenta um gráfico do percentual de ocorrências destes tipos.

Sommerville (2000) classifica os fatores que afetam a manutenção em técnicos e não-técnicos. Os fatores técnicos são:

- independência de módulos: deve ser possível alterar um componente de um produto de *software* sem afetar outros componentes;
- linguagem de programação: o produto de *software* escrito em uma linguagem de programação de alto nível é, normalmente, mais fácil de ser entendido e, conseqüentemente, de ser mantido, em detrimento daquele escrito em linguagem de programação de baixo nível;
- estilo de programação: a forma em que um produto de *software* é escrito contribui para seu entendimento e facilita a sua modificação;
- teste e validação de produtos de *software*: geralmente, quanto mais tempo e esforços forem gastos na validação do projeto e no teste, menor é a quantidade de erros;
- qualidade da documentação do produto de *software*: se um produto de *software* é suportado por uma documentação clara, completa e concisa, a tarefa de entendê-lo é facilitada;
- técnicas de gerência de configuração: há um custo significativo para manter a rastreabilidade e garantir a consistência da documentação de um produto de *software*.

Tabela 2.2 – Tipos de Manutenção

Tipo de Manutenção	Origem da Mudança	Descrição
Adaptativa	Usuário ou Ambiente	Realizada quando o produto de *software* precisa ser adaptado às novas tecnologias (*hardware* e *software*) implantadas no ambiente operacional. As atividades de manutenção adaptativa têm origem em uma solicitação de melhoramento.
Corretiva	Requisitos do Sistema	Realizada quando são corrigidos erros não identificados durante a etapa de teste. Visa a consertar defeitos de um sistema para que fique em conformidade com os requisitos sobre os quais foi desenvolvido. Estes defeitos dizem respeito a quaisquer problemas com o *hardware*, *software* ou documentação.
Evolutiva / Perfectiva	Equipe de Manutenção	Realizada quando o produto de *software* deve englobar novos requisitos ou melhorias decorrentes da evolução na tecnologia de implementação empregada. Isso acontece quando se modifica o sistema, de modo a aumentar a qualidade do *software* ou de sua documentação, sem modificar sua funcionalidade. Nesta categoria, estão incluídos os esforços para aumentar a legibilidade do produto de *software*, para melhorar sua performance, para incrementar a reusabilidade, entre outros.
Preventiva	Maturidade da Empresa	Realizada quando o produto de *software* é alterado para aumentar sua manutenibilidade e/ou confiabilidade, sendo relativamente raro em ambientes de desenvolvimento. Modificações realizadas neste tipo de manutenção não afetam o comportamento funcional do produto de *software*.

Os fatores não-técnicos são:

- domínio da aplicação: estando bem definido e entendido, a probabilidade da completude dos requisitos é considerável;

- estabilidade da equipe: custos de manutenção são reduzidos se os engenheiros que desenvolvem o produto de *software* são também responsáveis pela sua manutenção;
- tempo de vida do produto de *software*: quanto mais antigo é o produto de *software* (pode ter sofrido mais ações de manutenção), mais dispendiosa pode se tornar a sua manutenção, devido à degradação de sua estrutura;
- dependência do produto de *software* com o meio externo: quando o meio externo muda, o produto de *software* deve mudar para acompanhar as alterações no meio externo;
- estabilidade do *hardware*: se o produto de *software* foi desenvolvido para uma configuração específica do *hardware*, quando há alteração da configuração, o produto de *software* deve ser alterado para acompanhar a nova configuração.

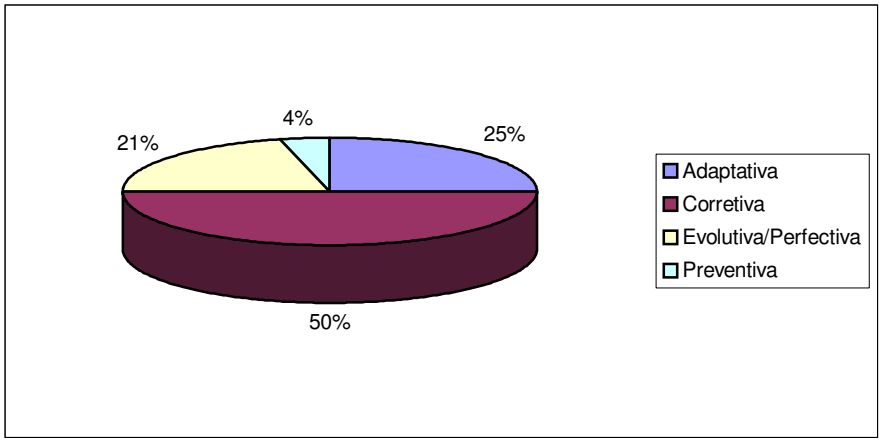

Figura 2.2 – Percentual dos tipos de manutenção (Fonte: Pfleeger (2001))

De forma a reduzir a problemática da atividade de manutenção, os produtos de *software* devem ser desenvolvidos visando à manutenibilidade desde o início de seu desenvolvimento. Para Costa (2005), tal atividade pode ser auxiliada pelo uso das subcaracterísticas da manutenibilidade da norma ISO/IEC 9126, as quais facilitam:

- diagnosticar as deficiências ou as causas de falhas ou identificar as partes a serem modificadas;
- modificar o produto de *software*, remover os seus defeitos ou adaptá-lo às mudanças ambientais;
- validar o produto de *software* modificado;
- identificar o risco de efeitos inesperados, ocasionados por modificações.

De acordo com Arthur (1988), existem atividades necessárias a cada um dos tipos de manutenção após a identificação de uma solicitação de mudança. A seguir, estão resumidas estas atividades para os três tipos mais comuns de manutenção:

- Manutenção Corretiva:
    - ⇒ elaborar hipóteses sobre a causa do defeito;
    - ⇒ testar as hipóteses para encontrar a causa exata do defeito;
    - ⇒ reparar o defeito encontrado.
- Manutenção Adaptativa:
    - ⇒ identificar partes da arquitetura envolvidas;
    - ⇒ elaborar alternativas;
    - ⇒ avaliar as alternativas;
    - ⇒ implementar a alternativa selecionada.
- Manutenção perfectiva:
    - ⇒ identificar aspectos de qualidade candidatos a melhoria;
    - ⇒ tratar os candidatos identificados como defeitos e utilizar a técnica para manutenção corretiva.

A partir desta relação de atividades, conforme Souza (2005), pode-se identificar três habilidades do mantenedor que influenciam sua produtividade:

- compreensão do código: influencia na produtividade das atividades de elaborar hipóteses sobre a causa do defeito e de identificar partes da arquitetura envolvida;
- elaboração de soluções: influencia as atividades de reparar o defeito encontrado e de elaborar alternativas;
- modificação do código: influencia as atividades de reparar o defeito encontrado, de avaliar as alternativas e de implementar a alternativa selecionada.

A diferença entre essas duas últimas é a primeira focar apenas em desenvolver soluções aplicáveis ao produto de *software* em manutenção sem, no entanto, considerar as conseqüências práticas disto. Estes detalhes são desvendados somente em um estágio posterior, onde a habilidade de maior peso será a modificação do código. Ainda se deve considerar que o desenvolvimento das três habilidades depende das capacidades do mantenedor e da qualidade do código-fonte.

Além do mais, a documentação deve ser elaborada e mantida de maneira correta, uma vez que é fonte poderosa de auxílio à manutenção. Tal documentação consiste nos artefatos que descrevem o produto de *software*, desde a especificação de requisitos até o plano de teste de aceitação.

Algumas soluções potenciais para os problemas de manutenção são [Liverpool (2000)]:

- realocação de esforço e de orçamento: tentativa de recuperar produtos de *software* quando a manutenção é a melhor alternativa;
- substituição completa do produto de *software*: quando a manutenção supera os custos de desenvolvimento de um novo produto de *software* – deve-se tentar aproveitar o conhecimento para, pelo menos, evitar reinventar a roda;
- manutenção preventiva: prover alterações no produto de *software* e em sua documentação, de forma a deixá-lo atualizado e torná-lo mais manutenível em caso de necessidade.

2.7. Considerações Finais

São escassos na literatura os trabalhos que visam a documentar as principais áreas de problemas durante a atividade de manutenção de produtos de *software*. Isso decorre do fato de que a manutenção é vista como a última atividade no ciclo de vida do produto de *software*. Entretanto, a manutenibilidade deve estar na mente da equipe durante o desenvolvimento do produto de *software* [Dart *et al.* (1993)].

Neste capítulo, foi apresentada a etapa de manutenção de um processo de desenvolvimento de produtos de *software*, de forma a elucidar seus conceitos, problemas, desafios e considerações históricas. No contexto deste trabalho, foram apresentados os termos qualidade e qualidade de *software* e a norma ISO/IEC 9126, a qual trata de características de qualidade de produtos de *software*, a fim de focar na característica de manutenibilidade e em suas subcaracterísticas. Isso é importante, uma vez que a manutenibilidade de produtos de *software* é de interesse na área de Engenharia de *Software*. Sabe-se que a melhor prática é aquela em que os produtos de *software* seriam projetados visando à manutenibilidade e na qual a sua documentação deveria permitir a correção e o reuso de código-fonte durante o seu aprimoramento, bem como eliminar a necessidade da engenharia reversa[6].

Além disso, como parte das referências sobre manutenção, vale ressaltar que Canning (1972) compara a manutenção de produtos de *software* a um *iceberg*, pois se

[6] A Engenharia Reversa é comum ao longo da manutenção de produtos de *software*, pois a documentação elaborada durante seu desenvolvimento costuma estar desatualizada com o produto de *software* existente. A técnica consiste na reconstrução da documentação a partir do código-fonte do produto de *software* existente [Pressman (2006)].

espera que o que é imediatamente visível seja tudo que existe. Entretanto, uma enorme massa de possíveis problemas e custos fica sob a superfície, isto é, a tarefa de manutenção não se resume apenas à manutenção por si só, mas ao aparato que envolve esta tarefa.

Dessa forma, a tarefa de manutenção tem caráter de equipe e poderia ser mais fácil e efetiva se conseguisse conjeturar melhor as possíveis fontes de erros. Conforme Pfleeger (2001), os pesquisadores estão em busca de novos modelos de qualidade para mostrar as ligações entre produtos e processos de *software*. Dessa forma, estes modelos ajudarão a identificar quando o esforço deve ser usado para manter um produto de *software* e quando é apropriado descartá-lo.

3. ORIENTAÇÃO A ASPECTOS

3.1. Considerações Iniciais

Este capítulo apresenta a programação orientada a aspectos (POA), uma nova tecnologia que tenta solucionar alguns dos problemas identificados na programação orientada a objetos.

Na seção 2, um breve histórico sobre a POA é descrito, abordando sua origem e sua importância e estruturando alguns de seus conceitos básicos. Algumas ferramentas e linguagens que dão suporte à programação orientada a aspectos são listadas na seção 3, com ênfase na linguagem de programação *AspectJ*, a qual é descrita com mais detalhes na seção 4.

3.2. Origem e Ascensão da Programação Orientada a Aspectos

A Engenharia de *Software* e as linguagens de programação coexistem em um relacionamento de suporte mútuo. A maioria dos processos de projeto de *software* considera um sistema em unidades cada vez menores. As linguagens de programação, por sua vez, fornecem mecanismos que permitem a definição de abstrações de unidades do sistema e a composição destas de diversas formas possíveis, para a produção do sistema como um todo [Becker; Geihs (1998)]. Uma linguagem de programação articula-se adequadamente com um projeto de *software* quando provê abstrações e mecanismos de composição que suportam claramente os tipos de unidades descritas no projeto. Os mecanismos de abstração das linguagens mais comumente utilizadas (sub-rotinas, procedimentos, funções, objetos, classes) podem ser enquadrados em um modelo de procedimentos generalizados, obtidos através da decomposição funcional do sistema.

À medida que aumenta a complexidade dos produtos de *software*, surge a necessidade de melhores técnicas de programação, objetivando organizar e apoiar o processo de desenvolvimento de produtos de *software*. Com isso, no âmbito da Ciência da Computação, essas técnicas de programação vêm evoluindo desde construções de baixo nível – como linguagens de máquina – até abordagens de alto nível – como programação orientada a objetos (POO) [Elrad *et al.* (2001)].

Segundo Resende; Silva (2005), inicialmente, desenvolviam-se produtos de *software* utilizando a programação desestruturada. Comandos de desvio de fluxo (por exemplo, o comando GOTO) eram usados indiscriminadamente, tornando muito difícil o entendimento e a reutilização do código-fonte. A programação estruturada foi um grande avanço, pois a funcionalidade era agrupada em funções ou procedimentos, que eram chamados a partir de um programa principal. Quem utilizava a programação estruturada podia reutilizar as funções e os procedimentos, desde que devidamente projetados. Nesta época, a reusabilidade foi explorada, utilizando o conceito de bibliotecas.

Com o advento da POO, em meados dos anos 70, com a linguagem de programação *Smaltalk*, inicia-se um novo paradigma de desenvolvimento de produtos de *software* que está presente até os tempos atuais. A POO trouxe grandes avanços para o desenvolvimento de produtos de *software*, permitindo a construção de sistemas mais fáceis de serem projetados, maior reusabilidade de componentes, modularidade, componentização, implementações mais robustas e redução do custo de manutenção [Junior; Winck (2006)]. Muitas aplicações não estão em apenas um único módulo de código contido em um único arquivo. Aplicações são coleções de módulos (classes) que trabalham juntas para fornecer determinadas funções para um conjunto de requisitos bem definidos [Gradecki; Lesiecki (2003)].

Entretanto, apesar de tais avanços, o aumento da complexidade inseriu novos problemas que a POO é incapaz de resolver. Desenvolver um produto de *software* é tão complexo que se tornou impossível exigir de um funcionário que domine as fases de desenvolvimento de produtos de *software*. Atualmente, atribuem-se determinadas preocupações para grupos de especialistas, sendo que a evolução da indústria de produtos de *software* tem forçado a divisão das áreas de preocupações de desenvolvimento em partes independentes.

Dessa forma, existem certas propriedades que se pretende programar em um produto de *software* que não são adequadamente encapsuladas em classes (não se enquadram em componentes da decomposição funcional), seja porque se aplicam às diversas classes de um produto de *software* simultaneamente, seja porque não pertencem à natureza intrínseca da(s) classe(s) à(s) qual(is) se aplica(m). A dificuldade de modularizar alguns tipos de funções causa um acúmulo de responsabilidade adicional que o projeto da

classe não previa, ocasionando um entrelaçamento da responsabilidade inicial – intrínseca da classe – com responsabilidade extra [Hugo; Grott (2005)].

Na concepção de Kiczales *et al.* (1997), nem as técnicas de POO nem as técnicas da programação procedural (estruturada) são suficientes para implementar com clareza importantes decisões do projeto. Isto conduz a uma implementação espalhada através do código-fonte, resultando em um emaranhado de código que se torna excessivamente difícil de desenvolver e manter. Assim, muitos interesses importantes espalham-se por vários módulos e misturam-se com outros interesses de um sistema de maneira intrusiva, dificultando a reutilização e a manutenção de seus componentes. Isto se deve ao fato de que as abordagens mais utilizadas concentram-se em encontrar e compor unidades funcionais da decomposição do sistema, enquanto que outras decisões importantes não são bem localizadas no projeto funcional. Isso afeta a performance ou a semântica da aplicação e aumenta a dependência entre os componentes funcionais, desviando-os de sua finalidade principal, tornando-os menos reusáveis e mais propensos a erros.

O princípio da separação de interesses (*separation of concerns*) foi introduzido por Dijkstra (1976), tendo como objetivo dividir o domínio do sistema em partes menores com o intuito de entender melhor cada parte isoladamente. Um interesse é alguma parte do domínio do problema que se deseja tratar com uma unidade conceitual única. No desenvolvimento de produtos de *software*, um interesse pode ser visto como um requisito funcional ou não funcional de um sistema. De forma geral, os vários interesses do sistema devem ser separados em módulos de acordo com as abstrações do desenvolvimento de produtos de *software* providas por linguagens, métodos e ferramentas. Os interesses de um produto de *software* podem ser classificados como [Laddad (2003)]:

- interesses do negócio – capturam a funcionalidade central de um módulo. Por exemplo, procedimento de quitação de uma compra;
- interesses em nível de sistema – capturam requisitos periféricos, no nível do sistema, e que atravessam múltiplos módulos. Por exemplo, segurança, *logging* e persistência.

A POO permitiu melhor separação dos diversos interesses do produto de *software*, com a estruturação de projetos e códigos mais próximos do que é idealizado naturalmente pelos desenvolvedores, favorecendo a reutilização, a flexibilidade, a manutenibilidade e a modularidade [Elrad *et al.* (2001) e Hugo; Grott (2005)]. Neste caso, as abstrações básicas para os interesses são classes, objetos, métodos e atributos. Entretanto, conforme

mencionado anteriormente, tais abstrações podem não ser suficientes para serem separadas em um único módulo (alguns dos interesses contidos em muitos produtos de *software* complexos), tendo assim os comportamentos distribuídos ao longo de vários e, às vezes não relacionados, módulos. Esses interesses são chamados de interesses transversais (*crosscutting concerns*), pois, inerentemente, a sua implementação se dá através da adição de código em diversos objetos ao longo do produto de *software*, não estando diretamente relacionado com a funcionalidade definida para estes objetos. Desta forma, a implementação mapeia os requisitos em uma única dimensão, como pode ser visualizado na Figura 3.1. Podem ser citados como exemplos de interesses transversais *logging*, integridade de transações, tratamento de erros, controle de concorrência, autenticação, segurança, desempenho, distribuição, persistência e *profiling* [Elrad *et al.* (2001) e Monteiro (2004)].

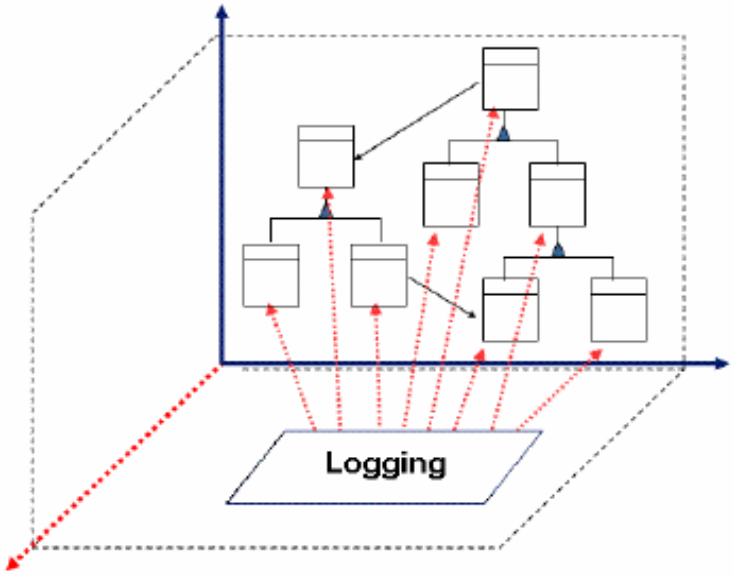

Figura 3.1 – Mapeamento de um interesse transversal (Fonte: Barbosa (2005))

A separação de conceitos (separação de interesses) [Pace; Campo (2001)] é um princípio fundamental na Engenharia de *Software* aplicado na análise, no projeto e na implementação de sistemas computacionais. A partir dessa separação, pode-se entender com maior facilidade como esta age no produto de *software* como um todo, uma vez que não é necessário procurá-la em diferentes lugares e separá-la de outras propriedades. Ainda é possível analisá-la, modificá-la, estendê-la e reusá-la mais facilmente [Czarnecki; Eisenecker (2000)].

Como exemplo de um interesse transversal, tem-se um editor de figuras apresentado em Elrad *et al.* (2001). O produto de *software* de edição de figuras é composto por duas

classes concretas de elementos de figura, Ponto e Linha. Sempre que os elementos da figura forem movimentados, é necessário que a tela seja atualizada. Desta maneira, observa-se através do Diagrama de Classes apresentado na Figura 3.2, que o interesse de atualização da tela não pertence às classes Ponto e Linha, mas entrecorta-as e, por isso, pode ser entendido como transversal a essas classes [Elrad *et al.* (2001)].

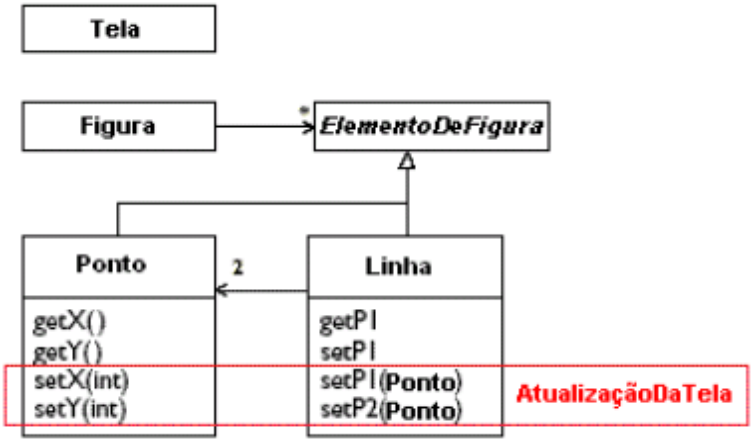

Figura 3.2 – Interesse transversal de atualização da tela (Fonte: Elrad *et al.* (2001))

Sem o uso de técnicas apropriadas para a separação de interesses e modularização, alguns fenômenos são observados e podem ser classificados nas seguintes categorias [Resende; Silva (2005)]:

- Código entrelaçado (*code tangling*) – acontece quando a implementação de um módulo em um produto de *software* interage simultaneamente com vários interesses, tais como *logging*, autenticação, *multi-threaded safety* e validações. Na Figura 3.3, é ilustrado o código entrelaçado em um módulo, causado pela execução simultânea de múltiplos interesses;

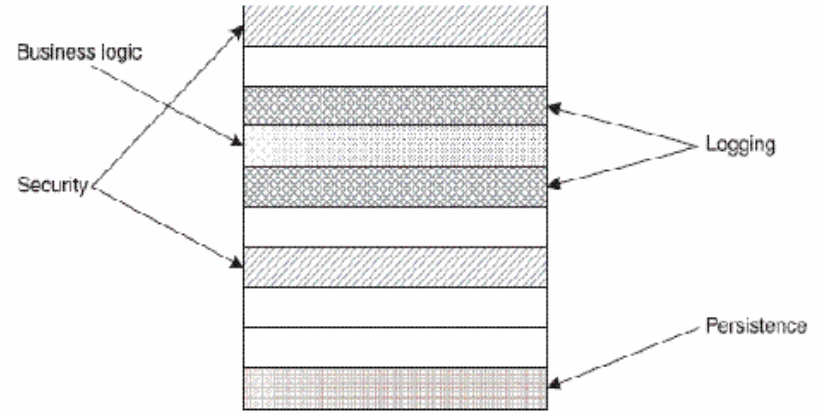

Figura 3.3 – Código entrelaçado causado pela execução simultânea de múltiplos interesses (Fonte: Laddad (2003))

- Código espalhado (*code scattering*) – acontece quando um interesse é implementado por múltiplos módulos. Desde que interesses transversais, por definição, são espalhados por vários módulos, a sua implementação também se espalha por esses módulos. Por exemplo, considere o mecanismo de *logging*. Na Figura 3.4, as colunas representam cada uma das classes do produto de *software* e as linhas grifadas são referentes à funcionalidade de *logging*. Na parte esquerda, verifica-se o encapsulamento de interesses transversais (ortogonais[7]) e na parte direita, a situação comum em POO de espalhamento e entrelaçamento (emaranhado).

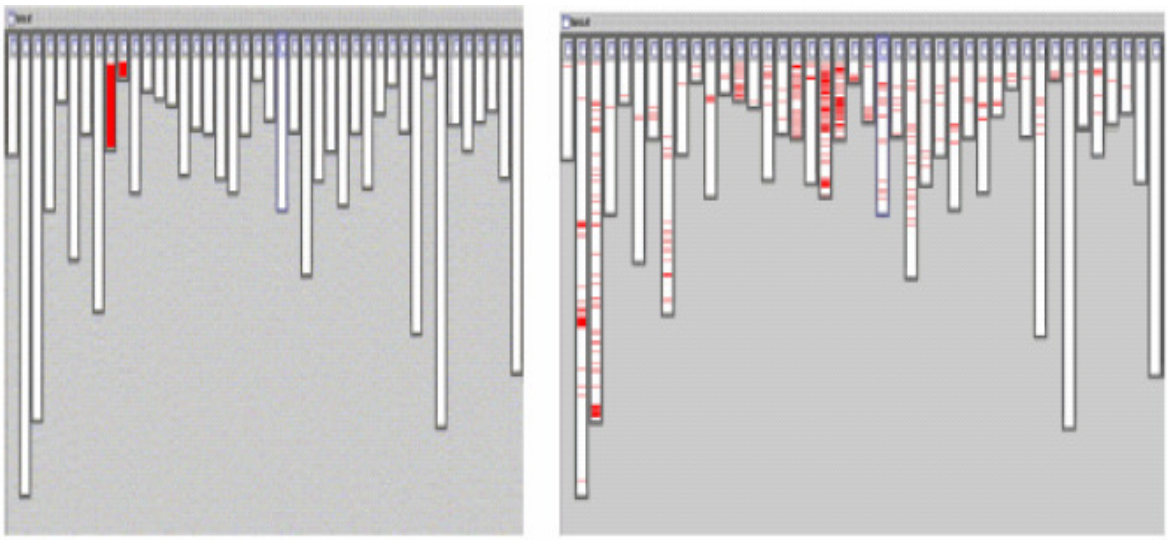

Figura 3.4 – Implementação de interesses transversais gerando código emaranhado (Fonte: Junior *et al.* (2004))

De acordo com Barbosa (2005), a implementação de códigos entrelaçados e espalhados afeta o desenvolvimento do produto de *software* de diversas maneiras:

- forte acoplamento – métodos das classes primárias precisam conhecer métodos das classes que implementam funções espalhadas (*crosscutting concerns*);
- fraca coesão – métodos das classes afetadas contêm instruções que não estão diretamente relacionadas às funções que implementam;
- redundância – muitos fragmentos de código semelhantes ocorrem em diversos pontos do código-fonte;
- dificuldades de compreender, manter e reusar – como conseqüência da implementação de *crosscutting concerns* ser dependente do código espalhado pelo produto de *software*, sua compreensão, manutenção e reutilização ficam prejudicadas.

[7] Quando duas ou mais propriedades sendo programadas devem ser compostas de maneira diferente e ainda coordenarem-se é dito que elas são ortogonais entre si [Piveta (2001)].

De acordo com Steinmacher (2003), entretanto, parte dessas barreiras pode ser eliminada com o uso de técnicas com a intenção de modularizar o desenvolvimento de produtos de *software* incluindo *mix-in* classes, padrões de projeto e soluções específicas de domínio. Por outro lado, existem algumas extensões da POO que tentam solucionar suas limitações visando a uma maior modularidade dos produtos de *software*, tais como programação orientada a aspectos (*Aspect-Oriented Programming*) [Kiczales *et al.* (1997)], Programação Orientada a Sujeito (*Subject-Oriented Programming*) [Osser; Tarr (1999)] e Programação Adaptativa (*Adaptive Programming*) [Lieberherr *et al.* (1994)]. Dentre essas extensões, a que tem se mostrado mais promissora é a programação orientada a aspectos (POA) [Elrad *et al.* (2001)].

Assim, a POA foi proposta por Kiczales *et al.* (1997), em Palo Alto, nos laboratórios da *Xerox*, como uma técnica visando a melhorar o suporte à modularização dos interesses transversais por meio de abstrações que possibilitem a separação e a composição destes interesses na construção dos produtos de *software*. Além disso, a POA não trabalha isoladamente; é um paradigma que estende outros paradigmas de programação, como a POO e a programação estruturada [Fernandes (2003)].

3.3. Programação Orientada a Aspectos

Conforme Monteiro (2004), a orientação a aspectos (OA) é uma proposta de solução para os problemas causados pela ineficiência dos atuais paradigmas de programação em prover a modularização adequada dos interesses multidimensionais no desenvolvimento de produtos de *software*. Ela propõe uma separação clara dos interesses multidimensionais de forma a evitar o entrelaçamento de interesses distintos e, por meio de um combinador (*weaving*), possibilita recompor tais interesses.

POA é uma abordagem que permite a separação dessas propriedades ortogonais dos componentes funcionais de uma forma natural e concisa, utilizando-se de mecanismos de abstração e de composição para a produção de código executável.

O Desenvolvimento de *Software* Orientado a Aspectos (DSOA) é uma prática de desenvolvimento baseado nos conceitos de OA e se utiliza de POA para implementá-los. POA propõe um novo tipo de abstração – o aspecto (*aspect*) – e novos mecanismos de composição que permitem a descrição modular e a composição de interesses que

geralmente se encontram espalhados e misturados (*crosscutting concerns*) em vários pontos de um produto de *software* orientado a objetos [Kiczales (2001)].

Segundo Garcia *et al.* (2004), a importância de DSOA no contexto da Engenharia de *Software* tem se mostrado evidente. Quando o tema começou a se destacar no Brasil (ano de 2003), vários trabalhos foram aceitos em eventos nacionais de relevância para a Engenharia de *Software*. O Simpósio Brasileiro de Engenharia de *Software* (SBES 2003), realizado em Manaus, em outubro de 2003, reuniu um tutorial e quatro artigos relacionados a DSOA. O SBES 2004 reuniu seis artigos, um tutorial e um mini-curso envolvendo tópicos associados com DSOA. Em 2005 e 2006, não foi diferente. Além disso, diversos tutoriais e *workshops* sobre DSOA têm sido organizados no contexto internacional. Eventos e sessões técnicas específicos sobre DSOA são geralmente incluídos no programa de conferências importantes, como ICSE (*International Conference on Software Engineering*), FSE (*Foundations on Software Engineering*), ETAPS (*European Joint Conference on Theory and Practice of Software*), ASE (*Automated Software Engineering*), OOPSLA (*Conference on Object-Oriented Programming, Systems, Languages, and Applications*), e ECOOP (*European Conference on Object-Oriented Programming*). Atualmente, no Brasil, existe o WASP (*Workshop* Brasileiro de Desenvolvimento de *Software* Orientado a Aspectos), o qual teve sua quarta edição em 2006, com trabalhos cada vez mais abrangentes. Em sua última edição, foi proposta a estruturação de um grupo sul-americano para estudos em Engenharia de *Software* Orientada a Aspectos, o qual está sendo implementado.

Ademais, a área possui uma conferência própria, denominada *International Conference on Aspect-Oriented Software Development* (AOSD *Conference*). A conferência é apoiada pela ACM (*Association for Computing Machinery*) e o seu programa se encontra em crescente expansão. A sexta edição será realizada em março de 2007. Vários livros sobre DSOA foram publicados nos últimos três anos, abordando projeto e programação de *software* orientados a aspectos. Existe ainda o periódico *Lecture Notes em Computer Science* (LNCS) *Transactions on Aspect-Oriented Software Development*, dedicado a explorar assuntos específicos da área. No Brasil, existem dois livros em português abordando o tema: **Programação Orientada a Aspectos com Java** [Resende; Silva (2005)] e **AspectJ: Programação Orientada a Aspectos com Java** [Junior; Winck (2006)].

A finalidade de POA não é substituir as técnicas de orientação a objetos, mas complementá-las para possibilitar a separação dos interesses multidimensionais em uma unidade – o aspecto. O aspecto permite que mudanças, quando necessárias, sejam definidas no código-fonte de um produto de *software* de forma menos invasiva (seja em sua estrutura ou em seu comportamento). Assim, é possível alcançar modularização adequada, beneficiando o desenvolvimento e a manutenção de um produto de *software*. A Figura 3.5 traz essa representação. Nela, podem-se observar as classes-base de um produto de *software*, representadas pelas barras mais claras, e os interesses multidimensionais, representados em um aspecto (a barra em vermelho). Assim, é possível perceber a separação dos requisitos do produto de *software*.

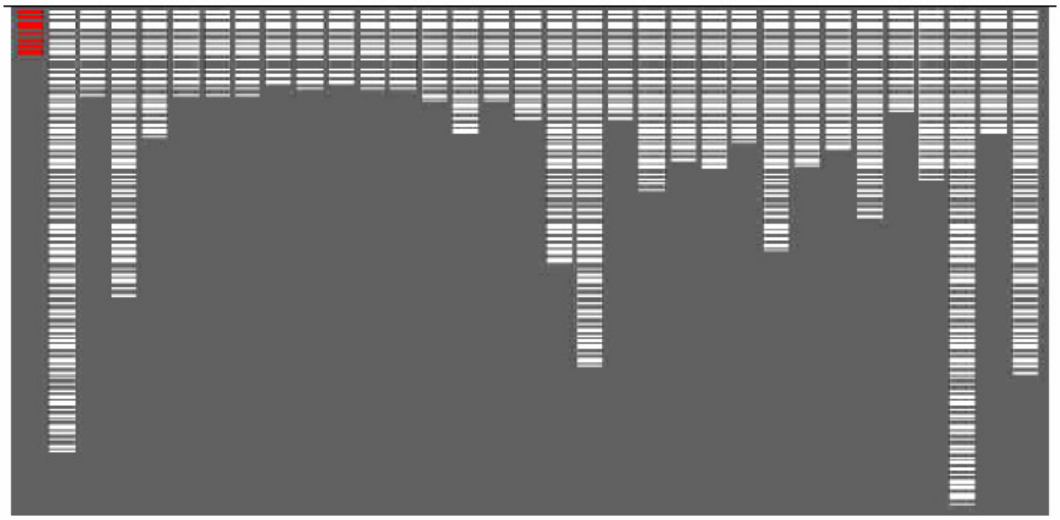

Figura 3.5 – Representação de um interesse multidimensional concentrado em um aspecto (Fonte: Kiczales (2001))

Uma implementação básica em POA compreende [Piveta (2001) e Steinmacher (2003)] (Figura 3.6):

- uma linguagem de componentes para a programação. A linguagem que atender a este requisito deve permitir ao programador escrever programas que implementem as funções básicas do produto de *software*, simultaneamente ao fato de que não prevêem nada a respeito do que deve ser implementado na linguagem de aspectos. Por exemplo: *Java*, *C++*, *Lisp* e *Cobol*;
- uma ou mais linguagens de aspectos para a programação. As linguagens que atenderem a este requisito devem suportar a implementação das propriedades desejadas de forma clara e objetiva, provendo construções necessárias para que o programador crie estruturas que descrevam o comportamento dos aspectos e definam em que ocasiões eles ocorrem. Por exemplo: *AspectJ*, *HyperJ*, *COOL* e *RIDL*;

- um combinador de aspectos (*aspect weaver*) para a combinação do código. O processo de combinação é o produto do cruzamento de um ou mais aspectos com os componentes descritos na linguagem de componentes. Ele processa o código, compondo-as de forma correta, e produz a interação desejada. Entretanto, esta interação não é a mesma que ocorre entre os módulos da linguagem de componentes, mas aquela que ocorrerá nos pontos de combinação. Tais pontos são locais dentro do código-fonte onde é possível agregar o comportamento que destaca POA (chamadas a métodos, construtores, destrutores ou funções), que oferecem a interface entre os aspectos e os componentes. O processo de combinação pode ser: i) estático, o qual consiste em modificar o código-fonte escrito em linguagem de componentes, inserindo sentenças nos pontos de combinação, que representam o código de aspectos, em tempo de compilação (exemplo: *AspectJ*); ou ii) dinâmico, no qual as combinações são feitas em tempo de execução e os aspectos devem existir em tempo de compilação e em tempo de execução (exemplo: *AOP/ST* e *AspectS*);
- um programa escrito na linguagem de componentes;
- um ou mais programas escritos na linguagem de aspectos.

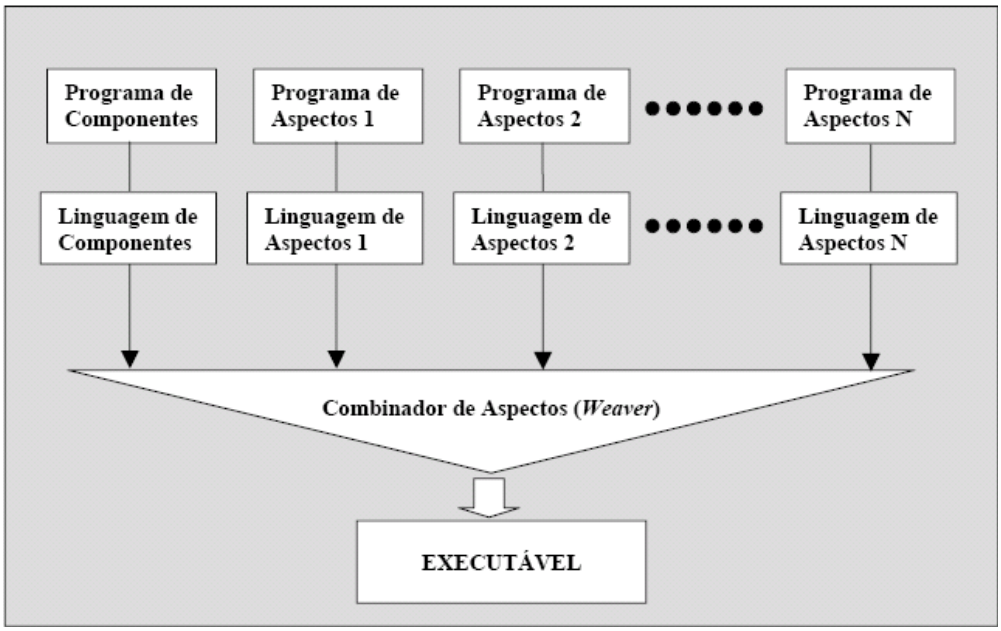

Figura 3.6 – Estrutura de um programa orientado a aspectos (Fonte: Steinmacher (2003))

POA envolve basicamente três etapas distintas de desenvolvimento [Barbosa (2005)]:

- decompor os interesses (*aspectual decomposition*) – identificar e separar os interesses transversais dos interesses do negócio;

- implementar os interesses (*concern implementation*) – implementar cada um dos interesses identificados separadamente;
- recompor os interesses (*aspectual recomposition*) – ter o integrador de aspectos que especifica regras de recomposição para a criação de unidades de modularização (aspectos). Tal processo de junção da codificação dos componentes e dos aspectos é a combinação (*weaving*).

Os requisitos funcionais, normalmente, são organizados em um conjunto de componentes, expresso por uma linguagem de POO, e os requisitos não funcionais (ou interesses multidimensionais) são organizados em aspectos [Kiselev (2002)].

Irwin *et al.* (1997) define que uma propriedade de um produto de *software* que deve ser implementada pode ser vista como um componente ou como um aspecto:

- a propriedade pode ser vista como um componente se puder ser encapsulada em um procedimento generalizado (objeto, método, procedimento, API – *Application Programming Interface*). Os componentes tendem a ser unidades da decomposição funcional do produto de *software*. Como exemplos, podem ser citados: contas bancárias, usuários e mensagens;
- os aspectos não são normalmente unidades da decomposição funcional do produto de *software*, mas propriedades que envolvem diversas unidades, afetando a performance e a semântica dos componentes funcionais sistematicamente. Exemplos de aspectos podem ser: i) controle de concorrência em operações em uma mesma conta bancária; ii) registro das transações de uma determinada conta; iii) política de segurança de acesso aos usuários de um sistema; e iv) restrições de tempo real associadas à entrega de mensagens.

Dada a definição de componentes e de aspectos, é possível colocar o objetivo de POA: oferecer suporte para o programador na tarefa de separar claramente os componentes dos aspectos, os componentes entre si e os aspectos entre si, utilizando-se de mecanismos que permitam a abstração e composição destas, produzindo o sistema desejado [Piveta (2001)].

Com relação ao mecanismo de composição em orientação a objetos, herança, parametrização e chamadas de funções são exemplos. Os mecanismos de composição de aspectos, conforme Czarnecki; Eisenecker (2000), devem permitir (idealmente) os seguintes requisitos:

- fraco acoplamento. Os aspectos devem possuir o mínimo possível de ligações com os componentes. Um fraco acoplamento entre aspectos e componentes é desejável;
- adição não-invasiva de aspectos ao código existente. É a capacidade de adaptar um componente ou um aspecto sem modificá-lo manualmente. Deve ser possível expressar mudanças como uma operação aditiva. Mecanismos de composição são utilizados para adicionar mudanças no código existente. O código não é fisicamente modificado, mas a expressão da composição indica que o código original possui uma semântica diferente. Um exemplo é o mecanismo de herança, onde a herança é não-invasiva com respeito à sua superclasse.

Conforme Piveta (2001) e Souza (2003), as seguintes propriedades podem ser vistas como aspectos:

- Sincronização de Objetos Concorrentes. Um programa em que dados globais sejam compartilhados necessita sincronizar suas atividades através de mecanismos que possibilitem sua execução. Um exemplo de política de sincronização: os objetos podem realizar operações de leitura em um determinado objeto, se ele não estiver sendo modificado, mas somente um pode realizar uma operação de escrita por vez;
- Distribuição. O projeto e a implementação de produtos de *software* distribuídos requer abordar diversos problemas que não existem em sistemas não-distribuídos. No projeto, é necessário mudar a perspectiva de como os componentes devem ser especificados, envolvendo o entendimento de diferentes componentes e como eles se relacionam. Na implementação, as linguagens de programação atualmente utilizadas deixam a desejar em prover suporte para programar em termos dessas diferentes perspectivas e de propriedades que são ortogonais uma às outras;
- Tratamento de Exceções. Algumas formas de tratamento de exceções são inerentes à definição de um tipo de objeto (por exemplo, tentar retirar um elemento de uma pilha vazia). Entretanto, políticas de tratamento de exceções mais genéricas podem ser vistas e implementadas como aspectos. Como exemplo, pode-se citar o uso de pré-condições e pós-condições que devem ser satisfeitas no momento em que uma operação é realizada. Estas condições podem ser implementadas como aspectos;
- Coordenação de Múltiplos Objetos. A implementação da coordenação de múltiplos objetos apresenta diversos problemas quanto ao reuso e quanto à extensão. Esta implementação é mais difícil que a primeira vista, devido à necessidade de sincronizar a integração entre os objetos ativos em busca de um objetivo global. Estes problemas

ocorrem por causa da falta de mecanismos de alto nível no modelo convencional de objetos que permitam abstrair o comportamento coordenado e devido ao algoritmo de coordenação estar espalhado, isto é, distribuído entre conjuntos de objetos, dificultando o seu entendimento. Os algoritmos usados para a coordenação de múltiplos objetos são normalmente ligados às implementações, não podendo ser reusados para objetos diferentes. Pode-se dizer que a coordenação é expressa através de mecanismos de sincronização específicos da linguagem de programação, tornando difícil o reuso das classes destes objetos. Um exemplo de coordenação pode ser a execução de um vídeo, onde a exibição de imagens, a reprodução de música e a locução são responsabilidade de objetos distintos que necessitam se coordenarem;

- Persistência. A persistência é um elemento importante em sistemas computacionais que aparece espalhado através do código do produto de *software*. Uma alternativa é a implementação desta característica como um aspecto, independente de como e quais são os componentes que devem ser tornados persistentes. A partir desta implementação, pode ser feita a composição entre o aspecto e os objetos. A recuperação de dados é feita no momento em que o objeto é acessado e a atualização da base de dados ocorre no momento da criação ou modificação do estado do objeto;
- Outras propriedades. Existem outras características que podem ser vistas como aspectos: serialização, atomicidade, replicação, segurança, visualização, *logging*, *tracing*, tolerância à falhas, obtenção de métricas, dentre outras.

Dessa forma, por meio da separação dos interesses em componentes e aspectos, POA permite uma modularização na construção do produto de *software*. Como conseqüência desta modularização, torna-se mais fácil escrever, compreender, reutilizar e manter um produto de *software*, porque seu código torna-se mais limpo por concentrar e tratar apenas um interesse do produto de *software*.

3.4. Linguagens e Ferramentas para a Programação Orientada a Aspectos

O conceito de orientação a aspectos não é novo; há muito tempo que se fala em separação de responsabilidades. Por isso, existem produtos de *software* que possuem propriedades de POA [Monteiro; Piveta (2003)]. Segundo Böllert (1998), existem alguns requisitos que devem ser observados na especificação de uma linguagem de aspectos:

- sua sintaxe deve ser fortemente relacionada com a da linguagem de componentes, de maneira a facilitar o aprendizado e obter maior aceitação por parte dos desenvolvedores;
- a linguagem deve ser projetada para especificar o aspecto de maneira concisa e compacta. Normalmente, as linguagens de aspecto são de mais alto nível de abstração que as linguagens de programação de uso geral;
- sua gramática deve possuir elementos que permitam ao combinador (*weaver*) compor os programas escritos usando as linguagens de aspectos e componentes.

Dessa forma, é apresentada uma descrição sucinta de algumas ferramentas de suporte à POA disponíveis na literatura:

- *HyperJ* [Ossher; Tarr (1999)]. As preocupações são agrupadas em uma estrutura multidimensional, baseando-se no conceito de MDSoC (*Multi Dimensional Separation of Concerns*). Esta ferramenta suporta separação multidimensional de preocupações em *Java*, com combinação dinâmica e propósito geral. O *HyperJ* permite compor um conjunto de modelos separado, em que cada um encapsula uma preocupação, definindo e implementando uma hierarquia de classes apropriada para esta preocupação. Cada modelo deve se estender por si próprio, aumentando seu comportamento juntando-se com outro. *HyperJ* inclui uma ferramenta que permite identificar preocupações, especificar módulos e sintetizar sistemas e componentes. A ferramenta oferece um módulo para a edição e a construção de regras de integração, permitindo um processo simples de teste para a integração de preocupações no produto de *software*. O usuário, primeiramente, cria *hypermódulos* escolhendo um conjunto de preocupações e um conjunto de regras de integração. O *HyperJ* cria *hyperslices* válidos de acordo com as escolhas. O usuário pode inserir novas regras ou modificar as existentes e, assim, continuar o processo de refinamento até obter o modelo desejado. Os principais benefícios de *HyperJ* são: i) separação e modularização de preocupações de maneira flexível; ii) composição não-invasiva com adaptação de componentes; iii) possibilidade da criação de extensões e da configuração de componentes sem modificação do código-fonte; e iv) desenvolvimento descentralizado de classes;
- *QIDL* (*Quality Interface Definition Language*) [Becker; Geihs (1998)]. A qualidade de serviço (*Quality of Service* – QoS) aborda o tratamento de propriedades não funcionais em produtos de *software* distribuídos, sendo originada de problemas inerentes à distribuição, os quais reduzem a sua transparência (tais como: flutuação dinâmica de

banda, transmissão de erros e falhas). Os autores ponderam que a QoS pode ser entendida como um aspecto na visão de POA, devendo, portanto, ser especificada em uma linguagem apropriada, o que pode ser feito juntamente com a especificação do aplicativo e integrada à implementação através de um *framework* de integração. Ao invés de usar um combinador de aspectos, os elementos de QoS são gerados em um *framework* através do compilador IDL (*Interface Definition Language*). Os aspectos são definidos em IDL e a implementação é feita na própria linguagem de componentes. QIDL é uma extensão de CORBA que funciona através de definições de QoS, que são projetadas para suportar uma grande variedade de restrições diferentes. O compilador QIDL pode auxiliar o usuário a implementar serviços que possuam QoS disponíveis;

- *AOP/ST* [Böllert (1998)]. Esta ferramenta provê suporte a POA com combinação de aspectos dinâmicos para *Visualworks*/*SmallTalk* e apresenta propósito específico. Ela é composta por um combinador para *SmallTalk* e duas linguagens de aspectos: uma para sincronização de processos e uma para acompanhamento de fluxo de execução de um produto de *software* (*tracing*). O combinador de *AOP/ST* é aberto para a integração com outras linguagens de aspectos, bem como possui um conjunto de componentes que possibilitam aos desenvolvedores criarem suas próprias linguagens de aspectos;
- *D* [Lopes (1997)]. Este ambiente de linguagens de aspectos fornece abstrações para a implementação de esquemas de sincronização e transferência remota de dados em sistemas distribuídos. Chama-se ambiente, pois oferece duas linguagens de propósito específico e com combinador estático: *COOL* (*Coordination Aspect Language*), para controlar a sincronização de *threads*, e *RIDL* (*Remote Interaction and Data transfers aspect Language*), para programar a transferência entre componentes remotos. Essas linguagens foram desenvolvidas de maneira independente de uma linguagem de componentes. Como *D* impõe algumas restrições sobre a linguagem de componentes, pode-se dizer que ela é semi-independente de linguagens de componentes. Essas restrições são satisfeitas pelas linguagens orientadas a objetos mais conhecidas (*C++*, *SmallTalk*, *Java* e *Eiffel*). Os aspectos descritos em *D* afetam o comportamento distribuído e concorrente dos componentes, possibilitando ao desenvolvedor preocupar-se primeiramente com a funcionalidade dos componentes e, a seguir, de uma maneira explícita, descrever como os aspectos afetam os componentes de um modo bem localizado e relativamente separado do restante da aplicação. A partir de *D*, foi

implementado um protótipo usando *Java* como linguagem de componentes. Esta implementação foi chamada de *DJ*;

- *IL* [Berger *et al.* (1998)]. Esta linguagem de aspectos descreve as interações entre objetos através de suas interfaces. Utilizando-se *IL*, independente da linguagem de componentes utilizada, as classes e os seus comportamentos são escritos na linguagem de componentes sem precisar se preocupar com interações. Elas são especificadas fora do objeto e a descrição das interações dos objetos é feita através de regras de interação que definem (para uma mensagem qualquer) um conjunto de mensagens parcialmente ordenado que corresponde ao fluxo de execução;
- D^2AL (*Design-Based Distribution Aspect Language*) [Becker (1998)]. Esta linguagem foi desenvolvida para a especificação de distribuição através de uma linguagem de aspectos, permitindo ao desenvolvedor separar o que diz respeito à distribuição do que faz parte da funcionalidade básica do produto de *software*. A diferença fundamental entre D^2AL e outras linguagens de aspectos é a construção das linguagens referir-se ao projeto e não à implementação da funcionalidade do produto de *software*;
- *JST* [Senturier (1999)]. Esta linguagem de aspectos para *Java* permite ao desenvolvedor expressar sincronização de objetos, separadamente do código de componentes, sendo baseada em uma linguagem de transição de estados. A idéia básica é definir em uma classe de sincronização os estados que um objeto da classe pode estar e quais os métodos que podem ser executados neste estado. Os benefícios de utilizar *JST* são: i) separar o código de sincronização da funcionalidade básica do produto de *software*; ii) prover uma linguagem onde os estados e as transições são elementos sintáticos do produto de *software*; e iii) prover uma abordagem semelhante a Diagramas de Estados para escrever classes de sincronização;
- *AspectIX* [Becker *et al.* (1998)]. Esta arquitetura possibilita a implementação e o controle de propriedades não funcionais de objetos distribuídos, proporcionando uma interface aberta e genérica para a especificação destas propriedades. A arquitetura permite que a implementação de objetos distribuídos seja modificada em tempo de execução, para permitir aos objetos refletirem as mudanças na configuração dos aspectos;
- *AspectJ* [AspectJ (2007)]. Esta linguagem de aspectos de propósito geral estende a linguagem *Java* para suportar POA de maneira genérica e apresenta combinação estática. É uma ferramenta *open source*, possuindo suporte a ambientes integrados de

desenvolvimento (*Emacs*, *Forte 4J*, *NetBeans*, *Jbuilder* e *Eclipse*). O projeto da linguagem é dirigido através da resposta dos usuários em relação ao ambiente. É descrita com mais detalhes na seção 3.5;

- *AspectC* [Coady *et al.* (2001)]. Esta linguagem de aspectos de propósito geral estende a linguagem C. É dito *AspectJ-like*, apesar de não oferecer qualquer tipo de suporte a POO ou módulos explícitos. O código de aspectos, conhecido como *advice*, interage com a funcionalidade básica nos limites de uma chamada a uma função e pode ser executado antes, depois ou durante a chamada. Os elementos centrais desta ferramenta têm como objetivo assinalar chamadas de funções particulares, acessando os parâmetros destas chamadas e anexar *advices* a elas. Como a linguagem C é uma linguagem estática, o combinador do *AspectC* é estático. Esta ferramenta, apesar de simples, é de grande utilidade devido ao amplo uso da linguagem C em diversos tipos de aplicações. *AspectC* é ideal para o desenvolvimento de produtos de *software* nos quais a eficiência é primordial, como na implementação de sistemas operacionais;
- *AspectC++* [AspectC++ (2007); Gal (2001)]. Esta linguagem de aspectos de propósito geral estende a linguagem *C++* para suportar a orientação a aspectos, com combinação estática. Nesta linguagem, os pontos de combinação estão no código de componentes onde os aspectos podem interferir. Estes pontos podem se referir a código-fonte, tipos, objetos e fluxos de controle. Os aspectos em *AspectC++* implementam de forma modular as preocupações e são extensões do conceito de classes em *C++*. Esta linguagem introduz importantes conceitos dentro do modelo de pontos de combinação, permitindo que estes atuem sobre classes, objetos e fluxo de controle;
- *AspectS* [AspectS (2007); Hirschfeld (2002)]. Esta linguagem de aspectos de propósito geral estende a linguagem *Squeak*/*SmallTalk* para suportar a orientação a aspectos. Esta ferramenta utiliza o modelo da linguagem *AspectJ*, relacionando os aspectos, e um ambiente dinâmico, empregando composição de meta-objetos. Nesta linguagem, os aspectos são implementados através de classes e suas instâncias atuam como um objeto, respeitando o princípio de uniformidade. Um aspecto pode conter um conjunto de *receptors*, *senders* ou classes *senders*. Estes objetos são adicionados ou removidos pelo código cliente e são utilizados pelo processo de combinação em tempo de execução, para determinar se o comportamento deve ou não ser ativado.

3.5. AspectJ

Dentre as propostas para implementar aspectos, pode-se destacar a ferramenta *AspectJ* [AspectJ (2007)]. Essa ferramenta é bem aceita por estender a linguagem de programação *Java* com construções eficientes para a implementação de interesses multidimensionais em separado em um produto de *software* (de uso geral) e, por meio dela, pode-se definir a implementação de aspectos em vários pontos de um produto de *software* [Resende; Silva (2005)].

AspectJ consistiu de uma proposta do *Palo Alto Research Center*, tradicional centro de pesquisas da *Xerox*, apresentando *Java* como a sua linguagem de componentes e possibilitando ao desenvolvedor especificar instruções nos seguintes pontos de combinação [Barbosa (2005)]: i) chamada e execução de métodos; ii) recebimento de chamadas a construtores; iii) execução de construtores; iv) acesso a campos; v) execução de inicialização estática; vi) pré-inicialização e inicialização de objetos; e vii) execução de manipuladores de exceção.

A Figura 3.7 mostra a estrutura do desenvolvimento utilizando *AspectJ*, no qual os componentes são implementados em *Java* e os aspectos em *AspectJ*. O *weaver* combina os dois tipos de códigos, gerando um novo produto de *software*.

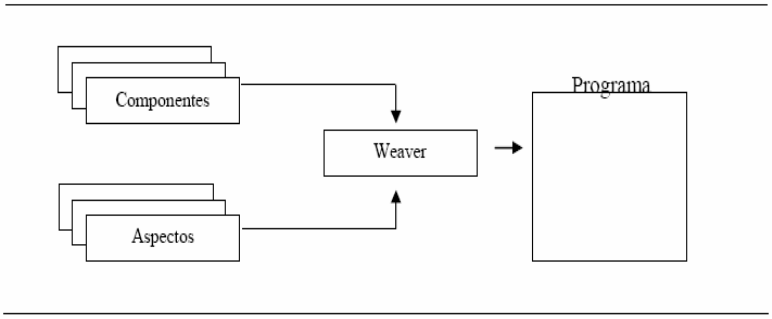

Figura 3.7 – Representação da estrutura do *AspectJ* (Fonte: Monteiro; Filipakis (2004))

A função do *weaver* é identificar, nos componentes, *join points* (pontos de junção) onde os aspectos se aplicam, produzindo o código final da aplicação que implementa as propriedades definidas pelos componentes e pelos aspectos [Chaves (2002)].

Os elementos principais em *AspectJ* são: *join points*, *pointcuts*, *advices*, *introductions* e *aspects* [Resende; Silva (2005), Junior; Winck (2006), Soares; Borba (2002), Garcia *et al.* (2004), Team (2000) e Team (2003)].

3.5.1. *Join points*

Os *join points* (pontos de junção) consistem nos pontos de execução de um produto de *software* de componentes nos quais ocorre a aplicação dos aspectos (conceitos abstratos em *AspectJ*, pois não possuem construção de programa para representá-los).

Conforme mostra a Figura 3.8, o modelo de *join points* do *AspectJ* baseia-se em um grafo dirigido de envio de mensagens a objetos. Os nós mostram-se como os pontos onde as classes e os objetos recebem uma invocação de método ou têm um atributo acessado. Os vértices representam as relações de fluxo de controle entre os nós, partindo daquele que chama para o chamado. O fluxo de controle ocorre nos dois sentidos: no sentido do vértice, quando a ação inicia-se, e no sentido contrário, quando a ação realizada pelo sub-nó estiver concluída [Stein (2002)]. Neste caso, o primeiro ponto de junção consiste na invocação de um método do "Objeto A", o qual pode retornar sucesso ou lançar uma exceção. O próximo ponto de junção consiste na execução deste método, que por sua vez pode retornar sucesso ou lançar uma exceção. Durante a execução do método do "Objeto A", invoca-se um método de um "Objeto B", podendo retornar sucesso ou lançar uma exceção. A invocação e a execução destes métodos representam pontos de junção [Soares; Borba (2002)].

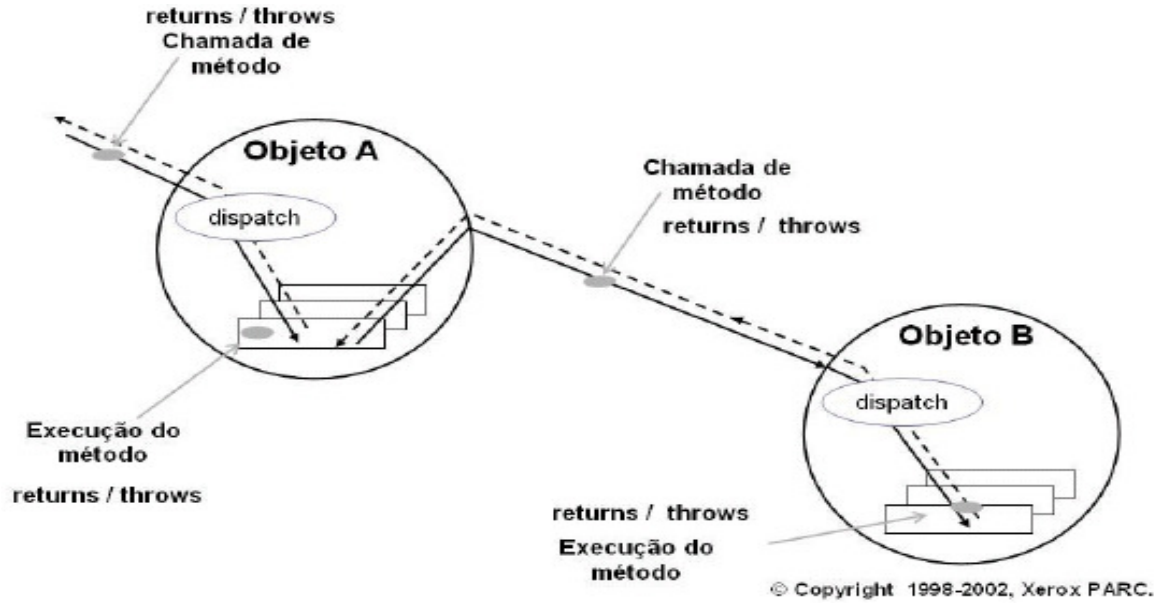

Figura 3.8 – Fluxo de controle em *AspectJ* (Fonte: Kiczales (2001))

Isto significa que um *join point* pode estar em uma das direções do fluxo de controle, ou seja, quando uma ação é iniciada e quando uma ação é terminada. Isto permite

maior exatidão quanto à aplicação dos aspectos em relação ao código-fonte de um componente qualquer.

3.5.2. *Pointcuts*

Os *pointcuts* (conjuntos de pontos de junção) são responsáveis por selecionar *join points*, ou seja, eles detectam em quais *join points* o aspecto deve realizar uma interceptação. Um *pointcut* é nomeado através de designadores (*pointcut designator* – PCD), que podem ser primitivos ou derivados (os programadores podem criar novos tipos a partir daqueles existentes). Um designador de *pointcut* primitivo abrange o tipo, a propriedade e a combinação dos *join points*. A Tabela 3.1 e a Tabela 3.2 mostram os designadores primitivos principais suportados pelo *AspectJ*.

Tabela 3.1 – Designadores de *pointcuts* baseados no tipo do *join point* (Fonte: Chaves (2002))

Designadores de *Pointcuts* (PCD)	*Join Points* selecionados (JP)
`call (<assinatura de método>)`	Quando o método é chamado
`execution (<assinatura de método>)`	Quando o método é executado
`get (<assinatura de atributo>)`	Quando o atributo é acessado
`set (<assinatura de atributo>)`	Quando o atributo é alterado
`handler (<tipo de exceção>)`	Quando a exceção é tratada
`initialization (<assinatura de construtor>)`	Quando o construtor é executado
`staticInitialization (<tipo>)`	Quando a inicialização de classe é executada

Tabela 3.2 – Designadores de *pointcuts* baseados na propriedade do *join point* (Fonte: Chaves (2002))

Designadores de *Pointcuts* (PCD)	*Join Points* selecionados (JP)
`within (Tipo)`	Qualquer JP que ocorra na classe Tipo
`withincode (Método)`	Qualquer JP que ocorra no método/construtor
`cflow (Pointcut)`	Qualquer JP que ocorra no contexto de um JP selecionado pelo PCD
`cflowbelow (Pointcut)`	Idem ao anterior, excluindo os JPs selecionados pelo próprio PCD
`this (Tipo)`	Qualquer JP que ocorra em um objeto da classe Tipo
`target(Tipo)`	Quando o objeto alvo do *call*/*get*/*set* é da classe Tipo
`args (Tipo, ...)`	Quando os argumentos são do tipo especificado
`if(ExpressãoLógica)`	Quando a expressão é verdadeira

Os *pointcuts* suportam os símbolos (elementos *wildcards*) "*", "+", e ".." para a especificação de um *join point*. Assim:

```
execution (* transferir(..))
```

seleciona os *join points* onde o método transferir, independentemente do tipo, e de zero ou mais argumentos, seja executado. Um *pointcut* é definido da seguinte forma:

```
Pointcut <Nome> ([argumentos]): <corpo>;
```

Um *pointcut* pode ser aninhado em outro *pointcut* com os operadores "e", "ou" e "não" ("&&", "||", e "!", respectivamente).

3.5.3. *Advices*

Os *pointcuts* somente selecionam os *join points*. Para implementar efetivamente os comportamentos multidimensionais, é usado o *advice* (adendo ou comportamento transversal), o qual corresponde a um trecho de código executado a cada *join point* descrito no *pointcut*. Existem três formas básicas de *advice*: *before*, *around* e *after* (Tabela 3.3). Como seus nomes sugerem, *before* executa antes do *join point*, *around* executa antes e após e *after* executa após [Resende; Silva (2005)].

Tabela 3.3 – Tipos de *advices* (Fonte: Resende; Silva (2005))

`before`	Executa quando *join point* é alcançado, mas imediatamente antes da sua computação
`around`	Executa quando *join point* é alcançado e tem total controle sobre a sua computação
`after`	Executa após a computação do *join point* em qualquer situação
`after returning`	Executa após a computação **com** sucesso do *join point*
`after throwing`	Executa após a computação **sem** sucesso do *join point*

Um *advice* é definido da seguinte forma:

```
TipoAdvice ([ListaDeParâmetros]): Pointcut {<corpo>}
```

Os *advices around* substituem o comportamento original do *join point*. Para isso, a linguagem oferece o método *proceed( )*, o qual permite que o comportamento original seja substituído e que comportamentos adicionais sejam realizados antes e após o comportamento original do *join point* selecionado [Chaves (2002)].

3.5.4. *Introductions*

AspectJ provê uma maneira de alterar a estrutura estática de uma aplicação. Isto ocorre por meio da *inter-type declaration* (declaração intertipos), que é descrita como interesse estático (*static crosscutting*). Estas declarações provêm uma construção chamada *introduction* (Tabela 3.4).

Tabela 3.4 – Construções do tipo *introdution* (Fonte: Soares; Borba (2002))

`Mofidicador Tipo PadrãoTipo.Id(Formais){<corpo>}`	Define um método nos tipos em *PadrãoTipo*
`abstract Mofidicador Tipo PadrãoTipo.Id(Formais)`	Define um método abstrato nos tipos em *PadrãoTipo*
`Mofidicador PadrãoTipo.new(Formais){<corpo>}`	Define um construtor nos tipos em *PadrãoTipo*

Os *introductions* modificam uma classe estruturalmente, acrescentando a ela novos membros, como construtores, métodos e campos, e novos relacionamentos (herança) por meio da cláusula *declare parents*. Analogamente aos *advices*, que atuam em pontos específicos do programa (denotados por *pointcuts*), *introductions* atuam sobre um conjunto de definições de classes [Stein (2002)].

3.5.5. *Aspects*

Aspects (aspectos) encapsulam *pointcuts*, *advices* e *introductions* em uma unidade modular de implementação e são definidos de maneira semelhante às classes. Enquanto estas encapsulam o código que se encaixa dentro de classes hierárquicas, *aspects* encapsulam o código que se entrelaça a essas classes hierárquicas. Embora o comportamento especificado pelos *advices* e pelos *introductions* declarados possa modificar o produto de *software*, eles são localizados no código-fonte dos aspectos.

AspectJ permite a definição de aspectos abstratos, os quais devem ser estendidos provendo a implementação do componente abstrato. Os componentes abstratos podem ser métodos, como em uma classe *Java*, e *pointcuts*, os quais devem ser definidos em um aspecto concreto, permitindo o reuso do comportamento dos aspectos abstratos.

3.6. Considerações Finais

Neste capítulo, foi apresentado o estado da arte sobre OA, abordando a sua origem e evolução, juntamente com seus conceitos, as suas ferramentas/linguagens que lhe dão suporte (ressaltando as construções em *AspectJ*) e as abordagens sobre modelagem orientada a aspectos.

Pelo exposto, OA traz vários benefícios devido à possibilidade de tornar os programas mais modulares. O objetivo da técnica é separar preocupações ortogonais (*crosscutting concerns*), de modo a evitar o entrelaçamento de código com diferentes propósitos e o espalhamento de código com propósito específico em várias partes do sistema. Desta forma, obtém-se ganhos com manutenção e evolução do produto de *software*, além de favorecer o reuso de suas partes [Fernandes (2003)].

Atualmente, a linguagem de programação *AspectJ* é a mais utilizada em POA e integrada com a linguagem de componentes *Java*. Com relação à modelagem orientada a aspectos, não existe ainda uma convenção ou uma abordagem dita padrão, tal como a UML está para a orientação a objetos, gerando a necessidade de levar as vantagens dos aspectos às etapas anteriores do processo de desenvolvimento de produtos de *software*. Isso decorreu do fato de que as pesquisas em modelagem são recentes em relação às pesquisas em implementação, além do fato do paradigma ainda não ter completado uma década, mostrando que existe muito trabalho a ser feito em uma tecnologia que mostra seu sucesso, atraindo a atenção de acadêmicos e do mercado.

4. CRITÉRIOS DE MANUTENIBILIDADE

4.1. Considerações Iniciais

Este capítulo define os critérios de manutenibilidade aplicáveis na avaliação das características de manutenibilidade do Modelo de Implementação de produtos de *software* orientados a aspectos, os quais podem ser utilizados para verificação do nível de facilidade de manutenção. As atividades para a avaliação dos critérios de manutenibilidade deste modelo devem ser inseridas no processo de desenvolvimento de produtos de *software* utilizado, uma vez que as recomendações não estão restritas a algum processo de desenvolvimento.

Na seção 2, são listados os critérios de manutenibilidade propostos com base na experiência do autor adquirida em congressos, pesquisas, artigos e discussões. Tais critérios estão separados em categorias para melhorar o entendimento e sua avaliação e são munidos de suas respectivas justificativas. A seção 3 apresenta um estudo de caso, o produto de *software* Gestão de Domínio Bancário (GDB), para automatizar as operações bancárias. Este visa ilustrar o uso dos critérios. O conceito de estudo de caso utilizado neste trabalho é aquele utilizado por Jacobson (1992), consistindo em apresentar um exemplo da aplicação dos recursos propostos, neste caso os critérios de manutenibilidade.

4.2. Critérios de Manutenibilidade

Esta seção exibe a configuração de um conjunto de critérios de manutenibilidade que devem ser atendidos pelos produtos de *software* orientados a aspectos, cujo código-fonte possui a característica de fácil manutenção.

A identificação de tais critérios foi suportada pela definição de manutenibilidade conforme consta na norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)] e nas boas técnicas programação para a construção de produtos de *software* com a característica de manutenibilidade.

Os critérios de manutenibilidade para o Modelo de Implementação identificados e definidos foram processados da união entre a experiência do autor (participação em congressos e discussões no núcleo de estudos do Grupo PqES – Pesquisa em Engenharia de *Software* do Departamento de Ciência da Computação da Universidade Federal de Lavras) com a literatura disponível, na qual buscou-se por: uso de métricas, convenções de

programação, bom estilo de programação e *guidelines* de programação. Alguns autores apresentaram grandes contribuições a esta pesquisa: Sun Microsystems (1999), Piveta *et al.* (2005), Soares; Borba (2002), Garcia *et al.* (2004), Junior; Winck (2006), Resende; Silva (2005), Costa (2005), Souza (2003), Hugo; Grott (2005), Kiczales (2001) e Camargo; Masiero (2005).

No decorrer da seção, os critérios de manutenibilidade são referenciados por CM_MI (**C**ritério de **M**anutenibilidade para o **M**odelo de **I**mplementação). Os CM_MI são apresentados conforme a estrutura (*template*):

- subseção: organiza os CM_MI segundo a sua natureza, a saber: Armazenamento, Comentário, Estrutura, Formato, Localização, Lógica e Requisitos;
- critério: identifica o CM_MI;
- justificativa: justifica o uso do CM_MI;
- subcaracterística(s): lista a(s) subcaracterística(s), da característica de manutenibilidade, definida(s) na norma ISO/IEC 9126 [ISO Std. 9126 (1991), ISO Std. 9126 (2001) e ABNT NBR13596 (1996)], as quais o CM_MI visa a melhorar, quando o critério é aplicado.

4.2.1. Armazenamento

São exigências relativas ao armazenamento do estado dos objetos em um meio persistente. A seguir, são apresentados os CM_MI que visam atender a essas exigências.

CM_MI 1 – O objeto de uma classe persistente, quando instanciado, tem seu estado armazenado em um meio persistente através de um aspecto.

Justificativa: Armazena as informações que estão em memória para uso posterior, devido à necessidade de manter um registro no meio persistente. A persistência é um elemento importante em sistemas computacionais que aparece espalhado através do código do produto de *software*. Logo, independente de como e quais são os componentes que devem ser tornados persistentes, pode ser feita a composição entre o aspecto e os objetos.
Subcaracterística(s): Analisabilidade, Modificabilidade e Testabilidade.

CM_MI 2 – O objeto de uma classe persistente, quando sofre alterações no seu estado, tem o seu estado atualizado no meio persistente através de um aspecto.

Justificativa: Mantém consistentes as informações que estão em memória com as informações no meio persistente. A persistência é um elemento importante em sistemas computacionais que aparece espalhado através do código do produto de *software*. Logo, independente de como e quais são os componentes que devem ser tornados persistentes, pode ser feita a composição entre o aspecto e os objetos.
Subcaracterística(s): Analisabilidade, Modificabilidade e Testabilidade.

CM_MI 3 – O objeto de uma classe persistente, quando excluído, tem o seu estado excluído do meio persistente através de um aspecto.

Justificativa: Mantém consistentes as informações que estão em memória com as informações no meio persistente. A persistência é um elemento importante em sistemas computacionais que aparece espalhado através do código do produto de *software*. Logo, independente de como e quais são os componentes que devem ser tornados persistentes, pode ser feita a composição entre o aspecto e os objetos.
Subcaracterística(s): Analisabilidade, Modificabilidade e Testabilidade.

4.2.2. Comentário

São exigências relativas à colocação de comentários estratégicos e/ou táticos. A seguir, são apresentados os CM_MI que visam atender a essas exigências.

CM_MI 4 – Os arquivos físicos (aspectos físicos[8]) têm comentário introdutório que fornece informações sobre o nome do arquivo, o conteúdo, o autor, os aspectos coesos, as classes atingidas pelo aspecto, versão, localização e outras informações relevantes para a caracterização do aspecto.

Justificativa: Facilita o entendimento do código-fonte presente no arquivo, uma vez que permite a identificação de quais classes e aspectos são interceptados pelo aspecto presente no arquivo.
Subcaracterística(s): Analisabilidade.

CM_MI 5 – Comentários descritivos antecedem *advices* e métodos auxiliares com funcionalidade complexa.

Justificativa: Facilita o entendimento de *advices* e métodos auxiliares complexos.
Subcaracterística(s): Analisabilidade.

4.2.3. Estrutura

São exigências relativas à disposição e à ordem das partes constituindo um todo. A seguir, são apresentados os CM_MI que visam a atender a essas exigências.

CM_MI 6 – Os elementos componentes estruturais de um aspecto são dispostos de maneira padronizada para facilitar a sua localização.

Justificativa: Facilita o entendimento do aspecto e a localização de seus elementos através de uma organização interna padronizada. A seção *public* é de interesse para um usuário do aspecto. A seção *protected* pode ser de interesse para os projetistas, quando consideram herança de aspectos. A seção *private* contém detalhes de interesse para o projetista deste aspecto. Assim, a estrutura interna de um aspecto deve estar organizada de uma forma padronizada. A Figura 4.1 apresenta uma sugestão de padronização:

- as constantes, utilizadas pelos elementos do aspecto;
- os atributos de aspecto, tanto aqueles cujos valores são únicos para as instâncias deste aspecto, quanto aqueles cujas instâncias possuam seu próprio conjunto de valores para estes atributos;

[8] Nesse contexto, um aspecto físico é aquele que se encontra implementado em um arquivo-fonte, em uma linguagem de programação adequada.

- o construtor[9] do aspecto, caso necessário;
- os *pointcuts* do aspecto, responsáveis por declarar os *join points*;
- os *advices* do aspecto, responsáveis por inserir novos comportamentos no produto de *software*;
- os métodos acessores, responsáveis em atribuir ou recuperar o valor de um atributo;
- os métodos *public*, que podem ser utilizados nos métodos de outros aspectos;
- os métodos *protected*, utilizados apenas nos outros métodos do próprio aspecto e em métodos de seus subaspectos;
- os métodos *private*, utilizados apenas nos outros métodos da próprio aspecto.

Constantes
Atributos de Aspecto
Construtor
Pointcuts
Advices
Métodos Acessores
Métodos *Public*
Métodos *Protected*
Métodos *Private*

Figura 4.1 – Organização interna de um aspecto

Subcaracterística(s): Analisabilidade.

CM_MI 7 – Os elementos componentes estruturais de um *advice* estão dispostos de maneira a facilitar a sua localização.

Justificativa: Facilita o entendimento do *advice* e a localização de seus elementos através de uma organização interna padronizada. Para isso, o *advice* deve estar estruturado de uma forma padronizada. A Figura 4.2 apresenta uma sugestão de padronização::

Variáveis/Objetos
Comandos
Retorno do *advice* (caso exista)

Figura 4.2 – Organização interna de um *advice*

- variáveis/objetos, utilizados apenas pelo *advice*;
- comandos, que correspondem ao corpo do *advice* (desempenham a sua funcionalidade);
- retorno do *advice*, caso seja do tipo *around* e o método interceptado pelo *pointcut* correspondente retorne algum valor.

Subcaracterística(s): Analisabilidade.

[9] Aspectos não são instanciados através do operador *new* (não são criados explicitamente pelo programador). São instanciados pelo próprio *weaver* (nome que se dá aos compiladores de linguagens orientadas a aspectos). O construtor em orientação a aspectos tem a função de prover mecanismos para controle de instanciação de aspectos [Resende; Silva (2005)].

CM_MI 8 – Os *advices* de um aspecto estão agrupados de acordo com a sua natureza (tipo). Por exemplo: *advices before*, *advices around*, *advices after*, *advices after returning* e *advices after throwing*.

Justificativa: Facilita a localização de um *advice* com funcionalidade específica, dentro de um aspecto, uma vez que os *advices* estão previamente agrupados de acordo com seu tipo. Essa prática é melhor do que posicionar *pointcuts* próximos aos seus *advices*, pois um *advice* pode tratar vários *pointcuts* e isso tornaria difícil tal tarefa. Para isso, o conjunto de *advices* deve estar estruturado de uma forma padronizada. A Figura 4.3 apresenta uma sugestão de padronização:

advices before
advices around
advices after
advices after returning
advices after throwing

Figura 4.3 – Organização interna dos *advices* em um aspecto

- *advices before*, executado após o *join point* ser alcançado, mas imediatamente antes de seu processamento;
- *advices around*, executado quando o *join point* é alcançado e tem total controle sobre o fluxo de execução do programa;
- *advices after*, executado após o *join point* ser alcançado, tendo ocorrido ou não exceção;
- *advices after returning*, executado após o método interceptado rodar com sucesso, ou seja, sem ocorrência de exceção;
- *advices after throwing*, executado após o método interceptado rodar sem sucesso, ou seja, ocorreu uma exceção.

Subcaracterística(s): Analisabilidade.

CM_MI 9 – Os comandos nos *advices* estão dispostos um em cada linha.

Justificativa: Permite, ao responsável pela manutenção do produto de *software*, a identificação mais rápida dos pontos a serem alterados e a modificação da linha de código.
Subcaracterística(s): Analisabilidade e Modificabilidade.

CM_MI 10 – A declaração dos atributos dos aspectos e das variáveis dos *advices* é posicionada uma em cada linha.

Justificativa: Dispondo os atributos por linha na declaração, permite identificar rapidamente os atributos que são utilizados, bem como viabiliza a colocação de comentários sobre o atributo à sua direita.
Subcaracterística(s): Analisabilidade.

CM_MI 11 – A declaração dos *pointcuts* dos aspectos é posicionada uma em cada linha, de forma que *pointcuts* extensos sejam divididos em *pointcuts* simples, os quais são agrupados em um novo *pointcut* geral formado a partir dos *pointcuts* simples.

Justificativa: Declarando um *pointcut* para cada *join point* e dispondo-os um por linha, é possível identificar rapidamente os *pointcuts* que são utilizados, além de facilitar o

entendimento, pois cada qual não ocupa mais do que uma linha de código. Assim, quando existir um *pointcut* formado de vários *join points*, cada um destes estaria declarado em um *pointcut* simples, os quais formariam um *pointcut* composto através de operadores lógicos para satisfazer a necessidade da aplicação.
Subcaracterística(s): Analisabilidade.

CM_MI 12 – Uma linha em branco separa blocos de comandos logicamente coesos.

Justificativa: A segmentação do código, utilizando linhas em branco e agrupando linhas de código logicamente relacionadas próximas, aumenta a legibilidade do código-fonte. Dessa forma, devem ser consideradas separações, por exemplo, entre os *advices*, entre as variáveis locais do mesmo tipo abstrato de dados nos métodos, entre as variáveis locais e a primeira linha de código nos *advices*, antes de comentários e entre os trechos logicamente ligados.
Subcaracterística(s): Analisabilidade.

CM_MI 13 – *Advices* possuem poucos parâmetros formais.

Justificativa: Os *advices* com poucos parâmetros formais facilitam a sua leitura e seu entendimento. Aplicando-se o correspondente CM_MI 14 para produtos de *software* orientados a objetos [Costa (2005)], este critério será respeitado, uma vez que os parâmetros que podem estar presentes nos *advices* são aqueles relacionados aos métodos das classes, onde são identificados os *join points*.
Subcaracterística(s): Analisabilidade, Modificabilidade e Testabilidade.

4.2.4. Formato

São exigências relativas à construção do identificador dos elementos presentes em um produto de *software*. A seguir, são apresentados os CM_MI que visam atender a essas exigências.

CM_MI 14 – O identificador de aspectos, *pointcuts* e *introductions* permite rápido reconhecimento destes elementos.

Justificativa: Facilita a identificação dos tipos dos elementos que compõem o produto de *software* orientado a aspectos pelo responsável da manutenção, através de um padrão para a construção de seus identificadores. Ao distinguir os identificadores de um aspecto, de um *pointcut* e de um *introduction*, o trabalho do responsável pela manutenção do produto de *software* é facilitado. Um sugestão de padronização é apresentada nas sentenças a seguir. O identificador de um aspecto é iniciado com a letra A, a primeira letra maiúscula e as palavras que seguem são iniciadas com letras maiúsculas e as demais letras são minúsculas. O identificador de um *pointcut* é composto com a primeira letra minúscula e as palavras que seguem a primeira são iniciadas com letras maiúsculas e as demais letras são minúsculas (não ter espaço entre elas). O identificador de um aspecto que contém um *introduction* é iniciado com a seqüência AI, a primeira letra maiúscula e as palavras que seguem a primeira são iniciadas com letras maiúsculas e as demais letras são minúsculas (não ter espaço entre elas).
Subcaracterística(s): Analisabilidade.

4.2.5. Localização

São exigências relativas à determinação da posição dos elementos no código-fonte. A seguir, são apresentados os CM_MI que visam atender a essas exigências.

CM_MI 15 – Os testes realizados no produto de *software* para a sua liberação estão disponíveis em um aspecto.

Justificativa: Permite que a tarefa do responsável pela manutenção do produto de *software* seja menos árdua, disponibilizando os testes realizados durante o desenvolvimento, uma vez que eles serviram para verificar e validar a funcionalidade do produto de *software*. Dessa forma, durante a manutenção, após serem realizadas as correções e/ou evoluções do produto de *software*, não é preciso formular novamente os testes para averiguar se houve descaracterização, pois os testes originais estão disponíveis. No caso de evolução, podem ser incorporados novos testes (complementares) para verificar e validar a sua nova funcionalidade. No caso de correção, os testes existentes, para se adequarem às modificações, podem ser alterados. Neste sentido, modificações são realizadas e testadas mais rapidamente, pois existe uma bateria de testes para ser aplicada, a fim de verificar os possíveis efeitos colaterais e validar a manutenção efetuada. Concentrar tais testes em um aspecto evita que os métodos de testes estejam inseridos em classes do produto de *software* e permite que, através dos conceitos de *pointcut*, *advice* e *introduction*, os aspectos atuem diretamente no código do produto de *software* para testar as funcionalidades.
Subcaracterística(s): Testabilidade.

CM_MI 16 – A localização de retorno de um objeto em um *advice* está na sua última linha de código.

Justificativa: Padroniza a localização do comando de retorno do objeto no *advice* (para *advices* do tipo *around* que retornam objeto). Com isso, permite ao responsável pela manutenção do produto de *software* achar facilmente o local de retorno do *advice*, bem como percorrer todo o *advice* durante a depuração.
Subcaracterística(s): Analisabilidade e Testabilidade.

CM_MI 17 – A responsabilidade de verificar as pré-condições[10] para iniciar a execução de um caso de uso[11] é destinada a um aspecto.

Justificativa: Separa as pré-condições necessárias ao desempenho de um caso de uso da implementação deste caso de uso, visando retirar do código funcional do produto de *software* responsabilidades de verificação do seu estado para a ocorrência de uma operação, e coloca tal tarefa em um aspecto. Dessa forma, ao receber uma requisição para realizar uma manutenção, o responsável identifica se esta requisição é uma situação em que o produto de *software* deve estar antes da execução do caso de uso ou se é uma falha na execução do caso de uso. Feito isso, ele direciona esforços para o aspecto específico que trata uma das situações.
Subcaracterística(s): Analisabilidade, Estabilidade, Modificabilidade e Testabilidade.

[10] Pré-condições são propriedades que devem ser verdadeiras antes da execução de um caso de uso. Define qual o resultado deve ser obtido e lança exceções quando as mesmas não são atendidas.
[11] Um caso de uso especifica o comportamento do sistema ou parte de um sistema e é uma descrição de um conjunto de seqüências de ações, incluindo variantes, que o sistema desempenha para obter um resultado significativo para o ator.

CM_MI 18 – A responsabilidade de verificar as pós-condições[12] para encerrar a execução de um caso de uso é destinada a um aspecto.

Justificativa: Separa as pós-condições necessárias para o desempenho de um caso de uso da implementação deste caso de uso, visando retirar do código funcional do produto de *software* responsabilidades de verificação do seu estado após a ocorrência de uma operação, e coloca tal tarefa em um aspecto. Dessa forma, ao receber uma requisição para realizar uma manutenção, o responsável identifica se esta requisição é uma falha na execução do caso de uso ou se é uma situação em que o produto de *software* deve estar após a execução do caso de uso. Feito isso, ele direciona esforços para o aspecto específico que trata uma das situações.
Subcaracterística(s): Analisabilidade, Estabilidade, Modificabilidade e Testabilidade.

CM_MI 19 – A validação de uma nova associação, através da verificação da cardinalidade de um relacionamento, é realizada por um aspecto.

Justificativa: Evita que outras responsabilidades, além da funcionalidade do produto de *software*, estejam espalhadas no seu código funcional. Os *pointcuts* do aspecto interceptarão os métodos responsáveis em estabelecer e em desfazer uma associação. Os seus respectivos *advices*, utilizando as constantes que representam a cardinalidade do relacionamento, podem permitir ou não esta nova associação.
Subcaracterística(s): Modificabilidade.

CM_MI 20 – Um aspecto que contém *introduction* (declaração inter-tipo) não contém nada além desta declaração.

Justificativa: Facilita o entendimento e a manutenção do código, além de evitar riscos e efeitos inesperados. Como um *introduction* é utilizado para alterar a estrutura da classe, introduzindo novos atributos, métodos, construtores, *getters*, *setters*, herança e interface, deve estar presente em um aspecto como seu conteúdo único.
Subcaracterística(s): Analisabilidade, Estabilidade, Modificabilidade e Testabilidade.

4.2.6. Lógica

São exigências relativas à seqüência coerente, regular e necessária de acontecimentos. A seguir, são apresentados os CM_MI que visam atender a essas exigências.

CM_MI 21 – Arquivos físicos contêm uma, e somente uma, definição de aspecto.

Justificativa: Facilita para quem usa e/ou mantém os aspectos, pois o identificador do arquivo físico corresponde ao nome do aspecto definido neste arquivo. Caso contrário, no mesmo arquivo físico, teria mais de uma definição de aspecto e, possivelmente, o identificador do arquivo físico não conseguiria representar os aspectos definidos.
Subcaracterística(s): Analisabilidade, Modificabilidade e Testabilidade.

[12] Pós-condições são propriedades que devem ser verdadeiras após a execução de um caso de uso. Define qual o resultado deve ser obtido e lança exceções quando as mesmas não são atendidas.

CM_MI 22 – Os atributos de um aspecto são caracterizados como privados ao aspecto.

Justificativa: Caracteriza uma extensão de uma das principais propriedades do paradigma de orientação a objetos, o encapsulamento. A implementação de um aspecto não deve ser uma preocupação para seus usuários.
Subcaracterística(s): Estabilidade.

CM_MI 23 – O método construtor de um aspecto possui duas, e somente duas, funções. Uma delas é instanciar um aspecto (a depender do tipo definido pelo programador) e a outra é atribuir valores iniciais aos atributos do aspecto instanciado (estado inicial).

Justificativa: Conserva a semântica do método construtor. O fato de atribuir mais responsabilidade, além das mencionadas, diverge da sua semântica. Além disso, os aspectos não são instanciados explicitamente pelo programador, mas pelo *weaver* (combinador).
Subcaracterística(s): Estabilidade.

CM_MI 24 – Os *advices* identificados são simples e com linhas de código suficientes para ter apenas uma função (tarefa).

Justificativa: Torna menos árdua a manutenção, pois *advices* extensos e complicados dificultam as alterações do código e o seu entendimento.
Subcaracterística(s): Analisabilidade, Modificabilidade e Testabilidade.

CM_MI 25 – É declarada a precedência em aspectos quando mais de um aspecto apresentam *pointcuts* que interceptam os mesmos *join points*.

Justificativa: Facilita o entendimento de situações onde o mesmo *join point* em uma classe é interceptado por mais de um *pointcut* e precisa-se entender a dinâmica de execução para verificação do produto de *software*. Dessa forma, deve ser explicitada uma ordem de precedência para evitar riscos durante a execução do produto de *software*.
Subcaracterística(s): Analisabilidade e Testabilidade.

CM_MI 26 – *Pointcuts* ou *advices* duplicados são refatorados[13].

Justificativa: Diminui a quantidade de duplicação de código, o que corresponde a uma das motivações do desenvolvimento de produtos de *software* orientados a aspectos. Ao prover mecanismos de abstração para a modularização de interesses transversais, a tendência é a duplicação ser reduzida. Caso a duplicação de código ocorra em diferentes *advices* de um mesmo aspecto, o código repetido deve ser extraído para um método que será chamado por estes *advices*. Caso o código dos *advices* seja idêntico, a combinação de *pointcuts* é possível, removendo o *advice* duplicado. Caso a duplicação ocorra em aspectos que possuem o mesmo super-aspecto, através do mecanismo de herança, a estrutura duplicada deve ser movida para os níveis acima na hierarquia. Outro problema relacionado é a definição de vários *pointcuts* ser parecida, com mudanças apenas em modificadores ou pequenas partes de sua declaração. Isso faz com que quando um *pointcut* for modificado, o mesmo deva ser feito em todos os outros. Dessa forma, deve-se extrair um novo *pointcut* a

[13] Refatorações são transformações de código-fonte utilizadas para melhorar o projeto de um *software* sem modificar o comportamento da aplicação [PIVETA *et al.* (2005)].

partir daqueles semanticamente iguais para evitar repetição de código e maior custo para a manutenção. Possíveis estruturas duplicadas: atributos, métodos, *advices*, *pointcuts* e *introductions*.
Subcaracterística(s): Analisabilidade, Modificabilidade e Estabilidade.

CM_MI 27 – Cada aspecto deve tratar um interesse específico.

Justificativa: Melhora o encapsulamento de interesses, contribuindo para a manutenibilidade do produto de *software*. Quando um aspecto tenta lidar com mais de um interesse (mesmo que seja do mesmo tipo), este deve ser dividido em tantos aspectos quantos forem os interesses. Normalmente, isto ocorre com a definição de *advices* com propósitos diferentes ou outros tipos de estruturas (por exemplo, atributos, *introductions*). Caso os membros do aspecto correspondam a interesses diferentes que possam existir em uma classe (ou em um aspecto), ele deve ser extraído para uma classe (ou para um aspecto). Quando diferentes interesses podem ser separados através de herança, devido à afinidade dos interesses tratados, eles podem ser extraídos na forma de sub-aspectos, a partir do aspecto maior.
Subcaracterística(s): Analisabilidade, Modificabilidade e Testabilidade.

CM_MI 28 – Os aspectos com poucas responsabilidades ou com generalidade especulativa são analisados para verificar possíveis mudanças ou sua eliminação.

Justificativa: Contribui para a simplicidade e clareza do código do produto de *software*. Quando um aspecto tem poucas responsabilidades, é possível que sua eliminação possa proporcionar benefícios na etapa de manutenção. Essa diminuição de responsabilidade está relacionada a modificações adicionadas em função de alterações que foram planejadas/previstas, mas que não ocorreram. Dessa forma, pode-se reduzir, por exemplo, a hierarquia, movendo membros de um aspecto para os seus sub-aspectos ou dos sub-aspectos para o super-aspecto. Aspectos vazios ou não utilizados podem ser removidos. Além disso, quando aspectos são criados para lidar com requisitos futuros, devido à facilidade de postergar algumas decisões do projeto, é possível remover a funcionalidade não utilizada. Pode-se, também, remover parâmetros que não usados no *advice* e renomear aspectos e *pointcuts* para melhorar a legibilidade.
Subcaracterística(s): Analisabilidade, Modificabilidade e Estabilidade.

CM_MI 29 – Os *pointcuts* são nomeados.

Justificativa: Favorece o entendimento e o reuso do *pointcut* para definir outros *pointcuts* e *advices*. Assim, a definição do *pointcut* não é usada diretamente no *advice*, não ocultando as intenções deste e evitando redundância de código, diminuição da legibilidade e prejuízo aos benefícios da orientação a aspectos. Caso contrário, pode se tornar necessário recorrer à descrição contida no *pointcut* anônimos (não-nomeado) para obter uma idéia dos pontos afetados pelo *advice*.
Subcaracterística(s): Analisabilidade e Modificabilidade.

CM_MI 30 – Em *AspectJ*, *pointcuts* declarados em classes são analisados para verificar sua mobilidade para um aspecto.

Justificativa: Diminui o acoplamento entre a classe e o aspecto e aumenta a coesão do aspecto. Em *AspectJ*, é permitido usar *pointcuts* em classes. Caso este *pointcut* seja utilizado por apenas um aspecto, é interessante que este seja movido para o aspecto que o

utiliza. Este mesmo problema pode ocorrer em classes, em situações tais que um método de uma classe faz referência aos atributos e aos métodos de outra classe ao invés de referenciar os membros da classe que o contém.
Subcaracterística(s): Analisabilidade, Estabilidade, Modificabilidade e Testabilidade.

CM_MI 31 – A inclusão de métodos abstratos via *introductions* é avaliada visando ser evitada.

Justificativa: Evita problemas advindos de hierarquia. Aspectos podem ser utilizados de forma a adicionar estado e comportamento às classes existentes através de *introductions*. Entretanto, o uso desta funcionalidade pode causar danos ao produto de *software*, caso sejam inseridos métodos abstratos nas classes da aplicação. Esta introdução força o desenvolvedor a prover implementações concretas para os métodos introduzidos nas classes afetadas e/ou sub-classes. Esta dependência aumenta de forma desnecessária o acoplamento entre o aspecto e as classes afetadas e a complexidade da solução utilizada, uma vez que implica em uma sub-classe de uma classe afetada ser criada. Implementações para os métodos abstratos inseridos pelo aspecto devem ser providas.
Subcaracterística(s): Analisabilidade, Estabilidade, Modificabilidade e Testabilidade.

4.2.7. Requisitos

São exigências relativas ao tratamento dos requisitos de um produto de *software*, de forma a identificar aspectos. A seguir, são apresentados os CM_MI que visam atender a essas exigências.

CM_MI 32 – Requisitos não-funcionais tais como distribuição, controle de concorrência, tratamento de exceções, depuração, sincronização de objetos concorrentes, coordenação de múltiplos objetos, serialização, atomicidade, replicação, segurança, visualização, *logging*, *tracing* e tolerância a falhas são implementados como aspectos.

Justificativa: Favorece o entendimento do produto de *software* e o reuso do aspecto. Além disso, dispõe o código de forma mais organizada e clara, uma vez que evita o espalhamento e o entrelaçamento de código relativo a interesses não-funcionais ao longo do código funcional do produto de *software*.
Subcaracterística(s): Analisabilidade, Estabilidade, Modificabilidade e Testabilidade.

CM_MI 33 – Requisitos funcionais, os quais implementam regras de negócio que ficam espalhadas e/ou entrelaçadas com a funcionalidade básica do produto de *software* (quando em uso de técnicas orientadas a objetos), são implementados como aspectos.

Justificativa: Favorece o entendimento do produto de *software* e o reuso do aspecto. Além disso, dispõe o código de forma mais organizada e clara, uma vez que evita o espalhamento e o entrelaçamento de código relativo a interesses funcionais repetitivos ao longo do código funcional do produto de *software*. Um exemplo pode ser visto em [Camargo; Masieiro (2005)], nos requisitos não-funcionais "reajuste de valor" e "cálculo baseado em tabela" (imposto de renda e vale-refeição, por exemplo), os quais são *frameworks* usados em um produto de *software* de oficina de aparelhos eletrônicos.
Subcaracterística(s): Analisabilidade, Estabilidade, Modificabilidade e Testabilidade.

4.3. Estudo de Caso

O aspecto permite que mudanças, quando necessárias, sejam definidas no código-fonte de um produto de *software* de forma menos invasiva. Dessa forma, as limitações da orientação a objetos são superadas através deste novo paradigma que não tem a pretensão de substituir esta metodologia bem sedimentada, mas complementar e melhorar para que problemas sejam resolvidos e novos desafios surjam, tornando a Engenharia de *Software* cada vez mais dinâmica.

Dessa forma, aliar estes benefícios de programação com pesquisas sobre manutenção é uma estratégia bem interessante, uma vez que esta consome mais da metade dos recursos financeiros alocados a um projeto.

Assim sendo, esta seção apresenta um estudo de caso, que visa a verificar os critérios de manutenibilidade, propostos na seção anterior, no desenvolvimento de um produto de *software* orientado a aspectos manutenível.

4.3.1. Descrição do Produto de *Software*

O produto de *software* Gestão de Domínio Bancário (GDB) gerencia operações requeridas por uma agência bancária. As operações (requisitos funcionais) consistem em: i) efetuar *login* e *logoff*; ii) cadastrar e remover clientes; iii) alterar senha no sistema; iv) ver saldo; v) realizar transferência; e vi) efetuar depósito e saque. Os usuários do GDB são classificados em dois tipos: i) o administrador, o qual tem permissões para realizar as operações descritas; e ii) o cliente, o qual apenas pode efetuar *login* e *logoff*, ver saldo, alterar senha e efetuar transferência de sua conta para outras. Para ilustrar, a Figura 4.4 apresenta o Diagrama de Caso de Uso para o GDB.

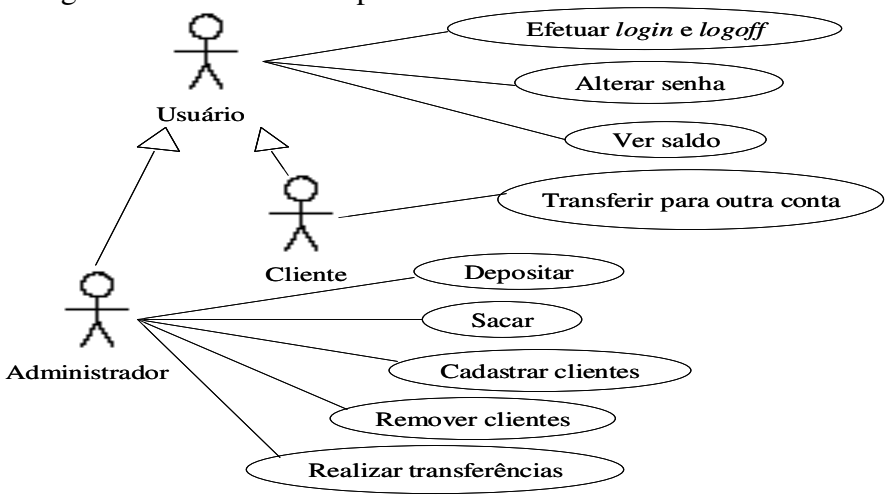

Figura 4.4 – Diagrama de Caso de Uso para o GDB

Como requisitos não funcionais, o GDB deve realizar autenticação do usuário com relação à funcionalidade disponível a ele (de acordo com seu tipo), cuidar do controle de fluxo de execução do produto de *software* (*tracing*), armazenar um histórico dos acessos em um banco de dados (*logging*), efetuar controle de transação, de forma a garantir a veracidade dos dados (operação de transferência) e dar suporte ao tratamento de exceções.

4.3.2. Aplicação dos Critérios de Manutenibilidade

Nesta subseção, é exemplificada a aplicação dos critérios de manutenibilidade sobre o produto de *software* GDB. Em alguns dos exemplos, vários critérios podem ser verificados, realçando apenas alguns deles visando a apresentar um maior número de exemplos. A intenção é avaliar o uso dos critérios de manutenibilidade.

A coerência entre objetos em memória e no meio persistente quando da sua instanciação, alteração ou exclusão deve ser respeitada para melhorar a manutenibilidade do produto de *software*. Como esta característica encontra-se normalmente espalhada ao longo do código-fonte, encapsulá-la em um aspecto é uma boa prática de programação. Os CM_MI 1, CM_MI 2 e CM_MI 3, os quais tratam esta prática, são verificados no aspecto `APersistencia`. Para melhorar o entendimento deste aspecto, o CM_MI 4 e CM_MI 6 são atendidos, através do comentário introdutório e disposição interna dos elementos componentes (*template* da Figura 4.1), respectivamente. Além disso, os *advices* possuem poucos parâmetros formais, o que verifica o CM_MI 13. Ainda neste aspecto, existe um construtor, o qual satisfaz o CM_MI 23. Este exemplo pode ser visualizado na Figura 4.5.

Para melhorar a legibilidade, os elementos fornecedores de comportamento transversal – *advices* – devem ser bem estruturados. No aspecto `ATransacoes`, existe um *advice* de funcionalidade complexa, o qual está documentado com um comentário inicial, além de seguir uma disposição interna concisa (*template* da Figura 4.2) e apresentar o retorno ao final de seu código. Assim, o CM_MI 5, CM_MI 7 e CM_MI 16 são atendidos, respectivamente, o que pode ser percebido na Figura 4.6.

No aspecto `ALogging`, os CM_MI 8, CM_MI 9, CM_MI 10, CM_MI 11, CM_MI 12, CM_MI 22, CM_MI 24 e CM_MI 29 são atendidos (Figura 4.7 e Figura 4.8). De forma sucinta: os *advices* estão agrupados de acordo com o seu tipo e seguindo o *template* da Figura 4.3, facilitando sua localização, além do fato de seus comandos estarem dispostos um por linha, melhorando a legibilidade e o entendimento, e de cada *advice* ser sintético o

suficiente para executar sua tarefa; os atributos no aspecto e as variáveis em seus *advices* estão dispostos um por linha, com linhas em branco separando trechos de comando coesos; o *pointcuts* são nomeados, a fim de ampliar a compreensibilidade dos aspectos, além do fato de que suas respectivas declarações estão dispostas uma por linha, com *pointcuts* simples (tais como `verSaldo()`e `transferencia()`) compondo *pointcuts* mais complexos (neste caso, `todosOsMetodos()`); e os atributos do aspecto são todos privados, o que representa uma boa prática de programação visando manutenção.

```
/*
 * APersistencia.aj
 *
 * Aspecto responsável pelo armazenamento de objetos de classes persistentes em
 * um meio persistente.
 *
 * Autor: Rodrigo Pereira dos Santos
 *
 * Aspectos relacionados: Nenhum relacionamento com outros aspectos.
 *
 * Classes atingidas: Banco; Conta; Usuário.
 *
 * ver. 1.01
 *
 * Localização: package aspectos;
 */
public aspect APersistencia {

	private IRepositorioContas repositorioContas;
	private IRepositorioUsuarios repositorioUsuarios;

	public APersistencia() {
		repositorioContas = new RepositorioContas();
		repositorioUsuarios = new RepositorioUsuarios();
	}

	pointcut metodoAlterarSenha() : withincode(void Banco.alterarSenha(..));
	pointcut novaConta(Conta conta) : ((execution(Conta.new(String, double, boolean))
		&& target(conta)));
	pointcut alterarSenha(Usuario usuario) : (call(* Usuario.setSenha(..))
		&& metodoAlterarSenha() && target(usuario));
	pointcut excluirCliente(Usuario usuario) : (execution(* Banco.excluirCliente(Usuario))
		&& args(usuario));

	after(Conta conta) : novaConta(conta) {
		try {
			repositorioContas.inserir(conta);
		} catch (SQLException e) {
			e.printStackTrace();
		}
	}

	after(Usuario usuario) : alterarSenha(usuario) {
		try {
			repositorioUsuarios.alterar(usuario);
		} catch (SQLException e) {
			e.printStackTrace();
		}
	}

	after(Usuario usuario) : excluirUsuario(usuario) {
		try {
			repositorioUsuarios.excluir(usuario);
		} catch (SQLException e) {
			e.printStackTrace();
		}
	}
}
```

Figura 4.5 – Código-fonte do aspecto *APersistencia*

```
/*
 * ATransacoes.aj
 *
 * Aspecto responsável pelo controle transacional da aplicação.
 *
 * Autor: Rodrigo Pereira dos Santos
 *
 * Aspectos relacionados: Relaciona-se com o aspecto TransacoesBancarias, o qual herda
 * as características e provê implementação para os métodos abstratos deste aspecto.
 *
 * Classes atingidas: RepositorioContas; ConexaoBD.
 *
 * ver. 1.01
 *
 * Localização: package aspectos;
 */
public abstract aspect ATransacoes {
        .
        .
        .
        /*
         * Este advice tem por finalidade desabilitar a confirmação automática
         * de operações do banco de dados. Isto é feito passando o valor false
         * como parâmetro para o método setAutoCommit() da classe Connection.
        */
        Connection around() throws SQLException :
                (obterConexao() && cflow(operacaoTransacionada())) {

                conn = proceed();
                conn.setAutoCommit(false);

                return conn;
        }
        .
        .
        .
}
```

Figura 4.6 – Código-fonte do aspecto *ATransacoes*

Uma das práticas que interfere na manutenção é o uso de identificadores para diferentes elementos de POA. Dessa forma, o aspecto abstrato `AIUsuario` mostra que o CM_MI 14 é atendido, uma vez que ele permite a rápida identificação de *pointcuts* e *introductions* (além do aspecto em si). Além disso, o aspecto atende ao CM_MI 20, ao apresentar um *introduction* responsável pela herança entre classes do produto de *software* como único conteúdo de tal aspecto. Isso está presente na Figura 4.9.

Costa (2005) propôs que testes realizados em produtos de *software* para a sua liberação estivessem disponíveis em métodos das próprias classes-alvo, para POO. Como os conceitos de orientação a aspectos vão além, os mesmos métodos de testes agora acabam por atuar sobre mais classes e aspectos simultaneamente. A idéia é encapsular testes em métodos dentro de aspectos relacionados, de forma a retirar responsabilidades desnecessárias das classes. Assim, o aspecto `ATracing`, por exemplo, respeita ao CM_MI 15, através de seu método `testeTracing()`. Além disso, este mesmo aspecto é único no arquivo `ATracing.aj`, o que atende ao CM_MI 21. Como os *pointcuts* do aspecto atuam em *join points* nos quais *pointcuts* de outros aspectos também atuam, a declaração de precedência deve ser utilizada para garantir a execução correta da funcionalidade do

produto de *software*. Isso faz com que o CM_MI 25 seja atendido. O CM_MI 26 é garantido: ao invés de utilizar dois métodos de rastreamento, `traceEntry(String str)` e `traceExit(String str)`, de mesmo conteúdo e correspondentes aos dois *advices* (*before* e *after*), o código dos métodos é extraído, eliminando-os e criando apenas o método `trace(String str)`, o qual recebe o conteúdo a ser exibido. O aspecto é não é complexo, o que atende ao CM_MI 27. O próprio aspecto acata ao CM_MI 32, uma vez que o rastreamento (*tracing*) é um requisito não-funcional espalhado ao longo do código, quando da implementação orientada a objetos. Por fim, durante a construção do GDB, este aspecto era do tipo *abstract*, com *pointcuts abstratc*. Como apenas um sub-aspecto seria implementado para rastrear as chamadas a métodos na aplicação, não havia a necessidade de estender o aspecto `ATracing` apenas para definir os pontos afetados. Dessa forma, isso foi feito diretamente no super-aspecto, o qual agora é apenas um aspecto comum. Isso atende ao CM_MI 28. Este exemplo pode ser visto na Figura 4.10.

O aspecto `APreEPosCondicoes` mostra também que o GDB respeita ao CM_MI 17 e ao CM_MI 18, implementando as pré/pós-condições do caso de uso "transferir" em um aspecto. Neste caso, as implementações não foram separadas em dois aspectos pelo fato delas estarem se comunicando para a verificação da coerência antes e após a ação de transferência de valores, o que é exibido pela Figura 4.11.

Por fim, algumas observações sobre os CM_MI restantes se fazem necessárias:

- o CM_MI 19 não foi verificado, uma vez que não foi implementada uma associação entre as classes `Usuário` e `Conta`, o qual seria de um para um (1:1). Os objetos dessas classes se comunicam via atributo `numeroConta` devido à sua simplicidade. Logo, não foi necessário verificar a cardinalidade do relacionamento e validá-lo via aspecto;
- o CM_MI 30 não foi verificado por não haver *pointcuts* declarados em classes. Como este critério está relacionado à linguagem *AspectJ*, isso mostra que o produto de *software* foi construído de forma mais genérica, independente da linguagem de programação;
- o CM_MI 31 não foi verificado porque métodos *abstract* não foram declarados em *introductions*, o que termina por respeitar ao critério em questão;
- o CM_MI 33 não foi verificado pois não existiram requisitos funcionais no GDB que pudessem ser implementados como aspectos.

```
/*
 * ALogging.aj
 *
 * Aspecto responsável pelo controle de logging do sistema.
 *
 * Autor: Rodrigo Pereira dos Santos
 *
 * Aspectos relacionados: Nenhum relacionamento com outros aspectos.
 *
 * Classes atingidas: Banco.
 *
 * ver. 1.01
 *
 * Localização: package aspectos;
 */
public aspect ALogging {

	declare precedence: ATracing, AAutenticacao, ALogging;

	private IRepositorioLogging repositorioLogging;
	private String data;
	private String hora;
	private Usuario usuario;
	private String operacao;
	private Date agora;
	private SimpleDateFormat formatarData;
	private SimpleDateFormat formatarHora;

	pointcut verSaldo(): (execution(* saldo(..)));
	pointcut transferencia(): (execution(* transferir(..)));
	pointcut efetuarSaque(): (execution(* debitar(..)));
	pointcut efetuarDeposito(): (execution(* creditar(..)));
	pointcut alterarSenha(): (execution(* alterarSenha(..)));
	pointcut cadastroUsuario(): (execution(* cadastrarCliente(..)));
	pointcut exclusaoUsuario(): (execution(* excluirCliente(..)));
	pointcut todosOsMetodos(): (verSaldo() || transferencia() || efetuarSaque() || efetuarDeposito()
		|| alterarSenha() || cadastroUsuario() || exclusaoUsuario());
	pointcut classes(): within(Banco);
	pointcut autenticacaoInicial(): call(public Usuario GuiAutenticacao.autenticar());

	public ALogging() {

		repositorioLogging = new RepositorioLogging();
		data = "";
		hora = "";
		usuario = new Usuario();
		operacao = "";
		agora = null;
		formatarData = new SimpleDateFormat("dd/MM/yyyy");
		formatarHora = new SimpleDateFormat("HH:mm:ss");
	}

	before() : todosOsMetodos() && classes() {

		agora = new Date(System.currentTimeMillis());
		data =  formatarData.format(agora);
		hora =  formatarHora.format(agora);
		operacao = "<inicio> " + thisJoinPointStaticPart.getSignature();

		try {
			repositorioLogging.gravarLog(data, hora, usuario, operacao);
		} catch (SQLException e) {
			JOptionPane.showMessageDialog(null, e.getMessage(), "Erro", JOptionPane.ERROR_MESSAGE);
			System.exit(0);
		}
	}

	Usuario around() : autenticacaoInicial() {

		usuario = proceed();

		return usuario;
	}
```

Figura 4.7 – Código-fonte do aspecto *ALogging* (I)

```
    after() returning : todosOsMetodos() && classes() {

        agora = new Date(System.currentTimeMillis());
        data =  formatarData.format(agora);
        hora =  formatarHora.format(agora);
        operacao = "<fim> " + thisJoinPointStaticPart.getSignature();

        try {
            repositorioLogging.gravarLog(data, hora, usuario, operacao);
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, e.getMessage(), "Erro", JOptionPane.ERROR_MESSAGE);
            System.exit(0);
        }
    }

    after() throwing(Exception e) : metodos() && classes() {

        agora = new Date(System.currentTimeMillis());
        data =  formatarData.format(agora);
        hora =  formatarHora.format(agora);
        operacao = "<excecao> " + thisJoinPointStaticPart.getSignature() + " - " + e.getMessage();

        try {
            repositorioLogging.gravarLog(data, hora, usuario, operacao);
        } catch (SQLException f) {
            JOptionPane.showMessageDialog(null, f.getMessage(), "Erro", JOptionPane.ERROR_MESSAGE);
            System.exit(0);
        }
    }
}
```

Figura 4.8 – Código-fonte do aspecto *ALogging* (II)

```
/*
 * AIUsuario.aj
 *
 * Aspecto responsável pela declaração de herança entre as classes filhas
 * Administrador e Cliente e a classe base Usuario.
 *
 * Autor: Rodrigo Pereira dos Santos
 *
 * Aspectos relacionados: Nenhum relacionamento com outros aspectos.
 *
 * Classes atingidas: Administrador; Cliente.
 *
 * ver. 1.01
 *
 * Localização: package aspectos;
 */
public aspect AIUsuario {

    declare parents : Administrador extends Usuario;
    declare parents : Cliente extends Usuario;

}
```

Figura 4.9 – Código-fonte do aspecto *AIUsuario*

```
/*
 * ATracing.aj
 *
 * Aspecto responsável pelo controle de rastreamento da aplicação.
 *
 * Autor: Rodrigo Pereira dos Santos
 *
 * Aspectos relacionados: Nenhum relacionamento com outros aspectos.
 *
 * Classes atingidas: Banco.
 *
 * ver. 1.01
 *
 * Localização: package aspectos;
 */
public aspect ATracing {

	declare precedence: ATracing, AAutenticacao, ALogging;

	pointcut verSaldo(): (execution(* saldo(..)));
	pointcut transferencia(): (execution(* transferir(..)));
	pointcut efetuarSaque(): (execution(* debitar(..)));
	pointcut efetuarDeposito(): (execution(* creditar(..)));
	pointcut alterarSenha(): (execution(* alterarSenha(..)));
	pointcut cadastroUsuario(): (execution(* cadastrarCliente(..)));
	pointcut exclusaoUsuario(): (execution(* excluirCliente(..)));
	pointcut todosOsMetodos(): (verSaldo() || transferencia() || efetuarSaque()
		|| efetuarDeposito() || alterarSenha() || cadastroUsuario() || exclusaoUsuario());
	pointcut classes(): within(Banco);

	before (): todosOsMetodos() && classes() {
		trace("--> " + thisJoinPointStaticPart.getSignature());
	}

	after (): todosOsMetodos() && classes() {
		trace("<-- " + thisJoinPointStaticPart.getSignature());
	}

	private void trace(String str) {
		System.out.println(str);
	}

	private void testesTracing() {
		Banco banco = new Banco();

		try {
			banco.saldo("1");
			banco.transferir("1", "2", 20.0);
			banco.debitar("1", 20.0);
			banco.creditar("1", 20.0);
		} catch(ContaNaoEncontradaException e) {
		} catch(SQLException f) {
		} catch(SaldoInsuficienteException g) {
		} catch(DadosInconsistentesException h) {}
	}
}
```

Figura 4.10 – Código-fonte do aspecto *ATracing*

```
/*
 * APreEPosCondicoes.aj
 *
 * Aspecto responsável por verificar as pré-condições e pós-condições de um caso de uso.
 *
 * Autor: Rodrigo Pereira dos Santos
 *
 * Aspectos relacionados: Nenhum relacionamento com outros aspectos.
 *
 * Classes atingidas: Administrador; Cliente.
 *
 * ver. 1.01
 *
 * Localização: package aspectos;
 */
public aspect APreEPosCondicoes {

	private IRepositorioContas repContas;
	private double saldoOriginalContaDeOrigem;
	private double saldoOriginalContaDeDestino;

	public APreEPosCondicoes() {
		repContas = new RepositorioContas();
		saldoOriginalContaDeOrigem = 0.0;
		saldoOriginalContaDeDestino = 0.0;
	}

	pointcut transferencia(String numeroDe, String numeroPara, double valor) :
		(execution(* Banco.transferir(..)) && args(numeroDe, numeroPara, valor));

	before(String numeroDe, String numeroPara, double valor)
		throws SQLException, ContaNaoEncontradaException,
		SaldoInsuficienteException:transferencia(numeroDe, numeroPara, valor) {

		Conta de = repContas.obterConta(numeroDe);
		Conta para = repContas.obterConta(numeroPara);

		if ((de == null) || (para == null)) {
			if (de == null) {
				throw new ContaNaoEncontradaException(numeroDe);
			} else {
				throw new ContaNaoEncontradaException(numeroPara);
			}
		} else {
			if (de.getSaldo() < valor) {
				throw new SaldoInsuficienteException(de.getSaldo(), de.getNumero());
			} else {
				saldoOriginalContaDeOrigem = de.getSaldo();
				saldoOriginalContaDeDestino = para.getSaldo();
			}
		}
	}

	after(String numeroDe, String numeroPara, double valor) returning
		throws SQLException, DadosInconsistentesException :
		transferencia(numeroDe, numeroPara, valor) {

		double novoSaldoContaDeOrigem = saldoOriginalContaDeOrigem - valor;
		double novoSaldoContaDeDestino = saldoOriginalContaDeDestino + valor;

		Conta de = repContas.obterConta(numeroDe);
		Conta para = repContas.obterConta(numeroPara);

		if ((de.getSaldo() != novoSaldoContaDeOrigem) ||
				(para.getSaldo() != novoSaldoContaDeDestino)) {
			throw new DadosInconsistentesException("Dados inconsistentes! A transação será desfeita!");
		}
	}
}
```

Figura 4.11 – Código-fonte do aspecto *APreEPosCondicoes*

4.4. Considerações Finais

Neste capítulo, foram definidos os critérios de manutenibilidade para a construção do Modelo de Implementação de produtos de *software* manuteníveis orientados a aspectos. Além disso, o uso dos critérios de manutenibilidade foi demonstrado através de um estudo de caso, o GDB, o qual nitidamente se mostrou mais legível, claro e conciso. Isso acontece, uma vez que tais critérios foram elaborados baseando-se em um denso estudo a respeito de manutenção de produtos de *software* e orientação a aspectos, através da identificação de boas técnicas de programação e do uso da Norma ISO/IEC 9126.

Entretanto, devido às peculiaridades de cada caso, ou seja, do domínio de negócio do produto de *software*, alguns critérios podem ser desconsiderados durante a avaliação do Modelo de Implementação.

Conforme observado no exemplo, as exigências analisadas atacam estratégias de construção de produtos de *software* orientados a aspectos e estendem boas práticas e técnicas presentes no consolidado paradigma orientado a objetos. Dessa forma, incorpora nessa nova tecnologia fortes características de manutenção, visando solidificar seus estudos e inserir seus conceitos no mercado.

5. CONSIDERAÇÕES FINAIS

A manutenção tornou-se inevitável por vários motivos ora discutidos neste trabalho. Dessa forma, abordou-se a manutenção de produtos de *software*, a qual é um fator relevante na indústria de *software*. Por isso, recomenda-se construir produtos de *software* com qualidade e informações que possam apoiar a sua manutenção. Assim, foram propostos critérios de manutenibilidade aplicáveis na avaliação das características de manutenibilidade do Modelo de Implementação de um produto de *software* orientado a aspectos.

Neste capítulo, são apresentados alguns aspectos resultantes do trabalho. A seção 1 apresenta uma breve conclusão da relevância do trabalho. A seção 2 cita algumas contribuições que a pesquisa gerou em seu desenvolvimento. Por fim, a seção 3 faz uma síntese dos trabalhos a serem desenvolvidos com base nesta pesquisa, objetivando a sua continuidade.

5.1. Conclusões

A atividade de manutenção de produtos de *software* é penosa. Esta atividade se torna menos árdua para aqueles produtos de *software* desenvolvidos segundo o paradigma de orientação a objetos, pois a OO a melhorou e a AO visa melhorá-la mais ainda. Além disso, o código-fonte dos métodos envolvidos no cumprimento desta função não está, normalmente, no mesmo arquivo físico, dificultando sua manutenção. Dessa forma, a orientação a aspectos surge para contornar os problemas de entrelaçamento e de espalhamento de código, presentes na orientação a objetos.

Não existe um método aprovado que sistematize o processo de manutenibilidade de *software* nem um grupo de regras que auxilie o controle das mudanças, dentro de um sistema metodológico. Isto é, mesmo sendo uma área de extrema importância no processo de desenvolvimento de *software*, a manutenibilidade ainda não apresenta um modelo, ou padrão, que possa ser usado e aplicado. Ela é considerada como a fase mais dispendiosa do ciclo de vida de um produto de *software* e, muitas vezes, a qualidade dos códigos reparados e atualizados apresenta-se baixa, podendo comprometer o seu desempenho. Enquanto as novas metodologias de desenvolvimento de produtos de *software* reduzem a necessidade da manutenibilidade corretiva, elas apresentam uma pequena influência na manutenibilidade perfectiva (evolutiva), que se trata da área mais dispendiosa.

Os critérios de manutenibilidade para o Modelo de Implementação de produtos de *software* orientados a aspectos foram estabelecidos a partir das informações necessárias para minimizar o esforço da tarefa de manutenção, obtidas da norma ISO/IEC 9126 e das boas práticas de programação. Ao se iniciar um processo de desenvolvimento de *software*, a importância do requisito de *software* deve ser avaliada. Em função disso, os critérios que devem ser atendidos podem ser selecionados, uma vez que o uso completo dos critérios pode não ser adequado para os tipos e os portes de produtos de *software*, pois algumas exigências estão relacionadas com determinadas características de uma aplicação particular. Além disso, este uso propicia o investimento cada vez maior no desenvolvimento de novos produtos de *software*, visto que o tempo necessário para efetuar a manutenção em um produto de *software* existente tende a diminuir, pois foi implementado considerando a questão da manutenibilidade.

A incorporação do ideal de manutenção durante a construção do Modelo de Implementação pode contribuir para melhorar o perfil das pessoas envolvidas em tal atividade, pois pode influenciar a cultura de desenvolvimento de produtos de *software* mais adequados, ao realçar a função de seus mantenedores. Desta forma, desmistifica-se a questão de que a manutenção é considerada uma tarefa que não precisa de criatividade e presença contínua para ser desempenhada.

Os critérios de manutenibilidade definidos neste trabalho podem ser utilizados em qualquer modelo de processo de desenvolvimento de produtos de *software* orientado a aspectos. A atividade da verificação dos critérios deve ser integrada nas fases do processo de desenvolvimento selecionado.

5.2. Contribuições

Conforme apresentado neste trabalho, sabe-se que são escassos na literatura trabalhos relacionados à manutenção, sobretudo relacionados à sua incorporação ao longo do processo de desenvolvimento de produtos de *software*. O desafio se torna maior ao se pensar na tecnologia da orientação a aspectos, a qual em menos de uma década desperta grande interesse da comunidade acadêmica e industrial, por atuar sobre algumas fendas dos paradigmas procedural e orientado a objetos.

Dessa forma, um produto de *software* que apresenta as características de manutenibilidade conduz às seguintes conseqüências:

- redução do tempo e do custo de manutenção do produto de *software* pois, de acordo com Sommerville (2000), ao investir esforço extra em fazer um produto de *software* manutenível, é provável obter retorno significativo na redução do custo da manutenção;
- o nível de abstração do artefato de *software* utilizado para a programação e para a manutenção pode ser diminuído, devido a uma implementação mais robusta e clara;
- menor equipe de manutenção e conseqüente realização de novos projetos, devido à maior disponibilidade de pessoal;
- o uso de POA, a qual proporciona maior clareza para a implementação dos sistemas, acarreta melhor manutenibilidade, uma vez que encapsula interesses espalhados e/ou entrelaçados, reduzindo o código-fonte e conduzindo a uma maior reutilização.

Dessa forma, o trabalho deixa como contribuição um conjunto de critérios de construção e avaliação de manutenibilidade elaborados tendo a norma ISO/IEC 9126 como guia e a partir de uma pesquisa ampla em torno dos temas manutenção e orientação a aspectos. Nesse sentido, buscou-se a identificação das características relevantes para a manutenção que devem ser consideradas no Modelo de Implementação. Tais características levaram aos critérios de manutenibilidade e permitem mostrar as informações importantes para o entendimento do produto de *software* e localização das alterações a serem feitas na manutenção.

Também foram verificados os critérios de manutenibilidade em um estudo de caso, de forma a exemplificar seu uso e sua necessidade, destacando a clareza e a legibilidade proporcionadas ao código-fonte e, conseqüentemente, aos desenvolvedores e mantenedores.

Somando-se a isto, o uso de POA constitui mais um reforço a este novo paradigma, de forma a colaborar para a futura Engenharia de *Software* Orientada a Aspectos.

5.3. Trabalhos Futuros

Finalmente, como trabalhos futuros, novas pesquisas podem ser realizadas a partir desta, como a verificação e solidificação dos critérios de manutenibilidade através de novos estudos de caso, preferencialmente em aplicações reais. Isso seria interessante por incorporar a manutenção no processo de desenvolvimento de produtos de *software* na indústria, além da orientação a aspectos, melhorando a qualidade do produto e do processo através de suas melhorias sobre a orientação a objetos.

Além disso, a partir desta base, desenvolver estudos sobre aspectos de manutenibilidade nas etapas anteriores do processo de desenvolvimento de produtos de *software*, ou seja, no Modelo de Análise e no Modelo de Projeto. O Modelo de Projeto é um subsídio importante para o entendimento do produto de *software*, pois, se bem construído e mantido atualizado, é uma representação mais realista da implementação do produto de *software*, auxiliando na avaliação de eventuais efeitos colaterais que podem ocorrer devido às modificações para o atendimento de novos requisitos ou sua correção. Por outro lado, considerando o apoio para a manutenção, o Modelo de Análise também é um subsídio importante, pois é o ponto de partida para o entendimento do produto de *software* na sua manutenção.

Além disso, outros temas de estudo estão relacionados a seguir, conforme [Costa (2005)], com o diferencial de agregar a orientação a aspectos:

- analisar os critérios de manutenibilidade propostos, considerando sua executabilidade, complexidade, relação de inclusão e nível de cobertura;
- realizar o mesmo enfoque utilizado neste trabalho, ou seja, a qualidade de produtos de *software*, porém considerando individualmente as outras características de qualidade presentes na norma ISO/IEC 9126: funcionalidade, confiabilidade, usabilidade, eficiência e portabilidade;
- agregar os trabalhos realizados, para cada uma das características de qualidade presentes no item anterior, em um único trabalho, avaliando as sobreposições e os conflitos;
- desenvolver um experimento de campo com dois grupos de engenheiros de *software*, tendo um gerente de projeto neste experimento. O gerente de projeto entrega um produto de *software* para o primeiro grupo desenvolver, utilizando os critérios de manutenibilidade apresentados neste trabalho. Após a conclusão da implementação do produto de *software*, o gerente de projeto propõe alterações nos requisitos do produto de *software* e o entrega ao segundo grupo para realizar a manutenção necessária. Após a conclusão dos trabalhos do segundo grupo, o gerente de projeto avalia o grau de facilidade de manutenção do produto de *software*;
- justificar detalhadamente o porquê do atendimento das subcaracterísticas de manutenibilidade de cada um dos critérios de manutenibilidade definidos.

REFERÊNCIAS BIBLIOGRÁFICAS

ABNT NBR13596. **Tecnologia de Informação – Avaliação de Produto de Software – Características de Qualidade e Diretrizes para o seu Uso**. ABNT, abr. 1996.

ALDAWUD, O.; ELRAD, T.; BADER, A. **A UML Profile for Aspect Oriented Modeling**. In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems – Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'2001). Tampa, Flórida, EUA, out. 2001.

ALVES, C. G. **Ambiente de Apoio ao Desenvolvimento e Manutenção de Software Científico**. Dissertação (Mestrado). COPPE – Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 1993, 140p.

AMBLER, S. W. **Análise e Projeto Orientados a Objeto**, v. 2. IBPI Press, 1998, 472p.

ARAÚJO, R. M. **Um Sistema de Suporte à Decisão em Grupos para o Desenvolvimento de Software**. In: II Workshop de Pesquisas de Tese em Engenharia de Software, 1993.

ARTHUR, L. J. **Software evolution: The software maintenance challenge**. Willey, New York, USA, 1988.

ASPECTC++. **AspectC++ Home Page**. Disponível em http://www.AspectC.org. Acessado em 10 fev 2007.

ASPECTJ. **AspectJ Home Page**. Disponível em www.aspectj.org. Acessado em 10/02/2007.

ASPECTS, **AspectS Home Page**. Disponível em http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/aspects/AspectS.html. Acessado em 10 fev 2003.

BARBOSA, A. E. **Verificação Automática da Conformidade de Código**. Proposta de Dissertação ao Centro de Engenharia Elétrica e Informática – Universidade Federal de Campina Grande, Campina Grande, PB, 2005.

BECKER, C.; GEIHS, K. **Quality of Service – Aspects of Distributed Programs**. In: Aspect-Oriented Programming Workshop – International Conference on Software Engineering (ICSE'1998). Kyoto, Japão, 1998.

BECKER, U.; GEIER, M.; HAUCK, F. J.; MEIER, E.; RASTOFER, U.; STECKERMEIER, M. **AspectIX: A Middleware for Aspect-Oriented Programming**. In: Aspect Oriented Programming Workshop – 12th European Conference on Object-Oriented Programmming (ECOOP'1998). Bruxelas, Bélgica, 1998.

BECKER, U. **D²AL: A design-based Aspect language for distribution control**. In: Aspect-Oriented Programming Workshop – 12th European Conference on Object-Oriented Programmming (ECOOP'1998). Bruxelas, Bélgica, 1998.

BERGER, L.; DERY, A. M.; FORNARINO, M. **Interaction Between objets: an Aspect of object-oriented languages**. In: Aspect-Oriented Programming Workshop – International Conference on Software Engineering (ICSE'1998). Kyoto, Japão, 1998.

BHATT, P.; SHROFF, G.; MISRA, A.K. **Dynamics of software maintenance**. In: ACM SIGSOFT Software Engineering Notes, v. 29, nº 5, pp. 1-5, set. 2004.

BOAVENTURA, I. Isso G. **Qualidade de Software Produto – Notas de Aula**, 2001.

BÖLLERT, K. **Implementing an Aspect Weaver in Smalltalk**. In: Smalltalk und Java in Industrie und Ausbildung (STJA'1998). Erfurt, Alemanha, 1998.

BOOCH, G. **Object-Oriented Analysis and Design with Applications**, 2ª ed. Benjamin/Cummings Publishing Company Inc, 1994.

BRUSAMOLIN, V. **Manutenibilidade de Software**. In: Revista Digital Online, v. 2, jan. 2004.

CAMARGO, V. V.; MASIERO, P. C. **Frameworks Orientados a Aspectos**. In: Simpósio Brasileiro de Engenharia de Software (SBES'2005). Uberlândia, MG, 2005.

CANNING, R. **The Maintenance Iceberg**, v. 10, nº 10, EDP Analyzer, out. 1972.

CAPRETZ, L. F.; CAPRETZ, M. A. M. **Object-Oriented Software: Design and Maintenance**, v. 6., World Scientific (Series on Software Engineering and Knowledge Engineering), 1996, 263p.

CHAVEZ, C. F. G.; LUCENA, C. J. P. **A Theory of Aspects for Aspect-Oriented Software Development**, In: 17º Simpósio Brasileiro de Engenharia de Software (SBES'2003). Manaus, AM, 2003.

CHAVEZ, C. F. G. **Um Enfoque Baseado em Modelos para o Design Orientado a Aspectos**. Tese (Doutorado). Departamento de Informática – Pontifícia Universidade Católica do Rio, Rio de Janeiro, RJ, 2004, 298p.

CHAVES, R. A. **Aspectos e MDA Criando modelos executáveis baseados em aspectos**. Dissertação (Mestrado). Universidade Federal de Santa Catarina, Florianópolis, SC, 2002.

CLARKE, S. **Composition of Object-Oriented Software Design Models**. PhD Thesis. Dublin City University, jan. 2001.

CLARKE, S. **Extending standard UML with model composition semantics**. In: Science of Computer Programming, v. 44, pp. 71-100. Elsevier Science, jul. 2002.

CLARKE, S.; BANIASSAD, E. **Aspect-Oriented Analysis and Design**. Addison-Wesley, 2005.

CLARKE, S.; WALKER, R. J. **Towards a Standard Design Language for AOSD**. In: 1th International Conference on Aspect-Oriented Software Development. Enschede, The Netherlands, abr. 2002.

COADY, Y.; KICKZALES, G.; FEELEY, M.; SMOLYN, G. **Using AspectC to improve the modularity of path-specific customization in operating system code**. In: European Software Engineering Conference (ESEC'2001) e 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9'2001), 2001.

CORDENONZI, W. H. **Mapeamento de Padrões Internacionais de Qualidade de Produto e de Software para um Modelo Conceitual de Gerência do Processo de Desenvolvimento de Software**. Dissertação (Mestrado). Instituto de Informática – Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, jan. 2000, 122p.

CÔRTES, M. L.; CHIOSSI, T. C. S. **Modelos de Qualidade de Software**. Editora da UNICAMP, 2001, 148p.

COSTA, H. A. X. **Critérios e Diretrizes de Manutenibilidade para a Construção do Modelo de Projeto Orientado a Objetos**. Tese (Doutorado). Escola Politécnica – Universidade de São Paulo, São Paulo, SP, 2005, 199p.

COSTA, J. C. P.; ROUILLER, A. C. **Uma Abordagem para Controlar Projetos de Manutenção de Produtos de Software em uma Pequena Empresa**. In: 2º Workshop de Manutenção de Software Moderna (WMSWM'2005) – Simpósio Brasileiro de Qualidade de Software (SBQS'2005). Manaus, AM, 2005.

CROSBY, P. B. **Quality Is Free – The Art of Making Quality Certain**. McGraw-Hill, 1979, 352p.

CZARNECKI K.; EISENECKER, U. **Generative Programming: Methods, Tools, and Applications**. Addison-Wesley, 2000.

DART, S.; CHRISTIE, A. M.; BROWN, A. W. **A Case Study in Software Maintenance**. Relatório Técnico CMU/SEI-93-TR-8, jun. 1993.

DEMING, W. E. **Out of Crisis: Quality, Productivity and Competitive Position**. MIT Press, 1982, 507p.

DEKLEVA, S. **Delphi study of software maintenance problems**, In: 8th Conference on Software Maintenance. Orlando, Flórida, EUA, nov. 1992.

DIAS, M. G. B. **Uma Experiência no Ensino de Manutenção de Software**. In: 1º Workshop de Manutenção de Software Moderna (WMSWM'2004) – Simpósio Brasileiro de Qualidade de Software (SBQS'2004). Brasília, DF, 2004.

DIAS, M. G. B.; ANQUETIL, N.; OLIVEIRA, K. **Organizing the Knowledge Used in Software Maintenance**. In: Workshop on Learning Software Organizations (LSO'2003) – 2nd Konferenz Professionelles Wissensmanagement Erfahrungen und Visionen. Luzern, Suíça, mai. 2003.

DIJKSTRA, E. W. **A Discipline of Programming**. Prentice Hall, 1976.

DROMEY, R. G. **Cornering the Chimera**. In: IEEE Software, v. 13, nº 1, pp. 33-43, jan. 1996.

EISENMANN, A. L. K.; ZACCONI, L. T.; MARTINS, R. ª G. C. **Preparação POSCOMP**, 2ª ed. Central de Ensino para Graduados, 2005.

ELRAD, T.; KICZALES, G.; AKSIT, M.; LIEBERHER, K.; OSSHER, H. **Discussing Aspects of AOP**. In: Communications of the ACM, v. 44, nº 10, pp. 33-38, 2001.

ENDO, C. **Um Estudo Geral sobre Qualidade de Software**. Monografia. Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo, São Carlos, SP, 1996.

FEIGENBAUM, A. V. **Controle da Qualidade Total**. Makron Books, 1994. 210p.

FERNANDES, A. P. **Programação Orientada a Aspectos**. In: Revista CCEI – URCAMP, v. 7, nº 11, p. 38-42, mar. 2003.

FILMAN, R. E.; ELRAD, T.; CLARKE, S.; AKSIT, M. **Aspect-Oriented Software Development**. Addilson Wesley – Pearson Education, 2005, 755p.

GAL, A.; SCHRÖDER-PREIKSCHAT, W.; SPINCZYC, O. **AspectC++: Language Proposal and Prototype Implementation**. In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems – Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'2001). Tampa, Flórida, EUA, out. 2001.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns – Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995, 416p.

GARCIA, A.; CHAVEZ, C.; SOARES, S.; PIVETA, E.; PENTEADO, R.; CAMARGO, V. V.; FERNANDES, F. **Relatório do 1º Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos** (WASP'2004), 2004.

GOMES, N. S. **Qualidade de Software: uma Necessidade**. Ministério da Fazenda, 2001.

GRADECK, J.; LESIECKI, N. **Mastering AspectJ: Aspect-Oriented Programming in Java**. Wiley, Indianópolis, Indiana, 2003, 453p.

GUEDES, L. C. **Um Processo de Re-engenharia Econômico e Eficaz**. Monografia. Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, RJ, 1993, 18p.

HIRSCHFELD, R. **Aspect-Oriented Programming with Aspects**. In: Net.ObjectDays (NODe). Erfurt, Alemanha, out. 2002.

HUGO, M.; GROTT, M. C. **Estudo de Caso Aplicando Programação Orientada a Aspecto**. Especialização em Tecnologia de Informática na Gestão Integrada de Negócio. Instituto Gene Blumenau, 2005.

IEEE Std. 610.12-1990. **IEEE Standards Collection: Software Engineering**. IEEE, 1993.

IEEE Std. 610.12-1990. **IEEE Standard Glossary of Software Engineering Terminology**. IEEE, 1994.

IRWIN, J.; KICKZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J. **Aspect-Oriented Programming**. In: 11th European Conference on Object-Oriented Programmming (ECOOP'1997). Springer-Verlag, Finlândia, 1997.

ISO Std. 8402. International Standard ISO/CD 8402-1. **Quality Concepts and Terminology, Part One: Generic Terms and Definitions**. International Organization for Standardization, dez. 1990.

ISO Std. 9126. **Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their use**. International Organization for Standardization, dez. 1991.

ISO Std. 9126. **Software Enginnering – Product Quality Part 1: Quality Model**. International Organization for Standardization, jun. 2001.

JACOBSON, I. **Object-Oriented Software Enginnering**. Addison-Wesley, 1992, 512p.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The Unified Software Development Process**. Addison-Wesley, 1999.

JUNG, C. F. **Metodologia para Pesquisa e Desenvolvimento: aplicada a novas tecnologias, produtos e processos**. Axcel Books do Brasil Editora, Rio de Janeiro, RJ, 2004.

JUNIOR, V. G.; WINCK, D. V. **AspectJ: Programação orientada a Aspectos com Java**. Editora Novatec, 2006, 228 p.

JUNIOR, V. G. S.; WINCK, D. V.; MACHADO, C. ª P. **Programação Orientada a Aspectos Abordando Java e AspectJ**. In: 1º Workcomp Sul. Florianópolis, SC, 2004.

JURAN, J. M.; GRYNA JR, F. M. **Quality Planning an Analysis From Product Development Through Use**. McGraw-Hill, 1970, 684p.

KAJKO-MATTSSON, M.; FORSSANDER, S.; OLSSON, U. **Corrective Maintenance Maturity Model (CM3): Maintainer's Education and Training**. In: 23rd International Conference on Software Engineering (ICSE'2001). Toronto, Canadá, mai. 2001.

KAN, S. H. **Metrics and Models in Software Quality Engineering**. Addison-Wesley, 1995, 344p.

KICZALES, G. **Aspect-Oriented Programming with AspectJ** (Tutorial). In: Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'2001). Tampa, Flórida, EUA, out. 2001.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J. M.; IRWIN, J. **Aspect-Oriented Programming**. In: 11th European Conference on Object-Oriented Programmming (ECOOP'1997), v. 1241 do LNCS, pp. 220-242. Springer-Verlag, Finlândia, 1997.

KISELEV, I. **Aspect – Oriented Programming with AspectJ**. Sams Publishing, 2002.

LADDAD, R. **AspectJ in Action: Practical Aspect-Oriented Programming**. Manning, Greenwich, 2003.

LAND, R. **Software Deterioration And Maintainability – A Model Proposal**. Mälardalen University, 2003.

LEHMAN, M. M.; BELADY, L. **Program Evolution: Processes of Software Change**. Academic Press, 1985, 538p.

LIEBERHERR, K. J.; HOLLAND, I. **Assuring Good Style for Object-Oriented Programs**. In: IEEE Computer, v. 6, nº 5, pp. 38-48, set./out. 1989.

LIEBERHERR, K. J.; SILVA-LEPE, I.; XIAO, C. **Adaptive Object-Oriented Programming Using Graph-Based Customization**. In: Communications of the ACM, v. 37, nº 5, pp. 94-101, 1994.

LIENTZ, B. P.; SWANSON, E. B. **Software Maintenance Management**. Addison-Wesley, Reading, MA, 1980.

LIVERPOOL, John Moores University. **CMSCD2010 – Software Maintenance using COBOL**, Edição de agosto de 2000. School og Computing and Mathematical Sciences, 2000.

LOPES, C. I. V. **D: a language framework for distributed programming**. Ph.D. Thesis. Northeast University, 1997.

LUCENA, C. J. P. **A Anatomia de um Ambiente de Desenvolvimento de Software**. Monografia. Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro. Rio de Janeiro, RJ, 1991, 46p.

MAFFEO, B. **Engenharia de Software e Especificação de Sistemas**. Campus, 1992, 526p.

MARCONI, M. A.; LAKATOS, E. M. **Fundamentos de Metodologia Científica**. Editora Atlas, São Paulo, SP, 2003.

MARTIN, J.; MACCLURE, C. **Software Maintenance: The Problem and its Solutions**. Prentice Hall, Englewood Cliffs, NJ, 1983.

MAYRHAUSER, A. Von; VANS, A. M. **Identification of Dynamic Comprehension Processes During Large Scale Maintenance**. In: IEEE Transactions on Software Engineering, v. 22, nº 6, pp. 424-437, jun. 1996.

MAYRHAUSER, T. E. Von. **Software Engineering – Methods and Management**. Academic Press, San Diego, Califórnia, EUA, 1990.

MEDINA, E. A. **Some Aspects of Software Documentation**. In: 3rd Annual International Conference on Systems Documentation (SIGDOC'1984), p.57-59. Cidade do México, México, 1984.

MCCALL, J.; RICHARDS, P.; WALTERS, G. **Factors in Software Quality**. Três volumes, NTIS AD-A049-014, 015, 055, nov. 1977.

MEYER, B. **Object-Oriented Softwware Construction**, 2ª ed. Prentice-Hall, 1997, 1260p.

MICHILINI, F. V. S. **Um Tutorial em Qualidade de Software**. Monografia. Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo, São Carlos, SP, jun. 1997.

MONTEIRO, E. S. **Implementação de um Aplicativo Utilizando AODM, Java e AspectJ**. Monografia. Centro Universitário Luterano de Palmas, Palmas, TO, 2004, 111p.

MONTEIRO, E. S.; FILIPAKIS, C. D. **Um Exemplo da Modelagem de um de Domínio Bancário Utilizando a Orientação a Aspectos**. In: VI Encontro de Estudantes de Informática do Tocantins (ENCOINFO'2004). Palmas, TO, pp. 313-322, out. 2004.

MONTEIRO, E. S.; PIVETA, E. K. **Programação orientada a Aspectos em AspectJ**. In: V Encontro de Estudantes de Informática do Tocantins (ENCOINFO'2003). Palmas, TO, nov. 2004.

MOURA, L. M. V. **Taxonomia de Ambientes de Desenvolvimento de Software**. Dissertação (Mestrado). COPPE – Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, 1992, 186p.

NASCIMENTO, S. J. **Pesquisa Qualidade e Produtividade no Setor de Software Brasileiro – 2004**. In: Encontro da Qualidade e Produtividade em *Software* (EQPS'2004). Manaus, AM, nov. 2004.

NASCIMENTO, S. J. **Pesquisa Qualidade e Produtividade no Setor de Software Brasileiro – 2005**. In: Encontro da Qualidade e Produtividade em *Software* (EQPS'2005). São Paulo, SP, ago. 2005.

NBR ISO/IEC 12207 – **Tecnologia de Informação: Processos de Ciclo de Vida de Software**. ABNT, 1998.

OSSHER, H.; TARR, P. **Multi-dimentional separation of concerns in Hyperspace.** In: Aspect-Oriented Programming Workshop – 13th European Conference on Object-Oriented Programmming (ECOOP'99). Lisboa, Portugal, 1999.

PACE, J. A. D.; CAMPO, M. R. **Analyzing the Role of Aspects in Software Design**. In: Communications of the ACM, v. 44, nº 10, pp. 67-73, out. 2001.

PADUELLI, M. M.; SANCHES, R. **Problemas em manutenção de software: caracterização e evolução**. In: 3º Workshop de Manutenção de Software Moderna (WMSWM'2006) – Simpósio Brasileiro de Qualidade de Software (SBQS'2006). Vila Velha, ES, 2006.

PAGE-JONES, M. **Gerenciamento de Projetos**. Makron Books, 1990, 330p.

PERIN, M. G. **Um Sistema de Gerenciamento de Hiperdocumentos para Ambientes de Desenvolvimento de Software**. Dissertação (Mestrado). Instituto de Informática, Universidade do Rio Grande do Sul, Porto Alegre, RS, 1992, 173p.

PFLEEGER, S. L. **Software Engineering – Theory and Practice**, 2ª ed. Prentice-Hall, 2001.

PIGOSKI, T. M. **Pratical Software Maintenance: Best Practices for Managing your Software Investment**. Wiley Computer Publishing, 1996.

PIVETA, E. K. **AURÉLIA: Um modelo de suporte a programação orientada a aspectos**. Dissertação (Mestrado). Universidade Federal de Santa Catarina, Florianópolis, SC, 2001, 80 p.

PIVETA, E. K.; HECHT, M.; PIMENTA, M. S.; PRICE, R. T. ***Bad Smells* em Sistemas Orientados a Aspectos**. In: XIX Simpósio Brasileiro de Engenharia de Software (SBES'2005). Uberlândia, MG, Brasil, 2005.

POLO, M.; PIATTINI, M.; RUIZ, F.; CALERO, C. **Roles in the maintenance process**. In: ACM SIGSOFT Software Engineering Notes, v. 24, nº 4, pp. 84-86, jul. 1999.

PRESSMAN, R. S. **Engenharia de Software**, 6ª ed. McGraw-Hill, 2006.

RAMASWAMY, R. **How to Staff Business-Critical Maintenance Projects**. In: IEEE Software, v. 17, nº 3, pp. 90-94, mai./jun. 2000.

RESENDE, A. M. P.; SILVA, C. C. **Programação Orientada a Aspectos em Java**. Editora Brasport, 2005, 190 p.

ROCHA, A. R. **Qualidade de *Software*: Teoria e Prática**. In: Encontro da Qualidade e Produtividade em *Software* (EQPS'2001). Porto Alegre, RS, ago. 2001.

ROCHA, A. R. **Qualidade de *Softwtare*: Processo ou Produto**. In: Encontro da Qualidade e Produtividade em *Software* (EQPS'2002). Petrópolis, RJ, ago. 2002.

ROCHA, A. R.; MALDONADO, J. C.; WEBER, K. C. **Qualidade de *Software*: Teoria e Prática**. Prentice Hall, São Paulo, SP, 2001.

ROTHERY, B. **ISO 9000**. Makron Books, 1993.

SANT'ANNA, C. N. **Manutenibilidade e Reusabilidade de Software Orientado a Aspectos: Um Framework de Avaliação**. Dissertação (Mestrado). Departamento de Informática – Pontifícia Universidade Católica do Rio, Rio de Janeiro, RJ, 2004, 113p.

SCHACH, S. R.; TOMER, A. **A Maintenance-Oriented Approach to Software Construction**. In: Journal of Software Maintenance: Research and ctice, v. 12, nº 1, pp. 25-45, 2000.

SENTURIER, L. **JST: An Object Syncronization Aspect for Java**. In: Aspect-Oriented Programming Workshop – 13th European Conference on Object-Oriented Programmming (ECOOP'1999). Lisboa, Portugal, 1999.

SHARON, D. **Meeting the Challenge of Software Maintenance**. In: IEEE Software, v. 13, nº 1, pp. 122-126, jan. 1996.

SILVA, J. B. **ProTESTE+ – Ambiente de Validação Automática de Qualidade de Software através de Teste de Técnicas de Teste e de Métricas de Complexidade**. Dissertação (Mestrado). Instituto de Informática – U niversidade Federal do Rio Grande do Sul, Porto Alegre, RS, fev. 1995, 185p.

SNEED, H.M. **Critical Success Factors in Software Maintenance**. In: International Conference on Software Maintenance (ICSM'2003). Amsterdam, The Netherlands, set. 2003.

SOARES, S.; BORBA, P. **AspectJ – Programação Orientada a Aspectos em Java** (Tutorial). In: VI Simpósio Brasileiro de Linguagens de Programação, pp. 39-55, Rio de Janeiro, RJ, jun. 2002.

SOMMERVILLE, I. **Software Engineering**, 6ª ed. Addison-Wesley, 2000.

SOUZA, L. C. **Orientação a Aspectos: O Paradigma para a Limitação dos Objetos**. Trabalho sobre Estudo Comparativo de Linguagens de Programação. Curso de Ciência da Computação – Universidade Federal da Bahia, Salvador, BA, 2003.

SOUZA, T. B. A. **Um Modelo para Avaliação da Manutenibilidade de Código-fonte Orientado a Objeto**. Trabalho de Graduação em Engenharia de Software. Universidade Federal de Pernambuco, Recife, PE, 2005.

STEIN, D. **An Aspect-Oriented Design Model Based on AspectJ and UML**. Dissertação (Mestrado). Mestrado em Gerenciamento de Sistemas de Informação – Universidade de Essen, Alemanha, 2002.

STEIN, D.; HANENBERG, S.; UNLAND, R. **A UML-based Aspect-Oriented Design Notation For AspectJ**. In: 1st International Conference on Aspect-Oriented Software Development (AOSD'2002). Enschede, The Netherlands, abr. 2002.

STEINMACHER, I. F. **Estudo de Princípios para a Modelagem Orientada a Aspectos**. Trabalho de Graduação. Universidade Estadual de Maringá, Maringá, PR, 2003, 71 p.

SUN MICROSYSTEMS. **Code Conventions for the Java Programming Language**, 1999. Disponível em: <http://java.sun.com/docs/codeconv/>. Acessado em 15 fev 2006.

SUZUKI, J.; YAMAMOTO, Y. **Extending UML with Aspects: Aspect Support in the Design Phase**. In: Aspect-Oriented Programming Workshop – 13th European Conference on Object-Oriented Programmming (ECOOP'1999). Lisboa, Portugal, jun. 1999.

SWANSON, E. B. **The Dimensions of Maintenance**. In: 2nd International Conference on Software Engineering (ICSE'1976), pp. 492-497. San Francisco, Califórnia, EUA, out. 1976.

SWEBOK. **Guide to the Software Engineering Body of Knowledge**, versão 2004. The Institute of Electrical and Electronics Engineers, Inc – IEEE, 2004.

TEAM, ASPECTJ.ORG. **Aspect-Oriented Programming with AspectJ**. In: 8th International Symposium on the Foundations of Software Engineering. San Diego, California, USA, 2000.

TEAM, ASPECTJ.ORG. **The AspectJ Programming Guide**, 2003. Disponível em http://aspectj.org. Acessado em 10 fev 2006.

TOLEDO, J. C. **Qualidade Industrial – Conceitos, Sistemas e Estratégias**. Atlas, 1987, 184p.

VELDWIJK, R. J. **Assessing the Software Crisis: Why Information System are Beyond Control**. In: Information Sciences, n.81, pp. 103-116, nov. 1994.

YOURDON, E. **Análise Estruturada Moderna**. Campus, 1990, 856p.

ZAKARIA, A. A.; HOSNY, H.; ZEID, A., **A UML Extension for Modeling Aspect-Oriented Systems**. In: Aspect-Oriented Modeling with AOP – 1st International Conference on Aspect-Oriented Software Development (AOSD'2002). Enschede, The Netherlands, abr. 2002.

ZVEGINTZOV, N.; PARIKH, G. **60 years of Software Maintenance: Lessons Learned**. In: 21st IEEE International Conference on Software Maintenance (ICSM'2005). Budapeste, Hungria, set. 2005.