PAULO AFONSO PARREIRA JÚNIOR

CRITÉRIOS DE TESTABILIDADE PARA AVALIAÇÃO DO MODELO DE PROJETO DE SOFTWARE ORIENTADO 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
2008

PAULO AFONSO PARREIRA JÚNIOR

CRITÉRIOS DE TESTABILIDADE PARA AVALIAÇÃO DO MODELO DE PROJETO DE SOFTWARE ORIENTADO 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:

Prof. Dr. Heitor Augustus Xavier Costa

LAVRAS
MINAS GERAIS – BRASIL
2008

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

Parreira Júnior, Paulo Afonso

Critérios de Testabilidade para Avaliação do Modelo de Projeto de Software Orientado a Aspectos / Paulo Afonso Parreira Júnior. Lavras – Minas Gerais, 2008. p. 62.

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

1. Qualidade de Software. 2. Testabilidade. 3. Orientação a Aspectos. I. PARREIRA JÚNIOR, P. A. II. Universidade Federal de Lavras. III. Critérios de Testabilidade para Avaliação do Modelo de Projeto de Software Orientado a Aspectos.

PAULO AFONSO PARREIRA JÚNIOR

CRITÉRIOS DE TESTABILIDADE PARA AVALIAÇÃO DO MODELO DE PROJETO DE SOFTWARE ORIENTADO 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 18/11/2008

_____

Prof. Dr. Antônio Maria Pereira de Resende

_____

Prof. Dr. Valter Vieira de Camargo

_____

Prof. Dr. Heitor Augustus Xavier Costa

(Orientador)

LAVRAS
MINAS GERAIS – BRASIL

"O coração do homem pode fazer planos, mas a resposta certa dos lábios vem do Senhor."
Pv.16:1

Dedico à minha mãe Eliana, ao meu pai Paulo, ao meu irmão Ricardo e à minha noiva Flávia Mendonça, que contribuíram de um modo muito especial para o meu crescimento como ser humano e como profissional Amo todos vocês!

AGRADECIMENTOS

Agradeço,

primeiramente a Deus, que tem me privilegiado com suas bênçãos enriquecedoras.

Aos meus pais Paulo e Eliana, pessoas fundamentais que me ensinaram a lutar pelos meus objetivos e a quem sou eternamente grato por todo amor e dedicação oferecidos a mim.

Ao meu irmão Ricardo, companheiro de todas as horas, amigo e amado irmão.

Às minhas tias Nadir, Sirlene e Rosilene, que estiveram presentes durante minha trajetória acadêmica, acreditando e ajudando a concretizar meus sonhos. Sempre serei grato a vocês.

À minha amada noiva Flávia Mendonça, que tornou os dias de minha vida mais felizes e sem a qual não posso mais viver. Obrigado por toda dedicação e paciência que teve durante esses anos. Amo você!

Aos meus amigos de república, especialmente Stenio Elias e Luis Paulo (vulgo Caboclo), pela cumplicidade e presença em vários momentos. Sempre me lembrarei de vocês. Obrigado pelo companheirismo e pelas agradáveis horas de confraternização.

Aos meus alunos, muito importantes para meu desenvolvimento. Foram momentos marcantes que estarão para sempre presentes em minha memória.

Por fim, agradeço ao meu orientador, o professor Heitor Augustus Xavier Costa, por todo seu caráter, profissionalismo, paciência e dedicação para comigo durante as diversas atividades de pesquisa e conclusão deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

DSOA	Desenvolvimento de Software Orientado a Aspectos
EROA	Engenharia de Requisitos Orientada a Aspectos
MOA	Modelagem Orientada a Aspectos
OA	Orientado a Aspectos ou Orientação a Aspectos
OO	Orientado a Objetos ou Orientação a Objetos
POA	Programação Orientada a Aspectos
POO	Programação Orientada a Objetos
UML	*Unified Modeling Language*

Critérios de Testabilidade para Avaliação do Modelo de Projeto de Software Orientado a Aspectos

RESUMO

A atividade de teste de software é realizada visando assegurar a maior qualidade possível do software. Apesar de ser impossível provar que um software é livre de defeitos, a aplicação de testes fornece evidências da conformidade com a funcionalidade especificada. Entretanto, sabe-se que as atividades relacionadas com os testes precisam ter o seu planejamento iniciado logo no princípio do ciclo de desenvolvimento, contribuindo para evitar defeitos e sua propagação nas demais fases do desenvolvimento. Desta forma, este trabalho propõe a avaliação do Modelo de Projeto de software orientado a aspecto, considerando as características de testabilidade. Para isso, foi definido um conjunto de critérios de testabilidade para ser usado na avaliação. Estes critérios foram utilizados numa aplicação para verificar a sua usabilidade..

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

Testability Criteria for Evaluation of Design Model of Aspect-Oriented Software

ABSTRACT

Software testing aims to ensure the best possible software quality. Despite being impossible to prove that software has no faults, testing supplies evidence of the conformity with the specified functionality. However, testing activities need to have their planning started at the beginning of the development cycle, contributing to avoid faults and their propagation throughout the development phases. This paper proposes an evaluation of the aspect-oriented software Design Model, considering testability characteristics. To achieve it, we have defined a set of testability criteria to be used in such an evaluation. These criteria were used in an application in order to evaluate their effectiveness.

Keywords: Software Quality, Software Test, Aspect-Oriented..

1. INTRODUÇÃO

1.1. Contextualização

O termo aspecto representa uma unidade modular projetada para implementar um interesse, que em Engenharia de Software pode caracterizar um objetivo particular, um conceito ou uma responsabilidade que possa ser adotada na solução de um software, por exemplo, persistência, controle de *logging* e segurança [LADDAD , 2003].

Ainda que existam várias afirmações na literatura [ZHAO; RINARD, 2003; ZHAO 2003; ZHAO, 2002] de que a adoção do Desenvolvimento de Software Orientado a Aspectos (DSOA) eventualmente leve a software com melhor qualidade, por conduzir a uma melhor arquitetura e de que uma linguagem Orientada a Aspectos (OA) force um estilo de codificação mais disciplinado, ambos não possuem imunidade contra erros de programadores, tampouco contra problemas de falta de entendimento na fase de especificação. Sendo assim, o teste continua sendo uma importante atividade, mesmo no DSOA [ZHAO, 2003].

Segundo Silveira (2007), a incorporação de interesses adicionais à lógica primária da aplicação (*core concerns*) deve passar por rigorosos e diferentes níveis de testes, com o intuito de garantir que esses interesses adicionais – conhecidos como funções auxiliares – comportem-se conforme o especificado, bem como garantir que não introduzam defeitos na aplicação.

1.2. Motivação

Um fator motivador para esta pesquisa é a importância, destacada por vários autores [BINDER, 2000; COLANZI, 1999; MCGREGOR; SYKES, 2001], de realizar as atividades de teste desde o início do ciclo de desenvolvimento, abrangendo as suas fases. Desta forma, esta pesquisa propõe um desenvolvimento que considere as atividades de teste desde as fases iniciais do desenvolvimento, incluindo as características de testabilidade nos Modelos de Projeto[1] e realizando a avaliação destas características pela da definição de critérios a serem seguidos.

[1] Modelo do Software é uma representação do software em vários níveis de abstração, dependendo da fase do processo de desenvolvimento (análise/projeto/implementação/teste/manutenção). Neste caso, o Modelo de Projeto está relacionado ao conjunto de artefatos gerados durante a fase de projeto, enquanto o Modelo de Implementação está relacionado à codificação dos sistemas de informação em desenvolvimento [Filman *et al*., 2005].

É importante salientar também, que outro fator motivador deste trabalho é o crescente uso do paradigma OA em desenvolvimento de software. Isso pode ser comprovado com os eventos que têm acontecido e tomado proporção considerável entre os pesquisadores: *Latin American Workshop on Aspect-Oriented Software Development* e *International Conference on Aspect-Oriented Software Development*.

1.3. Objetivos

O objetivo deste trabalho é definir critérios de testabilidade que possam ser aplicados ao Modelo de Projeto para aumentar a característica de qualidade testabilidade de software OA.

Segundo a norma ISO/IEC 9126, a característica de qualidade testabilidade consiste em atributos do software que evidenciam o esforço necessário, visando minimizá-lo, para validar o software modificado [ISO Std. 9126, 1991; ISO Std. 9126, 2001; ABNT NBR13596, 1996].

A linguagem de modelagem usada para representação os artefatos do Modelo de Projeto foi *aSideML*. Esta linguagem foi proposta na tese de doutorado de Chavez (2004) e consiste em uma linguagem para especificar e comunicar projetos (*design*) OA.

1.4. Metodologia

1.4.1. Tipo da Pesquisa

Conforme Jung (2004) e Marconi; Lakatos (2003) e observando o método científico, pode-se caracterizar esta pesquisa de natureza tecnológica, com objetivos de caráter exploratório, usando procedimentos experimentais fundamentados em um trabalho de campo.

1.4.2. Procedimentos Metodológicos

O trabalho foi iniciado realizando um levantamento bibliográfico na *Internet* e em bibliotecas de artigos científicos relacionados ao tema. Em seguida, foi estudado o paradigma OA, de forma a verificar suas características, suas técnicas e suas contribuições para a área de computação. Posteriormente, foram estudados os principais conceitos e técnicas relacionadas ao teste de software, em particular, teste de software OA.

A etapa seguinte constituiu-se em identificar os aspectos de testabilidade a serem incorporados nos artefatos de software do Modelo de Projeto, sob a luz da norma ISO/IEC 9126 [ISO Std. 9126, 1991; ISO Std. 9126, 2001; ABNT NBR13596, 1996], definindo os critérios de testabilidade.

Reuniões semanais ocorrerão entre o autor e o orientador visando manter a consistência e bom andamento do trabalho. Por fim, os resultados obtidos foram usados em um estudo de caso, possibilitando verificar a aplicabilidade dos critérios propostos.

1.5. Estrutura do Trabalho

Este trabalho está organizado da seguinte forma.

O Capítulo 2 apresenta a definição e os principais conceitos do paradigma OA, bem como algumas abordagens para a modelagem visual de programas OA.

O Capítulo 3 discorre sobre teste de software, qualidade de software, testabilidade em software e sintetiza a norma ISO/IEC 9126, mostrando as seis características de qualidade para software e suas respectivas subcaracterísticas.

O Capítulo 4 descreve os principais trabalhos da literatura relacionados a testabilidade de software.

O Capítulo 5 define os critérios de testabilidade aplicáveis na construção e/ou avaliação/adaptação do Modelo de Projeto de software orientados a aspectos. Além disso, apresenta um estudo de caso, o software Gestão de Domínio Bancário (GDB), visando ilustrar a aplicabilidade dos critérios.

O Capítulo 6 conclui o trabalho fazendo algumas considerações finais sobre a pesquisa realizada. Neste capítulo, além das contribuições, são apresentados os resultados alcançados e sugeridos trabalhos futuros.

2. ORIENTAÇÃO A ASPECTOS

2.1. Considerações Iniciais

Este capítulo apresenta os principais conceitos da Programação Orientada a Aspectos (POA), uma nova tecnologia que objetiva solucionar alguns dos problemas identificados na Programação Orientada a Objetos (POO) e descreve as abordagens para modelagem visual de software OA.

A terminologia utilizada neste trabalho, quando referente a POA, segue as traduções realizadas pela Comunidade Brasileira de Desenvolvimento de Software Orientado a Aspectos (AOSDbr) para a Língua Portuguesa [AOSDbr, 2008].

A seção 2.2 descreve breve histórico sobre a POA, abordando sua origem e sua importância. A seção 2.3 apresenta conceitos básicos envolvidos na POA. A seção 2.4 apresenta diversas abordagens propostas para a modelagem visual de aspectos, destacando a abordagem *aSideML*.

2.2. Surgimento da Programação Orientada a Aspectos

O conceito de interesse (*concern*) foi apresentado por Dijkstra (1976) e é utilizado em Engenharia de Software para separar os atributos ou as funções de um software [ELRAD *et al*., 2001a].

Um software pode ser visto como uma implementação combinada de múltiplos interesses, tais como regras de negócios, desempenho, persistência, registro de informações (*logging*), autenticação, segurança, gerenciamento de memória, balanceamento de carga e replicação [KICZALES *et al*., 2001]. A Figura 2-1 apresenta um exemplo de software visto como a implementação de um conjunto de interesses.

Os interesses em um software podem representar Requisitos Funcionais (RFs) e Requisitos Não-Funcionais (RNFs). Os RFs definem a funcionalidade de um software ou de seus componentes. Os RNFs podem estar relacionados a propriedades emergentes do software, como confiabilidade e tempo de resposta, e incluir limitações (desempenho e segurança) e/ou no processo de desenvolvimento (custos, atrasos, metodologias adotadas e componentes a serem utilizados) [SOMMERVILLE, 2001].

Com o constante crescimento da complexidade dos sistemas modelados em Engenharia de Software, diversas dificuldades têm sido ocasionadas no processo de desenvolvimento de software em conformidade com RFs e RNFs, dentro dos cronogramas e dos orçamentos planejados [SILVEIRA, 2007].

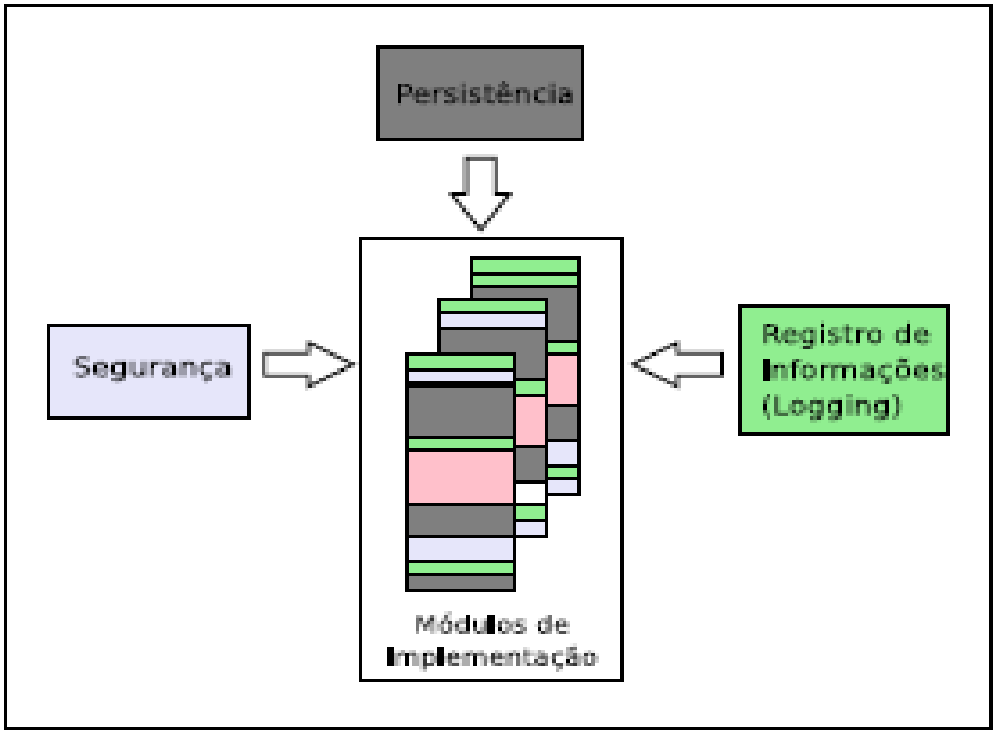


Figura 2-1 – Exemplo de um Sistema Visto como um Conjunto de Interesses (Fonte: Laddad (2003) *apud* Silveira (2007))

À medida que aumenta a complexidade do software, surge a necessidade de melhores técnicas de programação, objetivando organizar e apoiar o seu processo de desenvolvimento. No âmbito da Ciência da Computação, as técnicas de programação vêm evoluindo desde construções de baixo nível (linguagens de máquina) a abordagens de alto nível (POO) [ELRAD *et al.*, 2001b].

Os paradigmas de programação constituem forte influência no projeto de linguagens de programação, porque permitem o desenvolvimento de software cada vez mais complexo e contribuem para a diminuição do seu custo final. Cada novo paradigma que surge e alcança sua maturidade proporciona um avanço na tecnologia de programação [SEBESTA, 2008].

Segundo Resende; Silva (2005), inicialmente, desenvolvia-se software usando a programação desestruturada. Comandos de desvio de fluxo (por exemplo, *GOTO*) eram

utilizados sem critérios, tornando árduo o entendimento, a reutilização e a manutenção do código-fonte. A programação estruturada foi um grande avanço, pois as funções e/ou os procedimentos eram agrupados e chamados a partir de um programa principal. Quem usava a programação estruturada podia reusar as funções e os procedimentos, desde que devidamente projetados. Nesta época, a reusabilidade foi explorada usando o conceito de bibliotecas.

Com o advento da POO, em meados da década 70, com a linguagem de programação *Smalltalk*, inicia-se um novo paradigma de desenvolvimento de software presente até os tempos atuais [SANTOS, 2007]. Tendo o reuso como objetivo, a POO ganhou popularidade, por dividir software em um conjunto de classes e objetos que representam abstrações de entidades do mundo real ou de artefatos de implementação. Nota-se que, cada classe e objeto resultante deste estilo de decomposição representa um interesse [SILVEIRA, 2007].

A POO trouxe avanços para o desenvolvimento de software, permitindo a construção de projetos mais facilmente, maior reusabilidade de componentes, modularidade, componentização, implementações mais robustas e redução do custo de manutenção [JUNIOR; WINCK, 2006]. Diversas aplicações não estão em apenas um único módulo de código contido em um único arquivo. Conforme Gradecki; Lesiecki (2003), aplicações são coleções de módulos que trabalham juntas para fornecer determinadas funções para um conjunto de requisitos bem definidos.

Entretanto, apesar de tais avanços, o aumento da complexidade inseriu novos problemas que a orientação a objetos (OO) é incapaz de resolver. Isso ocorre principalmente quando se trata de RNFs, que afetam o comportamento das classes, tais como persistência de dados (*data persistence*) e pontos de verificação para fins de tolerância a falhas (*checkpoints for fault-tolerance*).

Na concepção de Kiczales *et al.* (1997), as técnicas de POO e da programação estruturada não são suficientes para implementar com clareza importantes decisões de projeto. Isto conduz a uma implementação espalhada pelo código-fonte, resultando em um emaranhado de código que se torna excessivamente difícil de desenvolver e manter. Conforme Silveira (2007), isto ocorre porque a solução para estas decisões não é facilmente encapsulada em uma estrutura de atributos e comportamentos, pois a sua

implementação requer que tais interesses estejam espalhados (*scattered*) e entrelaçados (*tangled*) ao longo de diversas classes, simultaneamente.

A necessidade de novas abstrações capazes de propiciar maior clareza na separação de interesses motivou a definição de novas técnicas de programação e de desenvolvimento de software [SILVEIRA, 2007]. Dentre as principais técnicas, há: *Adaptive Programming* (AP) [LIEBERHERR *et al*., 2001]; *Composition Filters* (CF) [AKSIT *et al*., 1992]; *Subject-Oriented Programming* (SOP) [OSSHER; TARR, 1999]; e *Aspect-Oriented Programming* (AOP) [KICZALES e*t al*., 1997]. Dentre essas técnicas, a que tem se mostrado mais promissora é a POA (ELRAD *et al*., 2001b), sendo ela a técnica abordada neste trabalho.

2.3. Programação Orientada a Aspectos

De acordo com Kiczales *et al*. (1997), aspectos não tendem a ser unidades de decomposição funcional do software, mas propriedades que afetam de forma sistemática o desempenho ou a semântica dos componentes. O estilo de decomposição da OA preserva os princípios estabelecidos pelo paradigma OO e adiciona a capacidade de modelar RNFs em estruturas bem definidas chamadas aspectos [SILVEIRA, 2007].

A POA não trabalha isoladamente; ela é um paradigma que estende outros paradigmas de programação, complementando-os [SANTOS, 2007]. Para melhor entender o paradigma OA, faz-se necessário descrever alguns de seus termos específicos [FILMAN *et al.*, 2005; RESENDE; SILVA, 2005]:

- **Interesses (*Concerns*)**. São preocupações presentes no processo de construção de um software, que envolvem desde assuntos e requisitos de alto nível aos relacionados a implementações de baixo nível. Algumas destas preocupações localizam-se em pontos bem específicos e bem definidos, enquanto outras se referem às propriedades mensuráveis do software como um todo. A Figura 2-1 apresenta alguns exemplos desses interesses;
- **Pontos de Junção (*Join points*)**. Este termo refere-se a pontos bem definidos em uma estrutura ou em um fluxo de execução de um software, onde um comportamento adicional pode ser anexado. Um modelo de pontos de junção provê um conjunto de referências que propiciam a definição de estruturas de aspectos. Os elementos mais comuns de um modelo de pontos de junção são as chamadas de métodos, apesar das

linguagens OA terem definido pontos de junção para outras circunstâncias (por exemplo, acesso, modificação e definição de atributos, exceções e execução de eventos e estados). Se uma linguagem OA permite chamadas de métodos no seu modelo de pontos de junção, um programador pode designar um código adicional para ser executado em uma chamada de método. A Figura 2-2 ilustra esta situação, apresentando o ponto de junção onde um comportamento extra pode ser acrescentado ao código original da aplicação;

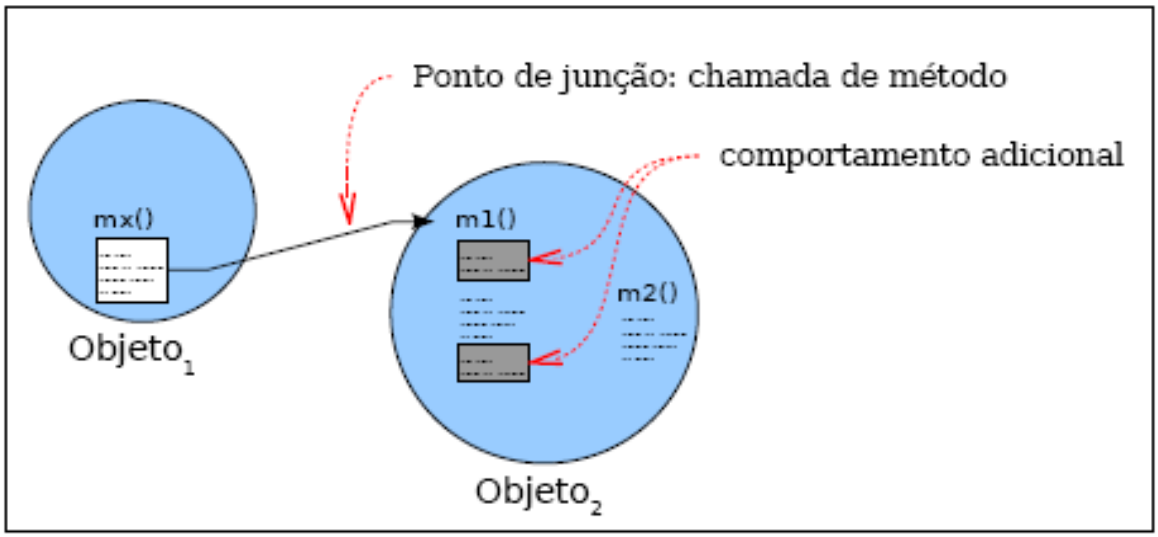


Figura 2-2 – Exemplo de Ponto de Junção na Chamada de Método (Fonte: Silveira (2007))

- **Interesses transversais (*Crosscutting concerns*)**. Freqüentemente, a implementação de um interesse deve ser espalhada ao longo de uma ou mais implementações. Neste caso, o interesse cuja implementação entrecorta a implementação de outros interesses é denominado interesses transversais. Segundo Araújo (2002), interesses transversais são responsabilidades que produzem representações entrelaçadas e espalhadas pelo ciclo de vida de um software. Como exemplo de um interesse transversal, tem-se um editor de figuras apresentado em Elrad *et al.* (2001). O 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 no Diagrama de Classes (Figura 2-3) o interesse de atualização da tela não pertencer às classes Ponto e Linha, mas entrecorta-as e, por isso, pode ser entendido como transversal a essas classes. As linguagens OA utilizam cincos elementos principais para modularizar os interesses transversais [ELRAD *et al.*, 2001]:

- Um modelo de pontos de junção, descrevendo os “ganchos” (*hooks*) onde melhoramentos ou funções podem ser adicionados;
- Meios de identificação desses pontos de junção;
- Meios de especificações de comportamento nesses pontos de junção;
- Unidades encapsuladas, combinando especificações e melhoramentos comportamentais nos pontos de junção;
- Um método para interligar ou combinar esses interesses ao programa alvo.

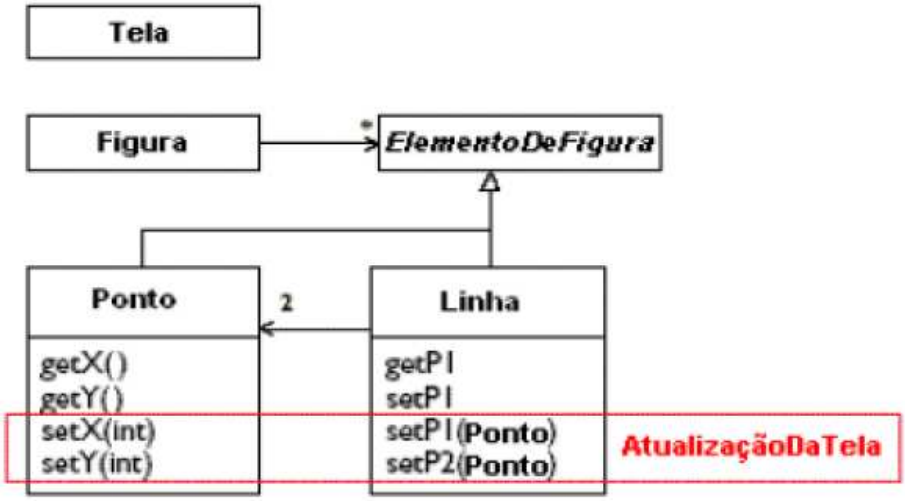


Figura 2-3 – Interesse Transversal de Atualização da Tela (Fonte: Elrad *et al*. (2001))

- **Aspecto (*Aspects*)**. Este termo representa uma unidade modular projetada para implementar um interesse. Uma definição de aspectos pode conter códigos e instruções sobre onde, quando e como será invocado;
- **Conjunto (de pontos) de junção (*Pointcut*)**. Este termo é utilizado para descrever um conjunto de pontos de junção (*join points*). Esta característica provê um mecanismo que possibilita a declaração de pontos bem definidos no fluxo de execução de um software para implementação de comportamentos adicionais;
- **Adendo (*Advice*)**. Este termo refere-se a construções similares aos métodos de objetos do paradigma OO e compreende o comportamento que deve ser executado em cada ponto de junção (j*oin point*) especificado por um conjunto de junção (*pointcut*). Por exemplo, o código de segurança responsável por realizar a autenticação e o controle de acesso. Muitas linguagens OA possuem mecanismos para executar o adendo antes

(*before*), depois (*after*), em substituição ou acerca (*around*) dos pontos de junção convenientes[2].

2.4. Modelagem Visual de Aspectos

Segundo Chavez (2004), software OA precisa ser visualizado e comunicado entre os desenvolvedores. Ademais, suas propriedades precisam ser especificadas de modo a permitir a análise de modelos e a evitar o entrelaçamento e o espalhamento nas especificações. A Modelagem Orientada a Aspectos (MOA) é uma parte crítica do DSOA que se concentra em notações e em técnicas para a especificação, a representação, a visualização e a comunicação de soluções OA por meio do percurso que leva da Engenharia de Requisitos Orientada a Aspectos (EROA) a POA.

A seguir, é apresentada uma descrição sucinta de algumas abordagens para MOA disponíveis na literatura:

- **Abordagem *Tema***: Clarke; Baniassad (2005) propuseram uma abordagem de desenvolvimento de software OA denominada *Tema*. A abordagem *Tema* é divida em dois níveis para diferentes fases do ciclo de vida do software: *Tema/Doc* e o *Tema/UML*. O nível *Tema/Doc* é utilizado para a fase de requisitos, fornecendo visões do texto da especificação de requisitos e expõe o relacionamento entre os comportamentos de um software. Ele permite ao desenvolvedor refinar a visão dos requisitos de modo a revelar quais funções do software são transversais e onde elas se entrecortam. O nível *Tema/UML* trabalha com dois tipos de temas: i) *Tema-base* (possui a funcionalidade do domínio do problema); e ii) *Tema-transversal* (encapsula os interesses transversais que afetam o *Tema-base*). Um tema é o encapsulamento de um interesse (*concern*) representado graficamente por meio das operações-gabaritos (*templates*). Definidos o *Tema-base* e o *Tema-transversal*, é necessário realizar a composição entre eles por meio de um relacionamento de ligação denominado *bind*. Com esse relacionamento, sabe-se quais métodos do *Tema-base* serão afetados pelo *Tema-transversal*. Após realizar o relacionamento de ligação, é possível fazer a composição para que seja gerado o modelo resultante da aplicação;
- **Abordagem Geral de Modelagem Arquitetural Orientada a Aspectos (*General Aspect-Oriented Architecture Design Approach* – GAADA):** esta abordagem foi

[2] Neste trabalho, a terminologia adotada para adendos do tipo *before*, *after* e *around* é anteriores, posteriores e de contorno, respectivamente.

proposta por Krechetov *et al*. (2006) e é baseada no Trabalho do Grupo de Pesquisa em Modelagem Arquitetural do AOSD-Europe [AOSD-EUROPE, 2006]. A GAADA pretende fornecer, em sua versão final, a representação dos seguintes conceitos: i) Aspecto; ii) Componente; iii) Conjunto de Junção; iv) Adendo; v) Entrecortes estático (declarações intertipos[3]) e dinâmico (*pointcut*); vi) Relacionamento Aspecto-Componente; e vii) Relacionamento Aspecto-Aspecto. A Figura 2-4 mostra a representação visual dos elementos da GAADA. Nota-se que, nesta abordagem, não existem notações gráficas para os elementos adendos e entrecortes estáticos. Os autores consideram estas representações e a modelagem visual das máscaras[4] de conjuntos de junção como questões em aberto e evidenciam a necessidade de mais estudos nesse tema;

- **Linguagem de Modelagem *aSideML***: Chavez (2004) propôs uma linguagem de modelagem chamada *aSideML* para especificar e comunicar projetos (*design*) OA. Esta linguagem oferece semântica, notação e regras que permitem ao projetista construir modelos cujo foco sejam os principais conceitos, mecanismos e propriedades de software OA, no qual os aspectos sejam explicitamente tratados como "cidadãos de primeira classe". A linguagem *aSideML* organiza os elementos de modelagem em diagramas gráficos que fornecem várias visões do software em desenvolvimento. A Figura 2-5 apresenta descrições dos modelos, dos diagramas, dos novos elementos de modelagem e das perspectivas relevantes para cada diagrama de aSideML.

Apesar das abordagens apresentadas anteriormente proverem suporte para MOA em nível de projeto, para este trabalho, a abordagem *aSideML* se apresentou mais adequada, pois ela propõe um modelo de aspectos de alto nível, independente de linguagem e contempla os principais conceitos, propriedades e arquitetura introduzidos pelo projeto OA. Desta forma, *aSideML* oferece recursos suficientes para o levantamento de informações para os testes.

[3] É uma característica presente em algumas linguagens OA que permite alterar a estrutura de uma classe introduzindo novos atributos, métodos, construtores, *getters*, *setters*, herança e interface [RESENDE; SILVA, 2005].

[4] Estrutura disponibilizada em algumas linguagens OA para facilitar o trabalho de escrever *pointcuts*. Por exemplo, o uso de "curingas", como *, +, .. (dois pontos seqüenciais), pode simplificar a declaração de pontos de junção (salvar*() = métodos que começam com o prefixo "salvar") [RESENDE & SILVA, 2005].

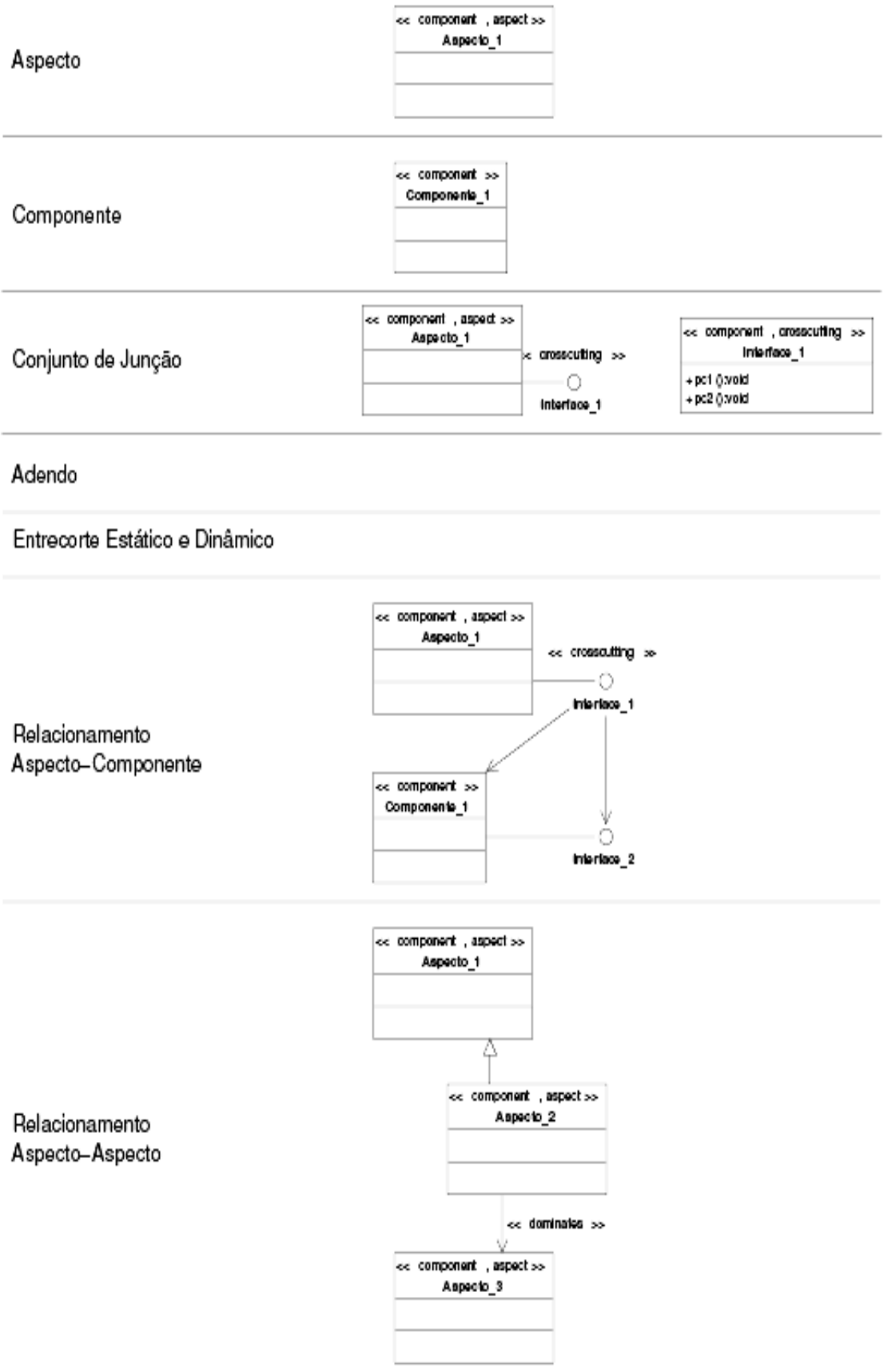


Figura 2-4 – Representação Visual dos Elementos da GAADA (Fonte: Krechetoc (2006) *apud* Silveira (2007))

Modelo	Diagramas	Perspectivas	Elementos
estrutural	diagrama de aspectos		aspecto, interface transversal, característica transversal
	diagrama de classe estendido	centrado em aspecto	aspecto, *crosscutting*
		centrado na base	interface transversal, *order*
comportamental	diagrama de sequência estendido	ponto de combinação	ponto de combinação dinâmico
	diagrama de colaboração aspectual		instância de aspecto, colaboração aspectual
	diagrama de sequência		instância de aspecto, interação aspectual
processo de combinação	diagrama de classes combinadas		classe combinada
	diagrama de colaboração combinada		colaboração combinada
	diagrama de sequência combinada		interação combinada

Figura 2-5 – Dimensões da Modelagem Orientada a Aspectos com aSideML (Fonte: Chavez (2004))

2.4.1. A Abordagem aSideML

A aSideML é uma dentre as várias abordagens existentes para MOA. Proposta na tese de doutorado de Chavez (2004), aSideML consiste em uma linguagem de modelagem desenvolvida para especificar e comunicar projetos OA. Segundo Chavez (2004), o nome aSide vem do termo em inglês *aside* (*to or toward the side*), para denotar elementos que estão ao lado de outros. O nome pode também ser interpretado como denotando "um lado" (*a side*), no sentido de uma faceta ou um aspecto. Os principais objetivos relacionados ao projeto aSideML são:

- fornecer uma linguagem de modelagem visual expressiva a fim de desenvolver e de comunicar modelos de aspectos;
- fornecer semântica e notação para oferecer apoio aos novos requisitos de modelagem introduzidos pelos aspectos;
- fornecer um *framework* conceitual que incorpora o consenso da comunidade OA sobre os principais conceitos do DSOA e uma base rigorosa para compreender a linguagem de modelagem;

- oferecer apoio às especificações independentes de linguagem que podem ser mapeadas para linguagens de programação OA particulares;
- seguir os padrões da indústria.

A linguagem *aSideML* define três dimensões de modelagem para sistemas AO (Figura 2-5):

- **modelagem estrutural** fornece visão estática de um software na presença de aspectos. Os principais elementos estruturais são: i) *aspectos*; ii) *elementos base* (de UML – *Unified Modeling Language*) que os aspectos afetam (as classes); e iii) seus *relacionamentos*. O comportamento dinâmico destes elementos é descrito por modelos comportamentais;
- **modelagem comportamental** oferece visão de interação de um software na presença de aspectos. Os principais elementos comportamentais são: i) *objetos*; ii) *instâncias de aspectos*; e iii) suas *colaborações* e *interações*. Esta modelagem caracteriza o comportamento de aspectos em termos da forma como interagem com objetos;
- **modelagem de processo de combinação (ou composicional)** descreve as visões estáticas e de interação de um software depois da combinação de modelos de objetos e modelos de aspectos. Os principais elementos composicionais são: i) *classes combinadas*; ii) *colaborações combinadas*; e iii) *interações combinadas*. Esses elementos são apresentados em diagramas de processo de combinação. Os resultados de combinação dependem da estratégia de combinação adotada, que depende do modelo de implementação suportado pela ferramenta ou linguagem de programação OA[5].

As próximas subseções apresentam a descrição geral dos diagramas e dos elementos de modelagem presentes em *aSideML*.

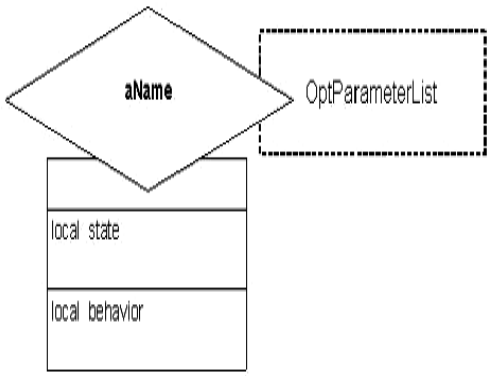
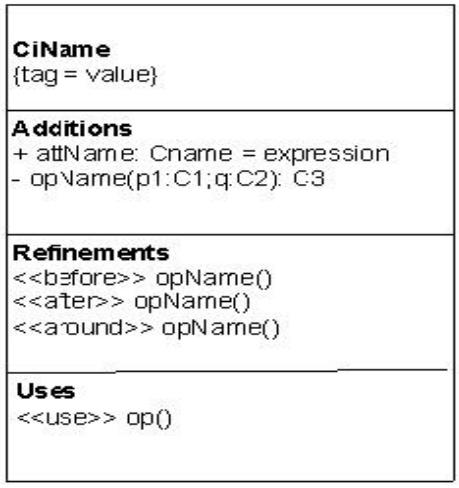
2.4.1.1. Diagrama de Aspectos

Um Diagrama de Aspectos fornece uma descrição completa de um aspecto. A descrição incorpora as interfaces transversais, as características locais e os relacionamentos de herança. Cada característica transversal comportamental pode ser visualizada em um

[5] O escopo deste trabalho baseia-se na definição e na incorporação de características de testabilidade nos artefatos gerados durante a fase de projeto de software AO. Assim, a modelagem de processo de combinação não é abordada neste trabalho.

Diagrama de Colaboração Aspectual. A Tabela 2-1 apresenta breve descrição e respectivas representações gráficas dos elementos que compõem um Diagrama de Aspectos.

Tabela 2-1 – Elementos do Diagrama de Aspectos

Elemento	Representação	Descrição
Aspecto		Um aspecto é uma descrição de um conjunto de características que alteram a estrutura e o comportamento de classes usando entrecortes de forma sistêmica.
Interface Transversal e Característica Transversal		Características transversais descrevem uma propriedade nomeada (atributo ou operação) definida em um aspecto que pode afetar um ou mais elementos base em locais específicos usando entrecortes. Uma interface transversal é conjuntos de características transversais com nome associado, que caracterizam o comportamento entrecortante de aspectos.

Um aspecto é representado como um retângulo tracejado, com um losango contendo o nome do aspecto, que toca um retângulo usado para descrever suas características locais (atributos e métodos). Um aspecto pode conter vários elementos internos, como interfaces transversais, classes, interfaces e relacionamentos. A notação para descrever interfaces transversais é apresentada na Figura 2-6. Esta figura apresenta *Observation*, um aspecto que descreve o padrão de projeto (*design pattern*) *Observer* [GAMMA *et al*. 1996].

Um aspecto pode ser representado de forma condensada, onde se omite as informações sobre seus elementos internos, exceto os nomes das interfaces transversais; cada interface transversal é exibida como um pequeno círculo com seu nome colocado ao lado (Figura 2-7). O círculo está ligado por uma linha sólida ao losango que representa o

aspecto. A visão condensada é o padrão para a apresentação de aspectos em Diagramas de Classes Estendidos.

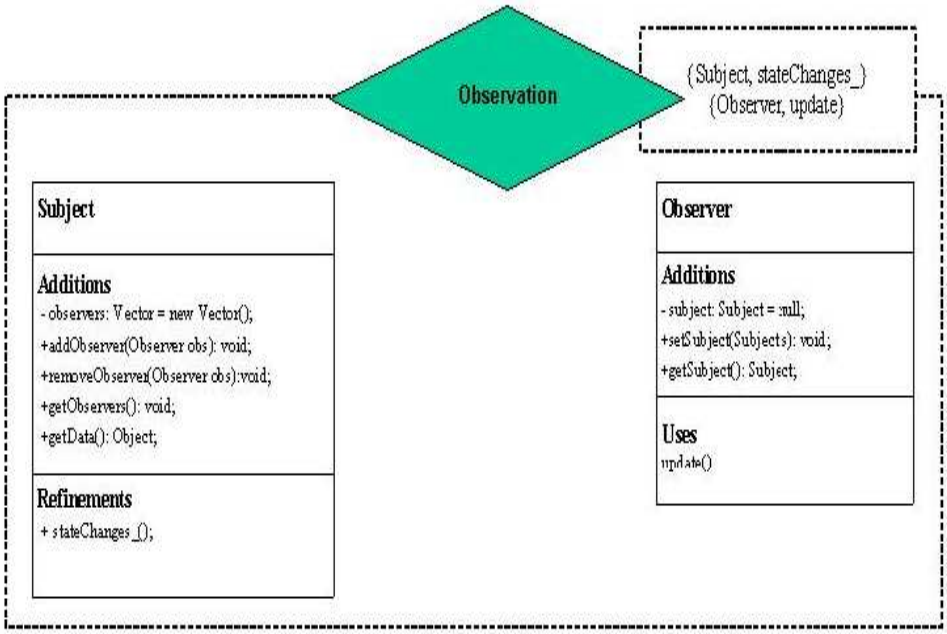


Figura 2-6 – Padrão de Projeto *Observer* Modelado como um Aspecto (Fonte: Chavez (2004))

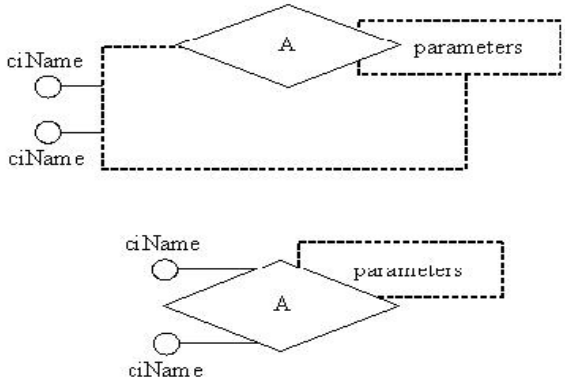


Figura 2-7 – Declaração Condensada de Aspecto (Fonte: Chavez (2004))

Uma interface transversal é representada por um retângulo com linhas sólidas com compartimentos separados por linhas horizontais. No primeiro compartimento, o nome no topo representa o nome da interface transversal. O segundo compartimento contém um conjunto de características comportamentais e estruturais que o aspecto adiciona a uma classe base (*additions*). O terceiro e o quarto compartimentos (*refinements* e *redefinitions*) contêm os conjuntos de características transversais comportamentais que refinam ou redefinem algumas características comportamentais existentes na classe base (obrigatoriamente, um deles não deve estar vazio). Um compartimento opcional (*uses*)

pode ser fornecido para definir um conjunto de assinaturas para características requisitadas (*required features*) usadas pelo aspecto.

Características transversais comportamentais são listadas em diferentes compartimentos, dependendo do tipo de melhoria (adição, refinamento, redefinição). Características transversais comportamentais que permitem adição são listadas no compartimento *Additions* e não são parametrizadas. Características transversais comportamentais que permitem refinamento são listadas no compartimento *Refinements*. Nesse compartimento, o nome das operações é representado com o símbolo "_" (*underscore*), em três combinações permitidas: `_op`, `op_` e `_op_`. Esses adornos indicam que a operação entrecortante oferece o comportamento que deve ser combinado antes, depois ou antes/depois do comportamento da operação base. Características transversais comportamentais que permitem redefinição são listadas no compartimento *Redefinitions*. Neste caso, comportamento de `op` é redefinido (ou sobreposto) pelo comportamento de `_op_`.

Adornos textuais (*before*, *after*, *around* e *use*) podem substituir o símbolo _. Por exemplo, pode-se escrever *before* `op` em vez de `_op`. A Tabela 2-1 apresenta uma interface transversal na qual o símbolo "_" é substituído por adornos textuais que descrevem o tipo de entrecorte.

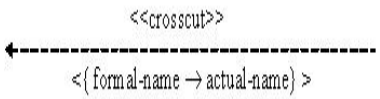
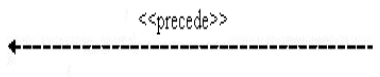
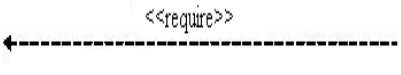
2.4.1.2. Diagrama de Classes Estendido

O Diagrama de Classes Estendido é um grafo de aspectos e de classes conectado por seus diferentes relacionamentos estáticos e oferece uma apresentação gráfica da visão estática de projeto de um software em que as classes e os aspectos residem como cidadãos de primeira classe. Cada aspecto pode ser apresentado em uma visão detalhada separada por um Diagrama de Aspectos.

O Diagrama de Classes Estendido agrega novos elementos de modelagem ao Diagrama de Classes definido pela UML; por isso, alguns elementos deste diagrama são omitidos neste trabalho. A Tabela 2-2 apresenta breve descrição e respectivas representações gráficas dos elementos que compõem o Diagrama de Classes Estendido. Uma tabela equivalente à Tabela 2-2 que apresenta a descrição e a representação gráfica

dos elementos que compõem o Diagrama de Classes da UML pode ser encontrada em Souza (2003).

Tabela 2-2 – Elementos do Diagrama de Classes Estendido (Fonte: Chavez (2004))

Elemento	Representação	Descrição
Entrecorte		Classifica um relacionamento entre um aspecto e um elemento base. Ele especifica que o aspecto deve atravessar os limites do elemento base em pontos de combinação bem definidos e modificar incrementalmente a base nesses pontos.
Precedência		Define que um aspecto (o cliente) precede outro aspecto (o fornecedor). Isso significa que o comportamento do aspecto cliente tem precedência em relação ao comportamento do aspecto fornecedor no tipo de entrecorte que esperam realizar.
Requirement		Oferece uma conexão explícita entre um aspecto (o cliente) que requer a presença de outro aspecto (o fornecedor) para sua implementação ou funcionamento correto.

Um relacionamento de entrecorte é representado como uma seta tracejada, com a parte final no elemento entrecortante e a parte inicial no elemento base, e o estereótipo <<*crosscut*>>. Cada associação relaciona um parâmetro definido na interface transversal do aspecto (nome da interface ou nome da operação) a um nome (ou seqüência de nomes) definido na interface de uma classe (nome de classe ou um nome de método). O relacionamento *precedence* é apresentado como uma seta tracejada entre dois elementos do modelo e rotulada com o estereótipo <<*precede*>>. O aspecto na parte final da seta (o cliente) precede o aspecto na parte inicial da seta (o fornecedor). O relacionamento *requirement* é apresentado como uma seta tracejada entre dois elementos do modelo e rotulada com o estereótipo <<*require*>>. O aspecto na parte final da seta (o cliente) requer a presença do aspecto na cabeça da seta (o fornecedor).

A Figura 2-8 apresenta dois relacionamentos de entrecorteque associam o aspecto *Observation* a *Button* (ligando *stateChange* a *click*) e *ColorLabel* (ligando *update* a *colorCycle*). O aspecto *Observation* intercepta o método *click* da classe *Button*, a estrutura da classe *Button* recebe atributos e operações novos listados no compartimento *Additions*

do aspecto *Observation* e seu comportamento é alterado por *Observation* no ponto de combinação definido (*click*). *Observation* também afeta a estrutura de instâncias de *ColorLabel*.

Ainda na Figura 2-8, pode-se observar o aspecto *Tracing*, que rastreia chamadas a *click* e *colorLabel*. Um relacionamento *precedence* é representado do aspecto *Observation* ao aspecto *Tracing*. Isso significa que o comportamento de notificação ocorre antes do comportamento de rastreamento. A Figura 2-9 apresenta dois relacionamentos *requirement*: i) do aspecto *Autonomy* ao aspecto *Interaction*; e ii) do aspecto *Adaptation* ao aspecto *Interaction*.

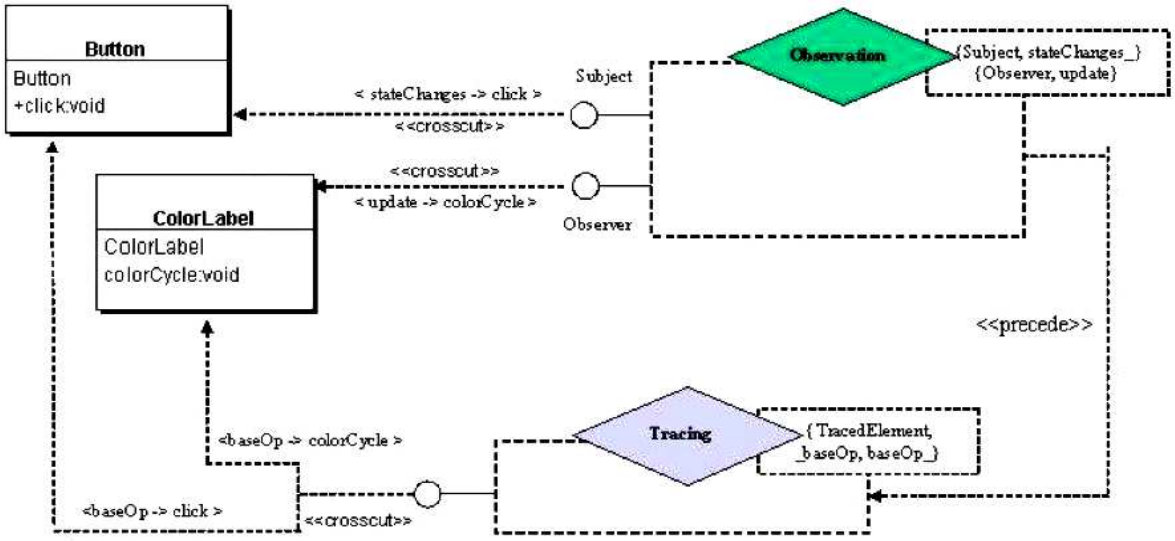


Figura 2-8 – Diagrama de Classes Estendido (Fonte: Chavez (2004))

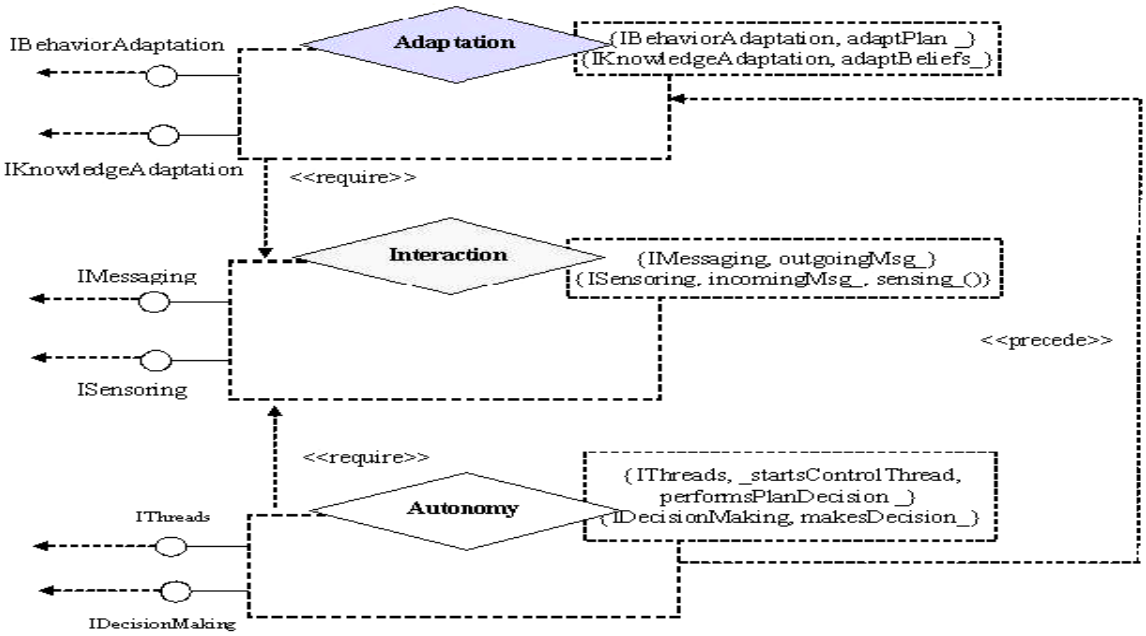
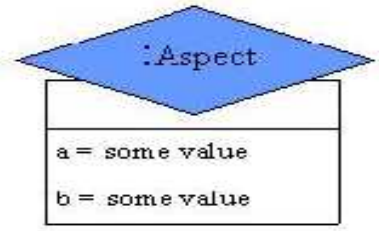
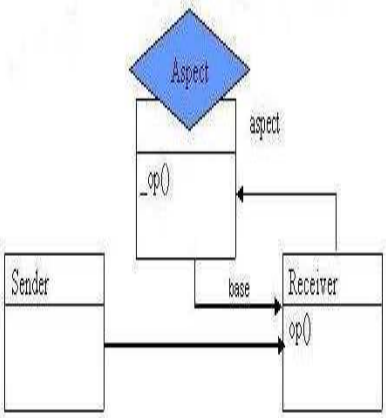


Figura 2-9 – Relacionamento *Requirement* (Fonte: Chavez (2004))

2.4.1.3. Diagrama de Colaboração e Diagrama de Interação Aspectuais

Um Diagrama de Colaboração Aspectual oferece uma apresentação gráfica de uma colaboração aspectual, um tipo especial de colaboração que modela a realização de uma operação transversal definida dentro de um aspecto. Esse diagrama oferece suporte à visão de interação (*interaction view*) que envolve instâncias de aspectos e elementos base. A Tabela 2-3 apresenta uma breve descrição e as respectivas representações gráficas dos elementos que compõem o Diagrama de Colaboração Aspectual.

Tabela 2-3 – Elementos do Diagrama de Colaboração Aspectual (Fonte: Chavez (2004))

Elemento	Representação	Descrição
Instância de Aspectos		Tem identidade, pode ter estado, não podem ser explicitamente criadas ou destruídas e só podem ser tratadas dentro de aspectos. Elas são elementos de modelagem que podem ser objetos comuns no nível de implementação.
Colaboração Aspectual		Define um conjunto de papéis e um conjunto de interações que descreve a estrutura e a comunicação entre a instância de aspecto e de objetos que exercem os papéis definidos.
Interação Aspectual	(Ver Figura 2-10)	Oferece uma conexão explícita entre um aspecto (o cliente) que requer a presença de outro aspecto (o fornecedor) para sua implementação ou funcionamento correto.

A notação da instância de aspecto é derivada da notação de objeto, sendo representada por um losango acima. Por definição, as instâncias de aspecto são instâncias anônimas, portanto o nome da instância de aspecto é sempre omitido, mas o nome do aspecto que dá origem está sempre presente, usando a sintaxe: *aspectName*.

Uma colaboração aspectual possui (Figura 2-10): i) uma parte estática que descreve os papéis que objetos e instâncias de aspecto exercem; e ii) uma parte dinâmica que mostra os fluxos de mensagens ao longo do tempo para realizar o comportamento entrecortante de acordo com os papéis. Há três papéis de associação: i) um papel do aspecto para um objeto, que denota a base; ii) um papel do emissor para o receptor; e iii) um papel de associação do objeto base para a instância de aspecto.

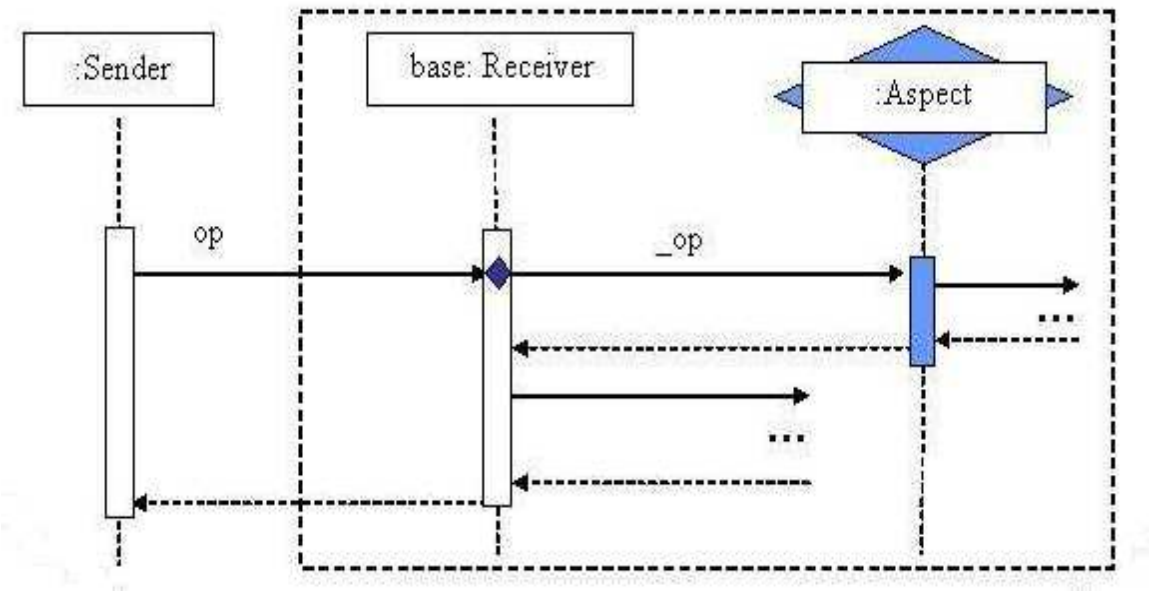


Figura 2-10 – Interação Aspectual (Diagrama de Seqüência com Adornos) (Fonte: Chavez (2004))

A parte superior da Figura 2-11 apresenta uma colaboração aspectual para a operação *stateChange*. Ela descreve o comportamento estendido de um objeto de um tipo denotado por *Sender* que chama a operação *stateChange*. As melhorias incorporam a inserção de um *link* chamado *aspect* a partir do objeto base (denotado por *Sender*) até a instância de aspecto e a inserção de uma chamada na operação de entrecorte (*stateChange_*) na sombra estática associada com o ponto de combinação dinâmico. Nesse caso, a chamada a *stateChange_* é inserida depois da chamada a *stateChange*.

As interações aspectuais são mostradas como Diagramas de Seqüência (da UML) com alguns adornos. A interação aspectual para a operação *stateChange* é apresentada na parte inferior da Figura 2-11. As colaborações e as interações aspectuais são apresentadas nos Diagramas de Colaboração Aspectuais e Diagramas de Interação Aspectuais, respectivamente.

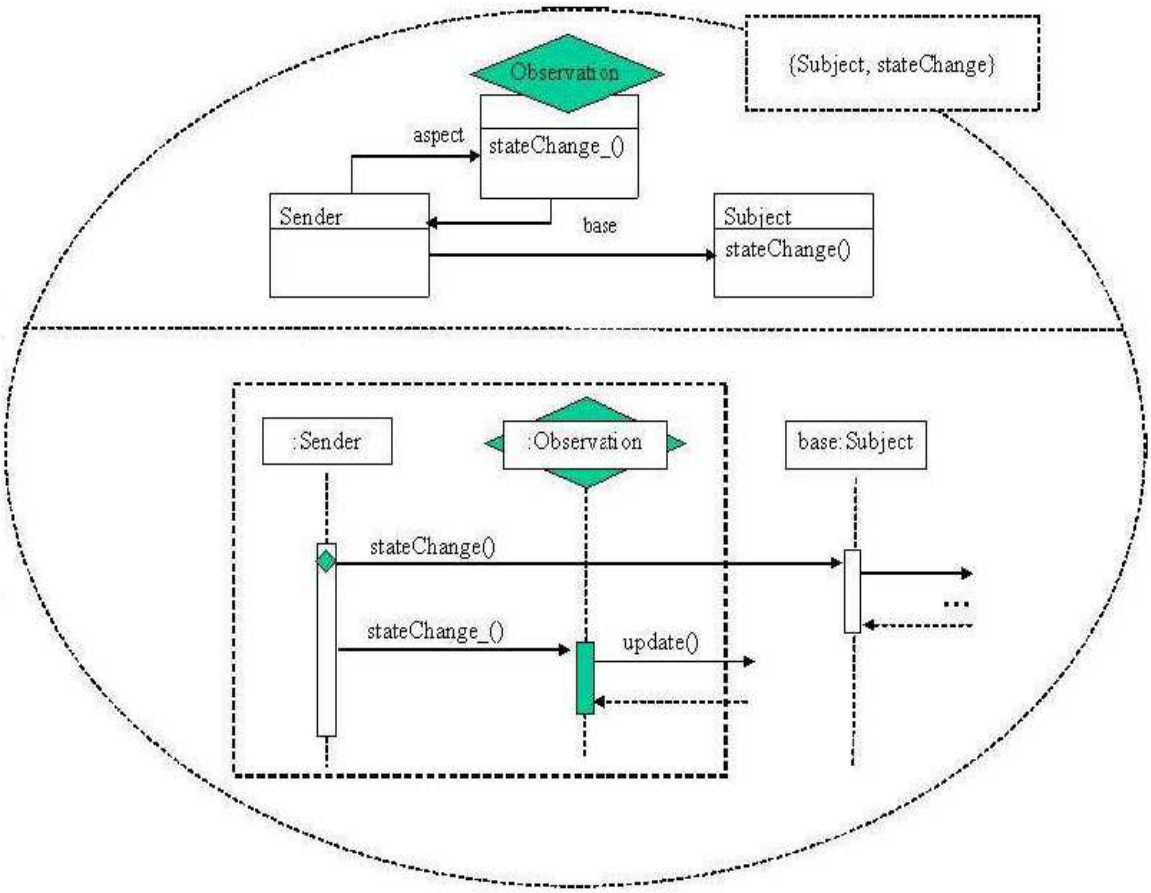


Figura 2-11 – Diagrama de Colaboração Aspectual para *stateChange_* (Fonte: Chavez (2004))

2.5. Considerações Finais

Neste capítulo, foram apresentados os principais conceitos sobre OA, abordando a sua origem e evolução e as abordagens sobre MOA. Pelo exposto, OA traz vários benefícios mediante a possibilidade de tornar os programas mais modulares. O objetivo da técnica é separar interesses transversais (*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 software.

Foram apresentadas algumas das abordagens para MOA existentes na literatura, enfatizando a abordagem *aSideML*, que consiste de uma linguagem de modelagem criada para especificar e comunicar projetos OA. Não existe ainda uma convenção ou uma abordagem padrão, tal como a UML é para a OO, gerando a necessidade de levar as vantagens dos aspectos às etapas anteriores do processo de desenvolvimento de software. Isso decorre, pois as pesquisas em modelagem são recentes em relação às pesquisas em implementação.

3. TESTE DE SOFTWARE

3.1. Considerações Iniciais

Este capítulo apresenta conceitos e terminologias envolvidos com as atividades de teste que devem ser realizadas durante o processo de desenvolvimento de software. Além disso, ele apresenta uma descrição sucinta da norma ISO/IEC 9126 e uma discussão sintética de testes e de testabilidade de software.

A seção 3.2 apresenta a definição de qualidade e de qualidade de software, sob a visão de alguns autores relevantes da literatura. A seção 3.3 e a seção 3.4 apresentam breve discussão sobre a norma ISO/IEC 9126 sob o contexto de teste e de testabilidade de software. A seção 3.5 descreve as atividades relacionadas com o teste que devem ser executadas durante as fases do desenvolvimento de software. A seção 3.6 apresenta as diversas técnicas de teste existentes que auxiliam no projeto de casos de teste. A seção 3.7 e a seção 3.8 apresentam as estratégias de teste definidas na literatura para software OO e software OA.

3.2. Qualidade de Software

Conforme Costa (2005), a literatura oferece várias definições de qualidade. A seguir, são apresentadas algumas destas definições:

- 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 é um grau previsível de uniformidade e dependência, baixo custo e satisfação no mercado, ou seja, qualidade é 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 é uma estratégia que requer percepção de todos na empresa. Além disso, ele diz que é um compromisso para com a excelência, um modo de vida corporativa, um modo de gerenciamento. A liderança para a qualidade, a tecnologia moderna da qualidade e o compromisso organizacional são os passos da sua abordagem [CÔRTES; CHIOSSI, 2001];
- ISO Std. 8402 (1990) caracteriza o termo qualidade como a capacidade de satisfazer as necessidades implícitas e explícitas dos seus usuários;

- Juran; 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 de maneira adequada a sua função.

Considerando o contexto de software, o termo qualidade recebeu uma definição especial na norma ISO/IEC 9126 [ISO Std. 9126, 1991; ISO Std. 9126 , 2001; ABNT NBR13596, 1996], na qual a qualidade é a totalidade das características de um software, que lhe confere a capacidade de satisfazer às necessidades explícitas e implícitas. As necessidades explícitas são requisitos para o desenvolvimento de software explicitamente sugerido pelo cliente ou obtidos na modelagem do problema. Por outro lado, as necessidades implícitas podem não ser explicitadas nem documentadas, mas devem estar presentes quando o software é usado em condições particulares; elas são necessidades subjetivas dos usuários, inclusive operadores, destinatários dos resultados do 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].

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, pois:

- o termo qualidade possui um único sentido, mas com conceitos multidimensionais – a quem se interessa, os 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 referir-se à qualidade com sentido geral e outro grupo referir-se 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 software contida na norma ISO/IEC 9126 [ISO Std. 9126, 1991; ISO Std. 9126, 2001; ABNT NBR13596, 1996] foi usada como referência, pois apresenta caráter geral e altamente conceitual, habilitando ser aplicada a uma grande variedade de ambientes e condições.

3.3. 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 software. A qualidade de um dispositivo mecânico é freqüentemente medida em termos de tempo médio entre suas falhas, sendo uma medida da capacidade do dispositivo suportar desgaste. O 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 software. Além de ter força de norma internacional, ela cobre os aspectos mais importantes para qualquer software. Ainda assim, existe uma versão traduzida pela Associação Brasileira de Normas Técnicas (ABNT) para o seu uso no Brasil, a qual foi publicada em abril de 1996, propiciando a difusão dos conceitos de qualidade de software e ampliando a terminologia específica e a própria cultura da área entre os profissionais [ABNT NBR 13596, 1996].

A nova versão da ISSO/IEC 9126 (2001) é constituída de quatro partes:

- ISO/IEC 9126-1: define características e subcaracterísticas de qualidade, introduzindo 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 métricas externas para avaliar e garantir a qualidade externa de software;
- ISO/IEC 9126-3: define métricas internas para avaliar e garantir a qualidade interna de software;

- ISO/IEC 9126-4: define métricas para avaliar e garantir a qualidade em uso de software.

A norma ISO/IEC 9126 define seis características de qualidade de software e suas respectivas subcaracterísticas. A Tabela 3-1 apresenta esta organização, com perguntas chaves para melhor entendimento.

A seguir, é apresentada a definição das características e das subcaracterísticas da Norma ISO/IEC 9126:

- **Confiabilidade**: atributos que evidenciam a capacidade do software em manter bom nível de desempenho sob determinadas condições e em determinado período de tempo;
    - *Maturidade*: atributos do software que evidenciam a freqüência de falhas por defeitos no software;
    - *Recuperabilidade*: atributos do software que evidenciam a sua capacidade de retornar ao funcionamento normal, mediante a recuperação dos dados após as falhas, o tempo e os esforços necessários para isso;
    - *Tolerância a Falhas*: atributos do software que evidenciam a sua capacidade em continuar o nível de desempenho especificado nos casos de falhas no software ou de violação nas interfaces especificadas;

- **Eficiência**: atributos que evidenciam o relacionamento entre o nível de desempenho do software e a quantidade de recursos que utiliza sob determinadas condições;
    - *Comportamento em Relação a Recursos*: atributos do software que evidenciam a quantidade de recursos usados e a duração de seu uso na execução das funções;
    - *Comportamento em Relação ao Tempo*: atributos do software que evidenciam o seu tempo de processamento e de resposta e a velocidade na execução das funções;
- **Funcionalidade**: atributos que evidenciam a existência de um conjunto de funções e suas propriedades que satisfazem às necessidades implícitas e explícitas;
    - *Adequação*: atributos do software que evidenciam a presença de um conjunto de funções e a sua adequação para as tarefas especificadas;

Tabela 3-1 – Características e 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 usar o software?	Inteligibilidade	É fácil entender os conceitos usados?
	Apreensibilidade	É fácil aprender a usar?
	Operacionalidade	É fácil operar e controlar a operação?
Eficiência Os recursos e os tempos usados são compatíveis com o nível de desempenho requerido para o produto?	Comportamento em relação ao tempo	Qual é o tempo de resposta de processamento?
	Comportamento em relação aos recursos	Quanto recurso utiliza?
Manutenibilidade Há facilidade para realizar 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 de faz alterações?
	Testabilidade	É fácil testar quando se faz alterações?
Portabilidade É possível usar 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 software considerado?
	Capacidade para ser instalado	É fácil instalar em outros ambientes?
	Capacidade para substituir	É fácil substituir por outro software?
	Conformidade	Está de acordo com padrões ou convenções de portabilidade?

    - *Acurácia*: atributos do software que evidenciam a geração de resultados ou efeitos corretos ou conforme acordados;
    - *Conformidade*: atributos do software que evidenciam o atendimento de normas, padrões, convenções e/ou regulamentações previstas em leis e descrições similares, relacionadas à aplicação;
    - *Interoperabilidade*: atributos do software que evidenciam a sua capacidade de interagir com outro software;
    - *Segurança de Acesso*: atributos do software que evidenciam a sua capacidade de evitar acessos não autorizados, acidentais ou deliberados aos dados e ao software;
- **Manutenibilidade**: de atributos que evidenciam o esforço necessário para realizar modificações específicas no software;
    - *Analisabilidade*: atributos do software que evidenciam o esforço necessário para diagnosticar deficiências ou causas de falhas, ou para identificar partes a serem modificadas;
    - *Estabilidade*: atributos do software que evidenciam o risco de efeitos inesperados, ocasionados por modificações;
    - *Modificabilidade*: atributos do software que evidenciam o esforço necessário para modificá-lo, remover seus defeitos ou adaptá-lo a mudanças ambientais;
    - *Testabilidade*: atributos do software que evidenciam o esforço necessário para validar o software modificado;
- **Portabilidade**: atributos que evidenciam a capacidade do software de ser transferido de um ambiente para outro;
    - *Adaptabilidade*: atributos do 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 software considerado;
    - *Capacidade para ser Instalado*: atributos do software que evidenciam o esforço necessário para a sua instalação em um determinado ambiente;
    - *Capacidade para Substituir*: atributos do software que evidenciam o esforço necessário para substituir um outro software, no ambiente estabelecido para este segundo;

    - *Conformidade*: atributos do software que o tornam de acordo com os padrões ou com as convenções relacionadas à portabilidade;
- **Usabilidade**: atributos que evidenciam o esforço necessário para poder utilizar o software, bem como o julgamento individual desse uso, por um conjunto implícito ou explícito de usuários.
    - *Apreensibilidade*: atributos do software que evidenciam o esforço do usuário para aprender a sua aplicação;
    - *Inteligibilidade*: atributos do software que evidenciam o esforço do usuário para reconhecer o conceito lógico e a sua aplicabilidade;
    - *Operacionalidade*: atributos do software que evidenciam o esforço do usuário para a sua operação e o controle desta operação.

3.4. Testabilidade

De acordo com Chatterjee (2008), testabilidade é um requisito não-funcional importante para a equipe de testes e para os usuários envolvidos no teste de aceitação. A testabilidade pode ser definida como a propriedade que mensura o quão fácil é testar trecho de código ou funcionalidade de um software específico.

Bach (1994) *apud* Pressman (2006) descreve a testabilidade de software como a facilidade com que ele pode ser testado. Ainda segundo os mesmos autores, um conjunto de características que levam um software a ser testável é:

- **Operabilidade**. “Quanto melhor funciona, mais eficientemente pode ser testado”;
- **Observabilidade**. “Facilita a visibilidade dos pontos a serem testados”;
- **Controlabilidade**. “Quanto mais se pode controlar o software, melhor será o processo otimizado e a automatização do teste”;
- **Decomponibilidade**. “Controlando o escopo do teste, pode-se isolar problemas mais rapidamente e refazer os testes mais racionalmente”;
- **Simplicidade**. “Quanto menos há a testar, mais rapidamente pode-se testá-lo”;
- **Estabilidade**. “Quanto menos modificações, menos interrupções no teste”;
- **Compreensibilidade**. “Quanto mais informações possuir, mais racionalmente testa-se”.

Um software tem alta testabilidade se ele tende a expor suas falhas durante os testes. Caso contrário, ele tende a ocultar as falhas durante a fase de testes. Requisitos incompletos, desatualizados, ambíguos ou contraditórios trazem baixa testabilidade [TANNOURI, 2006].

Um ciclo típico de desenvolvimento de software envolve elicitação de requisitos, análise, projeto, codificação, testes e manutenção. Assumindo que um software possua alta testabilidade, as seguintes vantagens em relação ao seu custo de desenvolvimento, à sua qualidade e à sua manutenibilidade podem ser [CHATTERJEE, 2008]:

- A maioria dos defeitos provavelmente será detectada e corrigida antes do software ser liberado para o cliente;
- Será mais fácil e menos dispendioso manter este software;
- A possibilidade de alcançar a satisfação dos clientes é elevada.

Pelos motivos apresentados anteriormente, pode-se concluir que testabilidade é um importante atributo para manutenção e garantia de qualidade de software [CHATTERJEE, 2008]. A testabilidade pode ser incorporada nos vários estágios do desenvolvimento de software [TANNOURI, 2006]:

- **Fase de Especificação do Software:** Durante o processo de revisão da especificação, a equipe de testes deve ser questionada quanto ao seu entendimento dos requisitos;
- **Fase de Detalhamento do Projeto:** Para incorporar testabilidade nesta fase, as entradas e as saídas devem estar indicadas claramente. O projeto deve elucidar o caminho do software para que a equipe de teste saiba quais programas são tratados em cada cenário;
- **Fase de Codificação:** Esta fase é a mais crucial. A cobertura de cenários que a aplicação deve e não deve fazer deve passar por testes unitários;
- **Fase de Testes:** O plano e os *scripts* de teste projetados devem cobrir as medidas de testabilidade e ser testados de acordo com os requisitos funcionais e não-funcionais.

3.5. Teste de Software: Conceitos e Atividades Relacionadas

Antes de iniciar a discussão sobre teste de software, é importante esclarecer o conceito de defeito, erro e falha. Vários esforços têm sido realizados pelo IEEE (*Institute of Eletrical and Electronics Engineers*) com o intuito de padronizar a terminologia usada

em vários campos do conhecimento, dentre eles o da Engenharia de Software. Entretanto, existe uma divergência entre os pesquisadores da área de teste de software e de outras áreas correlatas sobre a terminologia adotada no Brasil, no que se refere aos termos defeito, erro e falha [SILVEIRA, 2007].

A norma IEEE número 610.12-1990 define os termos defeito, erro e falha da seguinte maneira [IEEE, 1990]:

- **defeito**: passo, processo ou definição de dados incorretos, por exemplo, uma instrução ou comando incorreto;
- **erro**: diferença entre valor obtido e valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do software constitui um erro;
- **falha**: produção de uma saída incorreta com relação à especificação.

Defeitos fazem parte do universo físico (a aplicação propriamente dita) e são causados por pessoas, por exemplo, pelo mau uso de uma tecnologia. Defeitos podem ocasionar a manifestação de erros em um produto, ou seja, a construção de software de forma diferente ao que foi especificado (universo de informação). Por fim, os erros geram falhas, comportamentos inesperados em um software que afetam diretamente o usuário final da aplicação (universo do usuário) e pode inviabilizar o uso do software [NETO, 2007]. A Figura 3-1 expressa a diferença entre os conceitos apresentados anteriormente.

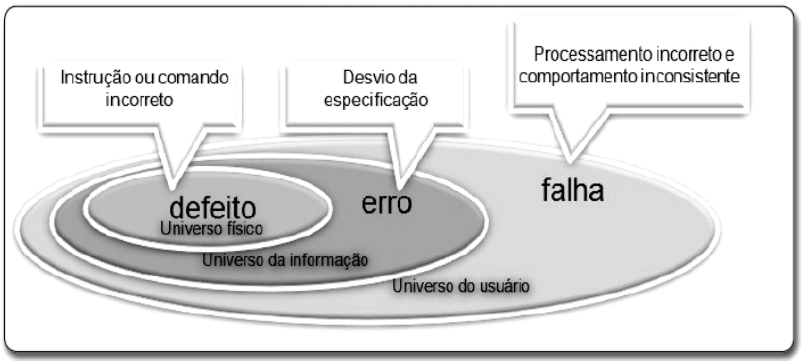


Figura 3-1 – Defeito x Erro x Falha (Fonte: NETO (2007))

O processo de Verificação e Validação compreende as atividades e as análises que asseguram que o software cumpre com as especificações e atende às necessidades para as quais ele foi desenvolvido [SOMMEVILLE, 2001]. A atividade de teste é uma das técnicas mais utilizadas de verificação e validação, constituindo-se em um dos elementos para

fornecer evidências sobre a confiabilidade do software em complemento a outras atividades, por exemplo, o uso de revisões e de técnicas formais e rigorosas de especificação e validação [MALDONADO, 1991].

Segundo Myers (1979), o principal objetivo da atividade de teste é revelar a presença de erros. Neste sentido, uma atividade de teste pode ser considerada bem sucedida quando consegue descobrir algum novo erro no software que está sendo testado. As atividades realizadas durante a execução de testes são consideradas dispendiosas em relação às demais fases do desenvolvimento de software [PRESSMAN, 2006; RAMAKRISHNAN, 1999]. Esta afirmação pode ser justificada por diferentes fatores:

- realização de testes sem planejamento: a fase de testes tende a ser longa e imprevisível;
- não formalização da arquitetura do software: mais esforço será despendido durante a identificação dos testes necessários, uma vez que as restrições existentes no software não foram claramente especificadas;
- não repetibilidade dos testes (testes de regressão): dados de entrada não documentados implicam em não uniformidade da verificação e re-trabalho;
- falta de histórico das atividades executadas: as atividades relacionadas com o teste "pesam" na estimativa de tempo e de recursos e, conseqüentemente, quando não consideradas nos cronogramas, surpreendem negativamente o gerente de projeto e o cliente.

Sendo assim, é necessário que as atividades relacionadas com o teste sejam identificadas de forma mais realística, tendo como base um processo de desenvolvimento onde, para cada fase do desenvolvimento, existam atividades relacionadas com o teste [BLINDER, 1995; KÖLLING; ROSENBERG, 1998; TSAI *et al.*, 1997]. As principais atividades relacionadas com o teste de software são (Figura 3-2) [AMBLER, 1998; FILHO, 2001; MCGREGOR; SYKES, 2001; BINDER, 2000; JACOBSON *et al.*, 1992; LI; WAHL, 1999; PRESSMAN, 2006]:

- **Planejamento dos Testes.** O plano de teste descreve o escopo, os recursos e o cronograma das atividades relacionadas com o teste. Para isso, são identificadas as características a serem testadas, as tarefas de teste a serem realizadas, as pessoas responsáveis pela realização de cada tarefa, os riscos possíveis e o plano de contingência;

- **Documentação dos Testes.** Os documentos de teste são gerados e baseados nas informações obtidas com o planejamento elaborado e nos artefatos do software. A atividade de documentação dos testes deve garantir que documentos adequados sejam gerados, permitindo que os casos de teste identificados possam ser re-executados ou reutilizados em novo software. Os documentos descritos no padrão IEEE de documentação de teste de software (IEEE Std 829, 1998) podem ser considerados como bom exemplo, visto que a norma é altamente conceituada entre os pesquisadores;

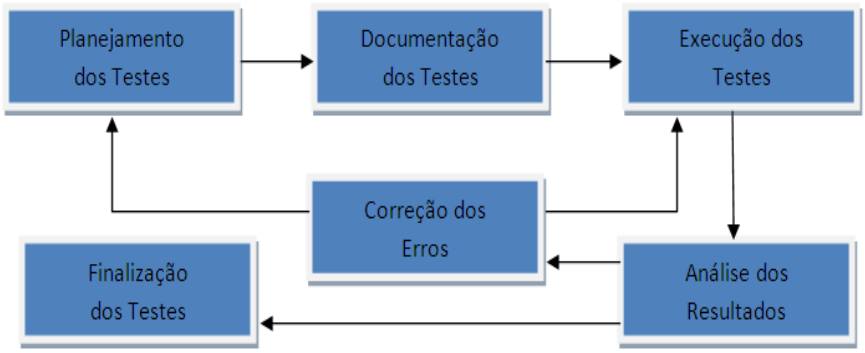


Figura 3-2 – Atividades Relacionadas com o Teste de *Software*

- **Execução dos Testes.** Os testes são executados conforme o plano e os documentos de teste;
- **Análise dos Resultados.** Análise dos resultados é realizada com o intuito de verificar se os testes foram ou não aprovados;
- **Correção dos Erros.** Caso sejam identificados erros, as alterações necessárias são feitas e os testes devem ser novamente executados ou, quando necessário, devem ser definidos novos testes para depois executá-los. Além disso, é necessário avaliar se os testes realizados foram aprovados antes das alterações, com o intuito de identificar a necessidade ou não de executá-los novamente. A execução de testes anteriores, após a correção, é denominada teste de regressão. Este teste é relevante em razão de garantir que as mudanças feitas no software não acarretaram erros nas partes que funcionavam adequadamente antes das alterações;
- **Finalização dos Testes.** Caso não sejam identificados erros, as atividades relacionadas com o teste são finalizadas.

A especificação de teste deve detalhar as características a serem testadas por um conjunto de casos e procedimentos de teste que deve ser realizado para validar as características apresentadas no plano de teste, além de fornecer os critérios de aceitação

dos resultados. Um caso de teste documenta os dados de entrada a serem usados durante a execução dos testes e os valores de saída desejados. O procedimento de teste especifica os passos que devem ser seguidos para exercitar o item a ser testado com os casos de teste especificados [SOUZA, 2003].

A elaboração destes documentos deve contar com um bom entendimento do software a ser testado e baseia-se nos artefatos gerados ao longo da análise, do projeto e da implementação. Portanto, se estes artefatos apresentarem características de testabilidade, eles podem contribuir para melhorar a qualidade dos documentos de teste e, conseqüentemente, do software.

3.6. Técnica de Teste de Software

Para a realização de testes em software, é necessário definir casos de teste com o intuito de exercitar as suas diferentes características [SOUZA, 2003]. Existem na literatura, diversas técnicas de teste definidas para o projeto de casos de teste de software, as quais podem ser divididas nas seguintes categorias [AMBLER, 1998; BERARD, 1994; BINDER, 2000; COLANZI, 1999; JACOBSON *et al.*, 1992; PERRY, 2000; PRESSMAN, 2006]: i) teste estrutural; ii) teste funcional; e iii) teste baseado em estados.

As técnicas de teste citadas anteriormente se diferenciam, basicamente, pela origem da informação usada para avaliar ou construir os conjuntos de casos de teste [COLANZI, 1999] e devem ser utilizadas de forma complementar para que as diferentes características do software sejam adequadamente testadas.

3.6.1. Técnica de Teste Estrutural

A técnica de teste estrutural avalia o comportamento interno do software (Figura 3-3). O deste teste, conhecido como teste de caixa-branca, é exercitar partes específicas do código do software, para garantir que a parte interna dos módulos é adequadamente testada. Nesta técnica, a estrutura interna do código-fonte deve ser conhecida [SOUZA, 2003].

Geralmente, os critérios desta técnica usam o Grafo de Fluxo de Controle (GFC), que representa o controle lógico de um software [PRESSMAN, 2006]. Para a construção do GFC, é necessário o uso do conceito de blocos de comandos. Um bloco de comando compreende uma seqüência de instruções sempre executadas de forma consecutiva. Assim,

desvios de controle só ocorrem para o começo ou a partir do final de um bloco [SILVEIRA, 2007].

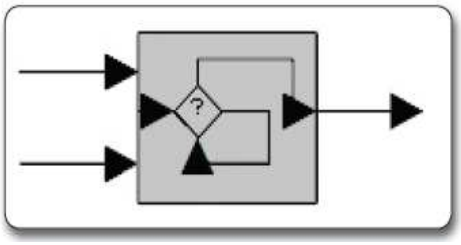


Figura 3-3 – Técnica de Teste Estrutural (Fonte: NETO (2007))

Como citado anteriormente, o teste de caixa-branca requer que o código-fonte seja conhecido. A Figura 3-4 apresenta um código-fonte de um programa escrito na linguagem C que valida um identificador recebido como parâmetro e a Figura 3-5 apresenta o GFC extraído a partir do código. A partir do GFC, deve ser escolhido algum critério para a geração dos casos de teste estruturais para o software.

```
/*01*/{
/*01*/char achar;
/*01*/int length, valid_id;
/*01*/length = 0;
/*01*/printf("Digite um possível identificador\n");
/*01*/printf("seguido por <ENTER>: ");
/*01*/achar=fgetc (stdin);
/*01*/valid_id=valid_starter (achar);
/*01*/if (valid_id)
/*02*/length=1;
/*03*/achar=fgetc (stdin);
/*04*/while (achar != '\n'){
/*05*/ if (!(valid_follower (achar)))
/*06*/    valid_id = 0;
/*07*/ length++;
/*07*/ achar = fgetc (stdin);
/*07*/}
/*08*/if (valid_id && (length >= 1) && (length < 6))
/*09*/ printf ("Valido\n");
/*10*/else
/*10*/ printf ("Invalido\n");
/*11*/}
```

Figura 3-4 – Código Fonte do Programa Identifier.c (Fonte: NETO (2007))

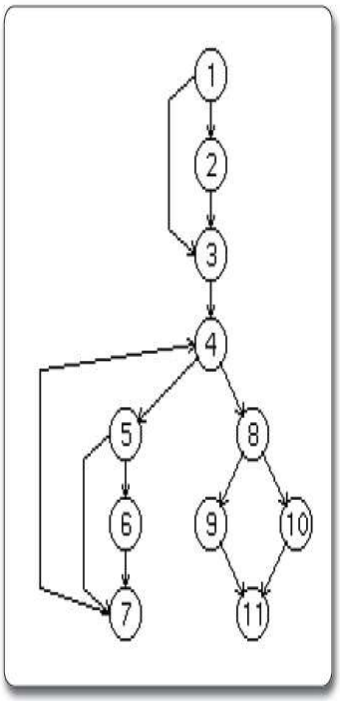


Figura 3-5 – GFC do Programa *Identifier.c* (Fonte: NETO (2007))

Entre os critérios de teste baseado no GFC, citam-se [MYERS, 1979; RAPPS; WEYUKER, 1982]:

- **Todos-Nós**: este critério garante que as instruções (comandos) de um dado módulo tenham sido executadas ao menos uma vez por algum caso de teste;
- **Todos-Arcos**: este critério requer que os arcos (ou arestas) de um GFC sejam exercitados ao menos uma vez por algum caso de teste;
- **Todos-Caminhos**: este critério requer que os caminhos possíveis do GFC sejam executados ao menos uma vez por algum caso de teste. Esse critério é o mais exigente do teste de caixa-branca e o número de casos de teste gerados para ele, mesmo em um software simples, pode ser demasiadamente grande (possivelmente infinito) [MYERS *et al*., 2004].

3.6.2. Técnica de Teste Funcional

Técnica de teste em que o software a ser testado é abordado como sendo uma caixa-preta, ou seja, não se considera o seu comportamento interno (Figura 3-6). O objetivo do teste de caixa-preta é mostrar que as funções do software estão operacionais, as entradas são adequadamente aceitas, as saídas são corretamente produzidas e a integridade da informação externa é mantida [SOUZA, 2003]. A abordagem funcional visa complementar a abordagem que as técnicas de teste estrutural apresentam.

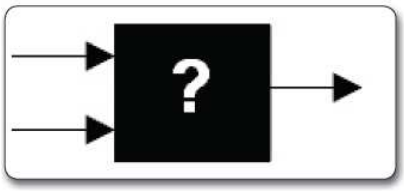


Figura 3-6 – Técnica de Teste Funcional (Fonte: NETO (2007))

A técnica de teste funcional não considera a estrutura interna do software, sendo aplicada quando ele se encontra em fase final ou completamente construído [SILVEIRA, 2007]. De acordo com Pressman (2006), os defeitos mais evidenciados neste tipo de teste são: i) defeitos de interface; ii) funções incorretas ou ausentes; iii) defeitos nas estruturas de dados ou no acesso a bancos de dados externos; iv) defeitos de desempenho; e v) defeitos de iniciação e término. Alguns critérios da técnica de teste funcional são [Pressman, 2006]:

- **Particionamento em Classes de Equivalência**: permite encontrar classes de erro, com base na divisão do domínio de entradas em classes de dados, contribuindo para a redução do número de casos de teste que precisam ser desenvolvidos. A Tabela 3-2 apresenta as classes de equivalência definidas para o programa *Identifier.c* (Figura

3-4). A partir desta tabela, é possível especificar quais serão os casos de teste necessários. Para ser válido, um identificador deve atender às condições (1), (3) e (5); logo, é necessário um caso de teste válido que cubra essas condições;

Tabela 3-2 – Classes de Equivalência do Programa *Identifier.c* (Fonte: NETO (2007))

Condições de Entrada	Classes	Classes
Tamanho t do identificador	$1 \leq t \leq 6$	$t > 6$
Primeiro caractere *c* é uma letra	Sim	Não
Só contém caracteres válidos	Sim	Não

- **Análise de Valor Limite**: permite avaliar se o código proposto é capaz de manipular situações de execução, ou seja, se o código trata corretamente as informações de entrada nas fronteiras das classes de equivalência definidas. As restrições identificadas para o software deverão ser exercitadas durante a realização dos testes para analisar a sua reação;
- **Grafo Causa-Efeito**: permite a avaliação de conjuntos complexos de condições de entrada (causa) e ações (efeito), de forma a definir um grafo causa-efeito para, em seguida, convertê-lo em uma tabela de decisão de onde serão extraídos os casos de teste. A Figura 3-7 e a Tabela 3-3 apresentam o exemplo de um grafo Causa-Efeito e a tabela de decisão gerada a partir deste grafo, respectivamente. O exemplo usado para criação do grafo causa-efeito e da tabela de decisão é de um software de comércio eletrônico apresentado em Neto (2007).

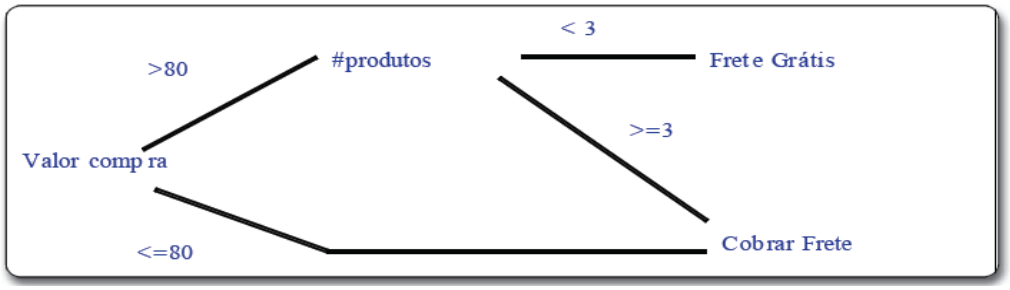


Figura 3-7 – Grafo Causa-Efeito (Fonte: NETO (2007))

Tabela 3-3 – Tabela de Decisão Derivada do Grafo de Causa-Efeito (Fonte: NETO (2007))

Causa	Valor da Compra	>60	>60	≤60
	# Produtos	<3	≥3	-
Efeito	Cobrar Frete	-	V	V
	Frete Grátis	V	-	-

3.6.3. Técnica de Teste Baseado em Estado

Esta técnica é utilizada em software que possui estados significativos durante a sua execução. O objetivo deste teste é validar o comportamento do software sendo testado [SOUZA, 2003].

Como exemplo de técnica de teste baseado em estados, tem-se a técnica denominada *W* [CHOW, 1978], a qual permite a geração de seqüências de testes baseados em máquinas de estado. Para modelar o comportamento do software e definir as seqüências de mensagens permitidas são utilizadas máquinas de estados finitos [BINDER, 2000].

3.7. Teste de Software Orientado a Objetos

O paradigma OO surgiu trazendo um novo enfoque comparado aos métodos tradicionais de desenvolvimento de software [SILVEIRA, 2007] e introduziu novos conceitos e novas abstrações, tais como classes, métodos, mensagens, herança, polimorfismo e encapsulamento. Para o paradigma OO, o mundo real é constituído de objetos autônomos, concorrentes e com interações entre si, no qual cada um desses objetos possui seus próprios estados e comportamentos, semelhantes aos seus correspondentes no mundo real.

Alguns benefícios do uso deste paradigma podem ser citados, tais como [SILVEIRA, 2007]: i) desenvolvimento mais rápido; ii) melhor qualidade; iii) manutenção facilitada; iv) estruturas de informação com melhor definição; v) bibliotecas de classes disponíveis; e vi) reutilização.

A atividade de teste para a abordagem OO geralmente é realizada em três fases distintas [SILVEIRA, 2007]: i) teste de unidade; ii) teste de integração; e iii) teste de sistema. É possível constatar na literatura pequenas variações quanto à divisão das fases de teste no paradigma OO. Colanzi (1999), por exemplo, inclui a fase de teste de classe. Segundo a visão de Colanzi (1999), as fases de teste são organizadas da seguinte forma [AMBLER, 1998; BINDER, 2000, JACOBSON *et al.*, 1992; PRESSMAN, 2006]:

- **Teste de Unidade**: fase onde se testam os métodos individualmente. Para a realização de testes nesses métodos ou unidades do software, entidades pseudocontroladoras (*drivers*) e pseudocontroladas (*stubs*) devem ser implementadas para, respectivamente,

coordenar e ativar a unidade a ser testada e substituir as unidades chamadas pela unidade em teste;

- **Teste de Classe**: fase onde se testa a interação entre os métodos de uma mesma classe. O objetivo deste teste é verificar o comportamento das classes diante dos estímulos durante a execução do software;
- **Teste de Integração**: fase onde se testa a integração entre classes. Este teste visa descobrir erros associados às interfaces dos módulos, durante a integração do software;
- **Teste de Sistema**: fase onde se testa a funcionalidade do software. O objetivo deste teste é identificar defeitos em funções e características de desempenho que não estejam em conformidade com a especificação.

É importante salientar que a menor estrutura a ser usada na fase de testes corresponde à estrutura atômica gerada pelo paradigma de Engenharia de Software escolhido [SILVEIRA, 2007]. Sendo assim, a menor estrutura pode ser uma função (paradigma estruturado), uma classe (paradigma OO) ou um aspecto (paradigma OA). Entretanto, alguns autores adotam nível menor de atomicidade a ser usado no teste. Em OA, por exemplo, alguns trabalhos consideram como a menor estrutura a ser testada um método, um adendo ou uma declaração intertipo (*inter-type declaration*) presente em um aspecto [ZHAO, 2003; LEMOS, 2005].

3.8. Teste de Software Orientado a Aspectos

Existem algumas referências na literatura [ZHAO; RINARD, 2003; ZHAO, 2003; ZHAO 2002] que discorrem sobre o fato de que a adoção do DSOA eventualmente propicie software com maior qualidade, porém a OA não provê corretude por si só [SILVEIRA, 2007].

Conforme descrito por Zhao (2003), ainda que a programação OA possa conduzir a uma melhor arquitetura e uma linguagem OA possa forçar um estilo de codificação mais disciplinado, ambos não possuem imunidade contra erros de programadores nem contra falta de entendimento de especificações. Sendo assim, a atividade de teste continua sendo essencial para o DSOA.

Diversas pesquisas recentes sobre teste de aplicações desenvolvidas sobre o paradigma OA representam importantes trabalhos nesta área. A seguir, é apresentada uma descrição sucinta das principais publicações na área de teste de software OA:

- **Modelo de Defeitos**: Um modelo de defeitos reflete as características estruturais e comportamentais do software. Esta técnica utiliza informações sobre os defeitos mais freqüentes presentes em um software para derivar casos de teste. Estas informações podem variar em função da linguagem, técnica ou método de desenvolvimento do software. Alexandre *et al.* (2004) propuseram um modelo de defeitos para programas OA, contemplando seis classes de defeitos:
    - *restrição incorreta em padrões de conjuntos de junção*: determina quais pontos de junção serão selecionados. Se a restrição for rígida, alguns pontos de junção requeridos podem ser não selecionados. Por outro lado, caso a restrição seja fraca (pouco restritiva), pontos de junção não requeridos podem ser selecionados, os quais deveriam ser ignorados. Em ambos os casos, comportamentos inesperados/indesejados podem surgir;
    - *precedência incorreta de aspectos*: a ordem em que os adendos de múltiplos aspectos são combinados na aplicação alvo pode afetar o comportamento do software, especialmente quando esses aspectos afetam conjuntos de junção coincidentes;
    - *defeitos para alcançar pós-condições de métodos*: espera-se que as pós-condições dos métodos sejam satisfeitas, independentemente de aspectos serem combinados ou não na aplicação alvo. Desta forma, definir adendos que não causem quebra de contrato de especificações em contexto onde são inseridos constitui-se um desafio para os desenvolvedores de aplicações OA;
    - *defeitos para preservar invariantes de classe*: analogamente aos defeitos referentes às pós-condições, o processo de combinação de aspectos não pode violar as invariantes de classe[6] da aplicação alvo;
    - *foco incorreto no fluxo de controle*: métodos que possuem chamadas recursivas podem ter seus pontos de junção capturados corretamente quando

[6] Invariante de classe é uma asserção a respeito de uma classe. A invariante é verdadeira para as instâncias da classe a qualquer momento que o objeto estiver disponível para ter uma operação executada nele. A invariante pode se tornar falsa durante a execução de um método, mas deveria retornar a ser verdadeira no momento em que outro objeto possa fazer qualquer coisa para o receptor [OLIVEIRA *et al*., 2001].

são chamados. Entretanto, o adendo correspondente pode não ser corretamente executado em decorrência de possíveis restrições no conjunto de junção;

- *mudanças incorretas em dependências de controle*: Adendos do tipo *around* podem alterar significativamente o comportamento semântico de um método e o controle original de dependências.

- **Teste De Unidade**: Zhao (2003) propõe uma abordagem de teste de unidade baseada no fluxo de dados para testar software OA codificado em *AspectJ*. Esta abordagem testa dois tipos de unidades de software OA: i) os aspectos, como unidades modulares que implementam interesses transversais; e ii) as classes, cujos comportamentos podem ser afetados por um ou mais aspectos. Para cada unidade, três níveis de testes distintos são aplicados: i) intramódulo; ii) intermódulo; e iii) intra-aspectos ou intraclasses. Para módulos individuais, como partes de um adendo, partes de uma declaração intertipos (*intertype declaration*) ou um método, testes intramódulo são realizados. Para módulos públicos, junto com outros módulos por eles chamados, testes intermódulos são conduzidos. Para módulos que podem ser acessados externamente por um aspecto ou classe, testes intra-aspectos ou interclasses são executados, respectivamente. São utilizados GFCs para determinar quais interações entre aspectos e classes devem ser testadas;
- **Verificação de Modelos**: Segundo Clarke *et al*. (2000), a verificação de modelos (*model checking*) é um método formal que verifica se uma estrutura (software) satisfaz ou não uma propriedade. Denaro; Monga (2002) utilizam a verificação de modelos para checar propriedades relevantes dos aspectos, por exemplo, ausência de travamentos (*deadlocks*) para o interesse em concorrência. Os autores defendem a idéia de que, se o código de um interesse pode ser encapsulado dentro de unidades em separado (principal característica do paradigma OA), provavelmente, a verificação destas propriedades específicas podem ser baseadas somente nos módulos que implementam tal interesse;
- **Representação de Programas**: Zhao; Rinard (2003) propõem uma extensão do Grafo de Dependência do Sistema (*System Dependence Graph* – SDG), para representar programas OA, bem como o algoritmo para sua geração. O SDG captura estruturas adicionais presentes nos programas OA como pontos de junção, adendos, declarações intertipos, aspectos, aspectos, herança de aspectos e outras interações entre classes e aspectos, servindo como uma sólida estrutura para posteriores tipos de testes OA;

- **Grafo de Fluxo de Aspectos**: Xu *et al*. (2004) propõem uma abordagem que combina modelos de estados (classes e aspectos) e grafos de fluxo (métodos e adendos) resultando em um modelo híbrido de teste, uma combinação de modelos de testes baseados em implementações (estruturais). Nesta abordagem, o modelo híbrido de teste, baseado nos relacionamentos entre classes e aspectos e um algoritmo para a construção do modelo de teste denominado *Aspect Scope State Model* (ASSM) são propostos. A partir do ASSM, um grafo de fluxo de aspectos (*Aspect Flow Graph*) é construído, provendo a derivação de conjuntos de casos de teste e garantindo que os caminhos entre as classes e os aspectos serão alcançados.

Outros trabalhos sobre teste de aplicações desenvolvidas sobre o paradigma OA são apresentados em Silveira (2007). Segundo este autor, a maioria dos trabalhos desenvolvidos refere-se à técnica estrutural, sendo poucas as abordagens referentes à técnica funcional. Ademais, nos trabalhos analisados anteriormente, não está clara a aplicação de testes na fase de sistema e de teste de regressão. Apesar de existirem trabalhos que apliquem testes de integração nas suas respectivas descrições, constam somente os relacionamentos existentes entre classes e aspectos.

3.9. Considerações Finais

Neste capítulo, foi apresentada uma fundamentação sobre teste de software, suas principais fases e técnicas. Foram relatados os termos qualidade de software e a norma ISO/IEC 9126, a qual trata de características de qualidade de software. Isso é importante, uma vez que o teste de software está intimamente ligado à garantia da qualidade de software.

Posteriormente, foi apresentado o teste de software OO e o teste de software OA, realizando uma investigação sobre os principais trabalhos relacionados com estas áreas de pesquisa.

4. ESTADO DA ARTE

4.1. Considerações Iniciais

Percebe-se que há algumas referências na literatura sobre a testabilidade de software. Este capítulo apresenta, sucintamente, alguns trabalhos relacionados investigados pelo autor.

4.2. Trabalhos Relacionados

Souza (2003) realizou um trabalho relativo a testabilidade de software OO, usando a *Unified Modeling Language* (UML), como representação dos modelos dos artefatos de software, e o Processo Unificado, como processo de desenvolvimento. Seu objetivo foi fornecer subsídios para elaborar os documentos de teste, mediante a identificação das características que podem ser inseridas no Modelo de Análise, para aumentar a testabilidade do software OO. As características de testabilidade do Modelo de Análise foram definidas em conseqüência do estudo do conteúdo dos documentos de teste e das informações que podem ser inseridas durante a modelagem. Foram propostos critérios, para a garantir que as características de testabilidade estejam adequadamente representadas nos modelos, e procedimentos de verificação dos modelos, visando a corretude e completeza.

Freedman (1991) definiu o conceito de testabilidade de domínio de software mediante a aplicação dos conceitos de observabilidade e controlabilidade de software. Observabilidade significa o quão fácil é determinar os fatores que afetam as saídas de uma determinada função do software, por exemplo. Controlabilidade significa o quão fácil é produzir uma saída específica a partir de uma determinada entrada. Um domínio é testável (domínio-testável) quando ele é observável e controlável, ou seja, quando ele não apresenta incoerências quanto às suas entradas e saídas. Freedman (1991) define também métricas que podem ser utilizadas para avaliar o nível de esforço necessário para modificar um programa domínio-testável, durante o processo de manutenção.

É importante salientar que ambos autores tratam testabilidade de software em software OO. Desta forma, o presente trabalho apresenta abordagem complementar, trazendo as preocupações relacionadas aos testes para o contexto de software OA.

4.3. Considerações Finais

Neste capítulo, foi descrita a investigação realizada sobre os principais trabalhos relacionados com esta pesquisa. Como é observado, há poucos trabalhos na literatura relacionados a testabilidade de software OA, sobretudo relacionados à sua incorporação ao longo do processo de desenvolvimento de software. Isto evidencia ainda mais a necessidade desta pesquisa para a comunidade científica.

5. CRITÉRIOS DE TESTABILIDADE

5.1. Considerações Iniciais

Este capítulo define os critérios de testabilidade aplicáveis na construção e/ou avaliação/adaptação do Modelo de Projeto de produtos de software orientados a aspectos. A seção 5.2 apresenta os critérios de testabilidade propostos. A seção 5.3 apresenta um estudo de caso, o software Gestão de Domínio Bancário (GDB), para ilustrar a aplicabilidade dos critérios. O conceito de estudo de caso utilizado neste trabalho é definido por Jacobson (1992), consistindo em apresentar um exemplo da aplicação dos recursos propostos; neste caso, os critérios de testabilidade.

5.2. Critérios de Testabilidade

Estendendo o trabalho de Souza (2003), com o diferencial de tratar a testabilidade no DSOA, esta seção apresenta um conjunto de critérios de testabilidade para o Modelo de Projeto de software OA do tipo "atende" ou "não-atende", que visam guiar a avaliação de seus artefatos. Esses critérios, denominados Critérios de Testabilidade para o Modelo de Projeto (*Testability Criteria for Design Model* – TC_DM), foram elaborados baseando-se nas características de testabilidade [ISO Std. 9126, 1991; ISO Std. 9126, 2001; ABNT NBR13596, 1996].

Os TC_DMs seguem a seguinte estrutura:

- **critério**: identifica o TC_DM com número e descrição;
- **justificativa**: justifica o uso do TC_DM.

TC_DM 1 – O relacionamento de precedência entre aspectos permite o rápido reconhecimento da ordem de execução de seus adendos.

Justificativa: A ordem em que os adendos de múltiplos aspectos são combinados na aplicação alvo pode afetar o comportamento do software, especialmente quando esses aspectos afetam conjuntos de junção coincidentes. Desta forma, a representação dos relacionamentos de precedência refletirá em implementações mais fáceis de serem testadas, uma vez que a ordem de execução dos adendos é definida a priori.

TC_DM 2 – Há conjuntos de pontos de junção para serem desconsiderados em um relacionamento aspecto-classe fraco.

Justificativa: Em um relacionamento entrecortante, caso a restrição seja fraca (pouco restritiva), pontos de junção não requeridos podem ser selecionados, os quais deveriam ser ignorados. Desta forma, uma lista contendo os pontos de junção que deverão ser descartados neste relacionamento facilitará a elaboração dos casos de teste.

TC_DM 3 – Há conjuntos de pontos de junção a serem considerados em um relacionamento aspecto-classe rígido.

Justificativa: Em um relacionamento entrecortante, se a restrição for rígida, alguns pontos de junção requeridos podem não ser selecionados. Logo, uma lista contendo os pontos de junção que deverão ser considerados neste relacionamento facilitará a elaboração dos casos de teste.

TC_DM 4 – O modelo de classes original se mantém inalterado.

Justificativa: Características transversais comportamentais que oferecem suporte à adição (*additions*) afetam elementos base (classes) adicionando-lhes novos atributos e/ou operações. Assim sendo, visando manter o modelo de classes original inalterado, uma nova representação do elemento base afetado deve ser criada. Isto permite a rápida identificação de classes que apresentam adições realizadas por aspectos, facilitando a elaboração dos casos de teste.

TC_DM 5 – Os relacionamentos de generalização/especialização do software representam hierarquias de herança forte.

Justificativa: Os relacionamentos de generalização/especialização definidos em um software representam uma importante fonte de informações para as atividades relacionadas com o teste, visto que, muitas vezes, é possível reduzir o número de casos de teste a serem definidos para o software. Para que isto seja possível, é necessário que haja uma hierarquia de herança forte, onde os subaspectos definam uma pequena quantidade de novas características ou sobreponham (*overriding*) poucas características existentes nos superaspectos.

TC_DM 6 – O tipo de relacionamento *Requirement* permite o rápido reconhecimento da dependência entre os aspectos de um mesmo software.

Justificativa: Um relacionamento do tipo *Requirement* oferece uma conexão explícita entre um aspecto (o cliente) que requer a presença de outro aspecto (o fornecedor) para sua implementação ou funcionamento correto. Desta forma, a representação deste relacionamento refletirá em implementações mais fáceis de serem testadas, uma vez que a dependência entre os aspectos do software é definida a priori.

5.3. Aplicabilidade dos Critérios de Testabilidade

A fim de verificar a aplicabilidade dos TC_DMs propostos na seção anterior, eles foram utilizados para guiar a construção do Modelo de Projeto do software GDB, porém os TC_DMs podem ser usados para avaliar um Modelo de Projeto pré-existente.

Dentre as tecnologias para DSOA, *aSideML* é utilizada para representar o Modelo de Projeto. Embora não haja consenso sobre qual a melhor linguagem de modelagem OA deve ser utilizada, a abordagem *aSideML* se apresentou mais adequada, pois ela propõe um modelo de aspectos de alto nível, independente de linguagem e contempla principais conceitos, propriedades e arquitetura introduzidos pelo projeto OA. Desta forma, *aSideML* oferece recursos (diagramas e elementos de modelagem) suficientes para o levantamento de informações para os testes.

5.3.1. Descrição do Software GDB

O GDB gerencia operações requeridas por uma agência bancária. As operações (requisitos funcionais) são: i) efetuar *login* e *logoff*; ii) cadastrar, consultar, editar e remover (CRUD – *Create, Read, Update, Delete*) clientes; iii) alterar senha; iv) ver saldo; v) realizar transferência; vi) efetuar depósito e vii) efetuar saque. Os usuários do GDB são classificados em dois tipos: i) o administrador (tem permissões para realizar as operações descritas anteriormente); e ii) o cliente (pode efetuar *login* e *logoff*, ver saldo, alterar senha e efetuar transferência de sua conta para outras, efetuar depósito e saque). A Figura 5-1 apresenta o Diagrama de Caso de Uso do GDB para ilustrar a interação de seus usuários.

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

Figura 5-1 – Diagrama de Caso de Uso do Software GDB

5.3.2. Uso dos Critérios de Testabilidade

Nesta subseção, é exemplificada a aplicação dos critérios de testabilidade no software GDB. Em alguns dos exemplos, o uso de mais de um critério podem ser verificado, realçando apenas alguns deles visando a apresentar maior número de exemplos. A intenção é avaliar a aplicabilidade dos critérios definidos.

O software GDB possui os seguintes aspectos: `APersistencia` (persistência), `ATransacoes` e seu sub-aspecto `ATransacoesBancarias` (controle de transação), `ALogging` (histórico de acessos), `AIUsuario`, `ATracing` (controle de fluxo de execução), `APreEPosCondicoes` (suporte à verificação de pré/pós-condições) e `AAutenticacao` (autenticação de usuário). A seguir, os artefatos do Modelo de Projeto do software GDB são analisados, exemplificando a aplicação dos três TC_DMs apresentados.

O relacionamento de precedência entre aspectos deve ser respeitado. Como a ordem em que os adendos de múltiplos aspectos são combinados no software GDB pode afetar o seu comportamento. Em especial, quando esses aspectos interceptam conjuntos de junção coincidentes, a representação dos relacionamentos de precedência entre aspectos é uma boa prática. O TC_DM 1 é aplicado no Diagrama de Classes Estendido dos aspectos `ALogging` e `ATracing` (Figura 5-2).

A Figura 5-2 apresenta dois relacionamentos de entrecorte que associam os aspectos `ALogging` e `ATracing` à classe `Banco`. Estes dois aspectos interceptam o método `saldo` da classe `Banco` e seu comportamento é alterado nos pontos de combinação definidos pelo *pointcut* `verSaldo` em ambos aspectos. O relacionamento *precedence* é apresentado como uma seta tracejada entre dois elementos do modelo e rotulada com o estereótipo <<*precede*>>. Neste caso, um relacionamento *precedence* é representado do aspecto `ATracing` ao aspecto `Alogging`, significando que o comportamento de *tracing* ocorre antes do comportamento de *logging*.

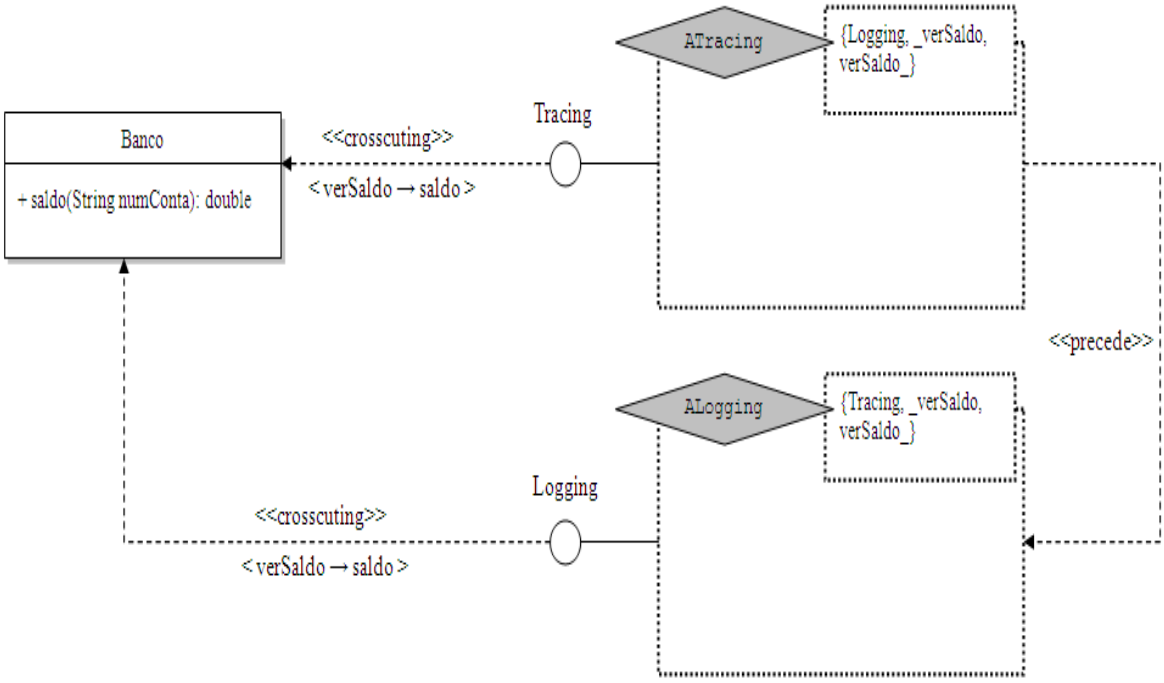


Figura 5-2 – Diagrama de Classes Estendido de `ALogging` e `ATracing`

O TC_DM 1 pode ser aplicado no diagrama da Figura 5-2, pois existe uma referência explícita ao relacionamento de precedência entre os aspectos envolvidos. A representação dos relacionamentos de precedência refletirá em implementações mais fáceis de serem testadas, uma vez que a ordem de execução dos adendos é definida a priori, contribuindo para aumentar a testabilidade de software OA. O TC_DM 2 e o TC_DM 3 contribuem para tornar a tarefa de elaboração de casos de teste menos árdua, pois exigem que os relacionamentos aspecto-classe a serem executados e/ou descartados sejam explicitados nos artefatos do Modelo de Projeto.

A Figura 5-3 apresenta o Diagrama de Aspectos de `AAutenticacao`. Pode-se observar que há dois *pointcuts*, `_classes` e `_verSaldo`, responsáveis por redefinir

características comportamentais na classe base. O *pointcut* `_classes` representa um relacionamento aspecto-classe fraco e, por isso, *join points* não requeridos podem ser afetados por ele. Por outro lado, o *pointcut* `_verSaldo` representa um relacionamento aspecto-classe forte, assim, *join points* requeridos podem não ser afetados por ele. Para verificar o TC_DM 2, um estereótipo denominado <<*ignore*>> foi incorporado ao Diagrama de Aspectos de `AAutenticacao` juntamente com uma listagem dos *join points* que devem ser desconsiderados para o relacionamento aspecto-classe em questão (*pointcut* `_classes`).

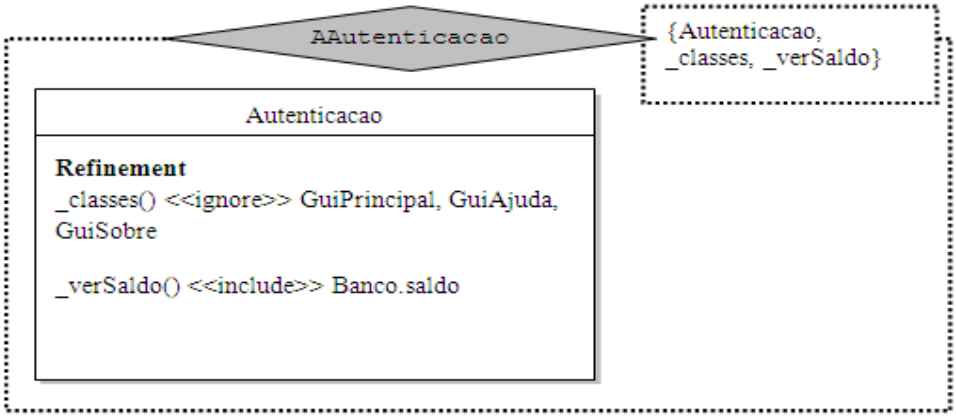


Figura 5-3 – Diagrama de Aspectos de `AAutenticacao`

De modo análogo, para verificar o TC_DM 3, um estereótipo denominado <<*include*>> foi incorporado juntamente com uma listagem dos *join points* que devem ser selecionados para o relacionamento aspecto-classe em questão (*pointcut* `_verSaldo`). A Figura 5-4 apresenta o Diagrama de Aspectos de `ATransacoes` e `ATransacoesBancarias`. Pode-se observar que há um relacionamento de herança entre estes dois aspectos. O aspecto `ATransacoesBancarias` herda as características e provê implementação aos métodos abstratos definidos no aspecto `ATransacoes`.

O TC_DM 5 pode ser verificado no diagrama da Figura 5-4, pois existe uma hierarquia de herança forte entre os aspectos identificados, uma vez que o subaspecto `ATransacoesBancarias` apenas implementa os conjuntos de junção abstratos definidos em `ATransacoes`, não inserindo novas características nem sobrepondo as características locais existentes em `ATransacoes`. Desta forma, casos de teste elaborados para o superaspecto poderiam ser reutilizados no contexto dos subaspectos.

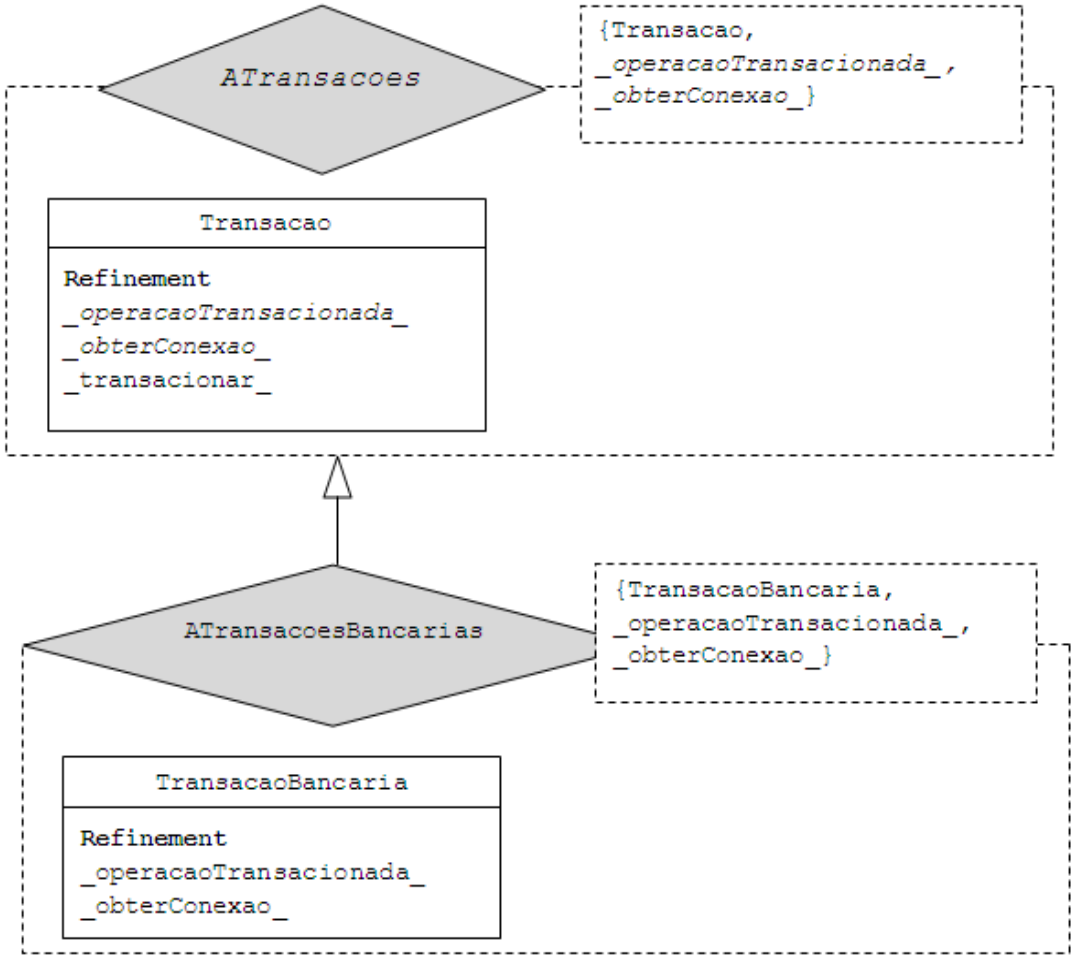


Figura 5-4 – Diagrama de Aspectos de `ATransacoes` e `ATransacoesBancarias`

5.4. Considerações Finais

Neste capítulo, foram definidos alguns critérios de testabilidade para a construção do Modelo de Projeto de software orientado a aspectos. Além disso, a aplicabilidade destes critérios foi demonstrada por meio de um estudo de caso, o software GDB.

Ao todo, foram aplicados 4 critérios, cobrindo 66,67% dos TC_DMs. Apenas o TC_DM 4 e o TC_DM 6 não puderam ser aplicados, por causa do escopo do software GDB. Entretanto, isso está previsto para os próximos exemplos de aplicação.

Conforme observado, os critérios de testabilidade definidos têm seu foco nas estratégias para a construção de software orientado a aspectos e baseiam-se nas características de testabilidade estabelecidas pela norma ISO/IEC 9126. Desta forma, incorporam-se à tecnologia de OA fortes características de teste de software, visando solidificar seus estudos e inserir seus conceitos no mercado.

6. CONSIDERAÇÕES FINAIS

A testabilidade é um importante atributo para manutenção e para garantia de qualidade do software. Embora as novas metodologias busquem reduzir a complexidade da sua organização, a atividade de teste continua sendo essencial para o seu desenvolvimento. Além disso, dada complexidade elevada do software, considera-se esta atividade responsável por metade do esforço despendido no seu desenvolvimento. Por isso, recomenda-se construir software com informações que possam minimizar os esforços necessários para testá-lo. Desta forma, foi apresentado um conjunto de critérios de testabilidade para guiar a construção do Modelo de Projeto de software OA (TC_DMs).

Neste capítulo, são apresentadas algumas observações resultantes deste trabalho. A seção 6.1 apresenta uma breve conclusão do trabalho. A seção 6.2 cita algumas contribuições que a pesquisa gerou em seu desenvolvimento. A seção 6.3 faz uma síntese dos trabalhos a serem desenvolvidos com base nesta pesquisa, objetivando a sua continuidade.

6.1. Conclusões

Os critérios de testabilidade para o Modelo de Projeto de software OA foram estabelecidos a partir das informações necessárias para minimizar o esforço das atividades de teste. Além disso, este trabalho possibilita que a preocupação pelo teste de software esteja presente desde as etapas iniciais do desenvolvimento, de forma que as atividades relacionadas com o teste possam ser amadurecidas de forma adequada.

Ao 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 software, pois algumas exigências estão relacionadas com determinadas características de uma aplicação particular. Além disso, o uso dos critérios propicia o investimento cada vez maior no desenvolvimento de software, visto que o esforço necessário para efetuar testes em um software tende a diminuir, pois foi implementado considerando a testabilidade.

Os critérios de testabilidade definidos neste trabalho podem ser utilizados em qualquer modelo de processo de desenvolvimento de software AO que utilize aSideML como linguagem de modelagem para representar os artefatos de software do Modelo de Projeto. A atividade da verificação dos critérios deve ser integrada nas fases do processo de desenvolvimento selecionado. A aplicação dos critérios em um estudo fez com que o conjunto de critérios definido fosse revisado e aprimorado, contribuindo para melhor entendimento do trabalho.

6.2. Contribuições

Conforme apresentado neste trabalho, observam-se poucos trabalhos na literatura relacionados a testabilidade de software OA, sobretudo relacionados à sua incorporação ao longo do processo de desenvolvimento de software.

Desta forma, o trabalho deixa como contribuição um conjunto de critérios de construção e avaliação de testabilidade elaborados tendo a norma ISO/IEC 9126 como guia e a partir de uma pesquisa ampla em torno dos temas teste de software e desenvolvimento de software OA. Assim, buscou-se a identificação das características relevantes para a fase de teste de software que devem ser consideradas no Modelo de Projeto. Tais características levaram aos critérios de testabilidade, permitem mostrar informações importantes ao entendimento do software e facilitam a elaboração de planos de teste de software.

Além disso, um estudo de caso foi elaborado visando verificar a usabilidade dos critérios propostos. Com ele, deixa-se como contribuição um exemplo prático da aplicação dos critérios, servindo como guia de referência para incorporação dos critérios em outros domínios de aplicações.

Foi publicado um artigo completo no *II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2008)* contribuindo para a comunidade acadêmica no que tange às pesquisas relacionadas ao DSOA.

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.

6.3. Trabalhos Futuros

Visando a completude dos critérios, experimentos e estudos de caso, referentes a avaliação/adaptação de Modelos de Projeto OA existentes, devem ser realizados, abrangendo outros domínios de aplicação, bem como a avaliação da efetividade dos TC_DMs estabelecidos. Além disso, uma análise do custo/benefício para a aplicação dos critérios propostos deve ser realizada a fim de verificar a contribuição destes critérios para a redução dos gastos inerentes às atividades relacionadas com o teste de software.

Realizar novos estudos de caso, preferencialmente em aplicações reais, seria interessante por incorporar características de testabilidade no processo de desenvolvimento de software na indústria, além da OA, melhorando a qualidade do produto e do processo com suas melhorias sobre a OA.

Outros desdobramentos deste trabalho podem ser: i) elaborar diretrizes de testabilidade para o Modelo de Projeto; ii) construir critérios e diretrizes de testabilidade para os Modelos de Análise e de Implementação; iii) manter uma tabela de rastreabilidade de critérios entre os três modelos; e iv) desenvolver um ambiente para apoiar a abordagem, visando sua integração ao processo de desenvolvimento.

Além disso, outros temas de estudo estão relacionados a seguir:

- analisar os critérios de testabilidade propostos, considerando sua executabilidade, complexidade, relação de inclusão e nível de cobertura;
- realizar o mesmo enfoque usado neste trabalho, ou seja, a qualidade de software, mas considerando individualmente as outras subcaracterísticas de manutenibilidade da norma ISO/IEC 9126: analisabilidade, estabilidade e modificabilidade;
- agregar os trabalhos realizados, para cada uma das subcaracterísticas de qualidade presentes no item anterior, em um único trabalho, avaliando as sobreposições e os conflitos;
- considerar as características (funcionalidade, confiabilidade, usabilidade, eficiência e portabilidade) e suas subcaracterísticas de qualidade presentes na norma ISO/IEC 9126;
- analisar as particularidades das diferentes linguagens de programação orientadas a aspectos, visando identificar quais informações podem ser inseridas nos artefatos do Modelo de Projeto para facilitar a implementação de software e posterior teste;

- considerar diferentes tipos de sistemas, tais como embutidos, cliente/servidor ou distribuídos, visando identificar particularidades importantes destes tipos de sistemas que devem ser consideradas durante a modelagem do software, contribuindo para a realização de testes adequados.

REFERÊNCIAS BIBLIOGRÁFICAS

ABNT NBR13596. **Tecnologia de Informação – Avaliação de Produto de *Software* – Características de Qualidade e Diretrizes para o seu Uso**. ABNT. Abril de 1996 (versão brasileira da Norma ISO/IEC 9126, 1991).

AKSIT, M.; BERGMANS, L.; VURAL, S. **An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach**. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 6., 1992, Utrecht. Proceedings... London: Springer-Verlag, 1992. p. 372-395.

ALEXANDER, R. T.; BIEMAN, J. M.; ANDREWS, A. A. **Towards the Systematic Testing of Aspect-Oriented Programs**. Fort Collins: Department of Computer Science, COlorado State University, 2004. (Technical Report CS-4-105).

AMBLER, I. S.; CARVER, D. L. A Testing Assistant for Object-Oriented Programs. In: IEEE AEROSPACE CONFERENCE, New York, 1998. **Proceedings**. v. 4. p. 149-158.

AOSDbr – COMUNIDADE BRASILEIRA DE DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS. **Terminologia em português para orientação a aspectos**. Brasília, 2008. Disponível em: <http://twiki.dcc.ufba.br/bin/view/AOSDbr/TermosEmPortugues>. Acessado em: Mar. 2008.

AOSD-EUROPE RESEARCH CENTER. **AOSD-Europe Group Homepage**. London, 2006. Disponível em: <http://www.aosd-europe.net/>. Acessado em: Jun. 2008.

ARAÚJO, J.; MOREIRA A.; BRITO, I.; Rashid, A. **Aspect-Oriented Requirements with UML**. Workshop on Aspect-Oriented Modelling with UML (held with UML 2002), 2002.

ASPECTJ. **AspectJ Home Page**. Disponível em <http://www.aspectj.org>. Acessado em: Jun. 2008.

BERARD, E. V. Issues in Testing of Object-Oriented *Software*. In: ELECTRO/94, 1994. **Proceedings**. Internacional combined volumes, p. 211-219.

BINDER, R. V. Object-Oriented Testing: Myth and Reality. **Object Magazine**, v. 5, n. 2, p. 73-75, May 1995.

BINDER, R. V. **Testing Object-Oriented Systems: Models, Patterns and Tools**. Boston: Addison-Wesley, 2000. ISBN 0-201-80938-9.

CHATTERJEE, I. ***Testing Testability.*** StickyMinds.com. Disponível em: <http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=8077&commex=1>. Acessado em: Maio 2008.

CHAVEZ, C. V. F. G. **Um Enfoque Baseado em Modelos para o Design Orientado a Aspectos**. PUC – Rio, Departamento de Informática, 2004. 298 f.

CHOW, T. S. **Testing Software Design Modeled by Finite-State Machines**. IEEE Trans. *Software* Engineering, v. 4, n. 3, p. 178-187, 1978.

CLARKE, S.; BANIASSAD, E. **Aspect Oriented Analysis and Design – The Theme Approach**. Editora Addison-Wesley, Primeira Edição, 2005.

CLARKE, E. M.; GRUMBERG, O.; PELED, D. A. **Model Checking**. Cambridge: The MIT Publisher, 2000. ISBN 0-262-03270-8.

COLANZI, T. E. **Uma abordagem Integrada de Desenvolvimento e Teste de Software Baseada na UML**. São Carlos, 1999. 135p. Dissertação – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo.

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.

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

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

DENARO, G.; MONGA, M. An Experience on Verification of Aspect Properties. In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, 4, 2002, Vienna. **Proceedings**. New York: ACM Press, 2002. p. 186-189.

DIJKSTRA, E. W. **A Discipline of Programming**. Englewood Cliffs: Prentice-Hall, 1976. 217 p. ISBN 013215871X.

ELRAD, T.; FILMAN, R. E.; BADER, A. **Aspect-Oriented Programming: Introduction**. Commun. ACM, 2001a.

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, 2001b.

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

FILHO, W. P. P. **Engenharia de *Software*: Fundamentos, Métodos e Padrões**. LTC – Livros Técnicos e Científicos Editora, 2001. 581p.

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

FREEDMAN, R. *Testability of Software Components*. IEEE Transactions on *Software* Engineering, 17(6):553-564, June 1991.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns – Elements of Reusable Object-Oriented *Software***. Addison-Wesley, 1996.

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.

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

IEEE Std. 829 – The Institute of Eletrical and Electronics Engineers, **IEEE Standard for *Software* Test Documentation**. 1998.

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.; CHRISTERSON, M.; JONSSON, P., ÖVERGAARD, G. **Object-Oriented *Software* Engineering – An Use Case Driven Approach**. Addison-Wesley, 1992. 528p.

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

JUNG, C. F. **Metodologia para Pesquisa e Desenvolvimento: Aplicada a Novas Tecnologias, Produtos e Processos**. Axcel Books do Brasil, Rio de Janeiro, RJ, 2004.

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

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

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.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. **An Overview of AspectJ**. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 15., 2001, Budapest. Proceeding… New York: Springer-Verlag, 2001. p. 327-353.

KÖLLING, M.; ROSENBERG, J. Support for Object-Oriented Testing. In: TECHNOLOGY OF OBJECT-ORIENTED LAGUAGES – TOOLS 28. Los Alamitos, 1998. **Proceedings**. p. 204-215.

KRECHETOV, I.; TEKINERDOGAN, B.; ALARCON, M. P.; FUENTES, L. **Initial Version of Aspect-Oriented Architecture Design Approach**. Enschede: Univesity of Twente, 2006. (AOSD-EUROPE Deliverable D37, AOSD-Europe-UT-D37).

LADDAD, R. **AspectJ in Action: Pratical Aspect-Oriented Programming.** 2003. Greenwich Mannig Publication Co., ISBN 1930110936.

LEMOS, O. A. L. **Teste de Programas Orientados a Aspectos: Uma Abordagem Estrutural para AspectJ**. 2005. 103 f. Dissertação – Instituto de Ciências Matemáticas e de Computação, Univerisdade de São Paulo, São Carlos, 2005.

LI, Y.; WAHL, N. J. An Overview of Regression Testing. ***Software*** **Engineering Notes**, v.24, n. 1, p. 69-73, Jan. 1999.

LIEBERHERR, K.; ORLEANS, D.; OVLINGER, J. **Aspect-Oriented Programming with Adaptive Methods**. Commun. ACM, v. 44, n. 10, p. 39-41, 2001.

MALDONADO, J. C. **Critérios Potenciais de Usos: Uma Contribuição ao Teste Estrutural de Software**. 1991. 169 f. Tese (Doutorado em Engenharia Elétrica). Universidade Estadual de Campinas, Campinas, 1991.

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

MCGREGOR, J. D.; SYKES, D. A. **A Pratical Guide to Testing Object-Oriented *Software***. Addison Wesley. 2001. 393p.

MYERS, G. J. **The Art of Software Testing**. New York: John Wiley & Sons, 1979. 177 p. ISBN 0471043281.

MYERS, G. J., SANDLER, C., BADGETT, T., AND THOMAS, T. M. **The Art of *Software* Testing**. 2004. John Wiley & Sons, 2nd edition.

NETO, A. C. D. **Introdução a Teste de *Software***. In: Revista de Engenharia de *Software*. Edição Especial. DevMedia. Brasil, 2007.

OLIVEIRA, W. R; BARRERO, J. S; LACERDA, R. M. **Diagrama de Classes: "Os Elementos Básicos"**. Varginha. 2001.

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.

PERRY, W. E. **Effective Methods for *Software* Testing**. John Wiley & Sons. 2000. 812p.

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

RAMAKRISHNAN, S. Visualizing O-O Testing in Virtual Communities – Distributed Teachung and Learnig. In: TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS. Los Alamitos, 1999. **Proceedings**. p. 300-310.

RAPPS, S.; WEYUKER, E. J. **Selecting Software Test Data Using Data Flow Information**. IEEE Trans. *Software* Engineering, v. 11, n. 4, p. 367-375, 1985.

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

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

SANTOS, R. P. dos. **Critérios de Manutenibilidade para Construção e Avaliação de Produtos de *Software* Orientados a Aspectos**. Lavras – Minas Gerais, 2007. 92.

SEBESTA, R. W. **Concepts of Programming Languages 8/E**. Addison-Wesley, 2008. 752 p. ISBN-10: 0321493621, ISBN-13: 9780321493620.

SILVEIRA, F. F. **METEORA: Um Método de Testes Baseado em Estados para Software de Aplicação Orientado a Aspectos**. 2007. 262f. Tese – Instituto Tecnológico da Aeronáutica, São José dos Campos.

SOMMERVILLE, I. ***Software Engineering*.** 6. Ed. Boston: Addison-Wesley Longman, 2001. 693 p. ISBN 0-201-39815-X.

SOUZA, R. C. G. **Características de Testabilidade nos Diagramas UML (*Unified Modeling Language*): Apoio aos Testes de Sistemas de *Software* Orientados a Objetos.** 2003. Tese – Escola Politécnica da Universidade de São Paulo, São Paulo.

TANNOURI, P. A. **O que é Testabilidade**. Linha de Código. 2006. Disponível em: <http://www.linhadecodigo.com.br/ITC_Artigo.aspx?id=923>. Acessado em: Maio 2008.

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

TSAI, B. Y.; STOBART S.; PARRINGTON N.; THOMPSON B. Iterative Design and Testing within the Software Development Life Cycle. ***Software* Quality Journal**, v. 6, p. 295-309, July 1997.

XU, W.; XU, D.; GOEL, V.; NYGARD, K. Aspect Flow Graph for Testing Aspect-Oriented Programs. In: INTERNATION CONFERENCE ON SOFTWARE ENGINEERING AND APPLICATIONS – IASTED, 8, 2004, Cambridge. **Proceedings**. Calgary: ACTA Press, 2004.

ZHAO, J. Slicing Aspect-Oriented Software. In: INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION, 10, 2002, Paris. **Proceedings**. Paris: IEEE Computer Society, 2002. p. 251-260.

ZHAO, J. Data-Flow-Based Unit Testing of Aspect-Oriented Programs. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 27, 2003, **Proceedings**. Dallas IEEE Computer Society, 2003. p. 188-197.

ZHAO, J.; RINARD, M. **System Dependence Graph Construction for Aspect-Oriented Programs**. Cambridge: MIT, Laboratory for Computer Science, 2003. (Technical Report MIT-LCS-TR-891).