INSTITUTO DE PESQUISAS TECNOLÓGICAS DO ESTADO DE SÃO PAULO

MARCIO BORGES FERREIRA

Uso da Programação Orientada a Aspectos em testes de integração: método e *framework* de aplicação

São Paulo

2008

Ficha Catalográfica
Elaborada pelo Departamento de Acervo e Informação Tecnológica – DAIT
do Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT

F383u Ferreira, Marcio Borges

Uso da programação orientada a aspectos em testes de integração; método e framework de aplicação. / Marcio Borges Ferreira. São Paulo, 2008.

78p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Jorsé Eduardo Zindel Deboni

1. Teste de software 2. Programação orientada a aspectos 3. Integração 4. Interface 5. Tese I. Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Coordenadoria de Ensino Tecnológico II. Título

08-165 CDU 004.415.538(043)

MARCIO BORGES FERREIRA

Dissertação apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT, para obtenção do título de Mestre em Engenharia da Computação.
Área de concentração: Engenharia de *Software*

Orientador: Dr. José Eduardo Zindel Deboni

São Paulo

2008

Agradecimentos

Agradeço a Deus pelas bênçãos e por me mostrar o caminho em que devo seguir; aos meus pais que sempre me incentivaram e acreditaram em mim; a minha esposa por tanto apoio e compreensão; aos meus amigos e familiares pelo estímulo constante ao longo dessa jornada e aos mestres, que repartiram comigo seus conhecimentos, com dedicação e amizade, no cumprimento dos seus deveres.

Resumo

Testes de integração são utilizados para encontrar falhas que os testes unitários são incapazes de detectar. Porém, as várias técnicas existentes acabam intervindo de maneira intrusiva na aplicação, incluindo alterações passíveis de novas falhas. Devido à interoperabilidade de componentes em aplicativos desenvolvidos para Internet, testes de integração têm tido sua importância ressaltada nesse ambiente. Com o aumento da complexidade do ambiente de Internet e o desenvolvimento de novas técnicas de programação, novas abordagens se fazem necessárias para atender de forma flexível os desafios desse tipo de teste. A proposta deste trabalho baseia-se na utilização da Programação Orientada a Aspectos para criar um método não intrusivo de execução de teste de integração em aplicativos desenvolvidos para Internet. Esse método permite capturar as interações entre os componentes que compõem uma funcionalidade avaliando, através de um critério de teste definido pela proposta, se ocorreu uma execução com falhas ou sem falhas do teste de integração de componentes.

Palavras Chaves: Teste de *Software*, Teste de Integração, Programação Orientada a Aspectos, Teste de Componentes, Teste de Interface.

Abstract

Integration Testing are used to find defects that the unitary tests are incapable to detect. However the several existing techniques end up intervening in a intrusive way within the application, including changes that may generate new defects. Due to great interoperability of components in applications developed for Internet solutions, integration tests have had its salient importance for tests execution into this environment. As consequence of the increase of the complexity of the Internet environment and the development of new programming techniques, new approaches are necessary to take care of flexible form the challenges of this type of testing. The proposal of this work is based on the use of the Aspect Oriented Programming to create a non-intrusive method of execution of integration testing in applications developed for Internet. This method allows to capture the interactions between the components that comprise a feature, evaluating, by a test criteria defined by the proposal, if there was a testing execution with or without flaws in the component integration testing.

Key Words: Test Software, Integration Testing, Aspect Oriented Programming, Component Testing, Interface Testing.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Acrônimos

AOP	*Aspect Oriented Programming*
AOSD	*Aspect Oriented Software Development*
DOM	*Document Object Model*
XML	*Extensible Markup Language*

1 Introdução

Teste de *Software* é uma disciplina da Engenharia de *Software* que estuda e propõe técnicas para auxiliar a difícil atividade de avaliar se um sistema mantém seu comportamento esperado durante o processo de desenvolvimento de *software*. Essa área acompanha de perto a evolução das técnicas de desenvolvimento, sempre atuando no sentido de oferecer novos métodos mais otimizados que garantam a qualidade do *software* e reduzam o esforço gasto pelos analistas de testes.

Segundo Myers (2004), o teste de *software* consiste em "*um processo de executar um programa de computador com a intenção de localizar erros*". As técnicas propostas pela disciplina de teste de *software* fornecem ao analista de teste abordagens para realizar a validação das funcionalidades presentes no programa, frente aos requisitos de negócio definidos na especificação(The Testing Standards, 2008). Abordagens que resultam em instrumentos que irão determinar se um componente atende às condições impostas na especificação ou não.

A atividade de teste de *software* pode ser dividida em três fases, associadas às fases do desenvolvimento de um sistema: testes unitários, testes de integração e testes de sistema. Testes unitários testam individualmente componentes de *software*, na fase de construção. Testes de integração realizam os testes referentes às interfaces entre esses componentes, quando os componentes se integram depois de construídos, e, por fim, os Testes de Sistema que validam o sistema de *software* completo.

Testes de integração são realizados para avaliar a interdependência de componentes dentro de um ou mais sistemas, avaliando a interface entre o componente e o restante do sistema, e assegurando os resultados esperados quando esses componentes são associados em uma funcionalidade ou serviço. Esses testes devem ser realizados em situações reais, muito semelhantes ao ambiente produtivo, pois não utilizam classes auxiliares para validar condições específicas como no caso dos testes unitários.

O sistema pode interagir de maneira distinta com um componente de acordo com as várias interações já realizadas com esse componente, o que pode revelar falhas que não são detectadas em testes unitários. Em outras palavras, não basta que se realizem testes unitários individualmente entre dois componentes para garantir que eles estão corretos, é preciso que se

teste a integração efetiva entre eles.

Em aplicações desenvolvidas para *Internet*, testes de integração são essenciais devido ao grande número de componentes e de módulos presentes em sua arquitetura típica de camadas (NGUYEN et al., 2003), esse ambiente possui novos desafios com relação à atividade de teste, pois a adaptação de técnicas utilizadas em software tradicional não cobre totalmente todos os pontos de integração dessa nova arquitetura.

A evolução das técnicas de programação, apesar de ser um fator importante nos desafios de testes de integração, pode também servir como ferramenta essencial no desenvolvimento de novas técnicas e abordagens para realização desse tipo de teste, por meio dos benefícios oferecidos por essa evolução.

1.1 Motivação

A principal motivação deste trabalho é realizar os testes de integração em um sistema de *software* de modo a não afetar ou intervir na codificação e no funcionamento original do sistema. Evita-se inserir nas classes de negócios elementos de código exclusivamente para avaliar a funcionalidade, e que não fazem parte do escopo das regras de negócio do sistema.

No processo de verificação dos testes, existe a dificuldade de acesso às informações por meio do fluxo do sistema, onde seja possível avaliar as entradas e saídas dos vários componentes que compõem as funcionalidades. O que nos remete ao problema de como ter acesso às informações pertinentes à execução das regras de negócio, sem intervir diretamente nestas regras incluindo funcionalidades que dizem respeito apenas ao teste e não à funcionalidade do aplicativo.

Testes de *software* são aplicados, em geral, em ambientes semelhantes ao ambiente produtivo, na intenção de simular o ambiente real para localizar o maior número possível de falhas antes da liberação do aplicativo ao usuário final; esses ambientes são chamados de ambientes de homologação. Portanto ao selecionarmos uma metodologia para realizar a verificação do aplicativo, o ideal é que essa metodologia apresenta baixo acoplamento com o aplicativo em teste para simplificar sua remoção no momento de enviar o aplicativo ao ambiente produtivo, pois caso contrário, isto causará necessidade de intervenção dos analistas de sistemas para separar as regras de negócios dos testes, o que pode gerar novas falhas e inviabilizar a

homologação do sistema.

A falta de consenso para uma sintaxe padrão de construção dessas validações, no que diz respeito à definição da organização e seqüência dos testes, dificulta a criação e a evolução de ferramentas de testes de integração.

No caso de aplicativos desenvolvidos para *Internet*, onde existe uma grande integração de componentes, entidades internas, entidades externas e parceiros de negócio; o desafio é ainda maior com relação à fase de testes de integração. Nesse ambiente, a maioria de métodos existentes:*Capture/Playback, Testability Hooks e Logs* (DUSTIN,2003) não permite uma inspeção do aplicativo sem alteração do código original, limitando-se a executar chamadas funcionais e posterior validação de sua resposta, o que acrescenta muito pouco para a avaliação de um aplicativo e a relação entre seus componentes.

Na comunidade da engenharia de *software*, existem atualmente algumas pesquisas para estudar linguagens de desenvolvimento que tratem melhor a questão de separação de interesses (KICZALES, 1997). Um interesse pode ser, por exemplo, um requisito funcional, ou não funcional, que é definido pela visão do usuário, analista, arquiteto ou patrocinador com relação ao *software*(JACOBSON,2004). Segundo Viega (2000), a meta dessas linguagens é reduzir a complexidade e promover a reutilização deste *software*, provendo mecanismos mais apropriados de abstração. Estes mecanismos de abstração fornecem uma solução para captura de interesses que são utilizados por todo o sistema, permitindo que esses mecanismos sejam definidos em um único ponto e aplicados uniformemente através de todo o *software*.

A abordagem que mais se destaca como solução para o problema apontado é a programação orientada a aspectos, pois suporta programação orientada a objetos e oferece poderosos recursos para definição de forma modular a captura de interesse de um aplicativo (KICZALES, 1997), e dada essa modularidade, tal abordagem auxilia a definição de métodos de testes de integração com baixo acoplamento.

1.2 Objetivo

O objetivo deste trabalho é desenvolver um método para testar a integração entre componentes de *software* sem a necessidade de extrair o componente, mantendo-o integrado ao restante do sistema. Esse método utiliza a tecnologia da programação orientada a aspectos, e é

definido de maneira não intrusiva, isto é, não se altera o código original do *software* pra incluir referências à atividade de testes, o que facilita a criação de casos de teste e assegura a independência do sistema a ser testado em relação aos testes.

1.3 Contribuições e Resultados Esperados

Uma contribuição deste trabalho à engenharia de *software* é o incentivo do uso da programação orientada a aspectos na área de teste e, em especial, nos testes de integração. Levando para a atividade de teste os inúmeros benefícios dessa abordagem de programação.

Outra contribuição esperada é propor a utilização de um roteiro de testes baseado em XML, que com regras claras e simples que facilitem a geração dos casos de testes de integração. Após a aplicação desse roteiro, é possível observar que diminui a inclusão de condições de testes em rotinas de rastreamento e o log em aplicativos desenvolvidos para Internet.

Como resultado final, este trabalho implementa um *framework* de testes baseado na programação orientada a aspectos a ser utilizado nos testes de integração dos componentes de um sistema de *software*.

1.4 Método de pesquisa

Este trabalho adota um método experimental para testar as hipóteses de pesquisa. Com base em uma revisão teórica das técnicas de teste de *software*, define-se a abrangência de cada uma, dirigindo a atenção, principalmente, aos testes de integração.

A partir das necessidades das técnicas de teste, introduz-se a técnica de programação orientada a aspectos para se avaliar como podem ser realizadas as implementações de interesses ortogonais em sistemas orientados a objetos. Definem-se, então, os requisitos para um mecanismo que implementa testes de integração baseado em AOP. Esse mecanismo é, então, construído para ser aplicado experimentalmente no desenvolvimento de um *software* para a Internet para analisar a efetividade de sua utilização no que diz respeito à captura de falhas, à simplicidade de definição e à manutenção do roteiro de testes, à redução ou eliminação da manutenção do código original do aplicativo para inclusão das referências de testes. Adicionalmente, estuda-se o uso de padrões XML na definição de um roteiro de testes.

1.5 Organização do Trabalho

Este trabalho está organizado da seguinte forma:

O Capítulo 2, Teste de *Software*, apresenta os principais conceitos de Teste de *Softwar*e, faz uma avaliação sobre a maneira como as técnicas de programação hoje capturam requerimentos de sistema, exibe alternativas neste campo com o desenvolvimento de novas técnicas e finaliza demonstrando padrões de roteiro de testes baseados em padrões XML, utilizados para aumentar a flexibilidade e padronização na definição do *script* de roteiro de testes.

O Capítulo 3, Programação orientada a aspectos, apresenta os fundamentos da programação AOP, onde são descritos os principais pontos deste novo paradigma de programação. Como complemento são exibidas as principais barreiras aos testes de integração em aplicativos desenvolvidos para Internet.

O Capítulo 4, Teste de integração apoiado por aspectos, apresenta a descrição da idéia dividida em abordagem, arquitetura e exemplo de uso. Ainda mostra o esquema para construção do mecanismo responsável pela realização dos testes em aplicativos desenvolvidos para Internet, como também o cenário onde ele atuará.

O Capítulo 5, Experimento de uso da técnica, apresenta a aplicação do trabalho realizado em um exemplo simples para demonstrar o potencial do *framework* na realização de um teste de integração de componentes, e avalia os resultados obtidos com o experimento.

O Capítulo 6, Conclusão, Limitações e Futuros trabalhos, realiza uma análise da aplicação do método de testes de integração e seus benefícios, no sentido de facilitar a definição dos testes de maneira não intrusiva em aplicativos desenvolvidos para *Internet*. Descreve as limitações encontradas durante o desenvolvimento do trabalho e sugere, para futuros estudos, a pesquisa de técnicas de cobertura de código para esse tipo de método, visando aumentar a confiabilidade dos testes definidos pelo método com relação ao aplicativo.

2 Teste de *Software*

Neste capítulo são conceituados os testes de *software*, bem como as dificuldades em realizar tais testes em sistemas desenvolvidos para a Internet.

2.1 Conceitos e Definições de Teste de Software

Na literatura, encontram-se registros formais a respeito do tema Teste de *Software* a partir de (MYERS, 1979), ao longo dos anos, várias técnicas foram criadas para auxiliar esse processo.

Existem, segundo Myers (2004), duas abordagens com relação aos testes de componentes: incremental e não-incremental. O Analista de Testes pode assumir a posição de integrar todos os componentes e testá-los em conjunto (não incrementais) ou testar cada componente individualmente e somente acrescentar um novo componente a um conjunto de componentes previamente testados (testes incrementais). Testes incrementais e não-incrementais utilizam-se de recursos como "*Stubs*" e *Driver*s que são definidos a seguir, para auxiliar o entendimento das abordagens:

***Stubs**:* são estruturas criadas para substituir um componente ou programa que ainda não está finalizado, mas que é necessário para validar o componente ou rotina a ser testada (DUSTIN, 2003).

Drivers: são programas criados para realizar a chamada específica de um determinado componente a ser testado passando os argumentos definidos para realização do teste.

As definições de Teste de *Software* são classificadas, segundo Whitaker (2000), quanto à aplicação do teste e ao tipo de teste. Segundo o escopo da aplicação do teste, ele pode ser dividido em:

Teste Unitário: É o processo de testes individuais de um programa, subprograma ou método de *software*, separadamente do restante do sistema. Em alguns casos, na ausência de uma interface de acesso ou de certas partes do sistema, fz-se necessária a criação de *stubs* e *drivers* para auxiliar a execução do teste unitário. (MYERS, 1979).

Teste de Integração: Teste de múltiplos componentes integrados. Tem como foco validar a interdependência dos componentes previamente testados individualmente.

(WHITAKER, 2000) ; ainda é necessário, muitas vezes, criar *stubs* e *drivers*.

Teste de Sistema: Teste de todos os componentes que constituem o aplicativo a ser entregue, integrados de forma a compor o sistema completo. Usualmente o ambiente onde o aplicativo é executado deve ser levado em conta para validação do teste de sistema. (WHITAKER, 2000).

De acordo com o tipo, podemos classificar os testes em:

Teste Estrutural: Teste realizado a partir das entradas de dados, que tem como base o código fonte do aplicativo e sua estrutura, também conhecido como teste de caixa branca ("*White-Box testing*"). (WHITAKER, 2000).

Teste Funcional: Teste realizado a partir da seleção de cenários, tendo como base a especificação do aplicativo e não seu código fonte, também conhecido como teste comportamental ("*Behavioral testing*") ou teste de caixa preta ("*Black-Box Testing*"). (WHITAKER, 2000).

Testes de *Software*, segundo as definições anteriores, atuam em todas as fases de desenvolvimento de um aplicativo, desde a concepção até a fase de distribuição do aplicativo em seu ambiente produtivo. A figura 1 exibe os limites de atuação de cada teste segundo sua classificação de acordo com a aplicação.

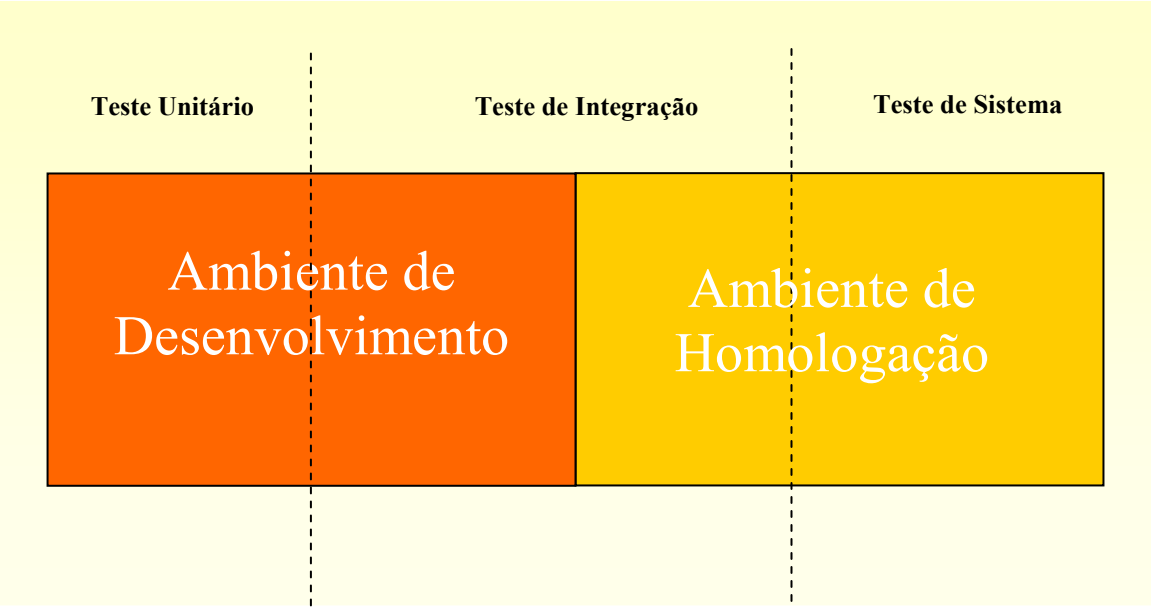

Figura 1. Limites de atuação dos testes quanto a sua aplicação

Na figura 1 são exibidos os ambientes onde são realizados os testes segundo sua divisão como descrito no tópico anterior. O ambiente de desenvolvimento diz respeito àquele onde o analista tem pleno acesso a realizar as alterações e configurações necessárias na implementação de seu aplicativo; neste ambiente não há processos de auditoria e de validação. Em muitos casos, o ambiente é a própria máquina do analista.

O ambiente de homologação por sua vez, é elaborado para garantir que o aplicativo seja executado com base em especificações bem definidas com processos de controle e auditoria que garantam essas premissas. Neste ambiente o analista tem acesso restrito, limitando-se a encaminhar um roteiro de instalação de seu aplicativo que é repetido em ambiente produtivo em caso de sucesso do processo de homologação do aplicativo em teste.

Observando a figura 1, nota-se que os testes, segundo sua classificação, podem ser realizados nestes ambientes sem uma definição específica de qual tipo de testes são realizados em qual ambiente. A linha tracejada estabelece um limite transacional entre os tipos de testes e os ambientes onde os mesmo são executados. Com isso um *framework* de testes deve levar em consideração essa transição dos aplicativos nos ambientes onde os mesmos são testados, assim como os níveis de acesso que um analista tem nestes ambientes para realizar ajustes e configurações necessárias para realizá-los.

Os testes de integração são utilizados para validar a união de componentes de *software* verificando se não apresentam falhas novas que não foram detectadas durante a execução dos testes unitários destes componentes. Em aplicativos desenvolvidos para Internet, este tipo de teste tem importância acentuada devido à grande quantidade de componentes e de interfaces integradas. Entre as abordagens para realização destes testes, segundo Dustin (2003), existe a necessidade de realizar alteração intrusiva para inclusão e remoção das referências para verificação da atividade de testes, em um ambiente de homologação controlado por processos bem definidos, o que resultaria no reenvio do aplicativo após a remoção das referências de testes causando atrasos e retrabalho na fase de desenvolvimento. O capítulo 2.3 descreve com mais detalhe a relação entre os tipos de teste e *softwares* desenvolvidos para Internet.Descrevem-se os principais desafios e as técnicas existentes para avaliar este tipo de *software*.

2.2 Testes de Integração

Segundo Gao (2003), sistemas de *software* eram constituídos por um programa principal que controlava a chamada de sub-rotinas. Cada sub-rotina era criada como parte específica de um programa baseado nos requisitos do sistema. Com isso, para reduzir esforços programadores em uma equipe utilizavam-se dessas sub-rotinas para implementar seus sistemas, logo essas sub-rotinas tornaram-se um dos primeiros formatos de reuso em *software*. Mcilory (1969) definiu o conceito de componentes e, a partir deste ponto, engenheiros e analistas passaram a dividir seus aplicativos em partes menores com funções específicas que, integradas, formavam o sistema de *software*. Desde então, muitas abordagens têm sido propostas no sentido de garantir a qualidade destes componentes tanto do lado dos desenvolvedores quanto do lado dos clientes, em processos de avaliação e aceitação desses componentes.

Testes de programas orientado a objetos segundo Howden (1987) podem ser classificados de duas maneiras: testes baseado em especificação e teste baseado em programa. Testes baseados em especificação dependem diretamente da qualidade da especificação do sistema, pois é a partir dela que são gerados os casos de teste. No caso de testes baseados em programa,um dos grandes problemas envolvendo esses testes é ausência do código fonte o que impossibilita a aplicação de técnicas de cobertura estrutural e técnica baseada em defeitos. Contudo, alguns critérios alternativos de testes baseados em programas vêm sendo propostos uma vez que não apresentam a restrição da presença do código fonte, são eles: reflexão computacional, polimorfismo, metadados e autoteste.

Reflexão computacional: Permite acesso à estrutura interna de um programa e inspeção do seu comportamento, sendo utilizada na área de teste para automatizar a execução dos testes, criando instâncias de classes invocando métodos em diferentes seqüências (MALDONADO et al., 2007).

Polimorfismo: Dada à especificação formal de um componente contendo as pré e pós-condições que devem ser satisfeitas na invocação de cada método, métodos polimórficos são criados de modo que, antes da invocação do método real, a versão polimórfica do método verifica se a pré-condição é satisfeita, coleta as informações usadas na invocação do método e constata se as pós-condições foram satisfeitas (MALDONADO et al., 2007).

Metadados: São utilizados pelos modelos de desenvolvimento de componentes para fornecer informações genéricas sobre o componente, tais como o nome de suas classes, o nome de seus métodos, bem como informações para o teste. Pelo uso de metadados, aspectos estáticos e dinâmicos do componente podem ser consultados pelo cliente para realização de diversas tarefas. O problema com essa abordagem é que ainda não há consenso sobre o conjunto de informações que deve ser disponibilizado, nem sobre a forma de como disponibilizá-lo (MALDONADO et al., 2007).

Autoteste: Disponibilizar componentes de *software* com capacidade de teste embutidas e que possam ser habilitadas e desabilitadas, dependendo se o componente encontra-se em operação normal ou em manutenção. É inserido no componente, através de um empacotador de metadados reflexivo, algumas informações tais como: especificação do componente, documentação, histórico de verificação e serviços de auto-teste. O problema com essa abordagem é que ela exige tempo e recursos adicionais por parte do desenvolvedor uma vez que é sua responsabilidade a coleta e inclusão de tais informações (MALDONADO et al., 2007).

Como descrito nesta seção, os critérios de teste têm por objetivo auxiliar a atividade de testes a partir da visão do cliente, uma vez que ao adquirir componentes de terceiros para integrar com as demais aplicações existentes não têm acesso ao código fonte e às informações sobre a documentação do componente não é suficiente para realização dos testes. Isto implica que, exceto se o componente possuir informações descritas nos critérios acima citados, nenhuma informação sobre o estado interno do componente pode ser obtida pelo cliente ao avaliar a cobertura estrutural em seus testes.

O papel do teste é avaliar se o componente está sendo utilizado de maneira coerente e que sua integração respeita a especificação fornecida pelo produtor (MALDONADO et al., 2007). Ao realizar testes de integração, um de seus inúmeros desafios é como obter o fluxo da informação através da execução dos métodos internos de seus componentes. Ao analisar este tipo de comportamento devemos citar os três níveis de testes que envolvem o fluxo de dados através das classes (HARROLD et al., 1994):

Teste intramétodo: testa os métodos individualmente. Esse nível é equivalente ao teste de unidade de programas.

Teste intermétodo: testa os métodos públicos em conjunto com outros métodos dentro de uma mesma classe. Esse nível de teste é equivalente ao teste de integração. Esse teste deseja descobrir defeitos nas interfaces de comunicação entre os métodos.

Teste intraclasse: testa a interação entre métodos públicos quando eles são chamados em diferentes seqüências. Como os usuários de uma classe podem invocar seqüências de métodos em uma ordem indeterminada, o teste intraclasse serve para aumentar a confiança de que essas diferentes seqüências de invocação colocam a classe em estado inconsistente.

Apesar da disseminação do desenvolvimento baseado em componentes a atividade de teste ainda possui grande esforço em realizar e automatizar testes de integração por não ter acesso a todas informações do comportamento interno de um componente, as abordagens propostas, com exceção da reflexão computacional, têm por objetivo adicionar informações relevantes à área de testes no momento do desenvolvimento dos componentes, porém, apesar de útil, levará muito tempo até que todo esse processo seja disseminado e adotado como padrão.

2.3 Testes de aplicativos para Internet

Existem hoje algumas abordagens para a realização de testes em aplicativos para Internet, que dizem respeito aos testes unitários, porém, segundo Hieatt (2002), há grande dificuldade em realizar esses testes. Isso ocorre devido ao grande número de entidades e interfaces que podem conectar-se ao aplicativo tornando-se parte da solução, inclusive a existência de relacionamentos com entidades externas, que possuem pouco ou nenhum controle para esses testes dentro do ambiente onde é desenvolvido o sistema. Como conseqüência, são gastas muitas horas para implementação de *Stubs* para realização desses testes, o que pode inviabilizar economicamente a construção das rotinas de testes destes sistemas de *software*.

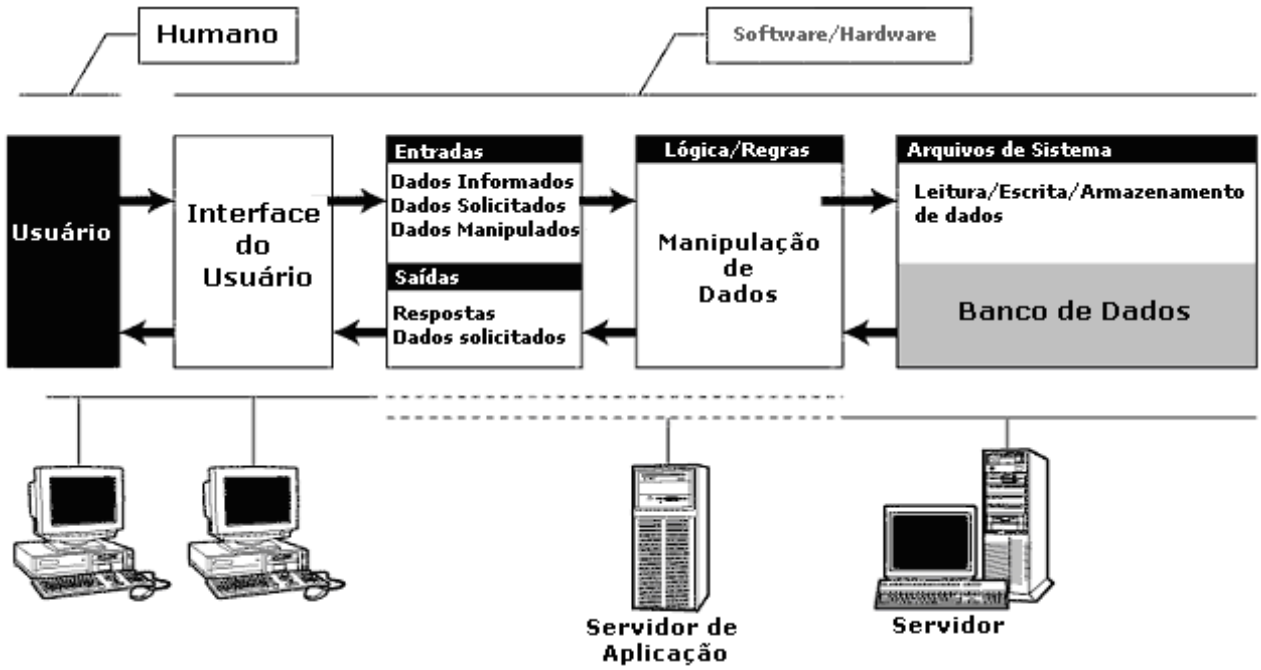

Figura 2: Arquitetura de uma aplicação distribuída segundo NGUYEN et al.,2003

A figura 2, baseada em Nguyen (2003) ilustra a arquitetura em camadas utilizadas para desenvolver uma aplicação para *Internet*, para cada solicitação do usuário: através de sua interface as informações são enviadas ao servidor que executa as funcionalidades relacionadas, persistindo as informações em base de dados ou arquivos de sistemas, e retorna a informação resultante desse processamento. Para realizar um teste de integração neste ambiente é necessário alterar os módulos de cada uma das camadas com referências que indiquem se a informação ao transitar de uma camada a outra continua a atender as especificações de suas funcionalidades. Essa abordagem, além de demandar grande esforço, é apenas parcial, pois trata apenas as transformações ocorridas na transição entre as camadas, não verificando a interação entre os componentes dentro de cada módulo.

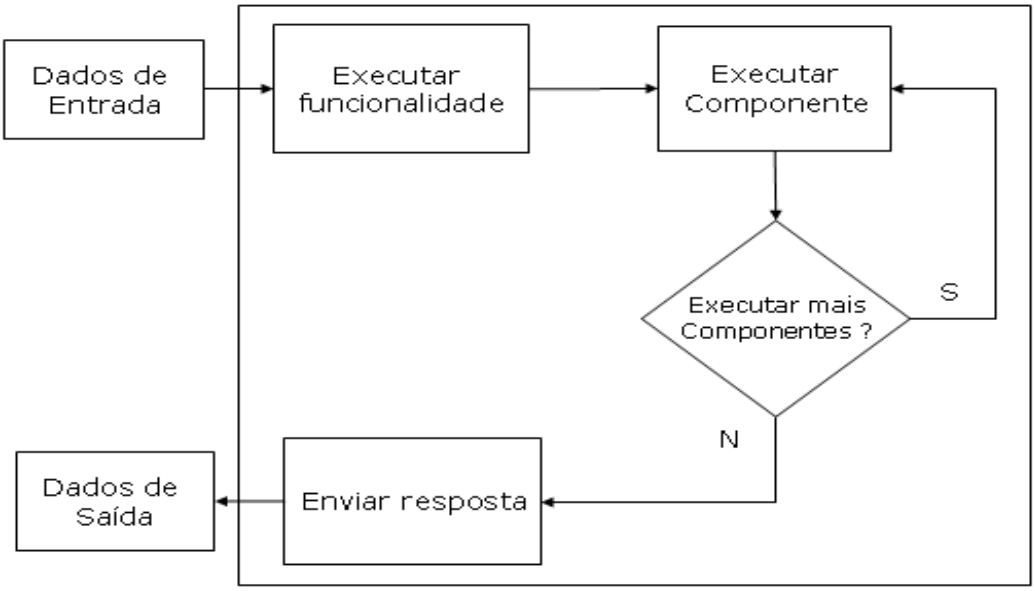

Figura 3. Detalhe do módulo de regras de negócio da arquitetura distribuída

A figura 3 ilustra uma situação onde é possível observar que, ao executar uma funcionalidade a partir dos dados de entrada, ocorre a execução de vários componentes intermediários que se integram para compor a funcionalidade e retornar com a resposta. Esses componentes, segundo a classificação de Harrold et al. (1994), executam integrações nos níveis intermétodo e intraclasse que não são avaliados. Quando ocorrem falhas em aplicativos com esta arquitetura é difícil afirmar com precisão qual componente apresentou o problema devido à superficialidade das abordagens (DUSTIN,2003) realizadas atualmente.

Para cada interação dos componentes do aplicativo, tanto no nível intermétodo quanto no intraclasse (HARROLD et al., 1994), é necessário prover mecanismos de captura e registro dessas interações para posterior avaliação. No caso de aplicativos desenvolvidos para internet, esses mecanismos confundem-se com rotinas de *log* e com servidores de aplicação.

As rotinas de *log* são utilizadas para registrar comportamentos do aplicativo em caso de exceção, são responsáveis por escrever, para os arquivos de sistema do servidor, informações que auxiliem o analista a rastrear possíveis problemas em seu aplicativo. No caso da abordagem de utilizar estas rotinas para a atividade de testes, são lançados nesses *logs*, estados de objetos, conteúdo de variáveis e resultados de eventos que não são permitidos por áreas de auditoria.

Ferramentas de teste como o HttpUnit (2003), realizam uma chamada remota ao aplicativo realizando a verificação do retorno dessa requisição. Com isso, é possível realizar apenas uma validação superficial dos serviços, deixando a integração dos vários componentes que formam o serviço sem verificação. O que obriga os programadores a utilizar rotinas de *log* para verificar o resultado destas interações intermediárias.

Dessa forma, ao utilizar rotinas de *log*, será necessário alterar cada componente a ser testado incluindo a referência dessa rotina para execução das rotinas de teste, e que, devido ao volume de alterações nos aplicativos, pode gerar problemas de concorrência entre as equipes de teste e de desenvolvimento.

E por fim, após realizar esses testes de integração, uma nova alteração é necessária para remover as referências das rotinas de teste dos serviços para promover o aplicativo para o ambiente produtivo, podendo, nesse momento, inserir novas falhas no aplicativo, ao remover acidentalmente parte do código funcional durante a exclusão dos trechos de código relacionado aos testes.

O fato de realizar Testes de Integração com rotinas de *log* (DUSTIN, 2003), que realizam alteração intrusiva do código do aplicativo, constituem um processo trabalhoso e pouco eficiente, mas que estão estritamente relacionados às técnicas de programação existentes hoje, e por mais que os componentes do código possuam baixo acoplamento, o relacionamento entre as várias funcionalidades do aplicativo necessitará de pequenas alterações que podem gerar novas falhas.

2.4 Caso de Teste

Um caso de teste pode ser definido como um conjunto de entradas, condições e resultados esperados que validam se uma determinada funcionalidade, segundo os requerimentos definidos para ela. Um caso de teste deve ser definido no início de um projeto, pois facilita o entendimento de todas as possíveis situações a que é submetida às funcionalidades do projeto. Dada a definição de um caso de teste, o mesmo pode ser utilizado durante todo o ciclo de vida de um projeto durante as várias fases de testes, como testes unitários, testes de integração e testes de sistema (COPELAND, 2004).

Segundo Copeland (2004), casos de testes bem definidos são compostos de três partes : Entradas, Saídas e Ordem de Execução.

- Entradas – Geralmente são informações digitadas pelo usuário, porém as entradas podem ter origem em interfaces de sistemas, da leitura de arquivos e de banco de dados, do estado do sistema quando o dado é recebido, e o ambiente dentro do qual o sistema é executado.
- Saídas – A informação gerada na tela de um computador é o raciocínio mais comum para uma saída, contudo ele apresenta a mesma variedade de origens descrita para as entradas. No desenvolvimento de casos de teste, a tarefa de determinar os resultados esperados é função de um oráculo. Esse oráculo pode ser definido como um programa, processo ou dado que irá prover ao testador um resultado para um caso de teste.
- Ordem de execução – Quanto à ordem de execução, existem dois estilos segundo Copeland (2004) : cascata e independente.
    - Cascata – Após a execução de um caso de teste que exercita uma funcionalidade, o *software* deixa o sistema em um estado preparado para que o segundo caso de teste possa ser executado.
    - Independente – Cada teste é auto-suficiente, ou seja, eles independem da execução com sucesso de testes anteriores para que sua execução seja efetuada. A principal vantagem é que os testes podem ser executados em qualquer número independente da ordem. A desvantagem é que estes tipos de testes podem se tornar complexos, difíceis de projetar, de criar e de manter.

Segundo Myers (2004) e Copeland (2004), é impossível testar completamente um programa, contudo existem técnicas e estratégias que podem ser adotadas que permitam detectar o maior número de erros possíveis, com o menor número de casos de testes. Essas técnicas são apresentadas com maiores detalhes visando analisar suas vantagens, limitações e algumas estratégias de utilização.

Em geral, segundo Copeland (2004), o processo de teste funcional é definido da seguinte forma:

- Os requerimentos e especificações são analisados.

- Entradas válidas são escolhidas baseadas na especificação para determinar que o aplicativo a ser testado processou-os corretamente. Entradas inválidas são também escolhidas para verificar que o aplicativo a ser testado consiga detectar essas entradas e tratá-las apropriadamente.
- Definição de saídas esperadas para as entradas estabelecidas.
- Testes são criados com as entradas selecionadas.
- Os testes são executados.
- As saídas são comparadas com as saídas esperadas.
- É determinado se o aplicativo em teste está funcionando apropriadamente.

Contudo, esse tipo de teste apresenta algumas desvantagens, pois o analista de teste nunca tem a certeza de que todo fluxo do aplicativo a ser testado é exercitado baseado nas entradas escolhidas. A definição do conjunto de entradas pode ser impossível de ser alcançada devido ao número muito alto de combinações possíveis nas entradas válidas e inválidas de uma funcionalidade. Para auxiliar a seleção das melhores combinações possíveis que determinem o funcionamento apropriado do aplicativo são avaliados as seguintes técnicas: *Equivalence Class Testing, Boundary Value Testing, Decision Table Testing, State-Transition Testing, Domain Analysis Testing, Use Case Testing, Error Guessing.*

***Equivalence Class Testing* (Teste de classe de equivalência)**

Segundo Myers (2004) e Copeland (2004), a técnica de classes de equivalência é utilizada para reduzir o número de casos de testes que são aplicados para validar um aplicativo com o objetivo de manter uma razoável cobertura. Essa técnica tem por objetivo eliminar testes equivalentes, ou seja, que, a partir de um conjunto de entradas, o resultado do módulo é sempre o mesmo. Com isso, não há a necessidade de repetir exaustivamente um grande número de testes, uma vez que essas entradas exercitam da mesma maneira o módulo a ser testado. Isso é válido desde que, no módulo a ser testado, não existam condições muito especificas para um determinado valor dentro do intervalo de entradas.

Essa técnica também define que para cada entrada inválida é gerado um teste separado, o que ocorre devido ao fato de que, se houver o agrupamento de casos de testes inválidos, durante a execução desses testes, a falha na verificação de uma das entradas inválidas pode mascarar a execução das demais.

***Boundary Value Testing* (Teste de fronteira)**

Testes de valores limites foram direcionados a partir da técnica de classes de equivalência onde é demonstrado como selecionar um subconjunto de entradas capaz de exercitar de forma razoável o aplicativo em teste. Caso de testes baseados em valores limites têm-se mostrado muito eficientes na revelação de falhas em aplicativos, por isso essa técnica visa selecionar um conjunto de casos de testes capazes de validar essas fronteiras. Myers (2004) afirma que as principais diferenças desses casos de testes, baseados em valores limites, com relação aos testes de classes de equivalência são:

- Não apenas selecionar qualquer valor de entrada dentro de um escopo válido. Devem ser considerados os valores limites das entradas das classes de equivalência.
- O foco deste teste não se resume apenas aos valores limites das entradas, como também aos valores limites dos resultados esperados.

***Decision Table Testing* (Teste de tabela de decisão)**

Tabela de decisão é utilizada para selecionar casos de testes para aplicativos que possuem regras de negócio complexas. Essas regras são combinadas em condições que resultem em ações associadas a estas condições (COPELAND, 2004).

O formato de uma tabela de decisão é representado abaixo:

	Regra 1	**Regra 2**	**Regra 3**	**Regra 4**
Condições				
Condição-1	Sim	Sim	Não	Não
Condição-2	Sim	Não	Sim	Não
Ações				
Ação-1	Faça X	Faça Y	Faça X	Faça Z
Ação-2	Faça A	Faça B	Faça B	Faça B

Tabela 1. Tabela de decisão segundo Copeland(2004)

Após a definição de uma tabela de decisão, escolher os casos de testes é uma tarefa simples, onde as colunas de regras tornam-se casos de testes, as condições são as entradas e as ações, os resultados esperados. Em certos aplicativos onde existe um grande número de condições, tabelas de decisão podem tornar-se grandes e de difícil manutenção, porém, em alguns casos, é possível utilizar intervalos de valores para as condições. Quando isso ocorre no momento de selecionar os casos de testes, são considerados o valor inferior e o valor superior do intervalo, onde é aplicado o princípio do teste de valor limite (COPELAND, 2004).

***State-Transition Testing* (Teste de transição de estado)**

Teste de transição de estado tem como base o diagrama de transição de estado, onde, a partir destes diagramas é possível formatar uma tabela com as seguintes características:

Estado Atual	**Evento**	**Ação**	**Próximo Estado**

Tabela 2. Tabela de transição de estados

A tabela contém todos as transições de estado possíveis dentro da definição do sistema, com isso é possível aplicar segundo Copeland (2004), quatro níveis de cobertura para geração dos casos de teste.

1. Criar um conjunto de casos de teste tal que todos os estados sejam visitados ao menos uma vez.
2. Criar um conjunto de testes tal que todos os eventos sejam disparados ao menos uma vez.
3. Criar um conjunto de testes tal que todos os caminhos são executados ao menos uma vez.
4. Criar um conjunto de testes tal que todas as transições de estado sejam exercitadas pelo menos uma vez.

Note-se que os níveis 1 e 2 geram um conjunto com baixa cobertura, enquanto 3 tem a melhor cobertura, porém pode gerar um número muito grande de testes, principalmente em sistemas que possuem “loop”, enquanto 4 provê a melhor cobertura sem gerar um grande número

de testes.

No nível 4, os casos de teste podem ser extraídos diretamente da tabela de transição de estado e ,dependendo do risco do sistema, podem-se criar alguns casos de teste com transições inválidas para garantir que não houve implementações errôneas em seu aplicativo.

***Domain Analysis Testing* (Teste de análise de domínio)**

Análise de domínio é uma técnica que pode ser utilizada para determinar casos de teste de múltiplos parâmetros sendo validados juntos. Essa técnica generaliza, em múltiplas dimensões, as técnicas de classe de equivalência e o valor limite que são focadas em parâmetros individuais para as entradas de um aplicativo.

A técnica consiste em determinar alguns casos de teste conforme as definições a seguir:

- Um ponto "**on"** que representa um valor no limite definido para variável.
- Um ponto "**off**" que representa um valor que não está no limite definido para variável.
- Um ponto "**in**" que representa todos os valores válidos do limite definido para variável e que não está na fronteira do limite definido para variável.
- Um ponto "**out**" que representa um valor fora do limite válido definido para variável.

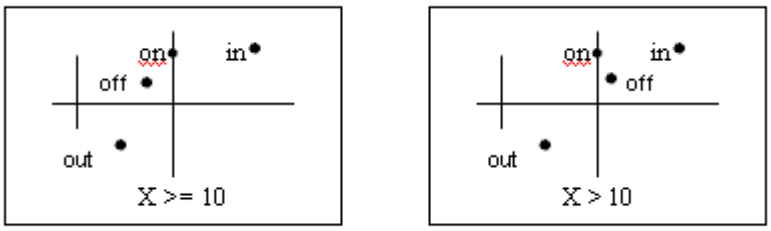

Figura 4 - Exemplo de teste de análise de domínio.(COPELAND, 2004)

A figura ilustra os pontos definidos na técnica de análise de domínio, tendo esses pontos definidos a técnica de análise de domínio nos guia na escolha dos casos de teste da seguinte forma:

- Para cada condição relacional (>=, >, <= ou <) escolha um ponto "**on**" e um ponto "**off**".
- Para cada condição de igualdade (=) escolha um ponto "**on**" e dois pontos "**off**", um logo abaixo do valor definido e o outro logo acima deste mesmo valor.

Para facilitar o entendimento do uso da técnica é utilizada uma tabela chamada de "Matriz de Domínio de Teste" conforme a seguir:

Variável			Casos de Teste		
Tipo Condição			1	2	3
X1	C11	on			
	C12	off			
	Comum	in			
X2	C21	on			
	C22	off			
	Comum	in			
Resultados esperados					

Tabela 3. Matriz de domínio de teste

Dado um método que possui dois parâmetros de entrada e tem como resultado esperado a soma destes dois valores. Basta utilizar a tabela 3 para definir os casos de testes de acordo com a obtenção dos pontos "on", "off" e "in" conforme as regras definidas.

Com a tabela, é possível visualizar com clareza a combinação das variáveis sendo validadas em conjunto, utilizando-se os valores limites estabelecidos para essas variáveis.

***Use Case Testing* (Teste de caso de uso)**

Casos de uso são baseados no exercício das funcionalidades de um sistema, enquanto técnicas como classes de equivalência e teste de valor limite preocupam-se com as entradas esperadas, tabelas de decisão com as regras de negócio e testes de transição de estado com o comportamento do aplicativo. Testes de casos de uso consideram as funcionalidades do início ao fim, testando cada transição individualmente.

Segundo Jacobson et al. (1999) ,casos de uso são definidos a partir da perspectiva do usuário. Um caso de uso é um cenário que descreve o uso do sistema por um ator para atingir uma meta. Um ator normalmente é um usuário exercendo um papel com relação ao sistema buscando realizar alguma ação dentro de um determinado contexto. Um cenário é uma seqüência de passos que descreve as interações entre o ator e o sistema.

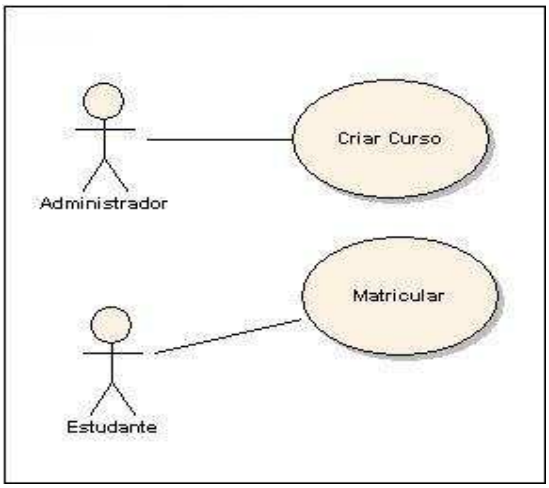

Figura 5. Exemplo de caso de uso (COPELAND, 2004)

Como cada caso de uso é passado por um processo de inspeção antes de ser implementado (COPELAND, 2004), para testar a implementação, a regra básica é criar ao menos um caso de teste para o cenário de sucesso principal e ao menos um caso de teste para cada cenário secundário do caso de uso. Como casos de uso não especificam entrada de dados, o analista de teste deve selecioná-la, para isso basta utilizar técnicas como classes de equivalência e valor limite já discutidas nas seções anteriores.

***Error Guessing* (Teste intuitivo)**

Para Myers (1979), algumas pessoas têm grande capacidade em definir casos de testes de forma intuitiva sem a utilização de nenhuma técnica específica para isso. Dado um particular problema essas pessoas utilizam-se de experiência e intuição para escrever casos de testes que exponham estes problemas. É difícil definir um procedimento para este tipo de teste, a idéia básica é escrever uma lista de possíveis erros ou situações de erro e definir os casos de testes que atendam a esta lista.

Analisando algumas técnicas de testes de caixa-preta pode-se observar que tais técnicas complementam-se, não sendo possível utilizar uma única técnica para selecionar os casos de testes mais efetivos. Com isso uma ferramenta de testes deve suportar o maior número possível destas técnicas para garantir ao analista de testes opções no momento da definição do roteiro de testes a ser executado.

Durante a elaboração deste capítulo viu-se a definição de testes, sua relação com o ambiente e as técnicas de critérios de testes mais conhecidas para auxiliar a definição do roteiro de testes que é parte fundamental no desenvolvimento de um *framework* de testes. Houve atenção especial aos aplicativos desenvolvidos para Internet, por ser este o cenário avaliado ao se aplicar algumas das técnicas de teste de integração baseadas nas abordagens existentes (DUSTIN,2003).

2.5 Uso de roteiro XML em testes

Segundo a proposta de Morris (2001) e Voas (2000), é possível realizar certificação de software através de uma metodologia padrão de geração de testes, escrita pelos próprios analistas, aplicando-a no local onde o sistema esta sendo executado. Por sua vez estes testes são entregues aos compradores junto com o aplicativo para demonstrar a eficiência.

Para que seja possível realizar com clareza e transparência os testes, foi criada uma especificação para geração do roteiro de testes com as seguintes características (MORRIS, 2001):

- Deve ser utilizado um padrão de *script* aberto e portável;
- O padrão deve ser simples e fácil de aprender;
- O padrão deve ser desprovido de características específicas de linguagem de programação;
- O padrão deve ser igualmente hábil para trabalhar com sistemas orientados a objetos, funções simples e componentes complexos;
- O padrão deve ser eficiente em lidar com a natureza repetitiva de muitos conjuntos de testes;
- O padrão deve ser capaz de oferecer uma grande quantidade de ferramentas de geração destes testes, fáceis de usar e de natureza não proprietária;

- O padrão deve ser livre de requisitos proprietários de software para interpretar e executar os testes;

- O padrão deve ter habilidade de suportar teste de regressão.

Na figura 10 é possível observar as tags da especificação de uma gramática que atenda aos requisitos acima citados:

```
<TestSpecification>
        <TestSet Name=”...”>+
                <TestGroup Name=”...”>*
                        <TestGroup>*
                        <Invariant DataType=”...”>*
                        <Operation Name=”...” Pré=”op_name”>*
                                <Invariant>*
                                <Constructor>
                                <MethodCall Target=”…”>*
                <Operation>
                <Invariant>
```

Figura 6. *Tags* de uma gramática de especificação de testes (MORRIS, 2001)

Observando a figura 6 nota-se a estruturação de uma especificação de testes, onde cada elemento descreve uma função especifica dentro do roteiro, e seguindo esta forma estruturada de declaração é possível definir uma seqüência de ações e seus atributos de forma a validar as operações envolvidas nos testes durante a interpretação do roteiro. Com isso é possível criar um roteiro de testes baseado em XML, que seja responsável pela execução dos testes de integração de um aplicativo. Esta definição existe para realizar a certificação de componente de software, onde a partir da geração de um roteiro de testes definido pelo analista, este roteiro é enviado para uma entidade certificadora independente que realizaria uma validação deste roteiro com relação ao aplicativo.

Porém, o próprio autor (MORRIS, 2001) define seu uso como não apropriado para sistemas de alto risco, onde o controle sob certificação deve ser bastante rígido.

Contudo, a utilização deste padrão para testes de integração em aplicativos desenvolvidos para Internet é uma alternativa interessante a ser avaliada, pois sua geração é simples baseada em padrões que são largamente utilizados no mercado.

2.6 Barreiras aos Testes de Integração

O grande desafio em realizar-se teste de integração nos modelos de sistemas existentes hoje, tanto estruturados como orientados a objetos, é o fato da inclusão de código para auxiliar a realização das validações necessárias aos testes, mas que não pertencem ao modelo de negócio inicialmente projetado. Isto porque, a cada execução dos testes é necessária uma nova intervenção por parte dos analistas para remoção ou indisponibilidade destes métodos tornando o sistema passível de novas falhas.

A figura 11 ilustra a execução de um teste de integração. Através de um *driver,* que possui o mapeamento necessário para executar determinada funcionalidade, é informado um valor de entrada específico para a mesma, esta funcionalidade então é executada e produz uma saída que é validada por um módulo denominado *checker* que armazena o resultado esperado da funcionalidade para comparação. Pode-se perceber que a execução de um simples teste envolve a integração de vários componentes, porém somente é verificado que dado uma simples entrada baseada na interface do usuário, apenas o resultado final da execução desta funcionalidade é validado, não nos sendo possível testar e avaliar as integrações intermediárias daquela execução.

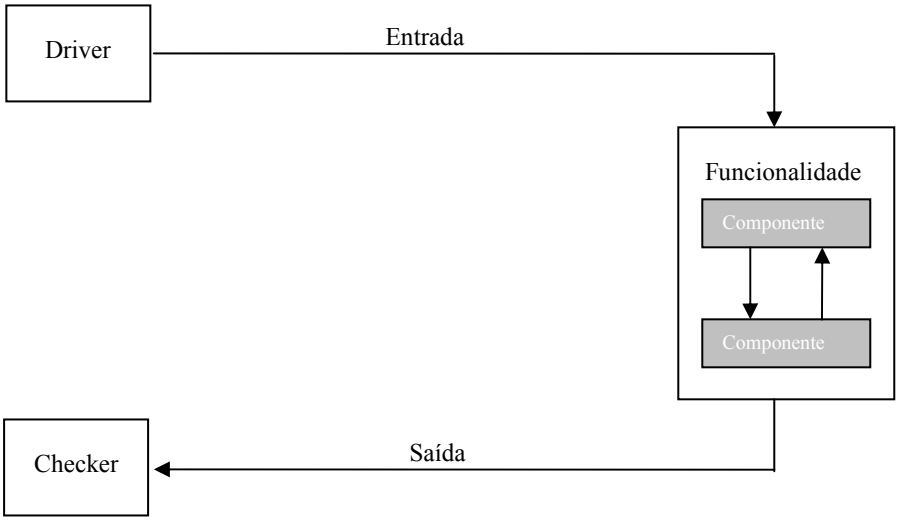

Figura 7. Execução de um teste de integração em uma funcionalidade

Há pouca literatura sobre o acesso a informações mais detalhadas sobre os componentes de um sistema para realização dos testes de integração, tais como, acesso à entrada e saída de dados destes componentes, posições de memória, estado dos componentes entre outros. Jorgensen (1994), enfatiza a necessidade dos testes de integração, devido ao fato de sistemas desenvolvidos em linguagens orientadas a objetos possuírem natureza comportamental, diferente de sistemas baseados em linguagens estruturadas, onde a partir de testes unitários é possível garantir um menor número de falhas destes sistemas.

Ao utilizar a estratégia não-incremental ("*big-bang*") para realizar os testes de integração conforme execuções exibidas na figura 11, é difícil afirmar com precisão o ponto exato de falha do aplicativo visto que, o acesso a validação dos componentes da funcionalidade não é possível sem intervenção no aplicativo original, o que faz ferramentas baseadas neste tipo de execução apresentarem resultado superficiais e imprecisos sobre as causas do problema. Isto ocorre, pois estas ferramentas de mercado imitam o comportamento do usuário final decorando uma seqüência de testes a partir da interface do aplicativo e repetindo esta seqüência a cada nova bateria de testes.

As intervenções de código em aplicativos desenvolvidos para Internet para realizar a atividade de teste, são utilizadas para complementar as ferramentas baseadas na abordagem "*Capture/Playback*" e segundo Dustin (2003) são chamados de "*testability hooks*", a figura 8 demonstra o mecanismo de *hooks*. A técnica "*Capture e playback*" consiste em gravar o comportamento do usuário armazenando a seqüência utilizada para execução de uma determinada funcionalidade para em seguida executá-la de forma sistematizada.

Ainda no processo de complementar tais ferramentas de "*capture/playback*" muitos analistas de testes utilizam processos de log dos aplicativos para exibir estados intermediários de componentes durante a execução de uma funcionalidade, na tentativa de verificar o resultado destes componentes integrados, garantindo que sua utilização é correta e reafirma os resultados dos testes unitários. Porém este tipo de utilização torna a manutenção do aplicativo complexa e confusa, pois é difícil determinar o que realmente está sendo utilizado como log operacional ou informações da abordagem de testes, sobrecarregando as logs e aumentando a complexidade do sistema em futuras manutenções.

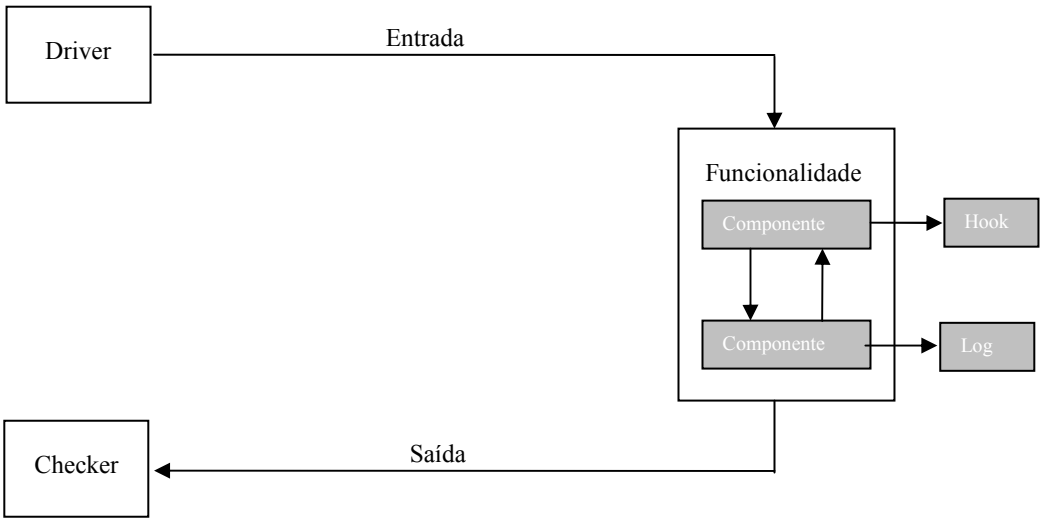

Figura 8. Execução de um teste de integração em uma funcionalidade com auxílio de Hooks e logs

Esta característica da necessidade de se ter acesso a valores intermediários no momento da integração dos componentes reforça a idéia de um modelo que permita tal acesso sem a intrusão nos fontes originais. Este modelo também é útil na automação dos testes de regressão uma vez que é independente do aplicativo, não causando conflitos no planejamento das áreas de sistemas e testes.

O ponto de discussão é como obter este modelo que atue independentemente do sistema onde o mesmo consiga integrar-se de maneira não intrusiva ao sistema a ser testado obtendo o estado intermediário dos componentes de uma funcionalidade para sua posterior validação. Alguns estudos (MORRIS, 2001) e (VOAS, 2000) citam a construção de um módulo de certificação de componentes (”*Test-Pattern Verifier*”), onde a partir de uma lista de testes executa-se a certificação de um componente. Porém não é claro o momento em que o procedimento é utilizado para a realização desta certificação, nem mesmo se existe alteração do código original resultante de sua aplicação.

Outra abordagem realizada por (JACOBSON, 2004) com base em AOSD (“*Aspect Oriented Software Development*”), mostra as vantagens de utilizar-se AOSD no desenho, definição e implementação de sistemas que possuem interesses ortogonais. Estes interesses ortogonais em sua forma geral não são bem interpretados por linguagens orientadas a objetos, uma vez que percorrem todo o sistema causando grande impacto nestes sistemas em processos de manutenção evolutiva.

Com AOSD é possível definir e manter interesses ortogonais separados dentro de um sistema.

No caso de teste de software se considerarmos os testes como um outro interesse dentro de um projeto de sistema, e como este interesse percorre todos os componentes de um sistema com o objetivo de validar sua funcionalidade, teste pode ser definido como um interesse ortogonal com relação aos componentes de um sistema. Com isso, considerar teste como um interesse ortogonal nos permite aplicar os recursos definidos em AOSD para definição de um modelo que venha a atender com maior visibilidade, baixo acoplamento e de forma não intrusiva os testes de integração de um sistema.

3 Programação Orientada a Aspectos

3.1 Introdução

Desenvolvimento de software representa uma evolução constante das técnicas de construção de aplicativos em virtude de ambientes complexos que acompanham a evolução da tecnologia. Além disso, o desenvolvimento de um aplicativo conforme as visões de analistas, clientes, gerentes e arquitetos, aumenta a dificuldade do seu desenvolvimento no sentido de atender a estas demandas de forma sincronizada resultando em um produto. Segundo Jacobson (2004) a resolução dos conflitos estabelecidos necessita de novas abordagens que permitam atender as necessidades de cada visão de forma modular. Jacobson (2004) afirma que não importa quanto esforço seja realizado no desenvolvimento, ainda há fragmentação do seu código em função dos requisitos de sistemas tanto funcionais como não funcionais. AOP representa este tipo de fragmentação de forma modularizada em classes especialistas com suas respectivas operações. Com a separação dos requisitos de forma modular torna-se mais simples tanto a manutenção do aplicativo como atender as necessidades das várias visões envolvidas em seu desenvolvimento.

3.2 Técnicas de Programação

No início, as técnicas de desenvolvimento de software concentravam-se no modelo estruturado (JORGENSEN, 1994). A partir de testes unitário e estrutural era possível garantir que os componentes, individualmente testados, teriam grande chance de não apresentar falhas quando integrados, pela própria natureza seqüencial desses modelos estruturados.

Com o advento da orientação a objetos, novas propriedades foram criadas tais como: herança, polimorfismo, persistência, reutilização de componentes. A partir disso, observou-se que não bastaria apenas testar estes aplicativos de maneira estrutural como antes pois, devido à relação de colaboração entre as várias instâncias de objetos, comportamentos diferenciados seriam capturados de acordo com a relação entre eles (figura 9).

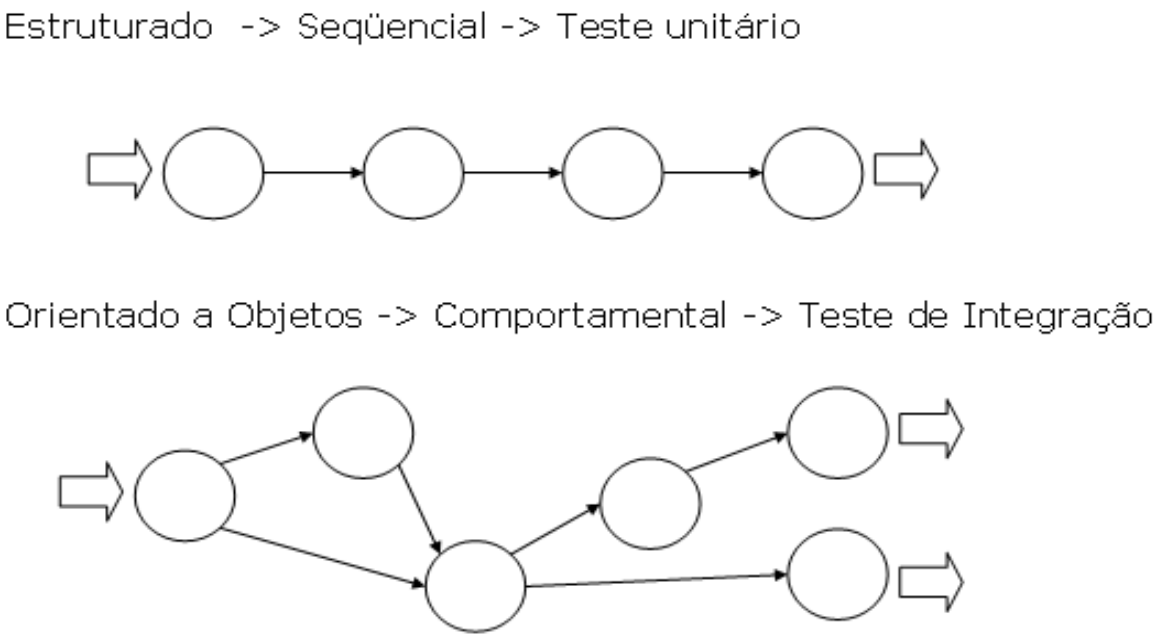

Figura 9. Relação entre componentes segundo Jorgensen (1994)

Estas novas variáveis deram uma característica mais comportamental do que estrutural aos aplicativos desenvolvidos. Tais definições integram-se com a própria fundamentação de teste de software que tem como base de interesse o que o sistema faz e não o que ele é (JORGENSEN, 1994). Com isso testes de integração tiveram sua importância acentuada, uma vez que revelava falhas impossíveis de serem identificadas por testes unitários.

A técnica de programação orientada a objetos, possibilita mapear em estruturas de classes, situações da vida real com maior fidelidade do que as técnicas de programação estruturada. Contudo ainda existem nesta técnica certos interesses que não são bem capturados gerando conseqüências na estrutura do sistema. Dentre essas conseqüências podemos citar: entrelaçamento, espalhamento e intrusão. Os interesses dentro de um módulo de um aplicativo podem estar relacionados a segurança, auditoria e rastreabilidade, a presença de referências que

representem estes interesses definidos dentro de um mesmo módulo de um aplicativo caracteriza o entrelaçamento. No caso dos interesses que possuem suas referências em vários módulos que compõe o aplicativo caracterizam o espalhamento. E por fim a intrusão que é a alteração do código original para incluir as referências dos interesses que estão presentes no aplicativo. Estas conseqüências podem também ser observadas ao realizar os testes de integração dos componentes de um aplicativo, pois a abrangência dos testes acaba espalhando por todo aplicativo referências temporárias necessárias para validação de seus componentes.

A meta de um sistema é reunir requisitos (JACOBSON, 2004), ou genericamente, interesses. Um interesse ("*concern*") é tudo aquilo que interessa ao cliente, usuário, gerente, ou analista. Um interesse pode ser um requisito funcional, um requisito não funcional, ou um comportamento sistêmico. Contudo a orientação a objetos possui dificuldades no mapeamento de situações da vida real dentro de sua estrutura de classes, principalmente no que diz respeito à separação de interesses dentro de um aplicativo. Segundo (RESENDE, 2005), "separação de interesses diz respeito ao trabalho de organizar requisitos de software em nichos de funcionalidades, interesses, preocupações ou responsabilidades". Alguns destes interesses são implementados em vários módulos de um aplicativo, o que o torna mais difícil de projetar, entender, alterar e evoluir.

Estes interesses são classificados da seguinte forma:

Interesses comuns ("*common-concern*"): São interesses específicos de um módulo ou função.

Interesses ortogonais ("*crosscutting-concern*"): São interesses mapeados a partir de requisitos mutuamente independentes dentro do projeto do aplicativo. Ex: autenticação, monitoração, performance e administração.

O desenvolvimento de um aplicativo é dividido em várias fases: requisitos, projeto, codificação e testes. Dada esta divisão, e a própria evolução da engenharia de software, tornou-se difícil que um mesmo analista conheça com profundidade todas estas fases até a concepção final do aplicativo. Esta evolução tem forçado os vários grupos envolvidos nas fases de desenvolvimento de um aplicativo a trabalharem de maneira independente. É preciso cada vez mais que grupos especializados sejam criados para atender desenvolvimento de soluções tais como: persistência, segurança, distribuição, testes etc.

Contudo, quando estes interesses se fundem dentro de um mesmo conjunto de programas de um aplicativo, a concorrência entre os vários grupos de desenvolvedores é inevitável, por mais claro que estejam definidas as regras de projeto deste aplicativo. Processos de desenvolvimento e manutenção tornam-se penosos de serem conduzidos, pois a cada alteração em um interesse de um determinado grupo, incorre na alteração de vários módulos do aplicativo.

A divisão de interesses por grupos atuando sobre um mesmo conjunto de módulos de um aplicativo gera uma série de conseqüências, segundo Laddad (2005). A figura 10 mostra a disposição dos vários interesses implementados em um aplicativo, as partes em verde dizem respeito às implementações referentes aos interesses de persistência, as partes amarelas dizem respeito as implementações de log, e as partes vermelhas as implementações de segurança:

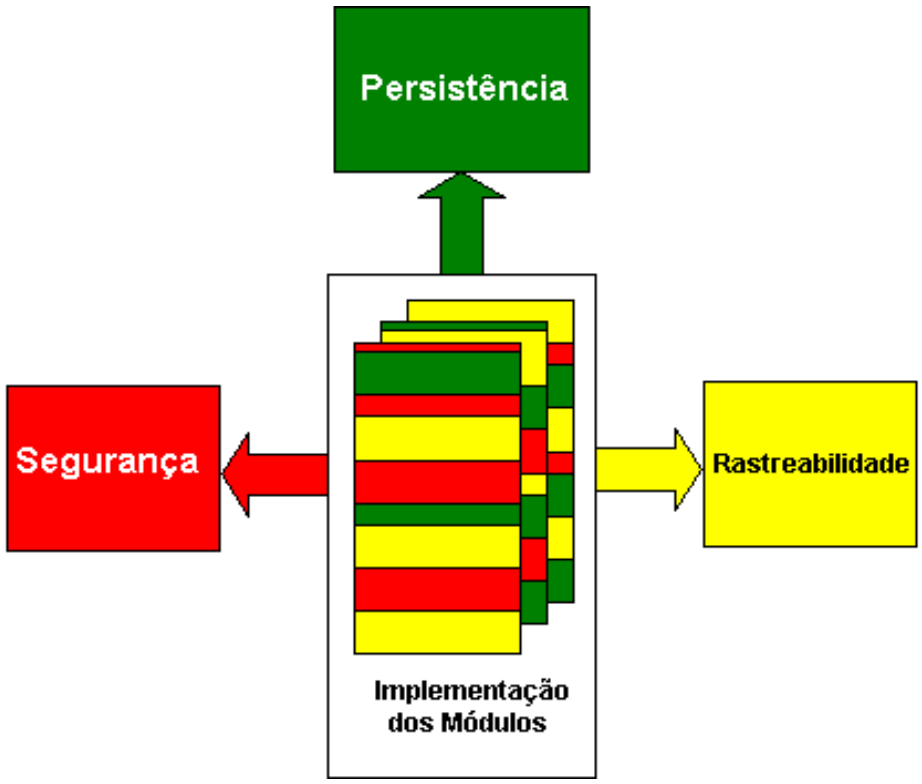

Figura 10. Implementação de módulos como um conjunto de interesses.(LADDAD, 2005)

Ao separar-se os módulos de um aplicativo por grupos de interesse, observa-se uma combinação desordenada que dificulta o entendimento e manutenção deste aplicativo, causando alguns sintomas que são classificados segundo Laddad (2005) como:

Código entrelaçado ("*Code tangling*"): Módulos em um aplicativo podem simultaneamente atender vários requisitos. Esta multiplicidade de requisitos resulta na presença simultânea de referências para cada implementação destes interesses, resultando em um código entrelaçado.

Código disperso ("*Code scattering*"): Uma vez que certos interesses são relacionados a vários módulos do aplicativo, suas implementações são espalhadas pelo aplicativo, o que em caso de alterações em sua implementação pode causar alteração em todos os módulos do aplicativo.

Ainda segundo Laddad (2005), estes sintomas afetam o desenvolvimento e projeto de aplicativos de muitas maneiras que, por conseqüência, afetam também a área de teste de software, uma vez que, quanto mais complexo o aplicativo, maior é a dificuldade de escrever testes para sua validação.

Os problemas gerados pelos sintomas ao separar módulos de um aplicativo por grupo de interesse, são:

Baixo rastreamento ("*Poor traceability*"): Implementação simultânea de vários interesses em um mesmo módulo, ofusca o entendimento da correspondência entre o interesse e sua respectiva implementação, resultando em baixa rastreabilidade.

Baixa produtividade ("*Lower productivity*"): Implementações simultâneas de múltiplos interesses fazem com que o analista desvie-se do foco principal do aplicativo tratando interesses periféricos, diminuindo sua produtividade.

Baixo reuso de código ("*Less code reuse*"): Sob estas circunstâncias com um módulo implementando vários interesses juntos, outros sistemas que requeiram alguma funcionalidade deste módulo talvez não sejam aptos a prontamente utilizá-lo sem antes avaliar todos os interesses contidos nele, reduzindo ainda mais a produtividade.

Baixa qualidade de código ("*Poor code quality*"): código entrelaçado produz código com problemas ocultos, por agrupar muitos interesses em um único módulo. Um ou mais destes interesses podem não receber atenção suficiente.

Dificuldade na evolução ("*More difficult evolution*"): Focalizar demais a solução de um problema e recursos altamente acoplados geralmente produzem um projeto que endereça somente interesses atuais. Novos requerimentos geram grande esforço de entendimento da implementação

deste sistema, e pelo fato da implementação ser muito acoplada acarretará a alteração de um grande número de módulos, aumentando o esforço com testes para garantir que as alterações não gerem novas falhas.

O problema de separação de interesses é melhor representado por meio da figura 11, onde a partir da passagem dos requisitos através de um prisma de identificação de interesses, os módulos são separados paralelamente em sua implementação. Neste ponto pode observar-se que, tal qual os interesses exibidos "*Security"*, "*Persistence"*, "*Logging"* e "*Business Logic"*. Testes de Software podem ser tratados como um interesse de uma área de qualidade de software dentro de um processo de desenvolvimento de um aplicativo, sendo mais um interesse identificado entre os apresentados na figura 11.

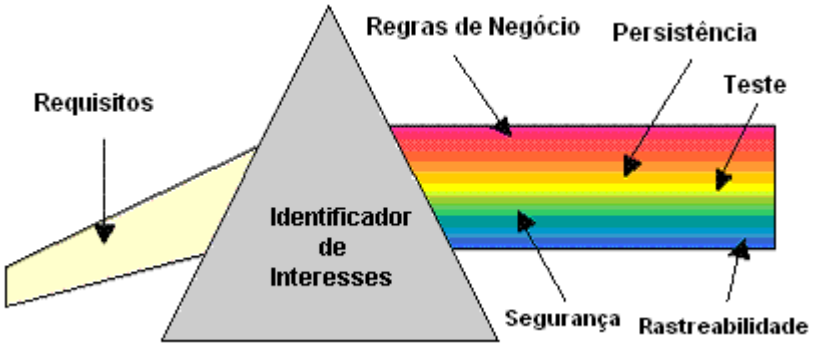

Figura 11. Decomposição de interesses: Analogia do prisma (LADDAD, 2005)

A separação de interesses foi denominada "programação orientada a aspectos" e foi proposta por (KICZALES, 1997). Em seu estudo foram definidos a sintetização dos conceitos de Aspecto como uma nova técnica complementar à técnica Orientado a Objetos.

A relação entre aspectos e classes é representada na figura 12 onde é possível observar a independência que o modelo propõe na separação dos interesses com relação às classes do aplicativo, interesses estes que são sintetizados em classes denominadas aspectos. Os aspectos são formados por "*Join-Point*", "*Pointcut*" e "*Advices*", os pontos de corte("*Pointcut*") contém as definições dos pontos de junção("*Join-Point*") e conselhos("*Advices*"). Os pontos de junção comportam-se como observadores do aplicativo em execução, quando a execução de determinada classe combina com a definição contida em um ponto de junção dizemos que este ponto foi alcançado e que o ponto de corte irá executar o conselho definido neste aspecto. Esses conselhos

podem ser qualquer requisito que atenda ao usuário como rotinas de segurança, rastreabilidade, auditoria entre outras, e podem ocorrer em três momentos da execução da classe original que são: antes, durante e depois. O processo de recomposição da classe com o aspecto que resulta na composição final do aplicativo é denominado “weaving” ou ”*integrating*” (LADDAD, 2005).

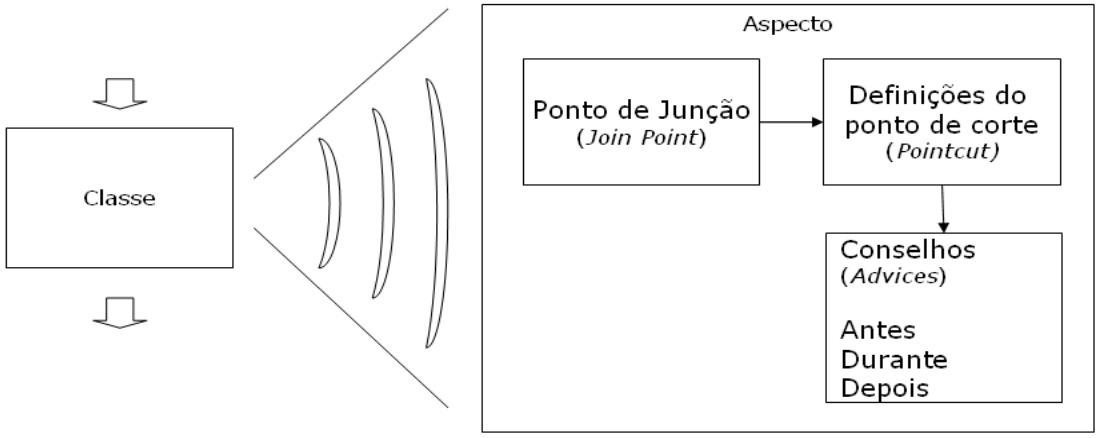

Figura 12. Detalhe de um aspecto segundo Kiczales (1997)

Segundo Resende (2005), outras áreas de engenharia mais antigas e estáveis conseguiram separar interesses a pontos de dividi-los em níveis. A engenharia Civil ora se preocupa com a fundação, ora com a forma dos cômodos, ora com sua localização, ora com seu acabamento etc. A programação orientada a aspectos tem este objetivo. Tentar separar os interesses durante o desenvolvimento de um aplicativo, logo é possível pensar separadamente nos problemas referentes à persistência de dados, modelagem de negócios, segurança, distribuição, auditoria, testes etc.

Ainda segundo Resende (2005) a proposta é desenvolver as partes do sistema sem se preocupar com as demais partes, e a cada interação da integração insere-se um novo nível de interesse sem ser necessário alterar o que já está pronto. Cada parte do sistema que se deseja trabalhar isoladamente pode-se chamar de Aspecto. Com isso, é possível, num determinado momento, trabalhar aspectos de segurança, em outro momento, aspectos de persistência, aspectos de teste de sistema e assim sucessivamente até que todos os aspectos importantes tenham sido tratados.

De acordo com os conceitos de programação orientada a aspectos podemos citar os três principais passos para realizar um desenvolvimento orientado a aspectos:

Decomposição em Aspectos ("*Aspectual decomposition*"): Decompor os requisitos para identificar os interesses tangenciais e os interesses comuns.

Implementação de Interesses ("*Concern Implementation*"): Implementar cada interesse tangencial separadamente.

Recomposição de Aspectos ("*Aspectual recomposition*"): Neste passo um integrador de aspectos, AspectJ (GRADECKI, 2003), determina as regras de recomposição das unidades modularizadas – aspectos. O processo de recomposição, também conhecido como "*weaving*", usa esta informação para compor o sistema final.

A programação orientada a aspectos tem facilitado principalmente a implementação de requisitos não funcionais que demandam soluções que não podem ser contidas em apenas um módulo, mas se espalham ao longo de todo o sistema (LADDAD, 2005).

Apenas a definição de um aspecto de testes não resolve por si só o problema de sua execução. Capturar as interações do aplicativo de maneira menos intrusiva é um ponto bastante desejável, porém para realização dos testes, é necessário que, após a captura, seja possível executar um roteiro que defina quais testes devam ser validados, com as respectivas verificações que possibilite indicar o sucesso ou falha de cada teste.

Este roteiro pode ser definido utilizando um padrão XML ("*Extensible Markup Language*"), uma vez que, o ambiente onde é empregado o método é o ambiente Web, e o mesmo utiliza largamente este padrão em vários aplicativos de diversas finalidades devido a sua flexibilidade e facilidade de manipulação, onde a partir de um conjunto de tags bem definidas é possível adicionar ou remover novos testes alterando-se algumas linhas deste XML.

A abordagem de utilização de XML para definição de roteiros de testes é apresentada no próximo tópico onde são discutidos alguns artigos de certificação de componente de software, que atualmente utilizam padrão XML. A proposta desta dissertação não é realizar certificação de componentes de software, contudo estas propostas possuem um padrão de definição dos testes que podem ser ajustados para realização dos testes de integração em aplicativos desenvolvidos para *Internet*.

4 Teste de integração apoiado por aspectos

4.1 Considerações iniciais

Diversos autores propõem estratégias específicas para elaborar testes em programas orientados a aspectos. Lemos (2005), direciona seu estudo em como realizar testes de aplicações que utilizam AOP com base em fluxo de dados, demonstrando técnicas e filtros para realização dos testes. Propõe ainda um modelo que estimula o software em teste por meio de interações de dados com o objetivo de localizar falhas entre a dependência de classes e os aspectos que entrecortam estas classes. Já Yuewei (2005), Xu et al (2004) e Rocha (2005) definem que é possível utilizar aspectos para testar aplicativos, colocando o teste como mais um interesse ortogonal ao sistema. Yuewei (2005) define um candidato a modelo de falhas baseado na relação entre classes e aspectos, onde é avaliado primeiro os interesses comuns, em seguida são avaliados o comportamento de cada interesse ortogonal e por último o comportamento da relação de *weaving* entre eles. Xu et al (2004) utiliza os recursos da AOP para estender os testes unitários definidos no JUNIT (GAMMA,2000) integrando-o a um oráculo baseado em aspectos. Rocha (2005) desenvolveu um framework para aplicações *cliente-servidor* onde a partir dos recursos da AOP é possível aplicar critérios funcionais e realizar sua posterior análise.

A AOP oferece novas possibilidades à atividade de testes, as propostas apresentadas utilizam seus recursos com idéias que permitem uma maior investigação dos sistemas de software em teste, mesmo quando esses não possuem disponibilidade de seus respectivos códigos fontes. Esta proposta também caminha na mesma direção de Yuewei (2005), Xu et al (2004) e Rocha (2005), ao utilizar os recursos da AOP para apoiar a atividade de testes, restringindo-se aos testes de integração de componentes.

Nas estratégias de testes descritas por Whitaker (2000), quanto ao tipo de teste estrutural ou funcional, a utilização de AOP para definir um método de testes de integração, pode não se adequar à classificação hoje definida, sendo necessário um terceiro tipo que ficaria nos limites entre os tipos estabelecidos atualmente. Isto ocorre, pois ao realizar um teste do tipo funcional o software em teste é considerado uma caixa preta onde os detalhes da implementação não são considerados. Em testes do tipo estrutural a implementação é considerada para estabelecer os requisitos de testes onde se requer a execução de componentes do sistema inclusive caminhos

lógicos deste componente. Ao utilizar os recursos da AOP é possível avaliar o sistema em teste com maior profundidade que o teste do tipo funcional, porém com menor detalhe que o teste do tipo estrutural onde há a presença do código fonte necessário para realizar os testes de caminho lógico. Com isso ao classificarmos um tipo de teste baseado em AOP ele encontra-se entre as duas definições existente.

O objetivo de realizar testes de integração sem isolar os componentes, e comunicando com as várias interfaces que fazem parte da solução, possui vantagem com relação ao fato de reduzir a necessidade de construir *Stubs* para validação dos testes.

Com base nos principais pontos discutidos até aqui envolvendo testes de software e AOP, apresenta-se um *framework* para auxiliar testes de integração dentro de um servidor de aplicativos, onde a partir da execução de testes manuais ou automáticos, é possível a validação de alguns critérios de testes apresentados em seções anteriores desta dissertação.

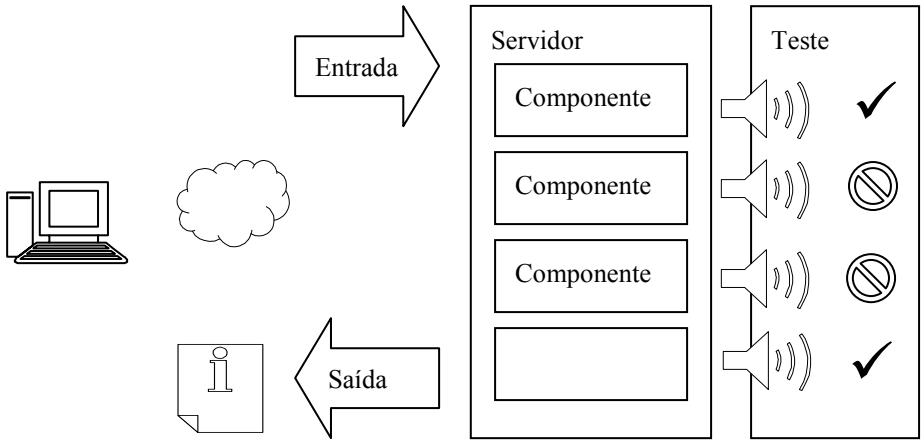

Figura 13. Execução de um teste de integração em uma funcionalidade apoiado por Aspectos

Ao considerar atividade de teste como um interesse ortogonal ao sistema é possível utilizar os recursos da AOP no sentido de propor um *framework* capaz de verificar de maneira não intrusiva a execução completa de uma funcionalidade em aplicativos para *Internet*. A figura 13 ilustra a execução de uma funcionalidade onde a partir da inclusão de um módulo de teste apoiado por aspectos é possível verificar todos os componentes até o retorno do resultado da requisição. Neste modelo não existe a intrusão no código original do aplicativo pela própria natureza do AOP, por isso ao realizar a homologação do aplicativo não se faz necessário novas

intervenções por parte dos analistas para ajustes com relação às referências de testes antes de enviá-lo ao ambiente produtivo.

Com a utilização de AOP é possível apontar com precisão o ponto onde a funcionalidade apresenta uma falha em testes com estratégias não-incrementais, pois é possível interagir com todos os componentes dentro da execução da funcionalidade atribuindo validações específicas a cada um destes componentes. Apesar da possibilidade de interagir com todos os componentes individualmente, a definição de qual componente verificar dentro de uma funcionalidade fica a critério do analista de teste com base na definição do roteiro de testes, a idéia de adotar um roteiro de testes com base no padrão XML permite definir de forma simples qual o nível de profundidade deseja-se testar em um aplicativo.

Uma limitação deste modelo diz respeito aos interesses definidos com base em aspectos que diferem da área de testes, ou seja, a área de testes é apenas um dos interesses ortogonais que entrecortam o aplicativo, caso existam outros interesses implementados dentro da solução do aplicativo alguns cuidados devem ser observados para que os testes destes outros interesses possam ser realizados da mesma maneira que o restante do aplicativo. Segundo Resende (2005) ao definir um aspecto é possível implementar em um único método o ponto de corte ("*pointcut*") e os conselhos ("*advices*") que serão executados no momento que o ponto de junção ("*joinpoint*") é alcançado, acarretando que os conselhos não serão executados de forma independente como um componente convencional o que impede sua verificação em atividades de testes apoiado por aspectos. Não foram encontradas na literatura propostas sobre a melhor forma de implementar aspectos, portanto como não é possível ter controle sobre os demais aspectos envolvidos dentro de um aplicativo, isto é uma limitação do modelo de testes proposto.

4.2 Cenário de uso

A evolução da *Internet* aumentou a complexidade da atividade de testes, pois o que no início era apenas exibição de conteúdo estático tornou-se um conjunto de aplicações dinâmicas distribuídas em várias camadas que interagem com um grande número de componentes e interfaces (MALDONADO et al., 2007).

Como visto no capítulo 3.4 existem grandes desafios ao realizar-se testes de integração em aplicativos desenvolvidos para *Internet*, as abordagens propostas baseadas em modelos de "*Capture/Playback*" necessitam de alternativas complementares para efetividade dos resultados

obtidos (DUSTIN, 2003).

Em aplicativos desenvolvidos para *Internet,* segundo sua arquitetura distribuída, para cada requisição de um usuário existe uma série de interações realizadas no servidor, o conjunto destas interações gera um caso de teste a ser validado pelo analista de teste. (NGUYEN et al. ,2003)

O *framework* proposto não se limita a programas desenvolvidos para a *Internet*, mas atende de forma não intrusiva a execução destes casos de teste através da execução das funcionalidades do aplicativo pela interface do usuário, verificando cada componente previamente definido em um roteiro de teste, ele elimina a necessidade de alteração do aplicativo original para complementar as verificações realizadas no servidor a partir da requisição do usuário na camada we*b*.

4.3 Arquitetura

A figura 14 mostra a arquitetura do *framework* de testes proposto. Nele constam as principais camadas envolvidas na execução dos casos de testes e a forma como o mesmo integra-se ao aplicativo a ser testado. Pode-se observar que não há interface direta do usuário e/ou máquina com o *framework*, pois para todo o teste a ser validado é utilizada a própria interface do aplicativo em teste. As propriedades da AOP permitem uma relação não intrusiva entre o aplicativo a ser testado e o *framework*, possibilitando uma total segregação de módulos, onde o *framework* pode ser incluído ou excluído de acordo com a fase do projeto sem alterações no aplicativo original.

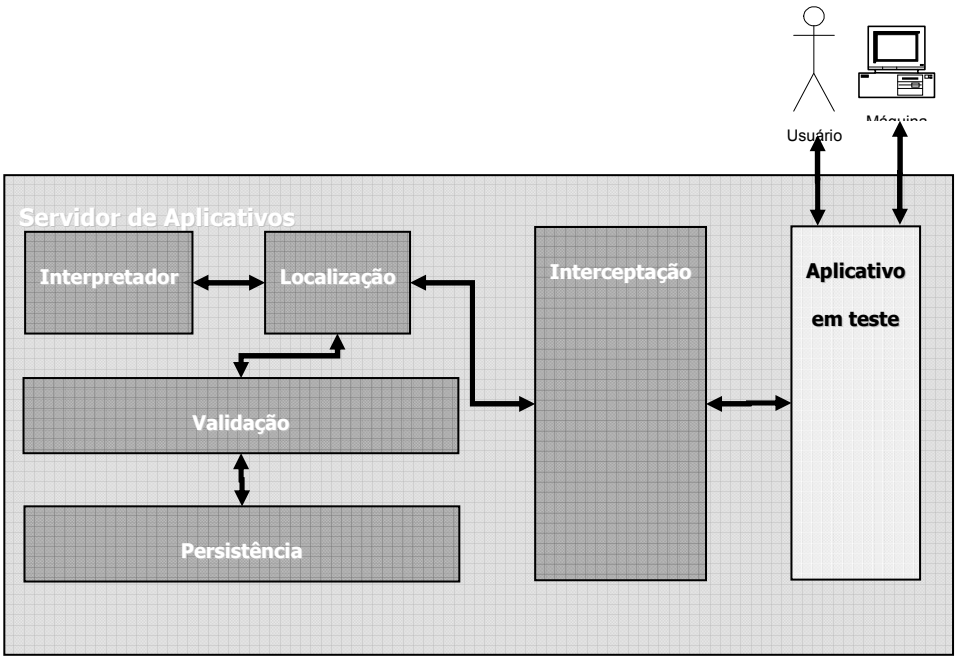

Figura 14. Arquitetura do framework de testes

Com a execução dos testes na camada de aplicativos a camada de interceptação é responsável por receber o processo de "*weaving"* gerado pelo aplicativo em teste, é nesta camada onde concentram-se os recursos da AOP para interceptação dos métodos do aplicativo em teste.

Através do evento *around* disponível na implementação de um aspecto, o *framework* verifica a assinatura da classe executada tanto no nível intermétodo quanto no nível intramétodo e realiza a chamada ao método *TestFinder* para localizar os casos de testes disponíveis para o método interceptado. Ao acionar a camada de localização a mesma percorre uma lista de casos de teste previamente armazenado em memória e retorna uma lista de testes definidos para o método em execução. A localização é realizada através das entradas esperadas no roteiro de teste: caso o método em execução possuir parâmetros de entrada igual aos parâmetros definidos no roteiro a camada de localização então retorna aquele caso de teste como um teste a ser avaliado para o método em execução.

A camada de interpretação realiza a leitura dos casos de testes previamente formatados em XML e organiza-os em memória no formato de árvore (DOM) para otimizar a localização dos casos de teste. Caso um teste seja localizado para aquela funcionalidade, ele então é checado pela camada de validação. Esta camada compara os valores de saída com os valores definidos no

roteiro de teste de acordo com a assinatura do método em execução e seu resultado é enviado a camada de persistência onde é armazenado se o teste ocorreu com falhas ou sem falhas.

O roteiro de testes utilizado pelo *framework* é gerado em formato XML que é carregado no momento da inicialização do servidor de aplicativos, este formato foi escolhido pela fácil utilização e manutenção na montagem dos casos de testes.

O analista de teste pode com o auxílio deste roteiro, definir a seqüência de execução de uma dada funcionalidade, validando a cada passo os valores esperados e resultados obtidos a partir dos casos de testes previamente estabelecidos. A seqüência e profundidade dos testes podem ser definidas de acordo com a necessidade dos analistas de teste, onde a partir da visão dinâmica de uma funcionalidade é possível definir quais classes e métodos são avaliados dentro ou fora de um contexto.

4.4 Critério de teste

Nesta seção é definido o critério de teste que o *framework* utiliza para validar se um teste ocorreu com falhas ou sem falhas. O critério está posicionado entre o modelo funcional e o modelo estrutural, pois apesar de definir os casos de teste a partir da funcionalidade é capaz de realizar um teste estrutural ao utilizar os benefícios da AOP para entrecortar todos os pontos a serem avaliados dentro do software em teste. Com isso ao executarmos uma funcionalidade é possível avaliar todos os pontos de uma seqüência tanto no nível intermétodo quanto intraclasse segundo as definições de Harrold et al. (1994).

Um caso de teste é composto por três partes: entradas, saídas e ordem de execução (COPELAND,2006). Ao aplicar um teste de integração temos que avaliar as interfaces dos componentes que formam a funcionalidade, com isso teríamos que para cada entrada e saída informada em um caso de teste, tem-se uma entrada que é o parâmetro da interface em execução e uma saída obtida que é o resultado da execução do componente.

A partir daí, é representado pela letra “E” a entrada de um caso de teste e pela letra “S” sua respectiva saída. Para os dados da execução do componente tem-se a letra “P” representando o parâmetro de entrada da execução do componente e a letra “R” como resultado obtido dessa execução. Com isso, é possível estabelecer uma relação entre os valores esperados e os resultados obtidos para determinar se um caso de teste encontrou uma falha ou não.

Em um caso de teste Ti temos que dado uma entrada Ei o caso de teste não encontra falha se o resultado Ri for igual a saída esperada Si. Ao confrontarmos este caso de teste com a execução do componente tem-se que:

Dados **E1** e **S1**

T1=(Se **E1** = **P1** (É caso de teste), logo se **S1** = **R1)**(Sem Falha)

T1=(Se **E1** = **P1** (É caso de teste), logo se **S1** <> **R1)**(Com Falha)

T1=(Se **E1** <> **P1** (Não é caso de teste))

Em testes funcionais também conhecidos como teste de caixa preta não é possível capturar o comportamento de execução dos componentes que formam a funcionalidade, apenas é possível avaliar a entrada do usuário e o resultado final esperado, com os benefícios do AOP é permitido avaliar todas as interfaces que compõe a funcionalidade, por isso o caso de teste não é apenas a avaliação do input do usuário e sua respectiva saída, mas também as ***n*** interações de entradas e saídas entre os componentes necessárias para retornar a resposta final. Ao utilizar AOP é possível entrecortar todos esses níveis internos de acordo com a necessidade do analista de teste, por isso ao avaliarmos a execução de um caso de teste de uma funcionalidade temos:

Dados (**E1** e **S1) + (E11** e **S11) + (E12** e **S12) + ...(E1n** e **S1n)**

Caso de teste "Sem falha"

T1=(Se **E1**=**P1**, logo se **S1**=**R1**(sem falha)) **+** (Se **E11**=**P11**, logo se **S11**=**R11**(sem falha)) +

(Se **E12**=**P12**, logo se **S12**=**R12**(sem falha)) +...(Se **E1n**=**P1n**, logo se **S1n**=**R1n**(sem falha))

Caso de teste "Com Falha"

T1=(Se **E1**=**P1**, logo se **S1**=**R1**(sem falha)) **+** (Se **E11**=**P11**, logo se **S11**=**R11**(sem falha)) +

(Se **E12**=**P12**, logo se **S12**<>**R12**(com falha)) +

... (Se **E1n**=**P1n**, logo se **S1n**<>**R1n**(com falha))

Neste caso o teste só será considerado sem falhas, se a soma da validação das interfaces internas não econtrar falhas, e caso encontre uma falha em alguma dessas interfaces o teste é considerado finalizado com falhas.

A vantagem de entrecortar todas as interfaces em uma seqüência de execução é que em caso de falha é possível dizer com precisão o ponto exato onde o erro ocorreu, recurso muito útil em testes que utilizam a estratégia não incremental(big-bang).

A figura 15 exibe um exemplo de um caso de teste utilizando o padrão XML estabelecido pelo framework.

```
1<?xml version="1.0" encoding="iso-8859-1"?>
 2<ScriptTest Name="XX">
 3  <TestSuite Name="Money">
 4    <Test Name="MoneyMethods" ClassName="junit.samples.money.Money">
 5      <Method Name="<init>">
 6         <Input Name="amount" DataType="int" value="12"/>
 7         <Input Name="currency" DataType="String" value="CHF"/>
 8         <Output DataType="String" value="[12 CHF]"/>
 9      </Method>
10    </Test>
11  </TestSuite>
12</ScriptTest>
```

Figura 15. Exemplo de formato de um roteiro de testes.

No exemplo a tag *Method* representa cada uma das interfaces a serem avaliadas pelo critério de teste, onde Ei é representado pelo conjunto de tags *Input* e Si representado pelo conjunto de tags *Output* que definem os parâmetros de entrada e saída esperados ao executar um caso de teste.

Ainda no sentido de exibir a atuação do critério de teste dentro da execução de uma funcionalidade a figura 16 mostra um diagrama de seqüência e os pontos que podem ser avaliados para determinar se a execução do caso de teste de uma funcionalidade foi realizada com ou sem falhas. Ao realizar este tipo de teste o analista de teste tem liberdade para determinar quais pontos internos da funcionalidade serão avaliados de acordo com a estratégia adotada para realização dos testes.

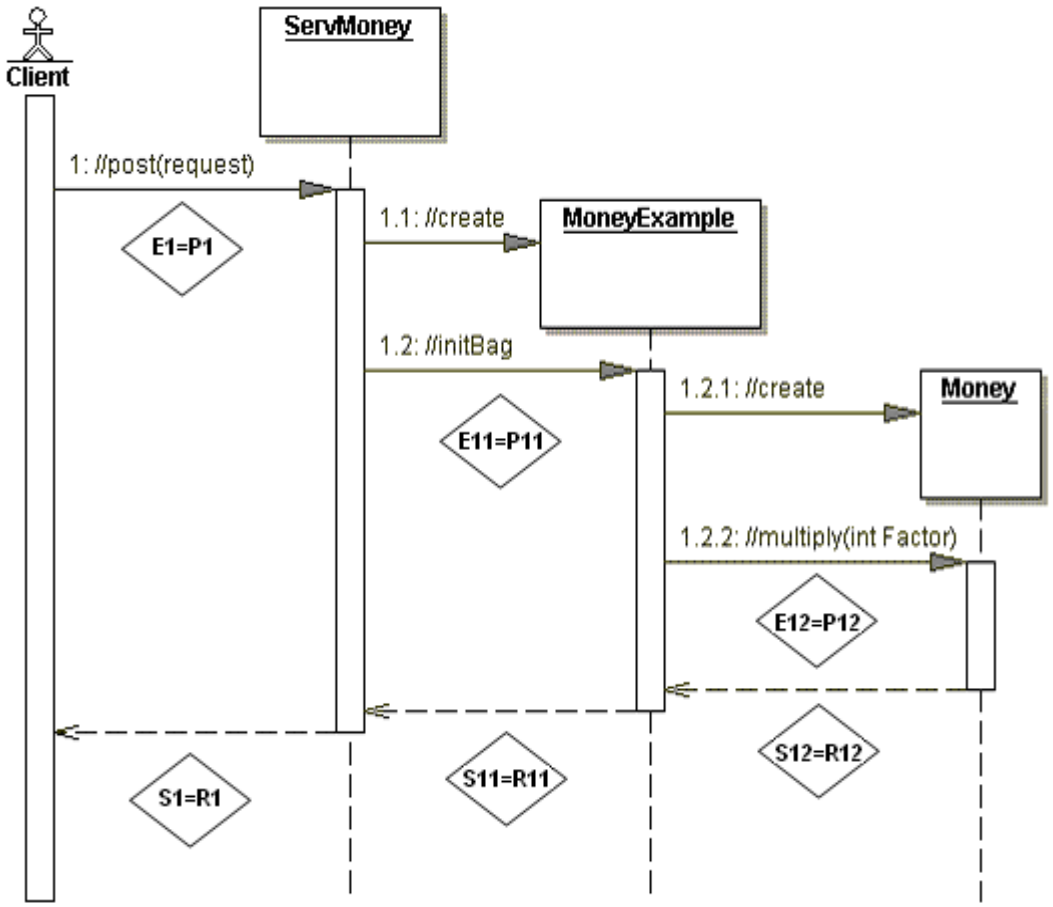

Figura 16. Exemplo do critério de teste aplicado sobre um diagrama de seqüência.

Utilizando o critério definido acima o *framework* é capaz de realizar a validação tanto no nível intermétodo, caso exista na funcionalidade interação entre métodos da mesma classe quanto intraclasse que diz respeito à seqüência de execução da funcionalidade, onde para cada ponto entrecortado é possível avaliar se um caso de teste foi executado com ou sem falhas de acordo com roteiro definido.

4.5 Uso do Framework

A construção do framework foi utilizada para comprovar a idéia de considerar a atividade de testes como mais um interesse ortogonal ao sistema e utilizar os recursos da AOP para propor uma nova abordagem em como realizar teste de integração de componentes. Para tanto o framework é composto pelos módulos: Interceptação, Localização, Interpretador, Validação e Persistência. O módulo de interceptação é responsável pela comunicação com as classes dos aplicativos a serem testados através do processo de "*weaving*" da AOP. Este processo de utilizar aspectos para interceptar a execução de uma classe e avaliar seus resultados foi discutido em

(MONK et al., 2002), onde a partir dos recursos da AOP é possível criar objetos mock virtuais para auxiliar os testes de uma determinada classe.

O modelo proposto pelo framework é baseado em ferramentas de "*Capture/Playback*" onde a partir do exercício do sistema a ser testado, é gerado um roteiro em padrão XML constando todas a interfaces que compõem as funcionalidades, em seguida é informado os valores dos casos de teste definidos pelos analistas de teste. Com o roteiro de teste definido, o sistema é novamente exercitado para validar as informações esperadas com os valores obtidos de acordo com o critério de teste definido nas seções anteriores.

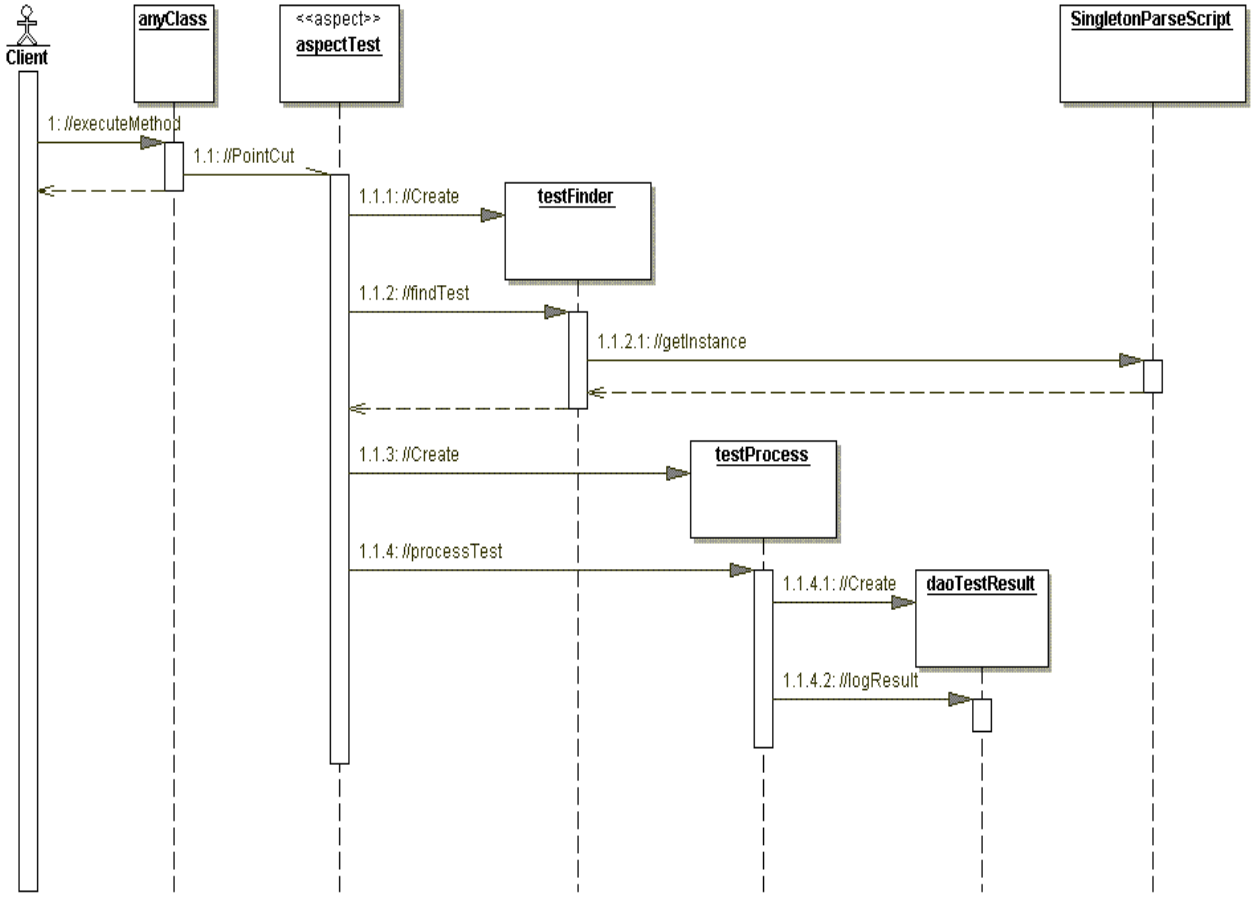

Figura 17.Diagrama de seqüência do framework de testes.

A figura 17 mostra o comportamento do framework quando uma classe da funcionalidade a ser testada gera um *weaving* para a classe AspectTest, o aspecto então instancia a classe TestFinder responsável por localizar os testes para determinado componente que recupera o roteiro de testes previamente carregado por uma única instância através da classe SingletonParseScript. Caso existam testes para classe que gerou o weaving os mesmos são avaliados pela classe TestProcess de acordo com os critérios estabelecidos e seus resultados armazenados pela classe DAOTestResult. Note que para todo processo de weaving gerado pelas classes em teste é gerada a seqüência do framework de testes, porém este não interfere na execução da classe original, apenas verificando os critérios estabelecidos no roteiro de testes para as classes do aplicativo.

No final da execução de cada funcionalidade é possível determinar com precisão quais funcionalidades apresentaram algum tipo de falha durante a utilização do framework o que acelera o processo de correção de possíveis defeitos do aplicativo em teste.

5 Experimento de uso da técnica

5.1 Introdução

Para demonstrar a aplicação do framework é utilizado a classe Money utilizada para apresentar o JUNIT (GAMMA,2000) onde é utilizado o método *multiply*(*int factor*). Este método prevê a multiplicação do montante atual da classe Money por um fator de reajuste informado pelo aplicativo resultando na classe Money atualizada. Para isso é aplicado o caso de teste que utiliza o critério de caso de uso onde são descritos o cenário principal e secundário da funcionalidade e seus respectivos resultados esperados. Este caso de uso prevê a criação de uma carteira de dinheiro a partir de um tipo de moeda e valor montante inicial informado pelo usuário.

Nome do Caso de Uso	Criação de um carteira de aplicação	
Responsável / Data	Nononono - 28/11/2008	
Recursos Necessários	Um servidor de aplicativos Um navegador padrão	
Procedimentos	O usuário deverá acessar a URL do sistema e informar o tipo da moeda e o montante inicial em seguida pressionar o botão para executar a operação	
Cenário Principal (Construir uma carteira de aplicação e multiplicar por um fator)		
Ações Recebidas	**Ações Executadas**	**Aprovado S/N**
Informar a moeda a partir da qual será criada a carteira Informar o valor montante inicial da carteira Informar o valor do fator multiplicativo	Será criada a carteira com os valores informados Será retornado a carteira com tipo da moeda e seu montante inicial atualizado	
Resultados, conclusões e sugestões		
Cenário Secundário (Construir uma carteira e aplicar um taxa administrativa sobre a carteira)		
Ações Recebidas	**Ações Executadas**	**Aprovado S/N**
Informar a moeda a partir da qual será criada a carteira Informar o valor montante inicial da carteira Informar o valor do fator multiplicativo	Será criada a carteira com os valores informados O montante da carteira será multiplicado pela taxa informada pelo usuário Será debitado do montante final o valor da taxa administrativa Será retornado a carteira com os valores atualizados	
Resultados, conclusões e sugestões		

Tabela 4. Caso de uso do exemplo proposto

5.1.1 Cenário principal

A execução da funcionalidade descrita no caso de uso é através de um Servlet que recebe como parâmetro o tipo de moeda, montante inicial e fator multiplicativo, o Servlet com este parâmetros cria o objeto da classe MoneyExample que inicia a carteira. Ao iniciar a carteira a classe MoneyExample cria uma instância da classe Money com os parâmetros tipo de moeda e montante inicial, após este passo multiplica o montante inicial pelo fator multiplicativo e retorna a carteira com o valor atualizado. Neste cenário principal é considerado que não existe nenhum aspecto além do AspectTest, isto ocorre porque a demonstração do framework é divida em duas

partes, a primeira com uma funcionalidade executando seu fluxo normal, e a segunda quando é incluído um aspecto na funcionalidade para calcular a taxa administrativa da multiplicação do montante no método multiply.

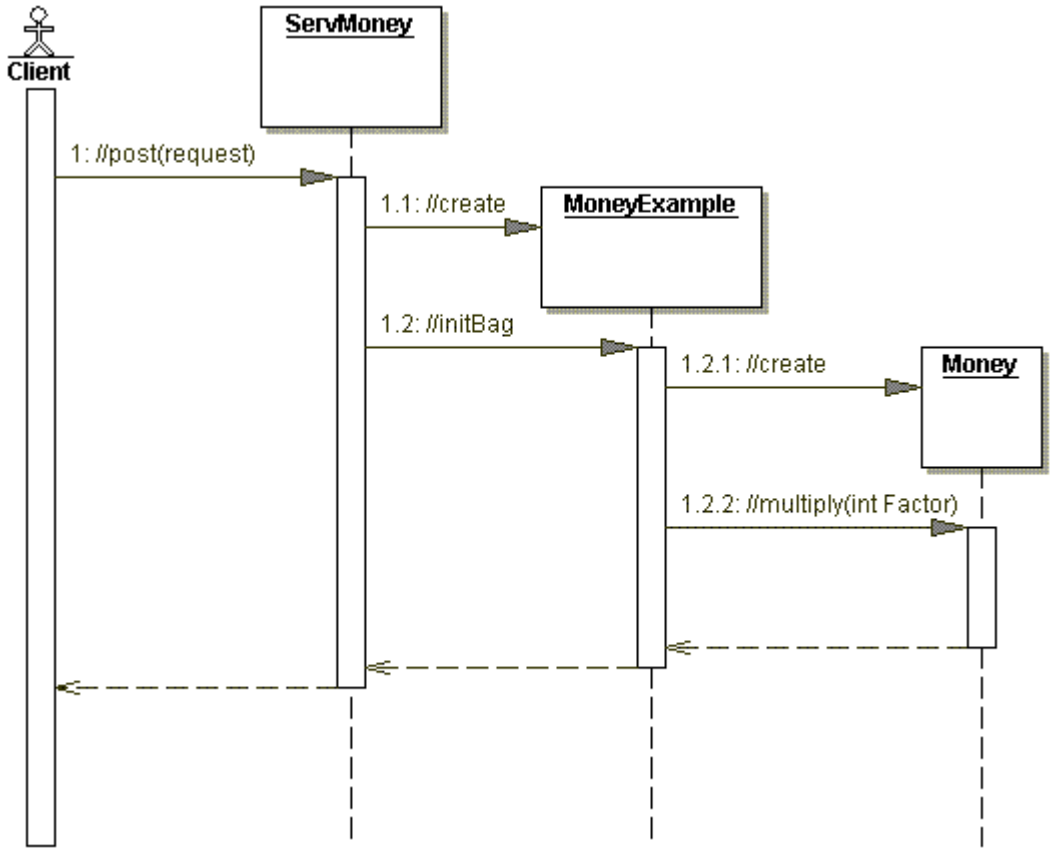

Figura 18. Diagrama de seqüência da classe money

Ao executar o cenário principal a cada passo da seqüência de execução o "*framework"* de testes recebe o processo de "weaving" gerado pelas classes, com isso ele avalia o roteiro de testes definido para este cenário e localiza os testes que unifiquem com as interfaces em execução na seqüência da funcionalidade.

Com isso dado o roteiro definido na figura 20 os seguintes testes são executados:

Teste de criação do objeto Money

Parâmetros

Tipo de moeda	=	CHF
Montante inicial	=	12

Resultados esperados

Objeto Money	=	[12 CHF]

Ordem de execução 1

Teste de multiplicação da carteira do objeto Money

Parâmetros

Fator	=	2

Resultados esperados

Objeto Money	=	[24 CHF]

Ordem de execução 2

```
1<?xml version="1.0" encoding="iso-8859-1"?>
2<ScriptTest Name="XX">
3  <TestSuite Name="Money">
4    <Test Name="MoneyMethods" ClassName="junit.samples.money.Money">
5      <Method Name="<init>">
6        <Input Name="amount" DataType="int" value="12"/>
7        <Input Name="currency" DataType="String" value="CHF"/>
8        <Output DataType="String" value="[12 CHF]"/>
9      </Method>
10     <Method Name="multiply">
11       <Input Name="factor" DataType="int" value="2"/>
12       <Output DataType="String" value="[24 CHF]"/>
13     </Method>
14   </Test>
15 </TestSuite>
```

Figura 19. Roteiro de teste do cenário principal

Substituindo-se os valores do roteiro de teste do cenário principal da figura 20 no critério de teste tem-se o seguinte esquema:

E1 = (12,"CHF") **S1** = ("12 CHF")

E11 = (2) **S11** = ("24 CHF")

Logo, para o caso de teste ser avaliado, suponha que o framework capturou os seguintes valores para as variáveis P e R:

P1 = (12,”CHF”) **R1** = (“12 CHF”)

P11 = (2) **R11** = (“24 CHF”)

Aplicando o critério teríamos a seguinte equação:

T1=(Se **E1**=**P1**, logo se **S1**=**R1**) + (Se **E11**=**P11**, logo se **S11**=**R11**)

Substituindo os valores esperados e os valores obtidos pelo framework, tem-se:

E1=(12,”CHF”), **S1**=(“12 CHF”), **P1**=(12,”CHF”), **R1**=(“12 CHF”)

E11=(2), **S11**=(“24 CHF”), **P11**=(2), **R11**=(“24 CHF”)

Logo **T1** é:

T1= ((12,”CHF”)=(12,”CHF”)(true), logo se (“12 CHF”)=(“12 CHF”)(true)) (sem falha) +

((2)=(2) (true), logo se (“24 CHF”)=(“24 CHF”)(true)) (sem falha))

Com isso ao analisar o resultado das validações dos dois testes realizados no roteiro, pode-se afirmar se o cenário principal é valido ou não para a funcionalidade de “Criar uma carteira de aplicação”.

O exemplo mostra como foi definida uma simples seqüência de teste para avaliar uma funcionalidade pelo uso do “*framework*”, o XML adotado como padrão para o roteiro de testes é bastante flexível, porém para soluções mais complexas pode ser uma tarefa trabalhosa sua geração, por isso estuda-se a possibilidade de geração automática deste roteiro baseado em uma execução para capturar a seqüência de comportamento do usuário operando as funcionalidades, muito comum em modelos do tipo “*Capture/Playback*”(DUSTIN, 2003).

5.1.2 Cenário secundário

No cenário secundário é apresentada a mesma seqüência do cenário principal, com a adição da implementação de um aspecto que irá calcular uma taxa toda vez que a carteira de dinheiro for modificada pelo método multiply. Este é o cenário secundário do nosso caso de uso que é utilizado para demonstrar que o framework é capaz de executar testes em aplicativos que também

utilizem AOP como solução para seus requisitos de negócio tanto funcionais como não-funcionais.

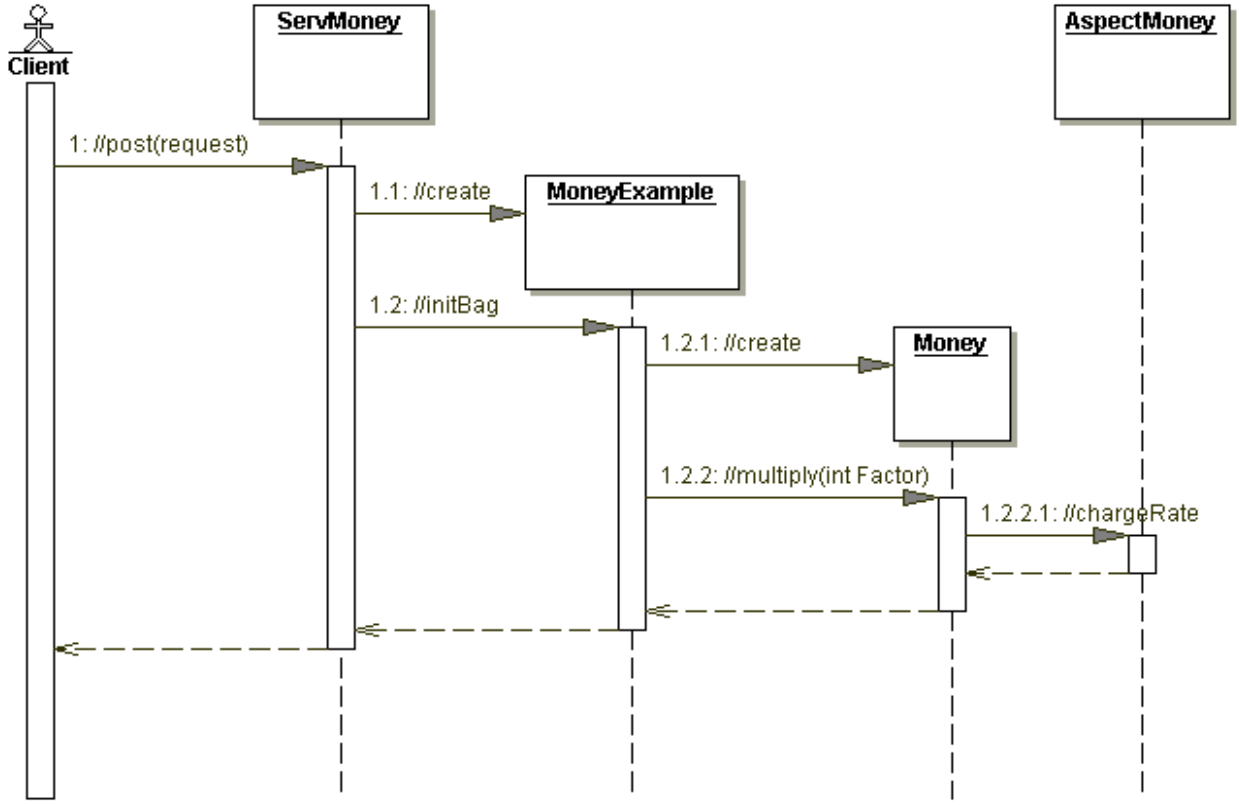

Figura 20. Diagrama de seqüência da classe money com aspecto

A seguir temos os testes que são executados no cenário secundário a partir do roteiro da figura 22.

Teste de criação do objeto Money

Parâmetros

Tipo de moeda	=	CHF
Montante inicial	=	12

Resultados esperados

Objeto Money	=	[12 CHF]

Ordem de execução 1

Teste de multiplicação da carteira do objeto Money

Parâmetros

Fator = 2

Resultados esperados

Objeto Money = [24 CHF]

Ordem de execução 2

Teste de cobrança da taxa administrativa do objeto AspectMoney

Parâmetros

Montante da carteira = 24

Resultados esperados

Objeto Money = [22 CHF]

Ordem de execução 1

O cenário secundário mostra que mesmo quando a aplicação em teste utiliza AOP para implementar seus requisitos de negócio é possível através do *framework* realizar os testes de integração do aplicativo salvo as limitações citadas por Resende(2005). Entretanto a sequência de execução pode variar de acordo com o tipo de evento do conselho(*advice*) for implementado(antes, durante, depois).

```
1<?xml version="1.0" encoding="iso-8859-1"?>
2<ScriptTest Name="XX">
3  <TestSuite Name="Money">
4    <Test Name="MoneyMethods" ClassName="junit.samples.money.Money">
5      <Method Name="<init>">
6          <Input Name="amount" DataType="int" value="12"/>
7          <Input Name="currency" DataType="String" value="CHF"/>
8          <Output DataType="String" value="[12 CHF]"/>
9      </Method>
10     <Method Name="multiply">
11         <Input Name="factor" DataType="int" value="2"/>
12         <Output DataType="String" value="[24 CHF]"/>
13     </Method>
14   </Test>
15   <Test Name="AspectMoneyMethods" ClassName="br.com.w3.mestrado.exemplo.AspectMultiply">
16     <Method Name="calculaCPMF">
17         <Input Name="result" DataType="int" value="24"/>
18         <Output DataType="int" value="22"/>
19     </Method>
20   </Test>
21 </TestSuite>
22</ScriptTest>
```

Figura 21. Roteiro de teste do cenário secundário

5.1.2.1 Exemplo de Teste (Sem Falha)

Analogamente ao cenário principal substituindo-se os valores do roteiro de teste do cenário secundário da figura 21 no critério de teste tem-se o seguinte esquema:

E1 = (12,”CHF”) **S1** = (“12 CHF”)

E11 = (2) **S11** = (“24 CHF”)

E12 = (24) **S12** = (22)

Logo, para o caso de teste ser avaliado, suponha que o framework capturou os seguintes valores para as variáveis P e R:

P1 = (12,”CHF”) **R1** = (“12 CHF”)

P11 = (2) **R11** = (“24 CHF”)

P12 = (24) **R12** = (22)

Aplicando o critério teríamos a seguinte equação:

T1 = (Se **E1**=**P1**, logo se **S1**=**R1**) + (Se **E11**=**P11**, logo se **S11**=**R11**) + (Se **E12**=**P12**, logo se **S12**=**R12**)

Substituindo os valores esperados e os valores obtidos pelo *framework*, tem-se:

E1=(12,”CHF”), **S1**=(“12 CHF”), **P1**=(12,”CHF”), **R1**=(“12 CHF”)

E11=(2), **S11**=(“24 CHF”), **P11**=(2), **R11**=(“24 CHF”)

E12=(24), **S12**=(22), **P12**=(24), **R12**=(24)

Logo **T1** é:

T1= ((12,”CHF”)=(12,”CHF”)(true), logo se (“12 CHF”)=(“12 CHF”)(true)) (sem falha) +
((2)=(2) (true), logo se (“24 CHF”)=(“24 CHF”)(true)) (sem falha) +
((24)=(24) (true), logo se (22)=(22)(true)) (sem falha))

Substituindo os valores exibidos na equação pelo resultado booleano das interações tem-se que o teste T1 obteve sucesso em sua validação do caso de teste do cenário secundário.

5.1.2.2 Exemplo de Teste (Com Falha)

Na seção 5.1.2.1 o caso T1 é avaliado como um caso de teste de sucesso para o cenário secundário proposto, agora vamos supor que o *framework* capturou para o resultado R12 o valor 23, ao aplicar o critério de teste com base no novo valor da variável R12 referente ao resultado do cálculo da taxa administrativa, tem-se:

Substituindo os valores esperados e os valores obtidos pelo *framework*, tem-se:

E1=(12,"CHF"), **S1**=("12 CHF"), **P1**=(12,"CHF"), **R1**=("12 CHF")

E11=(2), **S11**=("24 CHF"), **P11**=(2), **R11**=("24 CHF")

E12=(24), **S12**=(22), **P12**=(24), **R12**=(**23**)

Logo **T1** é:

T1= ((12,"CHF")=(12,"CHF")(true), logo se ("12 CHF")=("12 CHF")(true)) (sem falha) +
((2)=(2) (true), logo se ("24 CHF")=("24 CHF")(true)) (sem falha) +
((24)=(24) (true), logo se (22)=(23)(**false**)) (**com falha**))

Como o resultado de T1 depende de uma operação "and" entre as interações de teste das interfaces dos componentes que compõe a funcionalidade dizemos que o teste foi executado com falhas em sua validação.

5.2 Análise dos resultados

Na seção 5.1 deste capítulo apresenta-se um exemplo de uso do *framework* em dois cenários propostos pelo caso de teste do tipo caso de uso, no primeiro cenário é definido a criação de uma carteira de dinheiro, cujo montante inicial é multiplicado por um fator informado. O roteiro de teste é definido na figura 20 para demonstrar a composição das tags XML utilizada para validar a funcionalidade, com base no roteiro é aplicado o critério de teste definido neste trabalho para validar se o caso teste obteve sucesso ou falha. Em seguida é discutido o cenário secundário onde é acrescido o cálculo da taxa administrativa da carteira, mostrando a flexibilidade do roteiro de testes conforme exibido na figura 22, como também a capacidade do *framework* em testar requisitos de negócio implementados com o uso da AOP

Como resultado deste trabalho tem-se a implementação de um protótipo com base no modelo proposto onde é possível analisar alguns pontos interessantes no *framework* de testes. Foi observado no protótipo de testes que em casos onde o roteiro de testes tem métodos definidos sem parâmetros de entrada e sem contexto, estes testes são validados para toda execução da funcionalidade do aplicativo, gerando um resultado não significativo. Pois independente de quais valores forem executados pela aplicação o teste poderá ter sucesso ou falha, o que não implica na descoberta de uma nova falha do aplicativo testado.

No caso de aplicativos que implementam aspectos de seus requisitos, nota-se que ao definir os conselhos("*advices*") e estes alteram o valor original do método entrecortado isto pode resultar em falha no cenário de teste. Isto ocorre, pois os conselhos ao serem implementados nas opções antes e durante manipulando o resultado esperado afeta o teste descrito para o componente, pois o resultado esperado sofreu intervenção de um aspecto que alterou o resultado final. Uma abordagem proposta por (MASSICOTTE et al., 2005), é realizar uma estratégia incremental dos aspectos definidos para o aplicativo, onde a partir da inclusão da validação do aplicativo original sem a presença dos aspectos, acrescenta-se um aspecto por vez repetindo o roteiro de teste e verificando o resultado. A única deficiência desta proposta diz respeito aos aspectos que compõe o resultado esperado de um dado componente o que acarretaria na construção de "*Stubs*" para substituição integral dos aspectos implementados.

Nesta seção foi possível extrair algumas situações interessantes a partir do protótipo, porém uma análise mais aprofundada se faz necessário em futuros trabalhos.

6 Conclusões, Limitações e Trabalhos Futuros

6.1 Conclusões

Considerando teste como um interesse ortogonal ao aplicativo é possível utilizar a modularização de interesses oferecido pela técnica AOP na realização dos testes de integração. A abordagem AOP para realizar tais testes possibilita a separação dos testes com relação a aplicação a ser testada. Com isso, é possível eliminar a possibilidade de inclusão de novas falhas em manutenções para desativação destas intervenções do código original.

O método mostrou-se efetivo ao eliminar a inclusão de "*testability hooks*" e "*logs*" no aplicativo original como complemento aos aplicativos de teste baseados no modelo "*capture/playback*".

Com relação aos testes de integração o modelo AOP permite identificar com maior precisão os erros relacionados à abordagem não incremental ("*Big-bang*"), uma vez que consegue interceptar todas as execuções dos componentes ocorridas no aplicativo em teste. Isto é possível, pois toda a seqüência de execução que compõe a funcionalidade pode ser validada na definição do roteiro de testes, o que em caso de execução de caso de teste com falhas é possível o *framework* apontar exatamente qual interface apresentou a falha.

Não é necessário que a aplicação em teste seja convertida para o modelo AOP para que seja possível aplicar o *framework*, os recursos disponíveis da AOP estão inseridos dentro do *framework* de testes e interagem com o aplicativo em teste de maneira não intrusiva.

Quanto a classificação do tipo de teste o uso de AOP esta posicionado entre estrutural (Caixa Branca) e funcional (Caixa Preta). Isto ocorre, pois os recurso disponíveis pelo uso de AOP vão além do teste funcional permitindo um nível de introspecção nas interfaces dos componentes que compõe a funcionalidade mesmo sem a presença dos fontes do software.

Todo trabalho é desenvolvido orientado para aplicações desenvolvidas para Internet, por ser o ambiente onde está concentrado o maior desafio relacionado aos testes de integração objeto desta proposta, tanto pela complexidade da arquitetura quanto pelo dinamismo das aplicações desenvolvidas neste ambiente, e onde ainda existe um vasto campo a ser explorado e debatido em

futuros trabalhos. Contudo não há restrição referente a sua aplicação em softwares desenvolvidos para outras arquiteturas desde que seja possível utilizar os recursos da AOP, porém uma análise se faz necessário.

6.2 Limitações

A utilização do *framework* revelou a necessidade de um maior conhecimento das interfaces internas dos componentes do aplicativo pelo analista de teste, o que pode demandar um tempo mais elevado para execução dos casos de testes, uma vez que as propostas existentes apóiam-se no modelo funcional onde apenas a interface com o usuário é avaliada.

Uma das limitações apresentada pelo trabalho é a realização dos testes em programas que utilizam aspectos como base do desenvolvimento de suas funcionalidades, devido ao fato do *framework* também utilizar AOP, a implementação dos pontos de junção não é capturada nos processos de "*weaving*" gerados por estes aspectos. Segundo (RESENDE, 2005) caso ocorra a necessidade de iteração entre aspectos o que deve ser feito é declarar explicitamente o "*advice*" que é executado a partir de um ponto de junção. Com este tipo de implementação o "*framework*" mostrou-se capaz de realizar seus testes como uma classe comum, porém não há como garantir que este tipo de implementação ocorra sempre nos programas a serem testados.

6.3 Trabalhos Futuros

A partir dos roteiros de testes definidos no *framework* determinar mecanismos capazes de avaliar a cobertura dos testes com relação ao código. Permitir a avaliação de métricas a partir da amostragem do histórico dos testes de regressão realizados pelo *framework*. Desenvolver a persistências das informações do roteiro de teste para flexibilizar e agilizar as alterações referentes às mudanças nas bases de informações dos aplicativos.

Uma análise mais aprofundada se faz necessário em futuros trabalhos no sentido de avaliar características como o "*overhead*" do *framework* de testes sobre software em teste. Também questões como o isolamento dos testes em ambientes onde haja grande número de requisições concorrentes para evitar a inconsistência dos dados ao serem capturados para avaliação.

Incluir a capacidade de realizar testes de integração com servidores de banco de dados, onde a partir da execução de um cenário de teste seja possível preparar o banco de dados para realização dos testes com as informações constantes do roteiro, e posteriormente a execução do roteiro validar se as informações manipuladas pelo banco de dados permanecem consistentes. Este tipo de integração com banco de dados complementa os testes de integração no caso de parte das regras de negócio ficar definido em mecanismos como *StoredProcedure* e *Triggers* na estrutura do banco de dados.

7 Referências Bibliográficas

(ALEXANDER, 2001) ALEXANDER, Roger; BIEMAN, James. **Challenges of Aspect-oriented Technology.** Colorado State University – Department of Computer Science, September 2001.

(COPELAND, 2004) COPELAND, Lee. **A Practitioner´s Guide to Software Design.** 1a Edição. Artech House Publishers, 2004.

(DBUNIT, 2003) **DbUnit - Database Testing Framework**. Disponível em http://www.dbunit.org/ (Acessado em 20/12/2007).

(DUSTIN, 2003) DUSTIN, Elfriede. **Effective Software Testing.** 1a Edição. Press Addison Wesley, 2003.

(FILMAN, 2001) FILMAN, Robert E., and Elrad, Tzilla, and Clarke, Siobhan, and Aksit, Mehmet. **Aspect-Oriented Software Development**. Press Addison Wesley, 2001.

(GAMMA, 2000). GAMMA, Erich; Beck, Kent. **JUNIT Framework** . Disponível em http://www.junit.org (Acessado em 12/03/2005)

(GAO et al., 2003) GAO, Jerry Zeyu;Tsao, Jacob; Wu, Ye. **Testing and Quality Assurance for Component-Based Software**. 1a Edição. Press Artech House, 2003.

(GRADECKI, 2003) GRADECKI, Joseph D., and Lesiecki, Nicholas. **Mastering Aspectj**. 1º Edição. Press Wiley, March 2003.

(HARROLD, 1994) HARROLD, M. J.; Rothernel, G.. **Performing data flow testing on classes**. In: II ACM SIG-SOFT Symposium on Foundations of Software Engineering, p 154-163, Nova York, NY, EUA, dez. 1994. ACM Press.

(HIEATT, 2002) HIEATT, Edward and Mee, Robert. **Testing the Web Application**. IEEE Software, April 2002.

(HOWDEN, 1987) HOWDEN, W. E., **Software Engineering and Technology: Functional Program Testing and Analysis**. Software Engineering and Technology. McGrall-Hill Book Co, Nova York, NY, EUA, 1987.

(HTTPUNIT, 2003) **HttpUnit - Automated Testing Framework**. Disponível em http://httpunit.sourceforge.net (Acessado em 01/03/2007).

(JACKSON, 1995) JACKSON, Daniel, Aspect : **Detecting Bugs with Abstract Dependences**. Carnegie Mellon University, June 1995.

(JACOBSON et al., 1999) JACOBSON, Ivar; Booch, Grady; Rumbaugh, James. **The Unified**

Modeling Language User Guide. 1a Edição. Press Addison Wesley, 1999.

(JACOBSON, 2004) JACOBSON, Ivar, and Ng, PAN-WEI. **Aspect-Oriented Software Development with Use Cases**. 1a Edição. Press Addison Wesley, 2004.

(JORGENSEN, 1994) JORGENSEN, Paul C.; Erickson, Carl. **Object-Oriented Integration Testing**. Communications of ACM, vol .37, September 1994.

(KALLEPALLI, 2001) KALLEPALLI, Chaitanya; Tiany, Jeff.**Usage Measurement for StatisticalWeb Testing and Reliability Analysis**. Southern Methodist University, Dallas, Texas, USA.

(KICZALES, 1997) KICZALES, Gregor; **Palo Alto Research Center**. 1997. URL: http://www.parc.xerox.com/, acessado em 19/08/2004.

(LADDAD, 2005) LADDAD, Ramnivas; **I want my AOP!**. (*JavaWorld*), January 2002. URL: www.javaworld.com/javaworld/ jw-01-2002/jw-0118-aspect.html, acessado em 15/02/2005.

(LEMOS, 2005) LEMOS, Otavio Augusto Lazzarini, and Maldonado, Jose Carlos and Masiero, Paulo César, **Data Flow Integration Testing Criteria for Aspect-Oriented Programs**. Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo – São Carlos – SP. URL:http://twiki.im.ufba.br/pub/Wasp/AcceptedPapers/WASP-Lemos.pdf, acessado em 10/03/2005

(MALDONADO et al., 2007) Maldonado, José Carlos; Delamaro, Márcio Eduardo; Jino, Mario. **Introdução ao Teste de Software.** 1a Edição, 2007. Editora Campus.

(MASSICOTTE et al., 2005) Massicotte, Philippe; Mourad, Badri; Mourad, Linda. **Generating Aspects-Classes Integration Testing Sequences: A Collaboration Diagram Based Strategy.** University of Quebec at Trois-Rivières.Proceedings of the 2005 Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05), 2005.

(MCILORY, 1969) Mcilory, M.D.**. Mass Produced Software Components**. *Proc. of NATO Conference on Software Engineering*, Garmisch-Partenkirchen, NATO Science Committee, NATO, Brussels, 1969.

(MONK et al., 2002) **Virtual Mock Objects using AspectJ with JUnit**. URL: http://xprogramming.com/xpmag/virtualMockObjects.htm, acessado em 20/08/2006.

(MORRIS, 2001) MORRIS, John. **Software Component Certification**. Software Engineering Australia – University Western Australia, IEEE Software September 2001.

(MYERS, 1979) MYERS, Glenford J., **The Art of Software Testing**. 1a Edição. Press John Wiley & Sons, Inc, 1979

(MYERS et al., 2004) MYERS, Glenford J. ;Sandler, Corey; Badgett, Tom; Thomas, Todd M., **The**

Art of Software Testing. 2a Edição. Press John Wiley & Sons, Inc, 2004

(NGUYEN et al., 2003) NGUYEN, Hung Q. ; Johnson, Bob; Hackett, Michael. **Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems**. 2a Edição. Press John Wiley & Sons, inc, 2003

(RESENDE, 2005) RESENDE, Antônio M. Pereira, e SILVA, Claudiney C. **Programação Orientada a Aspectos em Java**. 1a Edição, 2005. Editora BRASPORT.

(ROCHA, 2005) ROCHA, André dantas; **Uma ferramenta baseada em aspectos para apoio ao teste funcional de programas java**. Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo – São Carlos – SP, 2005

(The Testing Standards, 2008) The Testing Standard **. The Testing Standard**. URL: http://www.testingstandards.co.uk/living_glossary.htm, acessado em 05/01/2008.

(TSAI et al.,2002) TSAI,W. T.; Bai, Xiaoying; Paul, Ray; Shao, Weiguang; Agarwal, Vishal. **End-To-End Integration Testing.** Design.Department of Computer Science and Engineering, Arizona State University, Department of Computer Science and Engineering,University of Minnesota, 2002.

(VIEGA, 2000) VIEGA, John; Voas, Jeffrey. **Can Aspect-Oriented Programming Lead to More Reliable Software?** IEEE Software November/December 2000.

(VOAS, 2000) VOAS, Jeffrey**. Developing a Usage-Based Software Certification Process**. The College of William and Mary, IEEE Software August 2000.

(W3C, 2005) W3C. **World Wide Web Consortium**. URL: http://www.w3c.org, acessado em 17/07/2005.

(WHITAKER, 2000) WHITAKER, James A., **What is Software Testing? And Why is it so hard?**. Florida Institute of Technology, IEEE Software January/February 2000

(XU et all, 2004) XU, Guoqing; Yang, Zongyuan;Huang, Haitao;Chen, Qian; Chen, Ling; Xu,Fengbin.**JAOUT: Automated Generation of Aspect-Oriented Unit Test**. Software Engineering Lab, Department of Computer Science, East China Normal University. Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004.

(YUEWEI, 2005) YUEWEI Zhou, Debra Richardson, and Hadar Ziv. **Towards a Pratical Approach to Test Aspect-Oriented Software.** Department of Informatics – University of California.URL:www.cs.colostate.edu/~rta/publications/CS-04-105.pdf, acessado em 15/01/2005.

Apêndice A

Classes da aplicação *Money*

Neste apêndice encontram-se as classes do pacote Money.

```
package junit.samples.money;
/**
 * The common interface for simple Monies and MoneyBags
 *
 */
public interface IMoney {
	/**
	 * Adds a money to this money.
	 */
	public abstract IMoney add(IMoney m);
	/**
	 * Adds a simple Money to this money. This is a helper method for
	 * implementing double dispatch
	 */
	public abstract IMoney addMoney(Money m);
	/**
	 * Adds a MoneyBag to this money. This is a helper method for
	 * implementing double dispatch
	 */
	public abstract IMoney addMoneyBag(MoneyBag s);
	/**
	 * Tests whether this money is zero
	 */
	public abstract boolean isZero();
	/**
	 * Multiplies a money by the given factor.
	 */
	public abstract IMoney multiply(int factor);
	/**
	 * Negates this money.
	 */
	public abstract IMoney negate();
	/**
	 * Subtracts a money from this money.
	 */
	public abstract IMoney subtract(IMoney m);
	/**
	 * Append this to a MoneyBag m.
	 */
	public abstract void appendTo(MoneyBag m);
}
```

Listagem A. 1 – Fragmento de código: IMoney.java

```
package junit.samples.money;
/**
 * A simple Money.
 *
 */
public class Money implements IMoney {

	private int fAmount;
	private String fCurrency;

	/**  Constructs a money from the given amount and currency.*/
	public Money(int amount, String currency) {
		fAmount= amount;
		fCurrency= currency;
	}
	/** Adds a money to this money. Forwards the request to the addMoney helper. */
	public IMoney add(IMoney m) {
		return m.addMoney(this);
	}
	public IMoney addMoney(Money m) {
		if (m.currency().equals(currency()) )
			return new Money(amount()+m.amount(), currency());
		return MoneyBag.create(this, m);
	}
	public int amount() {
		return fAmount;
	}
	public String currency() {
		return fCurrency;
	}
	public int hashCode() {
		return fCurrency.hashCode()+fAmount;
	}
	public boolean isZero() {
		return amount() == 0;
	}
	public IMoney multiply(int factor) {
		return new Money(amount()*factor, currency());
	}
	public IMoney negate() {
		return new Money(-amount(), currency());
	}
	public IMoney subtract(IMoney m) {
		return add(m.negate());
	}
	public String toString() {
		StringBuffer buffer = new StringBuffer();
		buffer.append("["+amount()+" "+currency()+"]");
		return buffer.toString();
	}
}
```

Listagem A. 2 – Fragmento de código: Money.java

Apêndice B

Classes e pacotes do *framework* de testes

br.com.w3.framework.test.script
SingletonParseScript
XMLTestSuite
XMLTestSingle
MethodDef
ParamDef
AtribDef

Tabela B. 1: **Classes do pacote script.**

br.com.w3.framework.test.data
DAOTestResult

Tabela B. 2: **Classes do pacote data.**

br.com.w3.framework.test.aspect
AspectTest

Tabela B. 3: **Classes do pacote aspect.**

br.com.w3.framework.test.util
Constants
Útil

Tabela B. 4: **Classes do pacote util.**

br.com.w3.framework.test.process
TestFinder
TestProcess

Tabela B. 5: Classes do pacote process.

Apêndice C

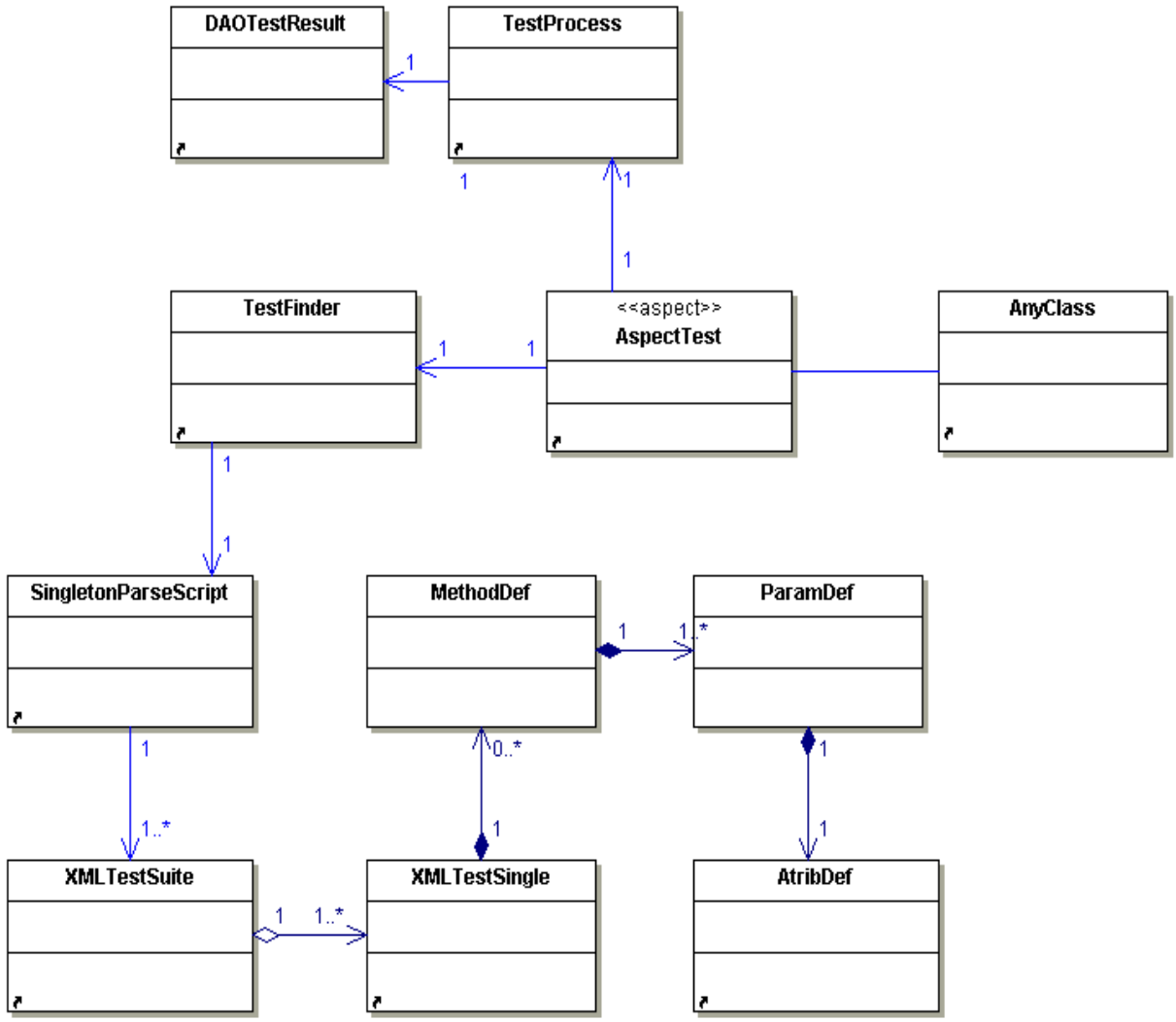

Figura C. 1. Diagrama de classes do framework de testes

Este apêndice descreve a estruturação das classes do framework e a respectiva função de cada classe dentro da proposta deste trabalho. A figura C.1 mostra um diagrama de classes, onde a partir da classe AspectTest é realizada a comunicação com as classes dos aplicativos a serem testados através do processo de "*weaving*" da AOP.

A classe *SingletonParseScript* é responsável por criar uma instância em memória do roteiro de testes a ser verificado, as demais classes diretamente relacionadas a *SingletonParseScript* tais

como : *XMLTestSuite*, *XMLTestSingle*, *MethodDef*, *ParamDef* e *AtribDef* são utilizadas para auxiliar a instância dos testes a partir do XML.

A classe *TestFinder* é responsável por localizar a existência de testes referentes aos componentes da funcionalidade que sofreu a interceptação pela classe AspectTest , caso exista algum teste, estes testes são enviados via parâmetros a classe TestProcess que executa a validação de acordo com os critérios estabelecidos no roteiro de testes e persiste o resultado através da classe *DAOTestResult* para posterior consulta e emissão de relatórios.

Apêndice D

Manual do *framework*

Nesta seção são demonstrados os passos necessários para utilização do *framework* de testes. Onde cada ponto é detalhado com o objetivo de esclarecer como definir um roteiro de testes suportado pelo *framework*. Com isso, temos os seguintes passos: "Preparação de ambiente", "Definição dos pontos de corte", "Montagem do roteiro XML", "Pré-compilação" e "Empacotamento e Publicação".

Preparação de ambiente

Ao utilizar o *framework* de testes é necessário definir alguns componentes utilizados em sua estruturação para auxiliar seu entendimento e evitar conflitos em cenários distintos do proposto. Com isso, segue a lista destes componentes utilizados para implementá-lo:

Compilador Java: JDK 1.5;

Implementação de Aspectos: Aspectj 1.5;

Parser : EXML(Eletric xml);

Container Web: TomCat 5.5.20

Definição dos pontos de corte

Ao montar seu roteiro de testes baseado nos requisitos do aplicativo em teste, o analista de teste deve selecionar com auxilio de um diagrama de seqüência quais serão os pontos de corte que serão entrecortados pelo *framework* de teste. Neste caso será necessário a alteração da classe "*AspectTest.aj*" para incluir estes pontos de corte que serão avaliados pelo *framework,* abaixo

segue um exemplo utilizado para demonstrar a prova de conceito realizada durante o desenvolvimento desse trabalho, temos:

```
pointcut callChecker() : ( ( within(br.com.w3.mestrado.exemplo.*) && call(new(..)) ) ||
                           ( within(br.com.w3.prototipo.web.servlet.*) && call(* *.initBag(..) ) ) ||
                           ( within(br.com.w3.mestrado.exemplo.*) && call(* *.*(..) ) ) );
```

O ponto de corte deve ser o mais restritivo possível, pois caso contrário irá gerar overhead desnecessário ao avaliar o aplicativo em teste, causando inclusive estouro de pilha em definições muito genéricas. No exemplo acima foi definido que o framework deve interceptar todas as chamadas das classes do pacote *br.com.w3.mestrado.exemplo.** e apenas o método *initBag* das classes do pacote *br.com.w3.prototipo.web.servlet.**. A definição dos pontos de corte pode ser exaustiva, porém deve ser realizada com cuidado, pois é a partir destas definições que será possível avaliar o aplicativo em teste com base no roteiro definido.

Montagem do roteiro XML

A partir da definição dos pontos de corte já é possível iniciar a confecção do roteiro de testes para avaliação do *framework*, a montagem do roteiro ainda é manual porém sua automação é possível de ser realizada incluindo um fase de captura e aprendizado dos pontos a serem avaliados. Para montar o roteiro de testes deve-se em primeira mão definir quais interfaces dos métodos serão avaliadas a partir dos pontos de cortes definidos na seção anterior. Para cada interface será criada uma entrada, uma saída esperada e uma ordem de execução, com isso dado o exemplo utilizado neste trabalho para avaliar a interface do método *multiply,* temos:

```
<Test Name="MoneyMethods" ClassName="junit.samples.money.Money">
    <Method Name="multiply">
        <Input Name="factor" DataType="int" value="2"/>
        <Output DataType="String" value="[24 CHF]"/>
    </Method>
</Test>
```

Inicialmente é definido o nome do teste *"Test Name"*, que pode ser definido livremente de acordo com o analista de teste, já o nome da classe deve ser o nome da classe ao qual pertence o método a ser avaliado. Note que o pacote *junit.samples.money* não consta das definições do ponto de corte definidos na seção anterior, porém como foi definido que qualquer chamada realizada a partir dos métodos da classe *br.com.w3.mestrado.exemplo* será entrecortado, os métodos da classe Money serão alcançados pelas propriedades do AOP e será possível realizar

sua avaliação. Na seqüência é definido o nome do método "*Method Name*" a ser avaliado pelo framework que segue o mesmo nome da implementação.

Um nível abaixo na estrutura do *XML* é definido os parâmetros de entrada e os resultados esperados informando seu nome "Name", tipo "DataType" e valor "value" por meio das propriedades do *XML* "Input" e "Output". A única diferença entre as definições de entrada e saída é que no caso do Output a propriedade "Name" é omitida. A partir desta definição para cada interface de método a ser avaliada pelo *framework* basta espelhar sua estrutura no formato *XML* definido acima, podendo inclusive devido sua simplicidade ser executada de forma automatizada.

Uma vez finalizado a definição do roteiro, basta indicar sua localização por meio da variável de ambiente "*–Dscript.test*" para que o *framework* possa carregá-lo durante sua execução.

Pré-compilação

Uma vez definido os pontos de corte e o roteiro XML, basta apenas realizar a pré-compilação do aplicativo em teste e o *framework*, a pré-compilação é necessária devido a escolha da implementação de aspectos utilizada nesse trabalho, a escolha do Aspectj foi feita pelo fato de possuir uma implementação mais didática dos conceitos de AOP, porém já existem implementações de Aspectos que dispensam esse passo da pré-compilação(Ex:. AspectWerkz). A pré-compilação é realizada de forma transparente por meio das opções do AspectJ no eclipse uma vez que a principal diferença está na definição do projeto que inclui os módulos necessários a sua execução. O destaque da pré-compilação só é feito para mostrar como é realizada a junção do *framework* com o aplicativo em teste, porém sua execução é muito semelhante aos procedimentos de um projeto sem aspectos.

Empacotamento e Publicação

Para realizar o empacotamento não é necessário carregar nenhuma classe do framework junto com a aplicação em teste, todos os procedimentos realizados pelo analista ao montar o pacote da sua aplicação não serão alterados, basta apenas que os pacotes referentes ao framework e a implementação de Aspectos, sejam referenciados no classpath do container onde a aplicação será publicada. Com isso, para o exemplo utilizado neste trabalho temos a seguinte divisão:

Pacote do exemplo a ser testado: aplicativo_em_teste.war.

Pacote do framework: framework.jar.

Pacote da implementação de Aspectos: aspectjrt.jar.

Pacote do parser do roteiro de Testes: exml.jar

Portanto, ao realizar a publicação do aplicativo em teste não haverá mudança alguma nos procedimentos utilizados pelo analista de delivery, somente a inclusão dos pacotes framework.jar, aspectjrt.jar e exml.jar no classpath do container onde a aplicação em teste é publicada.