UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

TESTES DE UNIDADE PARA PROGRAMAÇÃO ORIENTADA A ASPECTOS EM DELPHI

EDUARDO MAFRA

BLUMENAU
2007

2007/2-10

EDUARDO MAFRA

TESTES DE UNIDADE PARA PROGRAMAÇÃO ORIENTADA A ASPECTOS EM DELPHI

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Marcel Hugo , M. Eng. – Orientador

BLUMENAU
2007

2007/2-10

TESTES DE UNIDADE PARA PROGRAMAÇÃO ORIENTADA A ASPECTOS EM DELPHI

Por

EDUARDO MAFRA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Marcel Hugo, M.Eng. – Orientador, FURB
Membro:	Prof. Fabiane Barreto Vavassori Benitti, Dr. – FURB
Membro:	Prof. Everaldo Artur Grahl, M.Eng. – FURB

Blumenau, 21 de novembro de 2007

Dedico este trabalho aos meus pais, Osni Mafra e Mara Lúcia Mafra, por terem acreditado sempre no meu esforço e na vitória.

AGRADECIMENTOS

Agradeço aos meus pais por sempre acreditarem em mim e por me fornecerem os meios de vencer mais esta etapa da minha vida.

Aos meus amigos que sempre me incentivaram na realização deste.

Ao meu orientador, Marcel Hugo, pelos ensinamentos, confiança em mim depositada e por ter acreditado na conclusão deste trabalho.

À professora Joyce Martins e ao colega de curso Edmar Soares de Oliveira, pela atenção, ajuda e apoio, os mesmos de fundamental importância para a execução deste trabalho.

Enfim, a todas as pessoas que de alguma forma contribuíram para a realização deste trabalho.

Muito Obrigado.

Só se pode lutar pelo o que se ama, só se pode amar o que se respeita e respeitar o que pelo menos se conhece.

A. H.

RESUMO

O recente aumento da complexidade na criação de software exige novos desafios ao desenvolvedor que as técnicas de programação orientada a objetos não resolvem. Essa evolução tem forçado a indústria de software em dividir as áreas de preocupações de desenvolvimento em partes independentes. O trabalho de organizar as partes independentes em nichos de funcionalidade denomina-se separação de interesses. A programação orientada a aspectos foi concebida com a proposta de resolução deste problema, a partir do uso de mecanismos que permitem o isolamento dos interesses. Entretanto, por ser uma técnica nova, nesses primeiros anos os pesquisadores preocuparam-se em estabelecer os conceitos e técnicas básicas das linguagens orientadas a aspectos, deixando para uma segunda fase a investigação de outras características do desenvolvimento de programas orientado a aspectos, como métodos de projeto e abordagens de testes. Neste trabalho é apresentada uma revisão sobre tipos e categorias de testes além de uma revisão sobre planejamento de casos de testes. Um levantamento sobre a técnica de programação orientada a aspectos e seus benefícios. É estendida a ferramenta AOPDelphi (OLIVEIRA, 2006) para suportar a técnica de teste unitário para programação orientada a aspectos, composta por uma linguagem para definição dos testes e por um ambiente para a codificação dos mesmos. A ferramenta recebe como entrada as rotinas com as definições dos testes e gera como saída um projeto Delphi com as classes de testes unitários.

Palavras-chave: Programação orientada a aspectos. Teste unitário. Delphi.

ABSTRACT

With the development gained with the object-oriented programming, the recent increase of complexity in the creation of the software inserted new challenges to the developer that the object-oriented programming do not solve. This evolution has forced the software industry to divide the development concern areas in to independent parts. The work of organize the independents parts into functionality niches is called separation of concerns. The aspect-oriented programming was conceived with the resolution offer of this problem, and with the use of the mechanism use, that allow the interests separation. However, for being a new technique, on this first years, the researchers worry about to establish the concepts and basic techniques of aspects oriented languages, leaving for a second phase the investigation of others development characteristics of aspect-oriented programs, with methods of project and research tests. In this paper, its presented a research of types and categories tests over the research on the planning of test cases. The explanation about the aspect-oriented programming techniques and their benefits. It´s extended the AOPDelphi tool (OLIVEIRA, 2006) to support the unit test for the aspect-oriented programming, and made by test definition language and for a codification environment of themselves. The tool receive access the routines with the definitions of the tests and creates as an exit a Delphi project with the single test class.

Key-words: Aspect-oriented programming. Unit test. Delphi.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

BNF – *Backus-Naur Form*

IDE – *Integrated Development Environment*

POA – Programação Orientada a Aspectos

POO – Programação Orientada a Objetos

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

A qualidade hoje em dia é valorizada em vários setores e conduz a uma evolução natural. Assim como todos os setores, a informática busca aprimorar a qualidade de seus produtos e serviços.

O tema central da engenharia de software é a produção de um software de alta qualidade. A engenharia de software permite que se controle o processo de desenvolvimento, fornecendo ao engenheiro de software as bases para a construção de um software de alta qualidade (PARRINGTON; ROPER, 1989, p. 9, tradução nossa; INTHURN, 2001, p. 11). Um dos critérios indispensáveis na medição da qualidade de software é a verificação de confiabilidade e funcionalidade de sistema.

Segundo Pressman (2002, p. 724), qualidade de software é definida como "conformidade a requisitos funcionais e de desempenho explicitamente declarados, padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido."

Parrington e Roper (1989, p. 9, tradução nossa) afirmam que "fatores que contribuem para a qualidade do software são: especificações formais, análise de sistemas, linguagens de programação mais avançadas e ferramentas de suporte. Porém, o fator chave que determina a qualidade do software são os testes."

Devido aos altos custos de manutenibilidade, as empresas investem cada vez mais na qualidade de seus sistemas antes de os mesmos entrarem em funcionamento comercialmente. Freqüentemente, gasta-se muito tempo e dinheiro em testes e na correção dos erros encontrados. Teste de software é por si só, uma atividade muito dispendiosa e o custo de não testar é potencialmente muito maior (PARRINGTON; ROPER, 1989, p. 14, tradução nossa; INTHURN, 2001, p. 52).

O processo de teste é uma parte integral de qualquer programa de qualidade e, para um produto ficar estável em seu ciclo de vida conforme almeja-se, o mesmo deve ser testado (PARRINGTON; ROPER, 1989, p. 26, tradução nossa).

"O teste deve ser considerado uma atividade necessária de coleta de informações, de modo a permitir que avaliemos nosso trabalho com eficiência" (HETZEL, 1987, pg. 7). O teste de software não pode ser considerado uma fase ou etapa do ciclo de desenvolvimento, mas uma atividade abrangendo todo o ciclo de desenvolvimento.

Para Pol, Teunissen e Veenendaal (2002, p. 5, tradução nossa), "teste é apenas um dos

muitos esforços para a qualidade de software sendo que o objetivo principal é prevenção e não a detecção de defeitos do produto final."

> Durante o processo de desenvolvimento de um software existem atividades que procuram garantir a qualidade do produto final; entretanto, apesar dos métodos, técnicas e ferramentas utilizadas, falhas no produto ainda podem ocorrer. Assim, a etapa de teste, a qual representa uma das atividades de garantia de qualidade, é de grande importância para a identificação e eliminação de falhas, representando assim o último passo no desenvolvimento do software. (INTHURN, 2001, p. 51).

O estudo dos métodos de testes visa estabelecer um método apropriado para reduzir as taxas de erro em programas de uma maneira mais eficaz e em um tempo considerável, melhorando a qualidade do software.

O surgimento da técnica de Programação Orientada a Objetos (POO) abriu portas para a produção de softwares com maior qualidade. A mesma propôs uma mudança radical na maneira de se conceber softwares e facilitou a aplicação dos métodos de testes, sobretudo os testes de unidade.

Os testes de unidade têm por objetivo verificar um elemento que possa ser logicamente tratado como uma unidade de implementação, assegurando assim, que esta unidade de implementação opera de forma correta. Em produtos implementados com a tecnologia de POO, uma unidade é tipicamente uma classe (PAULA FILHO, 2001, p. 185).

Porém, com o crescimento dos sistemas e o respectivo aumento de sua complexidade, observou-se o surgimento de problemas como, por exemplo, entrelaçamento e espalhamento de código, que as técnicas de POO são incapazes de resolver (KICZALES, et al., 1997). Assim nasceu a Programação Orientada a Aspectos (POA), com o intuito de apresentar soluções para estes problemas.

Para Resende e Silva (2005, p. 12), "a POA tem por objetivo separar os níveis de preocupação durante o desenvolvimento de software. [...] e a proposta é poder desenvolver as partes do sistema sem se preocupar com as demais partes." Este trabalho de separar os níveis de preocupação denomina-se separação de interesses.

A POA é uma nova técnica de programação concebida para permitir o uso mais efetivo do princípio da separação de interesses no desenvolvimento de softwares. Contudo, até agora, a maioria das discussões sobre POA tem focado nos conceitos básicos sobre a mesma, deixando para um segundo momento os problemas relacionados com o desenvolvimento de software orientado a aspectos (LEMOS et al., p. 55).

A POA é reconhecida como uma técnica que facilita a manutenção dos diferentes interesses e a legibilidade de código, gerando assim, sistemas com melhores arquiteturas. Entretanto, a sua simples utilização não evita que erros sejam introduzidos ao longo do

desenvolvimento do software e, dessa maneira, técnicas de verificação, validação e teste continuam sendo importantes no processo de desenvolvimento de software orientado a aspectos (LEMOS et al., p. 55).

Em Oliveira (2006), foi desenvolvido um protótipo de um *weaver* para geração de código na linguagem Object Pascal com suporte a programação orientada a aspectos. O *weaver* recebe como entrada um projeto Delphi, e um ou mais programas de aspectos em uma linguagem própria da ferramenta para especificação dos mesmos. Como saída, a ferramenta gera um projeto Delphi que mescla as funcionalidades dos programas de aspectos com o projeto Delphi fornecidos como entrada.

O ambiente de desenvolvimento Delphi não possui suporte nativo a testes de unidade. Para suprir tal necessidade, o *framework* DUnit oferece um conjunto de classes que permite ao desenvolvedor implementar classes de testes para automatizar a execução dos testes de unidade. O DUnit não automatiza a criação dos testes de unidade, o mesmo apenas fornece uma estrutura para a implementação dos testes e executa as classes implementadas pelo desenvolvedor a partir do plano de testes, revelando o sucesso ou falha do código testado.

Este trabalho permite que o desenvolvedor elabore e automatize os testes de unidade em Object Pascal com suporte do DUnit. Considerando a importância da técnica de testes de unidade e a não existência da mesma no ambiente Delphi e considerando a inserção da técnica de POA neste mesmo ambiente, observa-se a necessidade de estender a ferramenta de Oliveira (2006) criando funcionalidades que auxiliem o desenvolvedor a elaborar os testes unitários nos aspectos e nas classes aspectadas[1].

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é elaborar uma ferramenta que auxilie o desenvolvedor a gerar código na linguagem Object Pascal para testar os aspectos e as classes aspectadas.

Os objetivos específicos são:

a) criar uma linguagem para definição de testes;
b) permitir que a ferramenta receba como entrada um ou mais programas fontes na linguagem criada para a definição dos testes;

[1] Trata-se de um neologismo. É o mesmo que ser afetado pelo aspecto.

c) realizar análise léxica, sintática e semântica dos programas fontes de testes;
d) disponibilizar um ambiente para a codificação das rotinas de testes;
e) gerar código fonte em Object Pascal com as classes de testes de unidade;
f) integrar a ferramenta criada no ambiente desenvolvido por Oliveira (2006).

1.2 ESTRUTURA DO TRABALHO

O conteúdo do trabalho está organizado em seis capítulos. No capítulo seguinte é feita uma apresentação sobre testes de software com seus conceitos, tipos e categorias de testes. No capítulo três é apresentado a tecnologia de programação orientada a aspectos com seus conceitos básicos, elementos que a compõem, benefícios e uma abordagem sobre testes de unidade para programação orientada a aspectos. No capítulo quatro é feito uma introdução ao *framework* DUnit e, ao final do capítulo, apresentado os trabalhos correlatos. O capítulo cinco trata da especificação e implementação da ferramenta desenvolvida. Para finalizar, no capítulo seis são apresentadas as conclusões alcançadas no desenvolvimento do trabalho, além de sugestões para extensões do mesmo.

2 TESTES

Para ter um melhor conhecimento sobre o assunto de testes de software, neste capítulo é visto as definições sobre o assunto. São elencados ainda os dois tipos de testes e as categorias de testes com ênfase em testes de unidade. Por fim, é realizado uma revisão sobre planejamento de testes.

2.1 TIPOS DE TESTES

Para executar a tarefa de teste de software são encontradas diversas técnicas e metodologias na literatura. Existem basicamente duas maneiras de construir testes: caixa branca e caixa preta (PAULA FILHO, 2001, p. 185).

2.1.1 Método da caixa branca

Este método tem por objetivo determinar defeitos na estrutura interna do produto, através do desenho de testes que exercitem suficientemente os possíveis caminhos de execução. Percorrer todos os caminhos, no entanto, acaba tornando a atividade de teste muito complexa. Em um sistema desenvolvido com a tecnologia de POO, essa complexidade torna-se muito maior devido principalmente ao polimorfismo, pois não é possível ver no código qual a unidade que está ativa em um determinado momento, visto que, quando um estímulo é mandado, qualquer instância de uma classe pode receber o mesmo (INTHURN, 2001, p. 55).

Segundo Pressman (2002, p. 793), o método da caixa branca garante que:

a) todos os caminhos independentes dentro de um módulo tenham sido exercitados pelo menos uma vez;
b) exercita todas as decisões lógicas para valores falsos ou verdadeiros;
c) executa todos os laços em suas fronteiras e dentro de seus limites operacionais;
d) exercita as estruturas de dados internas para garantir a sua validade.

2.1.2 Método da caixa preta

Este método tem por objetivo determinar se os requisitos foram total ou parcialmente satisfeitos pelo produto. Os testes de caixa preta não verificam como ocorre o processamento, mas apenas os resultados produzidos.

Este método tenta assegurar que as funções do sistema de aplicação atinjam os objetivos esperados e de forma correta. O teste de caixa preta concentra-se nos requisitos funcionais do software. Através dele, torna-se possível verificar as entradas e saídas de cada unidade (INTHURN, 2001, p. 56).

Inthurn (2001, p. 56) enfatiza que o método de caixa preta procura descobrir basicamente:

a) funções incorretas ou ausentes;
b) erros de interface;
c) erros nas estruturas de dados ou no acesso a bancos de dados externos;
d) erros de desempenho;
e) erros de inicialização e término.

O teste de caixa preta não é uma alternativa para as técnicas de caixa branca. Ao contrário, trata-se uma abordagem complementar que visa descobrir uma classe de erros diferente daquela dos métodos de caixa branca (PRESSMAN, 2002, p. 816).

2.2 CATEGORIAS DE TESTES

A atividade de teste pode ser classificada basicamente em quatro categorias, segundo Pressman (2002, p. 838): de unidade, de integração, de validação e de sistema.

2.2.1 Teste de unidade

Os testes de unidade têm por objetivo verificar um elemento que possa ser logicamente tratado como uma unidade de implementação. Em produtos implementados com a tecnologia de POO, uma unidade é tipicamente uma classe (PAULA FILHO, 2001, p. 185).

O teste de unidade concentra-se no esforço de verificação da menor unidade de projeto de software – o módulo ou classe. Usa-se a descrição do projeto detalhado como guia e os caminhos de controle importantes são testados para descobrir erros dentro das fronteiras do módulo. O teste de unidade baseia-se na caixa branca, e esse passo pode ser realizado em paralelo para múltiplos módulos (PRESSMAN, 2002, p. 844).

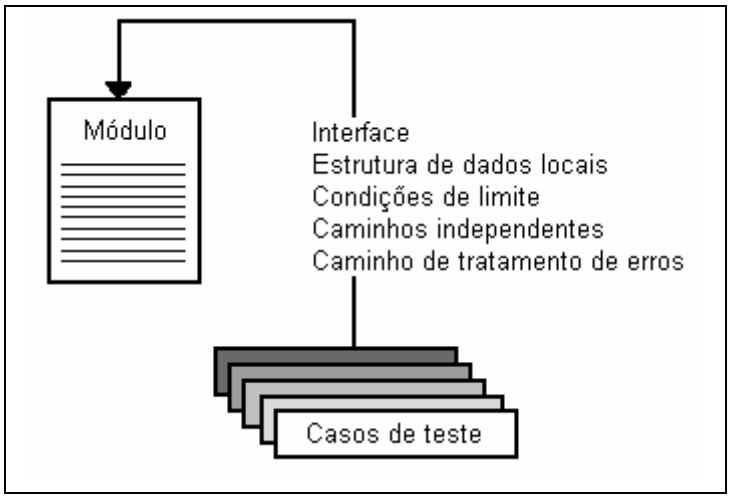
Como o objetivo do teste de unidade é verificar a menor unidade do projeto, pode-se ainda identificar como a menor unidade um procedimento ou operação de uma classe, o qual é a menor parte funcional de um programa (ROCHA; MALDONADO; WEBER, 2001, p.75).

Ainda segundo Rocha, Maldonado e Weber (2001, p. 75), "uma unidade é um componente de software que não pode ser subdividido."

De acordo com Pressman (2002, p. 844), os testes de unidade devem levar os seguintes pontos em consideração:

a) interface com o módulo: testa-se para ter a garantia que as informações fluem para dentro e para fora da classe que se encontra sob teste;
b) estrutura de dados local: examinada para ter a garantia de que os dados armazenados temporariamente mantêm sua integridade durante todos os passos de execução do algoritmo;
c) condições de limite: testes para ter a garantia que a classe opera adequadamente nos limites estabelecidos;
d) caminhos básicos: todos os caminhos da estrutura de controle de uma classe são exercitados para ter a garantia que todas as instruções foram executadas pelo menos uma vez;
e) caminhos de tratamento de erros: caminhos de tratamento de erros devem ser executados a fim de verificar se valores não verdadeiros são devidamente tratados.

Os testes que acontecem como parte da atividade de teste de unidade é ilustrado esquematicamente na Figura 1.



Fonte: adaptado de Pressman (2002, p. 845).

Figura 1 – Atividades do teste de unidade

Pressman (2002, p. 846) ainda aponta os erros mais comuns de computação que podem ser descobertos com testes de unidade, os quais são:

a) precedência aritmética incorreta ou mal compreendida;
b) operações em modo misto;
c) inicialização incorreta;
d) erro de precisão;
e) representação simbólica incorreta de uma expressão.

Sendo o objetivo encontrar defeitos no sistema, o desenvolvedor deve seguir alguns passos para alcançar o mesmo. Primeiro, examina-se o código para identificar defeitos no algoritmo e erros de sintaxe. Segundo, compila-se o código, eliminando-se os erros de sintaxe remanescentes e, finalmente, desenvolvem-se casos de testes para mostrar que a entrada é apropriadamente convertida na saída desejada (PFLEEGER, 2004, p. 279).

Os casos de testes, segundo Pressman (2002, p. 847), devem descobrir erros tais como:

a) comparação de diferentes tipos de dados;
b) operadores lógicos ou precedência incorretos;
c) expectativa de igualdade quando um erro de precisão torna a igualdade improvável;
d) comparação ou variável incorreta;
e) término de laço impróprio ou inexistente;
f) falha para sair quando é encontrado iteração divergente;
g) variáveis de laço impropriamente modificadas.

É necessário possuir um bom planejamento e desenho dos casos de testes; caso contrário pode obter-se um conjunto de casos testes que não encontram erros (o verdadeiro objetivo da atividade).

Para criar casos de testes eficientes, uma revisão das informações do projeto proporciona orientação para estabelecer os mesmos. Cada caso de teste deve estar associado a um conjunto de resultados esperados (PRESSMAN, 2002, p.848).

Para a criação dos casos de testes, coloca-se a necessidade de implementação de *drivers* e *stubs*. O *driver* é a unidade responsável pela ativação e coordenação dos testes de unidade. Ele é o responsável por receber os dados de teste fornecidos pelo testador e, de acordo com a interface da unidade, repassar esses dados para a unidade obtendo assim os resultados produzidos. Um *stub* é uma unidade que substitui, na hora do teste, uma unidade utilizada (chamada) pelo método ou classe. Em geral, o *stub* simula o comportamento da unidade chamada com o mínimo de computação ou manipulação de dados (ROCHA; MALDONADO; WEBER, 2001, p.75).

Em um teste de unidade, não é preciso que um determinado teste chegue ao fim para que outro possa ser iniciado. É possível, por exemplo, efetuar o teste de unidade e o de integração simultaneamente. Em outras palavras, no teste de unidade pode-se verificar a estrutura interna da classe e no teste de integração, a interface ou troca de mensagens entre classes (INTHURN, 2001, p. 58).

2.2.2 Teste de integração

O teste de integração verifica basicamente se as unidades testadas de forma individual executam corretamente quando integradas (INTHURN, 2001, p. 59).

Quando o desenvolvedor estiver convencido que os componentes individuais estão funcionando corretamente e de que atingem os objetivos esperados, o mesmo combina os componentes em um sistema em funcionamento. Esta integração é planejada e coordenada de modo que quando uma falha ocorre, o desenvolvedor tem idéia do que a causou (PFLEEGER, 2004, p. 290).

2.2.3 Teste de sistema

O teste de sistema é uma atividade de validação usada para demonstrar que o software inteiro está correto. Ele é executado após o teste de integração e é basicamente um teste de caixa preta (INTHURN, 2001, p. 62).

Existem, segundo Pfleeger (2004, p. 315), basicamente quatro etapas em testes de sistema:

a) teste funcional: verifica se o sistema integrado realiza as funções especificadas nos requisitos;
b) teste de desempenho: compara os componentes integrados com os requisitos não funcionais do sistema. Esses requisitos, incluindo segurança, precisão, velocidade e confiabilidade restringem o modo como as funções do sistema são realizadas;
c) teste de aceitação: assegura aos clientes que o sistema solicitado é o que foi construído;
d) teste de instalação: permite que o usuário final execute as funções do sistema e documente problemas adicionais específicos.

O objetivo principal deste tipo de teste é pôr completamente à prova o sistema, ou seja, após todas as outras categorias de testes, o sistema é testado em conjunto com outros softwares e elementos de hardware, os quais serão usados comumente após sua liberação (INTHURN, 2001, p. 62).

2.2.4 Teste de validação

O teste de validação ocorre ao término da atividade de teste de integração. A validação é bem sucedida quando o software funciona de uma maneira esperada pelo cliente, ou conforme a especificação (INTHURN, 2001, p. 59).

O teste de validação é usado para garantir que todos os requisitos funcionais são satisfeitos, todos os requisitos de desempenho são conseguidos, a documentação está correta e passou por um trabalho de engenharia e outros requisitos como portabilidade, compatibilidade, remoção de erros e manutenibilidade são efetivamente cumpridos (PRESSMAN, 2002, p. 859).

2.3 PLANEJAMENTO DE TESTES

O projeto de teste de software e de outros produtos trabalhados por engenharia pode ser tão desafiador quanto o projeto inicial do próprio produto. Contudo, os engenheiros muitas

vezes tratam a atividade de teste como uma reflexão tardia, desenvolvendo casos de testes que podem parecer certos, mas que apresentam pouca garantia de estar efetivamente completo (PRESSMAN, 2002, p. 791).

Decidir o que testar e quando terminar os testes de programas é quase sempre decisão do desenvolvedor. Em sua grande maioria os testes feitos são informais e não se mantêm registros dos testes efetuados e dos erros encontrados (HETZEL, 1987, p. 10).

Em algumas empresas, o teste de programa é mais formal. Planos de testes descrevendo o que será testado e os resultados esperados devem ser assinados pelo analista e pelo programador (HETZEL, 1987, p. 10).

"O documento que descreve as alternativas a serem testadas e os resultados esperados é conhecido como plano de teste." (HETZEL, 1987, p. 23)

O planejamento da atividade de teste deve fazer parte do planejamento global do sistema, culminando em um plano de teste que constitui um dos documentos cruciais no ciclo de vida de desenvolvimento de software (ROCHA; MALDONADO; WEBER, 2001, p. 74).

Em um plano de teste, são estimados recursos e são definidos estratégias, métodos e técnicas de teste, caracterizando-se um critério de aceitação do software em desenvolvimento. A falta de tempo e recursos para a realização de testes são os principais problemas enfrentados pelas equipes de testes de software (ROCHA; MALDONADO; WEBER, 2001, p. 74).

Para Hetzel (1987, p. 23), um plano de teste deve conter:

a) um plano para a realização do teste: este plano contém a descrição dos objetivos do teste e a descrição de como os objetivos do teste devem ser atendidos;
b) um plano para a verificação dos resultados do teste: este plano contém a descrição dos resultados esperados e a descrição dos procedimentos de validação dos resultados;
c) um veículo para documentação dos resultados: este veículo contem a descrição do que foi testado e a descrição dos resultados obtidos.

Inthurn (2001, p. 77) elenca que um plano de teste deve possuir ainda, resumidamente, os seguintes itens:

a) título;
b) identificação do software (nome, versão);
c) histórico de revisão do documento (autor, data);
d) proposta do documento;
e) objetivo do teste;

f) lista de documentos relevantes (requerimentos, documentos do projeto, planos de teste);
g) responsabilidades do teste;
h) análise de risco do projeto;
i) escopo e limite do teste;
j) ambiente de teste (*hardware,* sistema operacional, requerimentos de software);
k) descrição de ferramentas para captura de telas do software que serão utilizadas a fim de ajudar a descrever e relatar os erros;
l) necessidades de configuração do ambiente de teste;
m) testes automatizados;
n) relatórios de resultados do teste;
o) critérios de suspensão e re-início do teste;
p) métricas usadas;
q) alocação de pessoal;
r) necessidades de treinamento;
s) apêndice;
t) glossário.

O engenheiro de software deve projetar testes que tenham a mais alta probabilidade de descobrir a maioria dos erros com uma quantidade mínima de tempo e esforço (PRESSMAN, 2002, p. 791).

3 PROGRAMAÇÃO ORIENTADA A ASPECTOS

Neste capítulo é realizada uma introdução a tecnologia de POAPrimeiramente é visto o conceito da separação de interesses. Em seguida são apresentados conceitos básicos sobre a POA e os elementos que compõem a mesma. Ao final são vistos os benefícios da técnica de POA e uma abordagem de teste de unidade para POA.

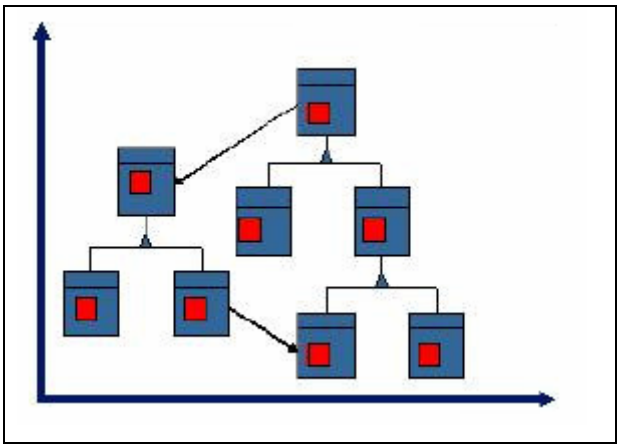
3.1 SEPARAÇÃO DE INTERESSES

O trabalho de organizar os requisitos de software em nichos de funcionalidades, interesses, preocupações e responsabilidades denomina-se *separation of concerns* ou separação de interesses (RESENDE; SILVA, 2005, p. 7). Um interesse (*concern*) é uma fração do problema que se deseja tratar como um componente a parte, podendo ser um requisito, uma funcionalidade ou uma propriedade do sistema (KULESZA; SANT'ANNA; LUCENA, 2005, p. 266). Em outras palavras, separação de interesses é a teoria estabelecida pela engenharia de software que prega que as preocupações envolvidas no desenvolvimento de um software devem ser focadas e trabalhadas separadamente. Essas preocupações que atravessam todo o sistema são denominadas de requisitos transversais ou ortogonais (KULESZA; SANT'ANNA; LUCENA, 2005, p. 266).

Requisitos transversais possuem este nome devido a sua implementação atravessar a estrutura de outros módulos do sistema, dificultando sua modularização (RESENDE; SILVA, 2005, p. 8). Dessa forma, aparecem os problemas de entrelaçamento e espalhamento de código, reduzindo a reusabilidade e dificultando a evolução do sistema (KULESZA; SANT'ANNA; LUCENA, 2005, p. 266).

Para tornar prática a teoria de separação de interesses, pesquisadores têm desenvolvido métodos, técnicas e ferramentas para componentizar e facilitar o desenvolvimento de software. A POA é uma dessas técnicas, sendo que o objetivo final de POA é reduzir a complexidade do desenvolvimento de software (RESENDE; SILVA, 2005, p. 9).

Na figura 2, supõem-se um modelo de classes onde os pontos destacados simbolizam os requisitos transversais. Percebe-se que eles estão espalhados por todas as classes, poluindo as classes com implementações que não são de interesse do negócio.



Fonte: adaptado de Barros (2004, p. 9).

Figura 2 – Exemplo de interesse transversal

3.2 CONCEITOS BÁSICOS SOBRE POA

Com o crescimento dos sistemas e o aumento de sua complexidade, tem-se observado alguns problemas, como por exemplo, entrelaçamento e espalhamento de código que as técnicas de POO são incapazes de resolver (KICZALES et al., 1997, p. 1). Assim nasceu a POA, com o intuito de apresentar técnicas capazes de cobrir essas falhas.

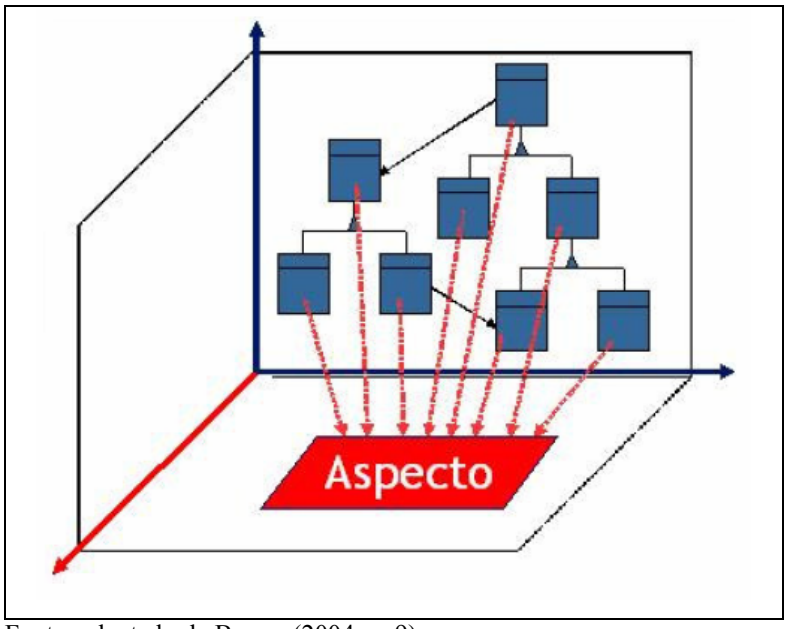
Para Resende e Silva (2005, p. 12), "a POA tem por objetivo separar os níveis de preocupação durante o desenvolvimento de software [...] e a proposta é poder desenvolver as partes do sistema sem se preocupar com as demais partes."

A POA tem o objetivo de resolver o problema dos requisitos transversais. Tenta-se separar os níveis de preocupação durante o desenvolvimento de software, sendo possível pensar separadamente nos problemas (RESENDE; SILVA, 2005, p. 12).

A POA oferece recursos para apoiar na separação de interesses transversais a fim melhorar a modularização desses interesses. Isso é feito através de abstrações que permitem a separação e composição desses interesses. Essa abstração é denominada aspecto (KULESZA; SANT'ANNA; LUCENA, 2005, p. 266).

O aspecto encapsula uma funcionalidade que atravessa vários módulos do sistema. Em definição, um aspecto é a unidade de modularidade para interesses transversais (KULESZA; SANT'ANNA; LUCENA, 2005, p. 267).

Levando em consideração a figura 2, a figura 3 exemplifica como fica encapsulada a implementação de um interesse transversal utilizando POA.



Fonte: adaptado de Barros (2004, p. 9).

Figura 3 – Exemplo de encapsulamento de um interesse transversal com POA

A implementação que antes estaria espalhada por diversas classes ou unidades do sistema, agora está encapsulada dentro de uma única unidade: o aspecto.

Resende e Silva (2005, p. 9) afirmam que "o objetivo final de POA é reduzir a complexidade do desenvolvimento de software, permitindo que as diversas preocupações possam ser componentizadas independentemente de sua natureza, funcional ou não funcional".

3.3 ELEMENTOS DA POA

A POA possui alguns elementos básicos, os quais são (CHAVEZ; LUCENA, 2004; TIRELO et al., 2004):

a) componente: unidade funcional expressa em uma linguagem de componentes. Propriedades de um sistema, no qual a implementação pode ser encapsulada de forma limpa em um procedimento generalizado;
b) aspecto: unidade não funcional expressa em uma linguagem de aspectos. Propriedades de um sistema no qual a implementação não pode ser encapsulada em um procedimento generalizado;
c) *crosscuting*: entrelaçamento entre os interesses de domínio e os interesses ortogonais;

d) pontos de junção (*join points)*: pontos definidos na execução de um programa;
e) conjuntos de junção (*point cuts*): elementos da semântica da linguagem de componentes com os quais os programas de aspectos coordenam; elementos compostos por pontos de junção que tem por objetivo reunir informações a respeito do contexto do ponto selecionado;
f) regras de junção: código relativo aos requisitos ortogonais que deve ser executado nos pontos de junção;
g) processo de combinação (*weaving*): composição entre os componentes e os aspectos;
h) combinador de aspectos (*aspect weaver*): processador de linguagem especial que oferece suporte à composição entre os componentes e os aspectos.

3.4 BENEFÍCIOS

Nelson (2005, p. 17) elucida benefícios da POA. Entre os explanados, cita-se:

a) menos responsabilidades em cada parte: como os interesses ortogonais são separados em seus próprios módulos, as partes do programa que lidam com a lógica de negócios não ficam poluídas com código que lida com interesses periféricos;
b) melhor modularização: como os módulos em POA não são chamados diretamente, há uma redução no nível de acoplamento;
c) evolução facilitada: novos aspectos podem ser acrescentados facilmente sem necessidade de alterar o código existente;
d) maiores possibilidades de reutilização: como o código não mistura interesses, aumentam-se as possibilidades de reutilizar-se módulos em sistemas diferentes.

3.5 TESTES DE UNIDADE PARA POA

A POA é uma nova técnica de programação concebida para permitir o uso mais efetivo do princípio da separação de interesses no desenvolvimento de software. Recentemente, além

de focarem-se apenas nos conceitos básicos de POA, os desenvolvedores têm voltado suas atenções para os problemas relacionados com o desenvolvimento de software orientado a aspectos, tais como técnicas de projeto e abordagens de verificação e teste de software (LEMOS et al., 2004, p. 55).

Zhao é um dos poucos pesquisadores a apresentar uma abordagem de testes para programas orientados a aspectos. O teste de unidade é, fundamentalmente, um teste estrutural (caixa branca), e o mesmo deriva seus requisitos a partir da estrutura lógica dos programas. A principal motivação da técnica é que não se pode confiar em um trecho de um programa se ainda existem certos caminhos que nunca foram executados (LEMOS et al., 2004, p. 56).

Segundo Delamaro, Maldonado e Jino (2007, p. 191), a atividade de teste de programas orientado a aspectos pode ser dividida nas seguintes fases:

a) teste de unidade: objetiva identificar defeitos na lógica e na implementação de cada método, método introduzido ou *advice*;
b) teste de integração: objetiva identificar defeitos na integração entre conjuntos de módulos.

Portanto, em POA testes de unidades basicamente são testes nos aspectos. Um aspecto é modelado para funcionar independentemente o máximo possível do código que está ao seu redor. Esta prerrogativa torna simples a tarefa do desenvolvedor em escrever testes para POA, pois o mesmo poderá escrever apenas um único teste que exercite apenas o aspecto (ZHAO, 2003, p. 2, tradução nossa).

Por outro lado, os aspectos afetam uma ou mais classes de acordo com os *advices*, tornando assim as classes afetadas mais complexas. Conseqüentemente, no momento de efetuar testes de unidade para POA, o desenvolvedor não deve ater-se apenas em testar os aspectos, deve também testar as classes afetadas pelos aspectos, pois as mesmas podem ter sido afetadas indevidamente pelo aspecto fazendo com que fuja do seu escopo original (ZHAO, 2003, p. 2, tradução nossa).

4 DUNIT E TRABALHOS CORRELATOS

Neste capítulo é feito uma introdução ao *framework* DUnit, seus conceitos e uma explanação de seu funcionamento. Em seguida são apresentados os trabalhos correlatos.

4.1 DUNIT

O DUnit é um *framework* de classes para Delphi desenvolvido para suportar teste de software. O propósito do *framework* é fornecer ao desenvolvedor de software, facilidade na criação de código para a automação de testes com a apresentação de resultados. O DUnit é inspirado no *framework* JUnit, ferramenta de mesmo propósito para linguagem Java (ANEZ, 2000).

A interface do DUnit pode ser vista na figura 4.

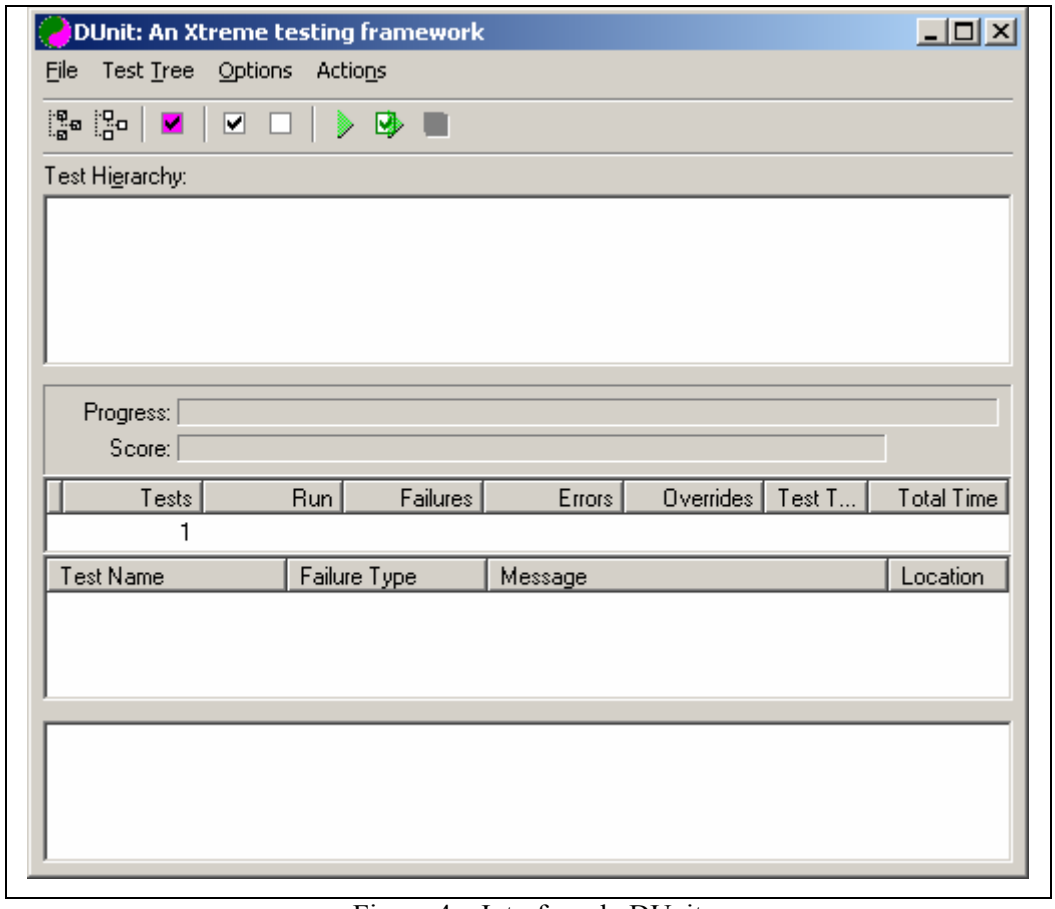


Figura 4 – Interface do DUnit

Com o *framework*, pode-se verificar se cada método de uma classe funciona de forma esperada, relatando possíveis falhas ou erros no código. É possível ainda ser executado em uma bateria de testes (ANEZ, 2000).

O Dunit não automatiza os testes. Os testes de unidade demandam a especificação de um plano de testes por parte do desenvolvedor. O *framework* propõe-se apenas a executar o código implementado pelo desenvolvedor a partir do plano de testes.

O *framework* fornece classes para facilitar a criação de testes de unidade. Estas classes possuem diversos métodos para efetuar testes de validação sobre as unidades que se deseja testar. Entre os métodos fornecidos, os principais estão indicados no Quadro 1.

Os métodos possuem como parâmetro uma condição booleana que visa descobrir se o método da classe a ser testada atende à condição booleana passada como parâmetro ao DUnit. Os métodos possuem ainda um parâmetro do tipo *string* opcional que é a mensagem de retorno, caso o teste seja falho.

MÉTODO	**DESCRIÇÃO**
`Check(condition: Boolean; msg: string = '')`	Verifica se a condição booleana do parâmetro "condition" tem retorno verdadeiro. O parâmetro requer uma expressão booleana
`CheckTrue(condition: Boolean; msg: string = '')`	Verifica se a condição booleana do parâmetro "condition" tem retorno verdadeiro
`CheckFalse(condition: Boolean; msg: string = '')`	Verifica se a condição booleana do parâmetro "condition" tem retorno falso
`CheckEquals(expected, actual: extended; msg: string = '')`	Verifica se o valor do parâmetro "expected" é idêntico ao valor do parâmetro "actual"
`CheckEqualsString(expected, actual: string; msg: string = '')`	Verifica se a string do parâmetro "expected" é idêntica à string do parâmetro "actual"
`CheckNotEquals(expected, actual: integer; msg: string = '')`	Verifica se o valor do parâmetro "expected" é diferente do valor do parâmetro "actual"
`CheckNotNull(obj :IUnknown; msg :string = '')`	Verifica se o objeto passado como parâmetro possui instância
`CheckNull(obj: IUnknown; msg: string = '')`	Verifica se o objeto passado como parâmetro não está instanciado

Quadro 1 – Métodos do *framework* DUnit

No Quadro 2 tem-se uma classe `TNotas` em linguagem Object Pascal para efeito de exemplo. A classe efetua a média de quatro valores. A mesma recebe como entrada quatro valores assimilados pelos seus atributos e devolve a média dos valores pelo método `Media`.

```
type

  TNotas = class
  private
    FNota1: integer;
    FNota2: integer;
    FNota3: integer;
    FNota4: integer;
    procedure SetNota1(const Value: integer);
    procedure SetNota2(const Value: integer);
    procedure SetNota3(const Value: integer);
    procedure SetNota4(const Value: integer);
  public
    property Nota1: integer read FNota1 write SetNota1;
    property Nota2: integer read FNota2 write SetNota2;
    property Nota3: integer read FNota3 write SetNota3;
    property Nota4: integer read FNota4 write SetNota4;
    function Media: Real;
  end;

implementation

function TNotas.Media: Real;
begin
  Result := (Nota1 + Nota2 + Nota3 + Nota4)/4;
end;
```

Quadro 2 – Classe para cálculo de médias

No Quadro 3, tem-se um exemplo de uma típica classe para testes de unidade utilizando o *framework* DUnit para efetuar as devidas validações. O teste irá validar o método `Media` da classe exemplo `TNotas`.

Para efetuar o teste do método, o desenvolvedor precisa instanciar a classe que deseja testar e, neste exemplo, fazer as devidas atribuições dos valores. Em seguida, deve-se chamar algum método do DUnit visto no Quadro 1.

```
  TNotasTestes = class(TTestCase)
  private
    FNotas: TNotas;
  protected
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure TestMedia;
  end;

implementation

{ TNotasTestes }

procedure TNotasTestes.SetUp;
begin
  //metodo executado antes do metodo de testes
  fNotas:= TNotas.Create;
  inherited;
end;

procedure TNotasTestes.TearDown;
begin
  //metodo executado apos o metodo de testes
  fNotas.Free;
  Inherited;
end;

procedure TNotasTestes.TestMedia;
begin
  FNotas.Nota1:= 10;
  FNotas.Nota2:= 10;
  FNotas.Nota3:= 10;
  FNotas.Nota4:= 10;
  CheckEquals(10, FNotas.Media, 'Erro no cálculo da Média');
end;

initialization
  TestFramework.RegisterTest('Tests Suite', TNotasTestes.Suite);
end.
```

Quadro 3 – Classe para testar o método `Media`

No caso visto no Quadro 3, o retorno do teste será de sucesso, pois o caso de teste valida a média dos quatro atributos. Essa validação é feita pelo método `CheckEquals`, o qual verifica se o retorno da classe instanciada `FNotas` é efetivamente o valor dez. O resultado da execução do método pode ser visto na figura 5.

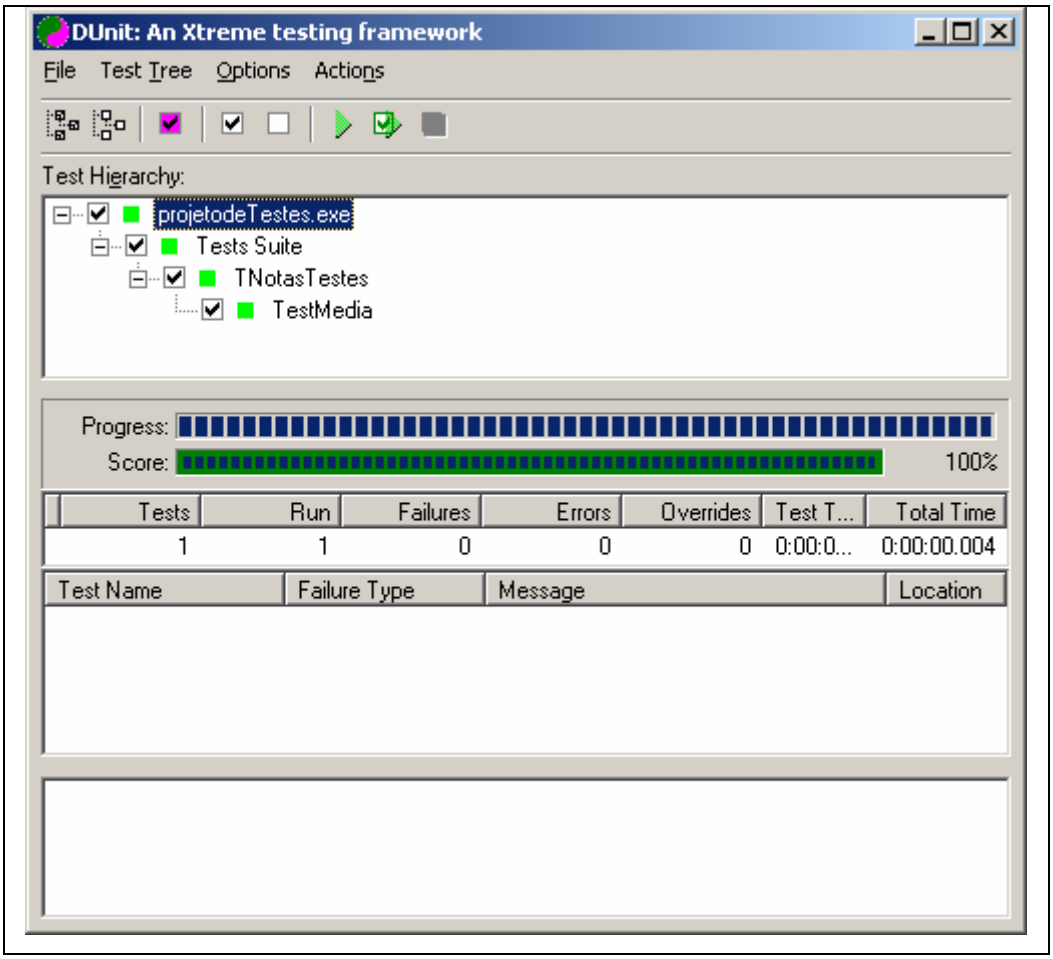


Figura 5 – Resultado da execução do método `TestMédia`

4.2 TRABALHOS CORRELATOS

Em Oliveira (2006) é proposto um protótipo de um *weaver* para POA em Delphi. A ferramenta propõe-se a gerar código na linguagem Object Pascal com suporte a POA. O *weaver* (OLIVEIRA, 2006) tem por objetivo complementar a programação em Object Pascal, já que o ambiente Delphi não possui suporte a POA nativo. O trabalho foi inspirado no AspectJ, um *weaver* bastante conhecido e difundido no mundo Java.

A ferramenta recebe um ou mais programas em uma linguagem de aspectos, e um ou mais programas fontes na linguagem Object Pascal. Realiza então análise léxica e sintática dos programas fontes de aspectos e permite o desenvolvedor especificar os pontos de junção, que são os locais onde serão executados os códigos de aspecto.

Por fim, a ferramenta gera programas na linguagem Object Pascal, mesclando as

funcionalidades dos programas fornecidos como entrada.

Silva (2005) faz um apanhado geral da tecnologia de POO e POAÉ visto sobre padrões de projetos aplicados a POO e a inserção da técnica de POA na engenharia de software. São apresentados detalhes teóricos do que é um aspecto e seus componentes. Por fim é elaborado um estudo de caso comparativo de uma implementação utilizando-se técnicas de POO em relação às técnicas de POA.

Ayroso (1998) analisa os vários métodos de testes e estabelece critérios de comparação entre eles. O mesmo levanta critérios para interpretar os resultados de um método de teste e avalia a importância dos testes de software no desenvolvimento de sistemas. É realizada uma revisão sobre os métodos de testes para sistemas baseados em banco de dados. Por fim é realizada uma análise geral dos métodos de testes estudados estabelecendo ligações entre eles.

5 DESENVOLVIMENTO

Neste capítulo é feita a apresentação das extensões adicionadas a ferramenta AOPDelphi (OLIVEIRA, 2006). Inicialmente são apresentados os requisitos principais e em seguida sua especificação e a operacionalidade da implementação.

5.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O AOPDelphi (OLIVEIRA, 2006) é uma ferramenta voltada ao desenvolvedor com o intuito de prover suporte à POA para a linguagem Object Pascal. As extensões adicionadas fornecem ao desenvolvedor um ambiente para codificação de rotinas de testes unitários para os aspectos implementados, além dos métodos do programa de componente afetados pelos aspectos.

A ferramenta AOPDelphi (OLIVEIRA, 2006) deve possuir um ambiente para codificação de testes dos aspectos além de um ambiente para codificação de testes das classes afetadas. A mesma pode receber como entrada, rotinas de testes previamente codificadas.

A ferramenta deve fazer análise léxica, sintática e semântica das rotinas de testes informadas como entrada no ambiente. Como saída, a ferramenta gera um projeto Delphi para os testes unitários dos aspectos e um projeto Delphi para os testes unitários dos métodos afetados pelos aspectos.

A seguir, estão relacionados os Requisitos Funcionais (RF), Requisitos Não Funcionais (RNF) e Regras de Negócio (RN):

a) realizar análise léxica e sintática da linguagem de definição de testes (RF);
b) receber como entrada uma ou mais classes na linguagem Object Pascal e um ou mais códigos fontes na linguagem que será criada para a definição dos testes (RF);
c) gerar código na linguagem Object Pascal com os casos de testes (RF);
d) disponibilizar um ambiente para a codificação dos casos de testes de unidade (RF);
e) ser implementado utilizando o ambiente Delphi 7 (RNF);
f) os códigos fontes na linguagem Object Pascal que servirão de entrada na ferramenta deverão estar implementados com POO (RN).

5.2 ESPECIFICAÇÃO

Nos tópicos seguintes é apresentada a especificação da ferramenta. Primeiramente é apresentada a especificação da linguagem criada para a definição dos testes unitários. Em seguida é apresentada a especificação das extensões adicionadas à ferramenta através dos diagramas de casos de uso, atividades e classes.

5.2.1 Especificação da linguagem de testes

A linguagem criada para a definição dos testes unitários é baseada na sintaxe do Object Pascal. Sua sintaxe e seu funcionamento são semelhantes e, assim como o Object Pascal, a linguagem é não *case sensitive.*

Para a especificação da linguagem foi utilizada a ferramenta GALS (GESSER, 2003). As definições regulares, *tokens*, e a gramática da linguagem foram definidas nesta ferramenta.

Na definição da gramática foi utilizada a notação BNF (*Backus-Naur Form*). O quadro 4 apresenta a gramática da linguagem de definição de testes unitários.

Definições Regulares

```
IDENTIFICADOR: (<LETRA> | "_") (<LETRA> | <DIGITO> | "_")*
CODIGO: .
LETRA: [a-zA-Z]
DIGITO: [0-9]
```

BNF

```
<programa>::= Test IDENTIFICADOR ; <corpo> End.
<corpo>::= <uses> <declaration> <setup> <teardown> <teste>
<uses>::= Uses <lista_uses>; | ε
<lista_uses>::= IDENTIFICADOR <lista_uses2>
<lista_uses2>::= , <lista_uses> | ε
<declaration>::= Declaration <declaration1> End; | ε
<declaration1>::= <declaration2> : IDENTIFICADOR ; <declaration4>
<declaration2>::= IDENTIFICADOR <declaration3>
<declaration3>::= , <declaration2> | ε
<declaration4>::= <declaration1> | ε
<setup>::= <codigo_setup> ; | ε
<codigo_setup>::= Setup (<codigo>)* EndSetup
<teardown>::= <codigo_teardown> ; | ε
<codigo_teardown>::= Teardown (<codigo>)* EndTeardown
<teste>::= <teste2> | ε
<teste2>::= Test IDENTIFICADOR ; <codigo_teste> ; <lista_teste>
<codigo_teste>::= BeginTest (<codigo>)* EndTest
<lista_teste>::= <teste2> | ε
<codigo>::= CODIGO
```

Quadro 4 – BNF da linguagem de definição de testes unitários

5.2.2 Especificação da ferramenta

A ferramenta foi especificada utilizando a *Unified Modeling Language* (UML) como linguagem de modelagem. A seguir é apresentado o diagrama de casos de uso, atividades e de classes, que foram concebidos na ferramenta Enterprise Architect.

5.2.2.1 Diagrama de casos de uso

A ferramenta possui sete casos de uso associados a dois atores: Desenvolvedor e Compilador Delphi. Desses sete casos de uso, três referem-se à implementação dos aspectos e quatro referem-se à implementação dos casos de testes unitários.

Os casos de uso que se referem à implementação dos aspectos são: Implementar programas de componentes; Implementar programas de aspectos; e Compilar projeto, sendo que os mesmos estão destacados na cor cinza. Estes casos de uso foram especificados no trabalho desenvolvido por Oliveira (2006). Como este trabalho é uma continuação do trabalho de Oliveira (2006), os casos de uso por ele desenvolvidos não serão explanados em profundidade.

O caso de uso Implementar programas de componentes consiste no desenvolvimento de um projeto escrito na linguagem Objetc Pascal, com técnicas de POO. Este projeto é fornecido como entrada na ferramenta para ser aspectado.

O caso de uso Implementar programas de aspectos é baseado na implementação dos programas de aspectos na linguagem criada por Oliveira (2006).

O caso de uso Compilar Projeto de Aspecto é a principal funcionalidade do *weaver* (OLIVEIRA, 2006). O desenvolvedor especifica os parâmetros de entrada, ou seja, seleciona um projeto Delphi (programa de componentes) além de selecionar os aspectos. O programa de componentes terá suas classes afetadas pelos aspectos assim que o desenvolvedor executar o comando *weaving*.

Os casos de uso que se referem à implementação dos casos de testes unitários são: Implementar testes de aspectos; Implementar testes de classes aspectadas; Compilar projeto de testes de aspectos; e Compilar projeto de testes de classes aspectadas.

Na figura 6 é apresentado o diagrama de casos de uso e nas figuras 7, 8, 9 e 10 são apresentados em detalhes os casos de uso Implementar testes de aspectos, Implementar testes

de classes aspectadas, Compilar projeto de testes de aspectos, e Compilar projeto de testes de classes aspectadas respectivamente. Para um maior aprofundamento nos demais casos de uso, deve-se consultar o trabalho de Oliveira (2006).

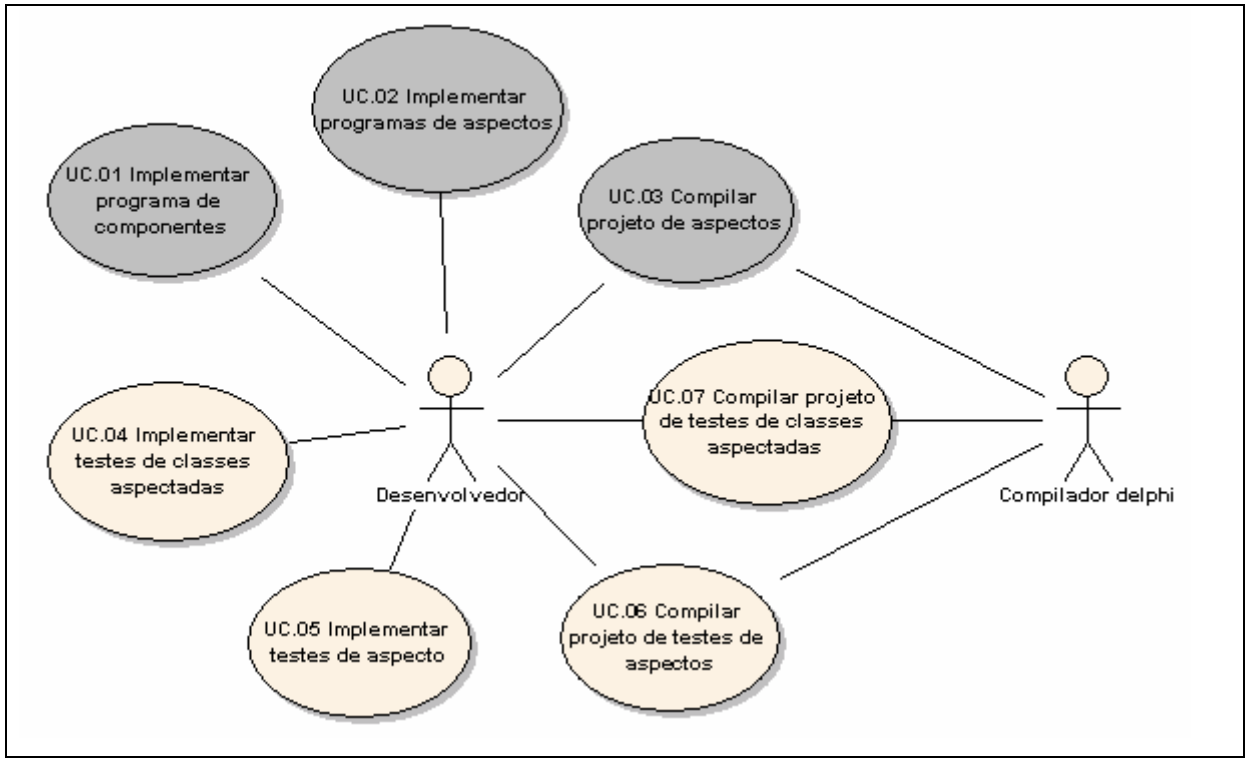
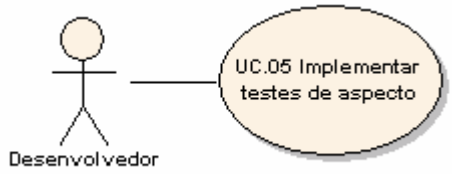
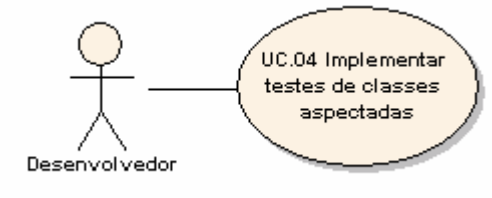


Figura 6 – Diagrama de casos de uso da ferramenta



Descrição	- A ferramenta disponibiliza um ambiente com as informações dos aspectos compilados como referência para o desenvolvedor. - A ferramenta disponibiliza um ambiente para codificação dos testes de aspectos.
Ator	Desenvolvedor
Requisitos atendidos	- As rotinas de testes devem ser escritas em linguagem própria. - A ferramenta deve receber como entrada uma ou mais rotinas de testes de aspectos. - Ferramenta deve possuir um ambiente para implementação das rotinas de testes de aspectos.
Pós-condições	- Rotinas de testes de aspectos podem ser fornecidas como entrada na ferramenta.

Figura 7 – Caso de uso implementar testes de aspectos

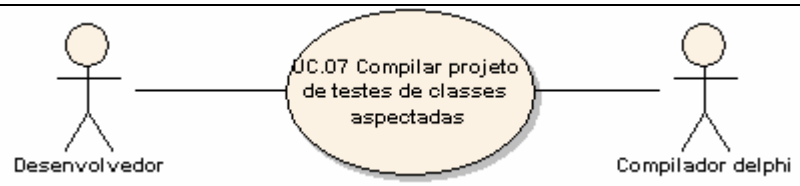


Descrição	- A ferramenta disponibiliza um ambiente com as informações das classes do programa de componentes que foram afetadas pelos aspectos como referência para o desenvolvedor. - A ferramenta disponibiliza um ambiente para codificação dos testes de classes aspectadas.
Ator	Desenvolvedor
Requisitos atendidos	- As rotinas de testes devem ser escritas em linguagem própria. - A ferramenta deve receber como entrada uma ou mais rotinas de testes de classes aspectadas. - A ferramenta deve possuir um ambiente para implementação das rotinas de testes de classes aspectadas.
Pós-condições	- Rotinas de testes de classes aspectadas podem ser fornecidas como entrada na ferramenta.

Figura 8 – Caso de uso implementar testes de classes aspectadas

Descrição	- A ferramenta realiza análise léxica, sintática e semântica das rotinas de testes informadas como entrada. - A ferramenta gera um projeto Delphi com as classes de testes unitários para os testes de aspectos.
Atores	Desenvolvedor e Compilador Delphi
Requisitos atendidos	- Realizar análise léxica, sintática e semântica das rotinas de testes de aspectos.
Pré-condições	- Deve existir a pasta Base no diretório corrente da ferramenta, com os arquivos .xml que armazenam os projetos. - Deve existir a pasta Saída no diretório corrente da ferramenta, que é a pasta onde será gerado o projeto após o processo de *weaving*. - Deve existir a pasta ProjetoTeste com a sub-pasta TestesAspectos que é a pasta onde será gerado os projetos de testes de aspectos
Pós-condições	É gerado um projeto Delphi com as classes de testes unitários com suporte do framework DUnit para testes de aspectos.

Figura 9 – Caso de uso compilar projeto de testes de aspectos



Descrição	- A ferramenta realiza análise léxica, sintática e semântica das rotinas de testes informadas como entrada. - A ferramenta gera um projeto Delphi com as classes de testes unitários para os testes de classes aspectadas.
Atores	Desenvolvedor e Compilador Delphi
Requisitos atendidos	- Realizar análise léxica, sintática e semântica das rotinas de testes de classes aspectadas.
Pré-condições	- Deve existir a pasta Base no diretório corrente da ferramenta, com os arquivos .xml que armazenam os projetos. - Deve existir a pasta Saída no diretório corrente da ferramenta, que é a pasta onde será gerado o projeto após o processo de *weaving*. - Deve existir a pasta a pasta ProjetoTeste com a sub-pasta TestesClasses que é a pasta onde será gerado os projetos de testes de classes aspectadas.
Pós-condições	É gerado um projeto Delphi com as classes de testes unitários com suporte do framework DUnit para testes de classes aspectadas.

Figura 10 – Caso de uso Compilar projeto de testes de classes aspectadas

5.2.2.2 Diagrama de atividades

No diagrama de atividades estão especificadas as funcionalidades da ferramenta de forma geral. Ele está dividido em duas raias, representadas pelo desenvolvedor e pela ferramenta, com suas respectivas atividades sendo as atividades em cor cinza, as atividades definidas por Oliveira (2006) em seu trabalho. Na figura 11 é apresentado o diagrama de atividades da ferramenta. A descrição detalhada do diagrama é apresentada no Quadro 5.

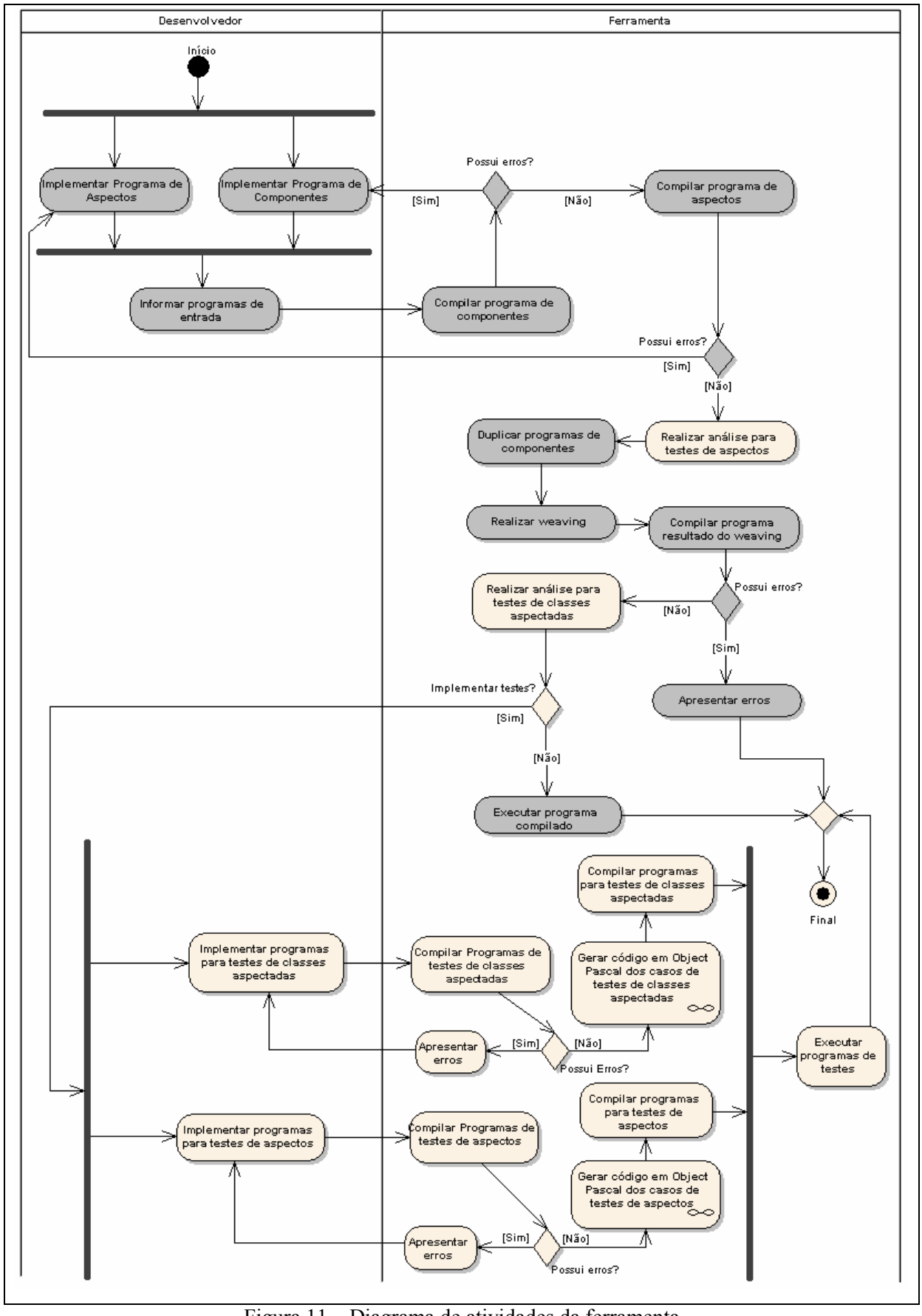


Figura 11 – Diagrama de atividades da ferramenta

Atividade	Raia	Descrição
Implementar programas de componentes	Desenvolvedor	O desenvolvedor implementa programas de componentes em Object Pascal. Este programa será dado como entrada na ferramenta.
Implementar programas de aspectos	Desenvolvedor	O desenvolvedor implementa programas de aspectos na linguagem própria da ferramenta.
Informar programas de entrada	Desenvolvedor	O desenvolvedor informa a ferramenta um ou mais programas de aspectos e um projeto Delphi. Pode informar ainda uma lista de units que não serão afetadas pelos aspectos.
Compilar programas de componentes	Ferramenta	A ferramenta compila o programa de componentes para certificar que o mesmo não possui erros.
Compilar programas de aspectos	Ferramenta	A ferramenta compila os programas de aspectos fornecidos como entrada para verificar se o mesmo possui erros.
Realizar análise para testes de aspectos	Ferramenta	Durante a compilação dos programas de aspectos, a ferramenta analisa o código de aspecto compilado e gera uma estrutura de dados com as informações necessárias para posteriormente o desenvolvedor implementar as rotinas de testes de aspectos.
Duplicar programas de componentes	Ferramenta	A ferramenta faz uma cópia de todos os arquivos fontes do projeto fornecido como entrada. As alterações feitas pelo *weaver* (OLIVEIRA, 2006) serão executadas sobre este projeto cópia.
Realizar *weaving*	Ferramenta	Baseado nos arquivos de aspectos fornecidos como entrada, a ferramenta localiza os *joinpoint* que devem ser interceptados e faz a junção do *advice*.
Compilar programa resultado do *weaving*	Ferramenta	A ferramenta compila o projeto resultado do *weaving* para certificar que o mesmo não possui erros.
Executar programa compilado	Ferramenta	Não havendo erros de compilação do projeto, a ferramenta executa o programa gerado.
Apresentar erros do resultado do *weaving*	Ferramenta	Havendo erros de compilação no projeto após o *weaving*, a ferramenta apresenta os erros e sua origem. A ferramenta distingue se o erro está na implementação do *advice* ou na junção dos mesmos no programa de componentes.
Realizar análise para testes de classes aspectadas	Ferramenta	Durante o processo de *weaving*, a ferramenta analisa o código aspectado e gera uma estrutura de dados interna com as informações das classes aspectadas para posteriormente o desenvolvedor implementar as rotinas de testes de classes aspectadas.
Implementar programas para testes de classes aspectadas	Desenvolvedor	O desenvolvedor implementa rotinas de testes unitários na linguagem própria da ferramenta para as classes aspectadas identificadas pela ferramenta.
Compilar programas de testes de classes aspectadas	Ferramenta	A ferramenta compila as rotinas de testes para certificar que a mesma não possui erros.
Apresentar erros	Ferramenta	Caso a compilação encontre erros, a ferramenta exibe ao desenvolvedor o erro e a categoria do erro (léxico, sintático ou semântico).
Gerar código em Object Pascal dos casos de testes de classes aspectadas	Ferramenta	Caso não haja erros nas rotinas de testes, a ferramenta gera os projetos Delphi e suas respectivas classes com as rotinas de testes

		unitários de classes aspectadas.
Compilar código em Object Pascal para testes de classes aspectadas	Ferramenta	Os projetos gerados são compilados usando o compilador do próprio Delphi.
Implementar programas para testes de aspectos	Desenvolvedor	O desenvolvedor implementa rotinas de testes unitários na linguagem própria da ferramenta para os aspectos identificados pela ferramenta.
Compilar programas de testes de aspectos	Ferramenta	A ferramenta compila as rotinas de testes para certificar que a mesma não possui erros.
Apresentar erros	Ferramenta	Caso a compilação encontre erros, a ferramenta exibe ao desenvolvedor o erro e a categoria do erro (léxico, sintático ou semântico).
Gerar código em Object Pascal dos casos de testes de aspectos	Ferramenta	Caso não haja erros nas rotinas de testes, a ferramenta gera os projetos Delphi e suas respectivas classes com as rotinas de testes unitários de aspectos.
Compilar código em Object Pascal para testes de classes aspectadas	Ferramenta	Os projetos gerados são compilados usando o compilador do próprio Delphi.
Executar programas de testes	Ferramenta	Por fim, a ferramenta executa todos os projetos compilados com as rotinas de testes.

Quadro 5 – Descrição das atividades

5.2.2.2.1 Diagrama de atividade dos processos de geração de código

Dentre as atividades apresentadas no diagrama ilustrado na figura 11, destacam-se as atividades de geração de código. Os diagramas de atividade dos processos de geração de código Object Pascal com os testes de aspectos e geração de código Object Pascal com os testes de classes aspectadas são ilustrados na figura 12.

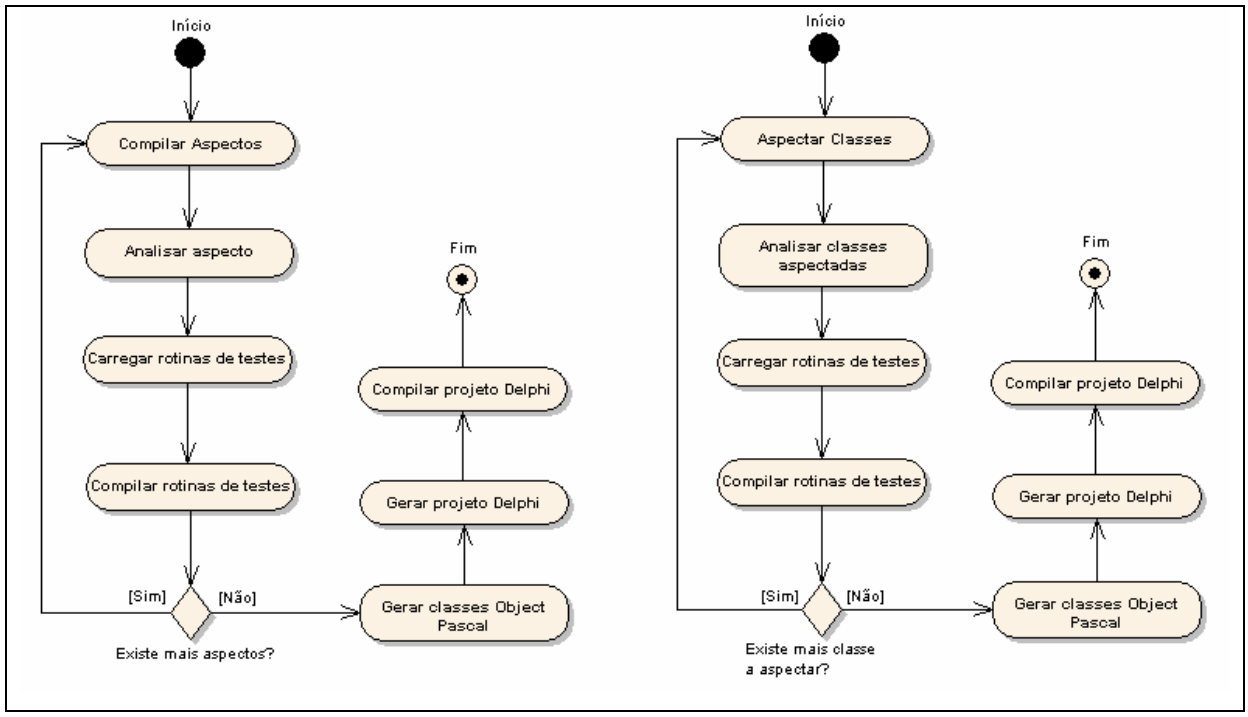


Figura 12 – Diagramas de atividades dos processos de geração de código

Para o processo de geração de código em Object Pascal para testes de aspectos, inicialmente a ferramenta compila as rotinas de aspectos. Durante o processo de compilação, o *parsing* do compilador retorna à ferramenta os *tokens* capturados. Com esses *tokens*, as estruturas de dados necessárias à geração de código são criadas e preenchidas. Em seguida, a ferramenta carrega as respectivas rotinas de testes previamente salvas (caso exista) e as compila. Após a compilação, são criadas as classes e adicionadas ao projeto Delphi. Por fim, o projeto Delphi é compilado pelo próprio compilador Delphi, criando assim um executável com os casos de testes unitários.

O processo de geração de código em Object Pascal para os testes de classes aspectadas é semelhante. A diferença fica na etapa da criação e preenchimento das estruturas de dados, estas preenchidas durante o processo de *weaving*. É durante o processo de *weaving* que a ferramenta descobre quais são as classes e métodos afetados pelos aspectos. Todo processo a seguir para geração de código, é idêntico ao processo para testes de aspectos.

5.2.2.3 Diagrama de classes

Na especificação da linguagem de testes da ferramenta foi utilizada a ferramenta GALS (GESSER, 2003). Esta ferramenta gera as classes necessárias para realizar as análises léxicas e sintáticas. Estas classes estão representadas na figura 13.

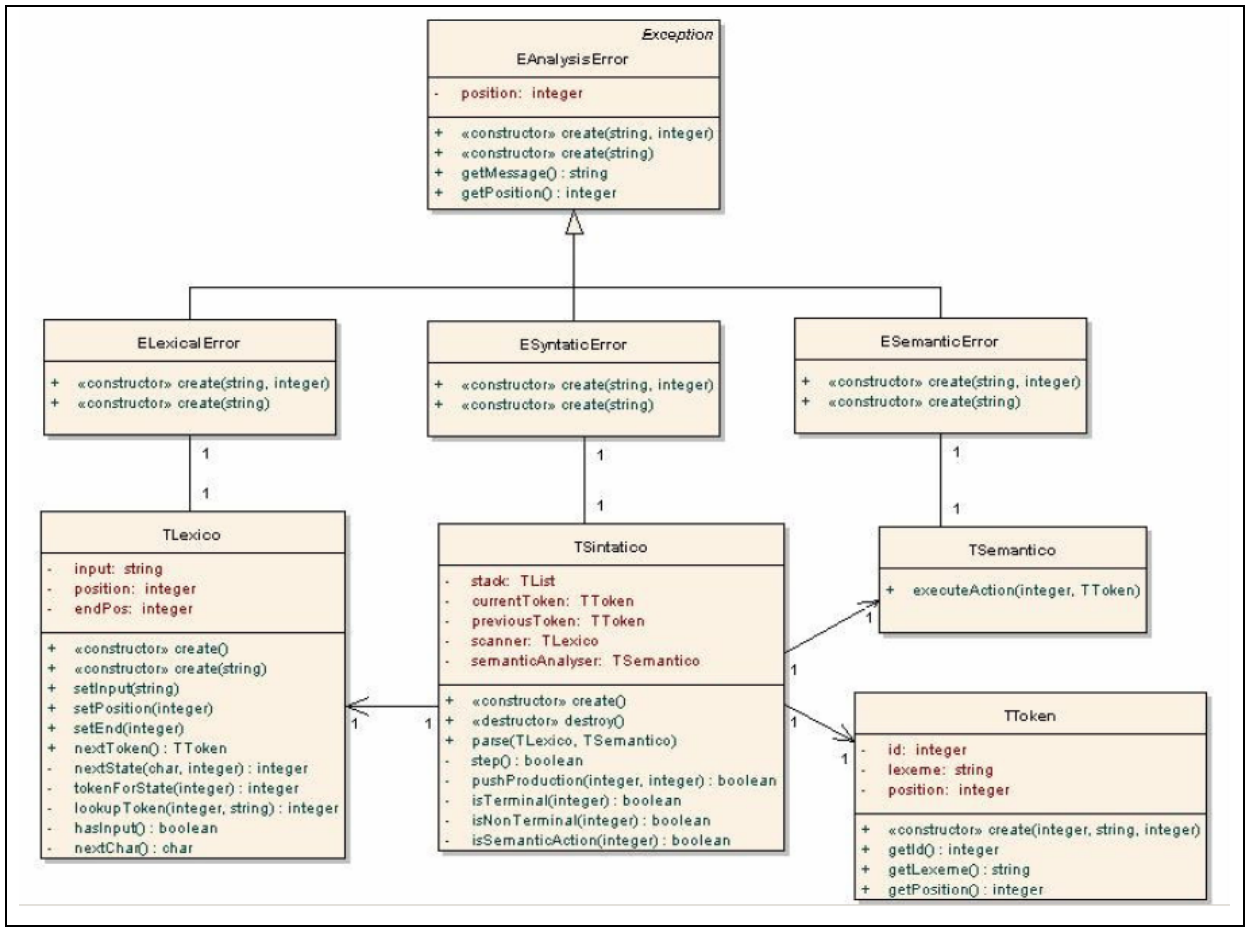


Figura 13 – Classes geradas pelo GALS (GESSER, 2003)

Das classes geradas pela ferramenta, a única que sofre modificações é a classe `TSemantico`, pois a finalidade do GALS (GESSER, 2003) é gerar analisadores léxicos e sintáticos, deixando a cargo do desenvolvedor implementar o analisador semântico.

Na figura 14 estão representadas as classes físicas criadas para o desenvolvimento da ferramenta. A classe `TSemantico` aparece novamente, desta vez com as alterações implementadas para atender as necessidades da ferramenta. Como a ferramenta possui duas linguagens especificadas, uma linguagem de definição de aspectos e outra para definição de testes unitários, a mesma possui duas classes de análise semântica representadas no diagrama.

Para a linguagem de definição de testes unitários, a classe de análise semântica implementada é a classe `TSemanticoTestes` para que não haja conflitos com a análise semântica da linguagem de definição de aspectos.

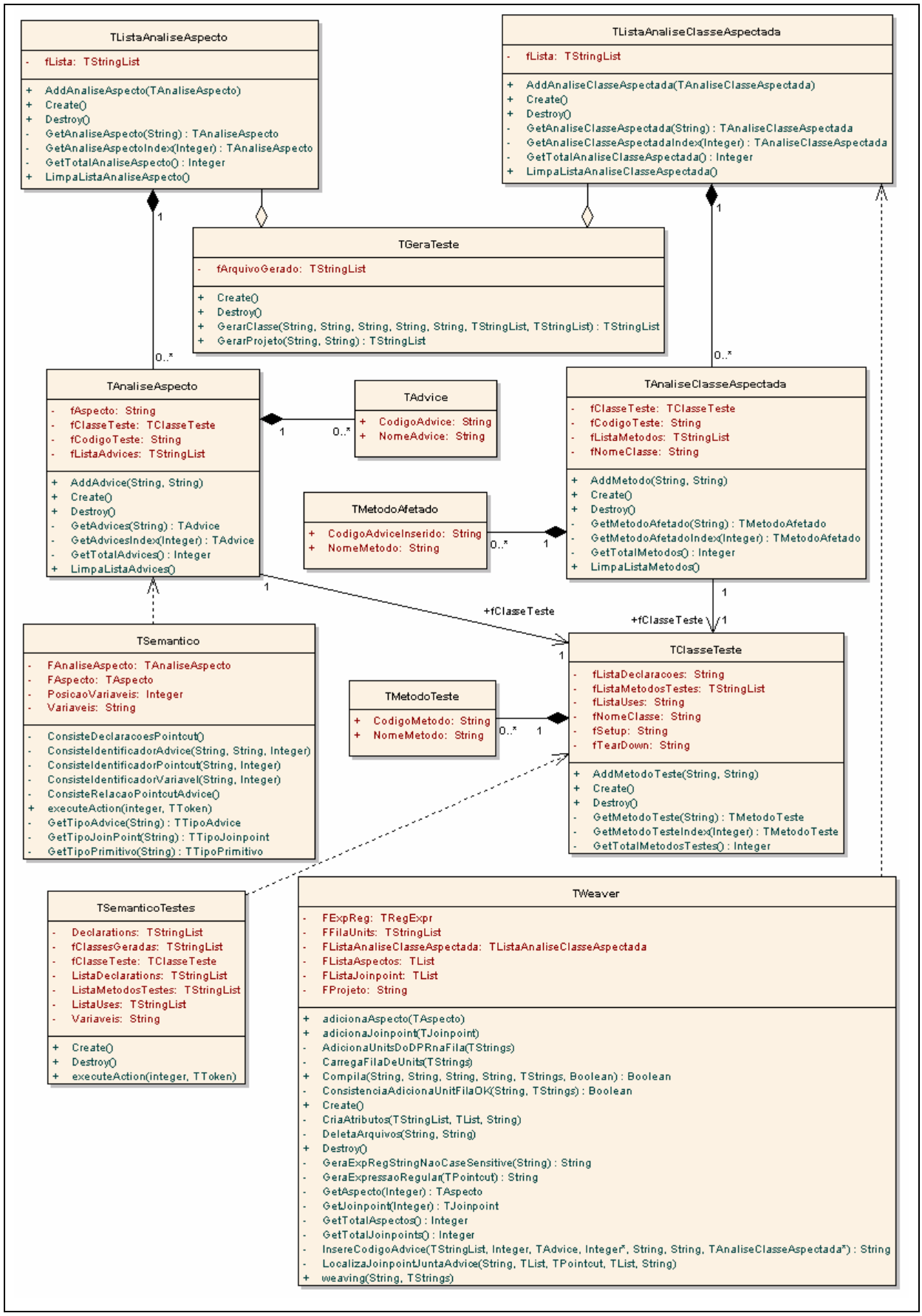


Figura 14 – Diagrama de classes da ferramenta

Na compilação de um projeto AOPDelphi (OLIVEIRA, 2006), para cada programa de aspecto compilado, é instanciado um objeto `TAnaliseAspecto`. Como um aspecto pode conter zero ou mais *Advices*, o objeto `TAnaliseAspecto` instancia, para cada *Advice,* um objeto `TAdvice` e guarda a mesma em uma lista. Todos os objetos `TAnaliseAspecto` são armazenadas em uma lista do objeto `TListaAnaliseAspecto` criando assim, uma estrutura de dados em forma de árvore. O objeto `TAnaliseAspecto` possui uma composição para cada objeto `TAdvice` instanciado e o mesmo ocorre com o objeto `TListaAnaliseAspecto` que, por sua vez, também possui uma composição para cada objeto `TAnaliseAspecto` instanciado.

Um processo semelhante ocorre quando a ferramenta faz o *weaving*. Para cada classe do programa de componentes aspectada pelo processo de *weaving*, é instanciado um objeto `TAnaliseClasseAspectada`. Como diversos métodos de uma classe podem ser afetados pelo aspecto, o objeto `TAnaliseClasseAspectada` instancia, para cada método afetado, um objeto `TMetodoAfetado` e guarda a mesma em uma lista. Todos os objetos `TAnaliseClasseAspectada` são armazenadas em uma lista do objeto `TListaAnaliseClasseAspectada` criando assim, uma outra estrutura de dados em forma de árvore. O objeto `TAnaliseClasseAspectada` possui uma composição para cada objeto `TMetodoAfetado` instanciado e o mesmo ocorre com o objeto `TListaAnaliseClasseAspectada` que, por sua vez, também possui uma composição para cada objeto `TAnaliseClasseAspectada` instanciado.

Após a compilação dos programas de aspectos e do processo de *weaving*, o desenvolvedor deve implementar as rotinas de testes de aspectos e classes aspectadas. Após a implementação, é feita a compilação das rotinas de testes. Quando o compilador faz o *parsing*, é invocado o método `executeAction` do objeto `TSemanticoTestes` para cada ação semântica definida na gramática da linguagem.

Durante o *parsing*, o compilador identifica nas rotinas de testes os *uses,* o nome da classe de testes, o método *Setup* e *TearDown* e assim por diante. Para cada rotina de testes compilada, a ferramenta instancia um objeto `TClasseTeste` e gera a composição da mesma com o objeto `TAnaliseAspecto` ou `TAnaliseClasseAspectada`, dependendo do tipo de teste que o desenvolvedor está criando (teste de aspecto ou teste de classe aspectada).

Após o *parsing*, a ferramenta instancia o objeto `TGeraTeste`, que é o objeto responsável pela criação dos fontes e projetos em Object Pascal. Este objeto tem por função varrer as listas dos objetos `TListaAnaliseAspecto` e `TListaAnaliseClasseAspectada`. Este objeto gera os programas fontes com os testes em Object Pascal e, posteriormente, a

ferramenta compila os projetos criados para efetivamente rodar os testes unitários.

5.3 IMPLEMENTAÇÃO

Nesta seção são abordados os aspectos sobre a implementação da ferramenta, as ferramentas utilizadas para a construção e seu funcionamento.

5.3.1 Ferramentas utilizadas

A ferramenta foi implementada utilizando a linguagem Object Pascal no ambiente Borland Delphi 7. No Quadro 6 é apresentado um trecho do código fonte da ferramenta. O código se refere a interface da classe `TClasseTeste` e sua composição `TMetodoTeste`.

```
TMetodoTeste = Class
  NomeMetodo,
  CodigoMetodo: String;
End;

TClasseTeste = Class
private
  fNomeClasse,
  fListaUses,
  fTearDown,
  fSetup,
  fListaDeclaracoes: String;
  fListaMetodosTestes: TStringList;
  Function GetMetodoTeste(Index: String): TMetodoTeste;
  Function GetTotalMetodosTestes: Integer;
  Function GetMetodoTesteIndex(Index: Integer): TMetodoTeste;
public
  Constructor Create;
  Destructor Destroy; Override;
  Procedure AddMetodoTeste(NomeMetodo, CodigoMetodo: String);
  Property NomeClasse        : String  Read fNomeClasse       Write fNomeClasse;
  Property ListaUses         : String  Read fListaUses        Write fListaUses;
  Property ListaDeclaracoes  : String  Read fListaDeclaracoes Write fListaDeclaracoes;
  Property Setup             : String  Read fSetup            Write fSetup;
  Property TearDown          : String  Read fTearDown         Write fTearDown;
  Property TotalMetodosTestes: Integer Read GetTotalMetodosTestes;
  Property MetodoTeste[Index: String]      : TMetodoTeste Read GetMetodoTeste;
  Property MetodoTesteIndex[Index: Integer]: TMetodoTeste Read GetMetodoTesteIndex;
end;
```

Quadro 6 – Interface da classe `TClasseTeste` e `TMetodoTeste`

Na implementação do compilador da linguagem de definição de testes foi utilizada também a ferramenta GALS (GESSER, 2003). O GALS é um gerador de analisadores léxico e sintático. Nele foram gerados as classes para a implementação do analisador léxico e sintático além da interface para o analisador semântico. A implementação do analisador semântico fica a cargo do desenvolvedor.

As classes geradas pelo GALS levam em consideração as definições regulares nele especificados além das palavras reservadas, símbolos especiais, gramática e outras informações definidas como entrada na ferramenta. No Quadro 7, é apresentada a interface da classe `TLexico` gerada pelo GALS.

```
TLexico = class
public
    constructor create; overload;
    constructor create(input : string); overload;

    procedure setInput(input : string);
    procedure setPosition(pos : integer);
    procedure setEnd(endPos : integer);
    function nextToken : TToken; //raises ELexicalError

private
    input : string;
    position : integer;
    endPos : integer;

    function nextState(c : char; state : integer) : integer;
    function tokenForState(state : integer) : integer;
    function lookupToken(base : integer; key : string) : integer;

    function hasInput : boolean;
    function nextChar : char;
end;
```

Quadro 7 – Interface da classe `TLexico`

5.3.2 Operacionalidade da implementação

Ao ser executada a ferramenta, a mesma verifica a existência e a necessidade da criação dos diretórios para o seu funcionamento. A estrutura dos diretórios está representada na figura 15.

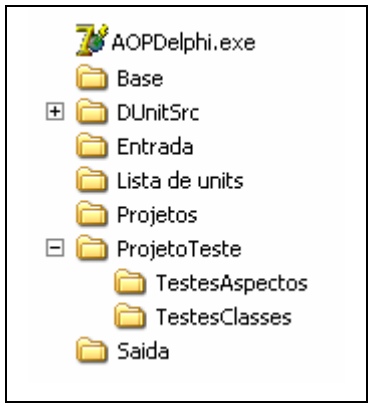


Figura 15 – Estrutura de diretórios da ferramenta

A pasta Base armazena os arquivos XML utilizados pela aplicação. Nesta pasta são salvos os arquivos Aspectos.xml, ListaUnits.xml e Projetos.xml que servem para armazenar as informações dos projetos em uso. Esta pasta não é criada automaticamente pela ferramenta, mas é de fundamental importância para o funcionamento da mesma. Caso a pasta não exista, a ferramenta faz um alerta e aborta a aplicação.

A pasta DUnitSrc contém o *framework* de classes do DUnit. Esta pasta contém todas as classes em Object Pascal do *framework* necessários à compilação dos projetos de testes unitários. Assim como a pasta Base, esta pasta também é de fundamental importância e não é criada automaticamente. Caso a pasta não exista, a ferramenta faz um alerta e aborta a aplicação. O *framework* DUnit pode ser encontrado em seu respectivo *site* e o endereço do mesmo pode ser encontrado no menu Sobre/Sobre.

A pasta Entrada guarda os arquivos de aspectos implementados além das rotinas de testes de aspectos e classes aspectadas. A ferramenta faz nesta pasta uma busca automática de programas testes implementados. Os arquivos de aspectos possuem extensão .dao e os arquivos com os testes, extensão .test.

A pasta Lista de *Units* é usada para armazenar os arquivos com as listas de classes que não serão afetadas por aspectos durante o processo de *weaving*.

A pasta Projetos é criada para armazenar os projetos AOPDelphi (OLIVEIRA, 2006) salvos. A ferramenta trabalha com conceito de projeto e este projeto é um conjunto de informações sendo os mesmos salvos neste diretório.

A pasta ProjetoTeste possui duas sub-pastas: TestesAspectos e TestesClasses. As respectivas pastas irão armazenar as classes e o projeto Delphi gerados pela ferramenta. A pasta TestesAspectos armazena os projetos e classes gerados para testes de aspectos e, a pasta TestesClasses, os projetos e classes para testes de classes aspectadas. O executável com os casos de testes implementados também serão salvos nestes diretórios, cada qual em seu

diretório específico.

A pasta Saída é utilizada para armazenar os arquivos fontes dos programas de componentes após o *weaving*.

Com exceção das pastas Base e DUnitSrc, todas as demais são criadas automaticamente pela ferramenta caso não existam. Quando executada, a ferramenta exibe a tela apresentada na figura 16.

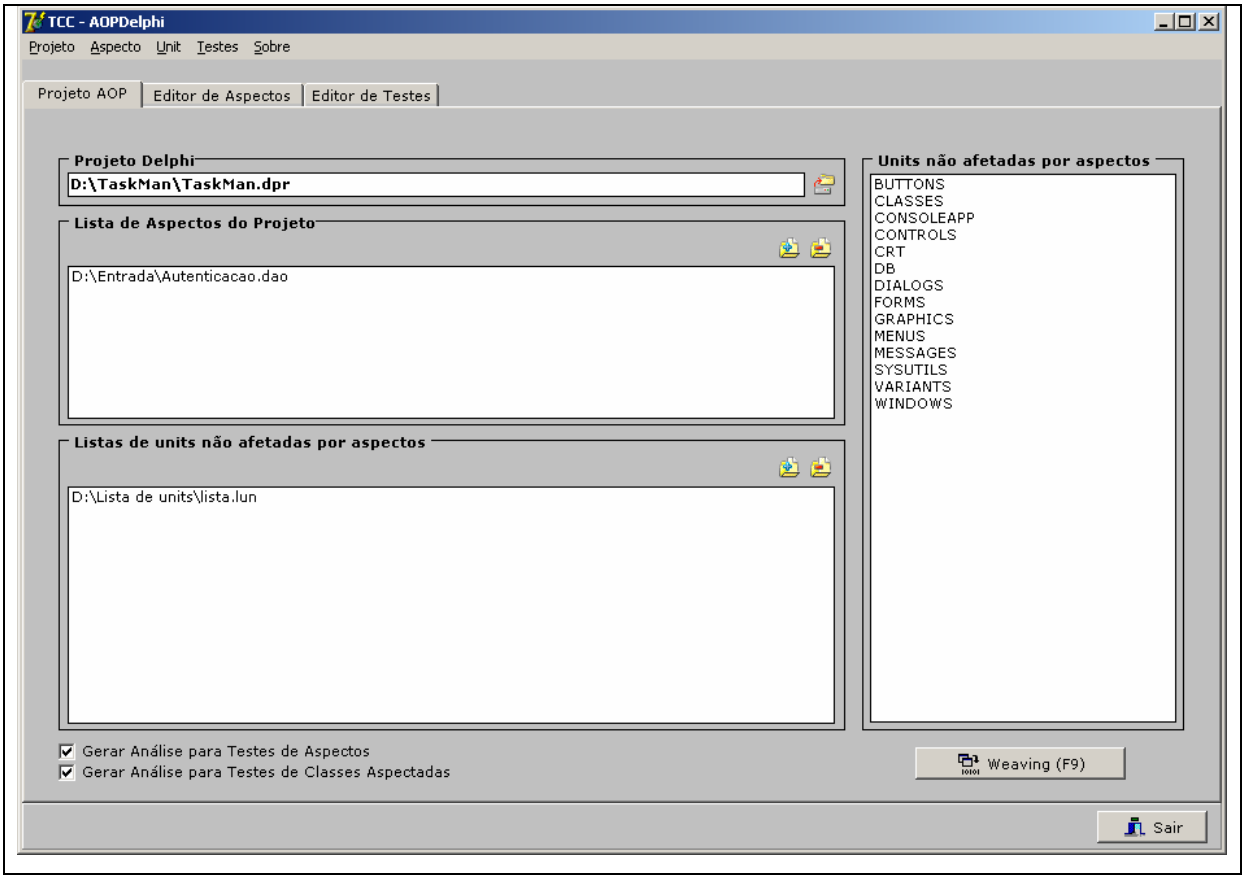


Figura 16 – Tela principal da ferramenta

A ferramenta é composta por três guias: Projeto AOP, Editor Aspectos e Editor Testes. Na guia Projeto AOP são informados os parâmetros do projeto que se quer aspectar. No campo Projeto Delphi é informado o caminho do projeto que se quer aspectar. Todas as *units* que fazem parte deste projeto serão analisadas e, levando em conta os programas de aspectos, aspectadas. Apenas as *units* informadas na lista de *units* não afetadas não serão analisadas e não serão afetadas pelos aspectos.

Na guia Projeto AOP existem ainda dois *checkboxes* que se referem à análise para geração dos casos de testes unitários. Caso estes *checkboxes* estejam habilitados, a ferramenta fará toda análise necessária dos aspectos e das classes aspectadas para que o desenvolvedor possa implementar os casos de testes.

Na guia Editor Aspectos é onde o desenvolvedor implementa os programas de aspectos

e está representada na figura 17. O ambiente possui comandos básicos para manipulação de arquivos como abrir, salvar, solicitar arquivo novo além da função compilar. Nesta guia ainda existe uma área destinada às mensagens de erros de compilação.

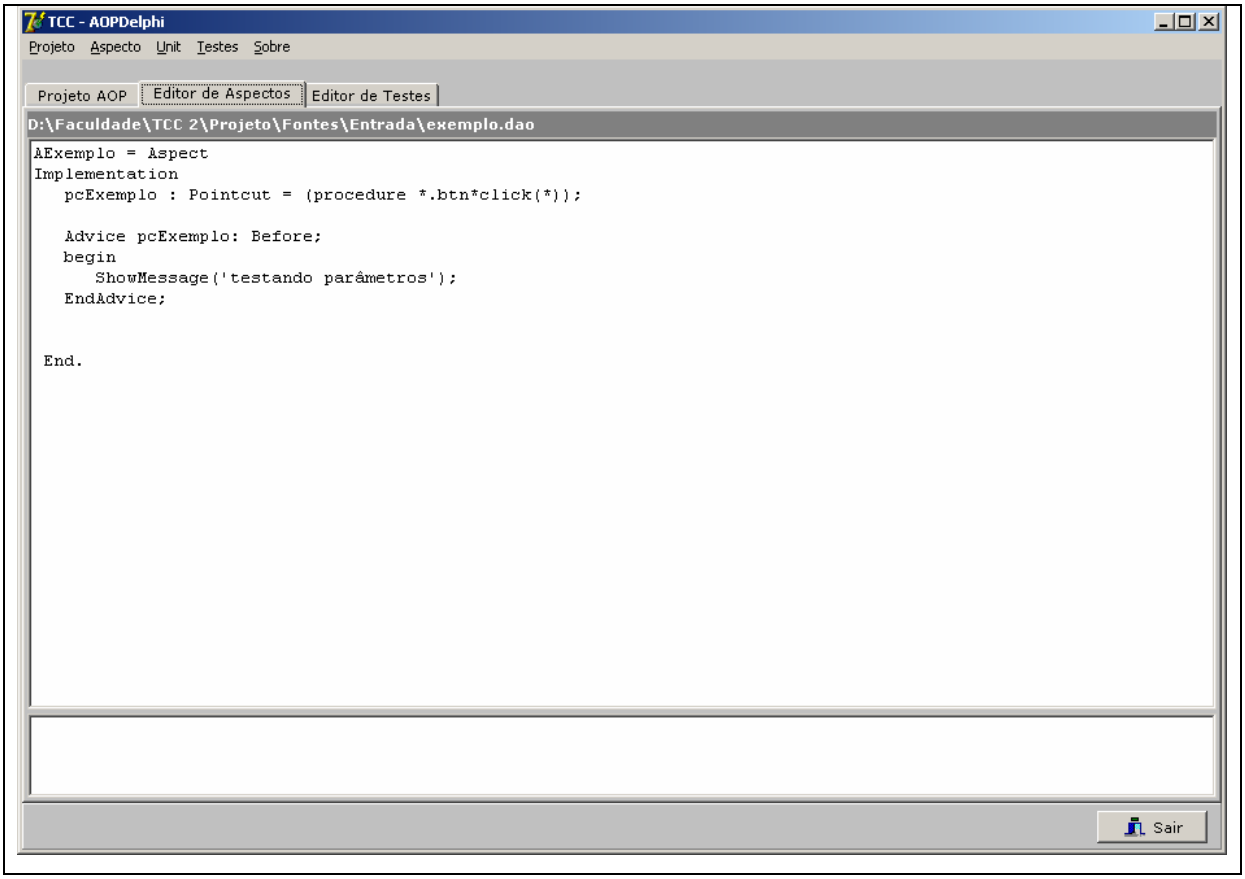


Figura 17 – Ambiente para programação dos aspectos

É na guia Editor de Testes onde o desenvolvedor implementa as rotinas de testes unitários. A guia é subdivida em outras duas guias: Testes de Aspectos e Testes de Classes Aspectadas. Cada guia se refere à implementação de um tipo de teste, testes de aspectos ou testes de classes aspectadas. A guia Editor de Testes é apresentada na figura 18.

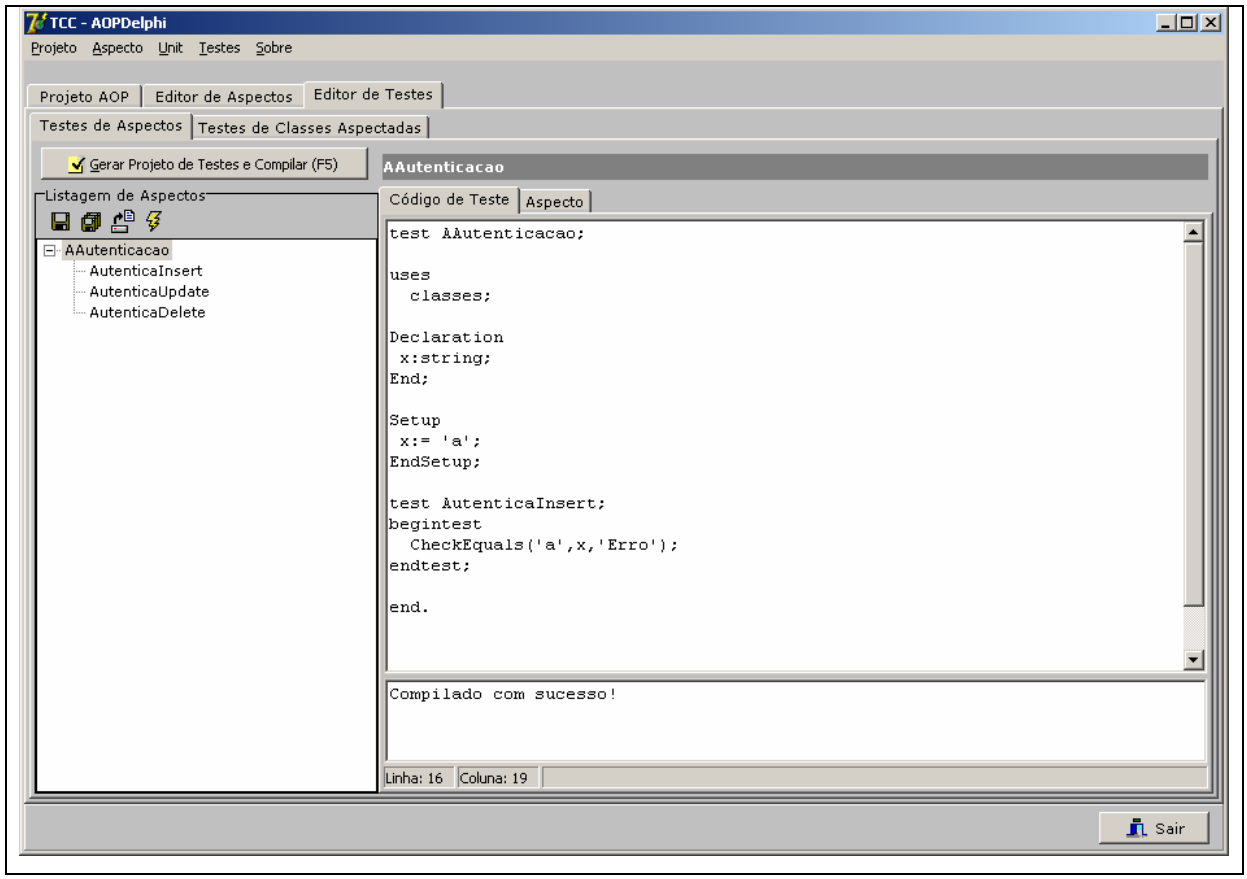


Figura 18 – Ambiente para implementação dos testes unitários

A figura 18 mostra o ambiente com um programa de testes de aspectos em edição. No programa exemplo está implementado um teste para o aspecto `AAutenticacao`, *advice* `AutenticaInsert`. O manual da linguagem de definição de testes está disponível no apêndice A deste trabalho.

O aspecto AAutenticacao refere-se a um aspecto de autenticação de usuários no sistema de componentes. Este aspecto visa restringir o acesso a determinadas funcionalidades do sistema de componentes por meio de autenticação com usuário e senha.

Após a implementação dos aspectos, o desenvolvedor compila as rotinas de aspectos usando o compilador criado no AOPDelphi (OLIVEIRA, 2006). Durante esta compilação, é feita uma análise das rotinas de aspectos para posteriormente ser desenvolvida as rotinas de testes de aspectos.

Após a análise, a ferramenta monta uma lista de aspectos e *advices* no *treeview* da guia de testes de aspectos. Este *treeview*, além de servir como guia de desenvolvimento de testes de aspectos, também serve para que o desenvolvedor crie rotinas de testes específicas para cada aspecto e *advice*Este *treeview* está representado na figura 18.

Este *treeview* serve como um menu para o desenvolvedor. Cada aspecto compilado pela ferramenta é representado nesta árvore em um nó pai e os *advices* são representados

como nós filhos. Na árvore da figura 18, está representado o aspecto `AAutenticação` em um nó pai e em seguida, seus respectivos nós filhos são os *advices* `AutenticaInsert`, `AutenticaUpdate e AutenticaDelete`.

Para cada nó pai da árvore de aspectos, ao lado é exibido o código de testes unitários referente ao aspecto selecionado. No caso da figura 18, está selecionado o aspecto `AAutenticacao` e ao lado, exibido o código de testes para este aspecto. Para cada aspecto, a ferramenta cria um nó pai na árvore e trabalha com um código de teste diferente.

O mesmo ocorre com os testes de classes aspectadas ilustrado na figura 19. No *treeview* da guia de testes de classes aspectadas são apresentadas as classes afetadas pelos aspectos e os respectivos métodos afetados. Cada nó pai da árvore indica a classe afetada e seus nós filhos, respectivamente, os métodos afetados da classe.

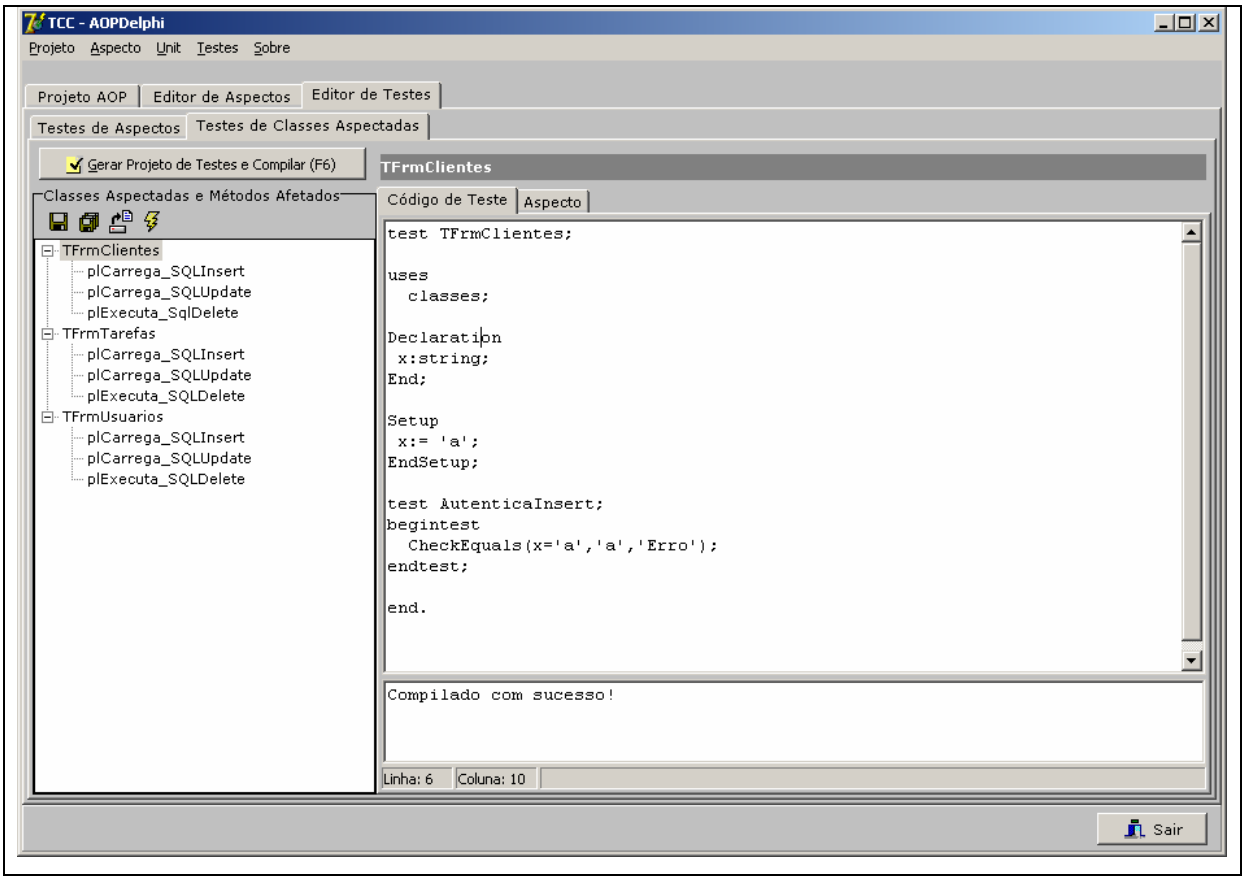


Figura 19 – Testes de classes aspectadas

Da mesma forma que o *treeview* serve como menu na guia de testes de aspectos, nos testes de classes aspectadas o *treeview* também serve de menu. Cada rotina de teste de classe é tratada separadamente, ou seja, cada classe afetada pelos aspectos possui sua própria rotina de teste.

Todas as duas guias de testes possuem comandos de acesso rápido e funcionalidades para facilitar a implementação e depuração dos testes unitários. As guias possuem uma área

destinada a informar os erros ou sucesso de compilação das rotinas de testes unitários na linguagem de testes, esta representada na figura 20.

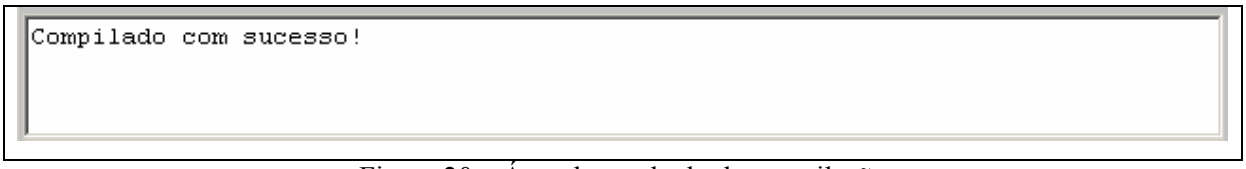


Figura 20 – Área de resultado de compilação

Nesta área, caso exista erro de compilação das rotinas de testes, é exibido o tipo de erro (léxico, sintático ou semântico) com sua respectiva descrição, além da linha e coluna do erro.

Sob as árvores das guias de implementação de testes, existe um menu de acesso rápido às funcionalidades da ferramenta, exemplificada na figura 21.

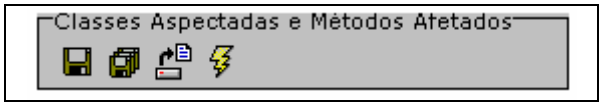


Figura 21 – Menu de acesso rápido

O primeiro botão refere-se a salvar em um arquivo texto a rotina de testes que está selecionada na árvore naquele instante. O segundo botão refere-se a salvar todas as rotinas de testes ao mesmo tempo automaticamente. Todas as rotinas de testes serão salvas com o nome do aspecto e carregadas posteriormente, automaticamente pela ferramenta, assim que a árvore for montada. O arquivo de testes salvo possui a extensão .test e o mesmo fica salvo no diretório Entrada.

O terceiro botão refere-se a carregar uma outra rotina de testes previamente implementada. Este botão abre uma segunda janela pedindo ao desenvolvedor para localizar o arquivo que deseja abrir. O quarto botão refere-se a compilar a rotina de teste exibida naquele momento, a fim de encontrar erros no código implementado. Havendo erros de compilação, os mesmos serão exibidos na área específica dos erros.

Para facilitar a implementação dos testes unitários com suporte do *framework* DUnit, existe um menu para auxílio do desenvolvedor. Este menu é encontrado clicando com o botão direito do *mouse* sobre a área de definição de testes e é representado na figura 22.

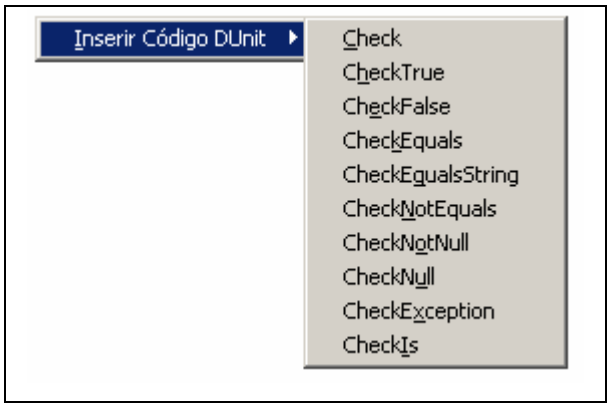


Figura 22 – Menu de acesso aos comandos do DUnit

Este menu tem por função inserir os principais métodos do DUnit no ambiente de implementação dos testes. O método selecionado no menu será inserido no ponto exato onde está o cursor no momento que se está escrevendo as rotinas de testes. Os métodos do DUnit podem ser consultados na Quadro 1.

Todos estes comandos e sub-menus também são encontrados no menu principal da ferramenta. O menu denomina-se Testes e está sub-dividido nas categorias de testes de aspectos e testes de classes aspectadas.

Após o desenvolvedor ter implementado todas as rotinas de testes unitários para aspectos ou classes aspectadas, o mesmo tem a opção de gerar as classes de testes. Esta funcionalidade pode ser disparada pelos botões que ficam cada qual em sua respectiva guia, (testes de aspectos ou classes aspectadas) ou ainda, disparados pelas teclas de atalho F5 para testes de aspectos e, F6 para testes de classes aspectadas.

Durante a geração das classes de testes, a ferramenta segue alguns passos. Primeiramente são compiladas todas as rotinas de testes implementadas na linguagem de definição de testes, a fim de excluir erros de compilação. Em seguida, são geradas as classes e o projeto Delphi e, posteriormente, as mesmas são compiladas usando o compilador do próprio Delphi. Todos estes passos são identificados ao desenvolvedor por meio de uma janela informativa que está representada na figura 23.

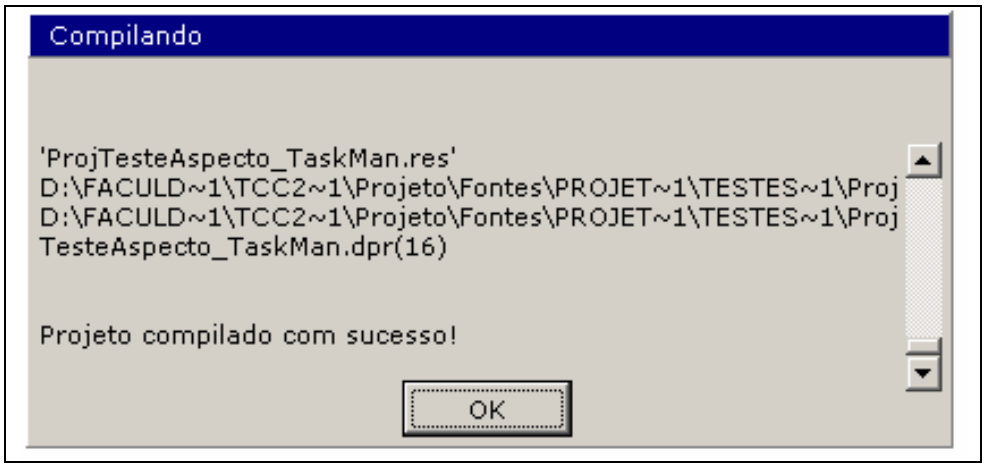


Figura 23 – Janela de compilação

Todas as classes e projetos gerados além do executável criado, são gerados na pasta ProjetoTeste e em sua respectiva sub-pasta, dependendo do tipo de teste. As classes já existentes serão sobrescritas e o mesmo ocorre com o executável gerado.

As classes geradas possuem o nome definido nas rotinas de testes implementadas. A assertiva `Test,` no início da gramática da linguagem de testes, requer um identificador logo em seguida. Este identificador tem função de dar nome à classe gerada. Já o projeto recebe o nome do projeto do programa de componentes que está sendo afetado pelos aspectos, este com algumas ressalvas. O projeto de testes de aspectos receberá o nome de "ProjTesteAspecto_" seguido do nome do projeto do programa de componentes. O mesmo ocorre com os testes de classes aspectadas; o projeto gerado recebe o nome de "ProjTesteClasseAspect_" seguido do nome do projeto do programa de componentes.

5.3.3 Estudo de caso

Como a ferramenta está subdividida em duas funcionalidades bem distintas, prover suporte a geração de testes de aspectos e testes de classes aspectadas, são apresentados então dois estudos de caso distintos. Em um primeiro momento é abordado um estudo de caso sobre a implementação de testes de aspectos e, posteriormente, testes de classes aspectadas.

No trabalho desenvolvido por Oliveira (2006), o mesmo faz um estudo de caso sobre a implementação de controle de *log* e autenticação no paradigma da programação orientada a aspectos através do AOPDelphi (OLIVEIRA, 2006). Os estudos de caso deste trabalho são baseados sobre o aspecto de autenticação desenvolvido por Oliveira (2006). O aspecto de autenticação é representado no Quadro 8.

O projeto de componentes envolvido no estudo de caso de Oliveira (2006) é um sistema com alguns programas de cadastro. Somente usuários autorizados podem ter acesso ao sistema e as suas funcionalidades. O objetivo do estudo de caso do aspecto de autenticação é consistir se determinada operação realizada pelo usuário (*Insert, Delete* e *Update*) é válida para o usuário ativo no sistema, ou seja, se o mesmo possui o direito de realizar tal operação. Para maiores detalhes do estudo de caso deve-se consultar o trabalho realizado por Oliveira (2006).

Para efeito do estudo de caso de testes de classes aspectadas, é abstraído o processo de *weaving*, partindo do princípio de que o sistema de componentes possui todas as classes já aspectadas.

```
AAutenticacao = Aspect
Implementation
  AutenticaInsert : Pointcut = (* *.*SQLInsert(*));
  AutenticaUpdate : Pointcut = (* *.*SQLUpdate(*));
  AutenticaDelete : Pointcut = (* *.*SQLDelete(*));

  Advice AutenticaInsert: Before;
  begin
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('OPERACAO').AsInteger := 1;  //1=Insert
      ParamByName('TABELA').AsString := Self.NomeTabela;
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão nesta atividade.');
    end;
  EndAdvice;

  Advice AutenticaUpdate: Before;
  begin
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('OPERACAO').AsInteger := 2;  //2=Update
      ParamByName('TABELA').AsString := Self.NomeTabela;
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão nesta atividade.');
    end;
  EndAdvice;

  Advice AutenticaDelete: Before;
```

```
  begin
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('OPERACAO').AsInteger := 3; //3=Delete
      ParamByName('TABELA').AsString := Self.NomeTabela;
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão nesta atividade.');
    end;
  EndAdvice;
End.
```

Quadro 8 – Aspecto de autenticação

5.3.3.1 Estudo de caso para testes de aspectos

O objetivo final de um teste unitário para aspectos é garantir que o código implementado e inserido pelo *advice* realmente funciona. O código do *advice* são as linhas de código em Object Pascal entre as palavras reservadas `Begin` e `EndAdvice` da linguagem de aspectos.

O objetivo do teste do aspecto autenticação é comprovar que as rotinas implementadas em seus respectivos *advices* funcionam. O programa com o caso de teste implementado na ferramenta e o código fonte em Object Pascal da classe gerada pela ferramenta estão ilustrados no Quadro 9 e 10, respectivamente.

```
Test AAutenticacao;

uses
  Classes, SysUtils, UDataModule;

Setup
  DMTaskMan:= TDMTaskMan.Create(Nil);
EndSetup;

TearDown
  DMTaskMan.Free;
EndTearDown;

Test AutenticaInsert;
Begintest
  Try
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
```

```
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := 3;
      ParamByName('OPERACAO').AsInteger := 1;
      ParamByName('TABELA').AsString := 'TAREFA';
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão para essa
atividade. ' + Chr(13) + 'Entre em contato com o seu superior.');
    end;
  Except
    On E:Exception Do
       Begin
         If   Pos('Você não tem permissão', E.Message) <> 0 Then
              Check(True, E.Message)
         Else
             Raise;
       End;
  End;
Endtest;

Test AutenticaUpdate;
Begintest
  Try
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := 3;
      ParamByName('OPERACAO').AsInteger := 2;
      ParamByName('TABELA').AsString := 'TAREFA';
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão para essa
atividade. ' + Chr(13) + 'Entre em contato com o seu superior.');
    end;
  Except
    On E:Exception Do
       Begin
         If   Pos('Você não tem permissão', E.Message) <> 0 Then
              Check(True, E.Message)
         Else
             Raise;
       End;
  End;
Endtest;

Test AutenticaDelete;
Begintest
  Try
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
```

```
      ParamByName('IDUSUARIO').AsInteger := 3;
      ParamByName('OPERACAO').AsInteger := 3;
      ParamByName('TABELA').AsString := 'TAREFA';
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão para essa
atividade. ' + Chr(13) + 'Entre em contato com o seu superior.');
    end;
  Except
    On E:Exception Do
       Begin
         If   Pos('Você não tem permissão', E.Message) <> 0 Then
              Check(True, E.Message)
         Else
             Raise;
       End;
  End;
Endtest;

end.
```

Quadro 9 – Programa com teste unitário do aspecto autenticação

```
Unit AAutenticacao;

Interface
Uses
  TestFramework, Classes,SysUtils,UDataModule;

Type
  TAAutenticacao = Class(TTestCase)
  Private
  Protected
    Procedure SetUp; Override;
    Procedure TearDown; Override;
  Published
    Procedure AutenticaInsert;
    Procedure AutenticaUpdate;
    Procedure AutenticaDelete;
  End;

Implementation

Procedure TAAutenticacao.SetUp;
Begin
   DMTaskMan:= TDMTaskMan.Create(Nil);
  Inherited;
End;

Procedure TAAutenticacao.TearDown;
Begin
  DMTaskMan.Free;
  Inherited;
End;

Procedure TAAutenticacao.AutenticaInsert;
Begin
  Try
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
```

```
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := 3;
      ParamByName('OPERACAO').AsInteger := 1;
      ParamByName('TABELA').AsString := 'TAREFA';
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise  Exception.Create('Você  não  tem  permissão  para  essa
atividade. ' + Chr(13) + 'Entre em contato com o seu superior.');
    end;
  Except
    On E:Exception Do
       Begin
         If   Pos('Você não tem permissão', E.Message) <> 0 Then
              Check(True, E.Message)
         Else
             Raise;
       End;
  End;
End;

Procedure TAAutenticacao.AutenticaUpdate;
Begin
  Try
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := 3;
      ParamByName('OPERACAO').AsInteger := 2;
      ParamByName('TABELA').AsString := 'TAREFA';
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise  Exception.Create('Você  não  tem  permissão  para  essa
atividade. ' + Chr(13) + 'Entre em contato com o seu superior.');
    end;
  Except
    On E:Exception Do
       Begin
         If   Pos('Você não tem permissão', E.Message) <> 0 Then
              Check(True, E.Message)
         Else
             Raise;
       End;
  End;
End;

Procedure TAAutenticacao.AutenticaDelete;
Begin
  Try
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
```

```
      ParamByName('IDUSUARIO').AsInteger := 3;
      ParamByName('OPERACAO').AsInteger := 3;
      ParamByName('TABELA').AsString := 'TAREFA';
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise  Exception.Create('Você  não  tem  permissão  para  essa
atividade. ' + Chr(13) + 'Entre em contato com o seu superior.');
    end;
  Except
    On E:Exception Do
       Begin
         If   Pos('Você não tem permissão', E.Message) <> 0 Then
              Check(True, E.Message)
         Else
             Raise;
       End;
  End;
End;
Initialization
  TestFramework.RegisterTest('Tests Suite', TAAutenticacao.Suite);
End.
```

Quadro 10 – Classe em Object Pascal gerada pela ferramenta

O caso de teste implementado irá instanciar um objeto do tipo `TDMTaskMan` necessário para efetuar o `Select` que o *advice* contém. Esta instanciação é realizada no método `Setup` e o referido objeto, destruído no método `TearDown`.

Após realizada a instanciação do objeto, o *framework* DUnit encarrega-se de disparar os métodos de testes e validar o resultado dos mesmos. Ao final o resultado é exibido ao desenvolvedor. O resultado da execução dos métodos de testes implementados pode ser visualizado na figura 24.

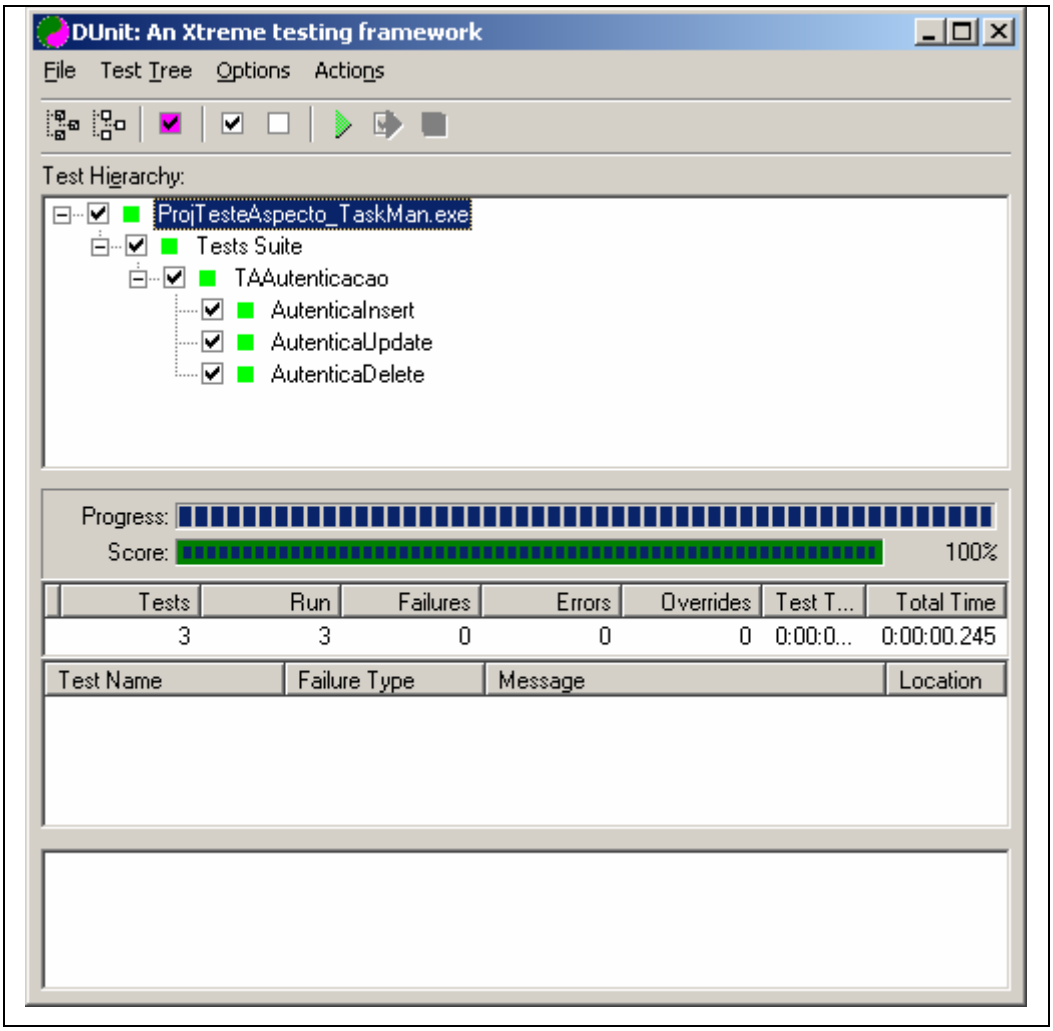


Figura 24 – Retorno do *framework* DUnit dos casos de testes de aspectos

O resultado do teste unitário do aspecto autenticação foi bem sucedido, porém, deve-se salientar que o usuário 1 não possui, de fato, acesso as operações de *update* e i*nsert* Devido a este fato, as rotinas de testes têm tratamentos de exceções para tratar as exceções que por ventura venham a ocorrer. Caso a exceção tratada seja a exceção gerada pelo código do próprio *advice,* podemos concluir com isso que o caso de teste foi bem sucedido e os *advices* implementados estão corretos.

5.3.3.2 Estudo de caso para testes de classes aspectadas

O objetivo maior de um caso de testes para classes aspectadas é descobrir se o aspecto que afetou determinada classe não afetou o resultado final do método interceptado, ou seja, se o método não fugiu do seu escopo original após o *weaving*.

Para o estudo de caso de testes de classes aspectadas é usada a classe afetada

`UClientes`, esta apresentada no apêndice B. Esta classe refere-se ao cadastro, no sistema de componentes, ao cadastro de clientes. Esta classe possui métodos *Insert, Update* e *Delete* que foram interceptados pelo processo de *weaving*. No estudo de caso é verificado se o método interceptado `flCarrega_SQLInsert` não teve seu escopo original afetado. Este método é do tipo função e seu retorno é um booleano. O método tempor função fazer a inserção de novos clientes no banco de dados. Caso a inserção seja bem sucedida, a função tem retorno verdadeiro, caso contrário, o retorno é falso.

O programa com o caso de teste implementado na ferramenta e o código fonte em Object Pascal da classe gerada pela ferramenta estão ilustrados no Quadro 11 e 12, respectivamente.

```
Test FormularioClientes;

Uses
  Classes, SysUtils, UDataModule, UClientes;

Declaration
  frmClientes: TFrmClientes;
End;

Setup
  DMTaskMan:= TDMTaskMan.Create(Nil);
  frmClientes:= TFrmClientes.Create(Nil);
EndSetup;

TearDown
  DMTaskMan.Free;
  frmClientes.Free;
EndTearDown;

Test flCarrega_SqlInsert;
BeginTest
  //CheckTrue(Condicao: Boolean , MsgErro: String = '')
  CheckTrue(frmClientes.flCarrega_SQLInsert, 'Erro de validacao');
EndTest;

end.
```

Quadro 11 – Programa com teste unitário do método afetado

```
Unit FormularioClientes;

Interface

Uses
  TestFramework, Classes,SysUtils,UDataModule,UClientes;

Type

  TFormularioClientes = Class(TTestCase)
  Private
    frmClientes: TFrmClientes;

  Protected
```

```
    Procedure SetUp; Override;
    Procedure TearDown; Override;
  Published
    Procedure flCarrega_SqlInsert;
  End;

Implementation

Procedure TFormularioClientes.SetUp;
Begin
   DMTaskMan:= TDMTaskMan.Create(Nil);
  frmClientes:= TFrmClientes.Create(Nil);
  Inherited;
End;

Procedure TFormularioClientes.TearDown;
Begin
  DMTaskMan.Free;
  frmClientes.Free;
  Inherited;
End;

Procedure TFormularioClientes.flCarrega_SqlInsert;
Begin
  giCodUsuario:= 3;
  frmClientes.NomeTabela := 'TAREFA';
//CheckTrue(Condicao: Boolean , MsgErro: String = '')
  CheckTrue(frmClientes.flCarrega_SQLInsert, 'Erro de validacao');
End;

Initialization
  TestFramework.RegisterTest('Tests Suite', TFormularioClientes.Suite);
End.
```

Quadro 12 – Classe em Object Pascal gerada pela ferramenta

Assim como o caso de teste de aspecto explanado no item 5.3.3.1, o caso de teste implementado irá instanciar um objeto do tipo `TDMTaskMan` e `frmClientes` necessários para efetuar o `Select` que o *advice* contém e para a chamada do método a ser testado, respectivamente. Estas instanciações são realizadas no método `Setup` e os referidos objetos, destruídos no método `TearDown`.

Após realizadas as instanciações dos objetos, o *framework* DUnit encarrega-se de disparar os métodos de testes e validar o resultado dos mesmos. No método de teste ainda é informado qual usuário conectado ao sistema de componentes e qual tabela este usuário esta tentando editar. Ao final o resultado é exibido para o desenvolvedor. O resultado da execução dos métodos de testes implementados podem ser visualizados na figura 25.

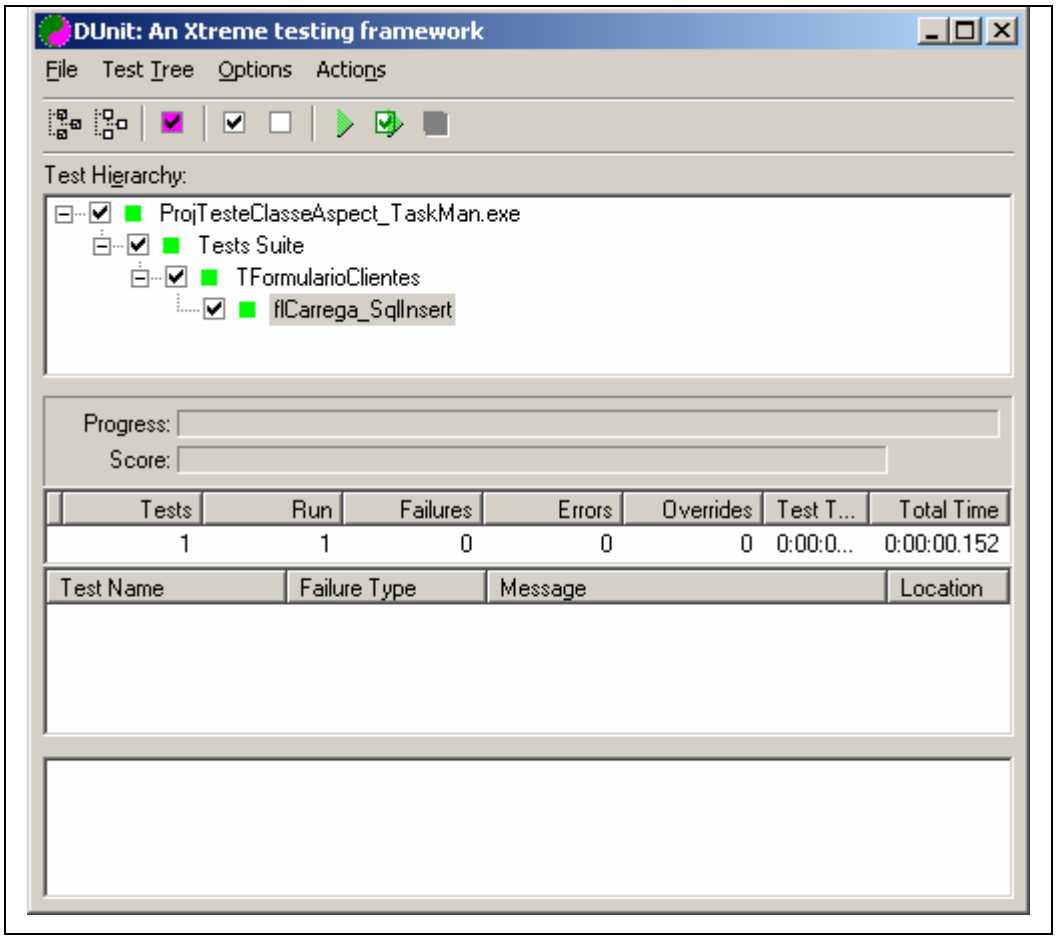


Figura 25 – Retorno do *framework* DUnit dos casos de testes de classes aspectadas

O resultado do caso de teste implementado para validação do método *Insert* foi sucesso, este representado pelo quadro verde na IDE do *framework*. O DUnit fez a instância do objeto `frmClientes` e chamou o método publico `flCarrega_SQLInsert` para verificar o seu retorno através do método `CheckTrue` do próprio *framework*.

5.4 RESULTADOS E DISCUSSÃO

Como se pode observar, o trabalho baseou-se em um estudo sobre tipos e categorias de testes de software, sobretudo testes de unidade, e uma explanação sobre POA. Porém, sendo o objetivo principal do trabalho apresentar um levantamento sobre testes de unidade para programação orientada a aspectos, a pesquisa bibliográfica realizada obteve poucas referências.

Considerando a proposta da ferramenta, foram obtidos bons resultados, no sentido de prover suporte a testes de unidade para programação orientada a aspectos em Object Pascal. A

ferramenta mostrou-se eficiente em testar as classes aspectadas e principalmente em testar os aspectos implementados. Porém, apesar da facilidade propiciada pela ferramenta em elaborar os testes, a linguagem de definição de testes é limitada. A falta do recurso de declarações de variáveis locais e a declaração de métodos locais tornam o trabalho do desenvolvedor mais penoso, ou seja, o desenvolvedor poderá ter que implementar mais métodos de testes além de declarar as variáveis sempre no escopo privado da classe. A falta do recurso de declarações de métodos locais torna a implementação dos métodos de testes redundante. Um exemplo disto é o estudo de caso da seção 5.3.3.1. Na implementação dos métodos de testes, o código implementado em todos os métodos são praticamente idênticos, alterando apenas o valor que é passado para o parâmetro Operação. Com a implementação de procedimentos locais, este comportamento poderia ser diferente, podendo o desenvolvedor declarar apenas um método de teste e o valor da operação poderia ser passado como parâmetro.

Considerando o estudo de caso elaborado na seção 5.3.3.1, observa-se a facilidade propiciada ao desenvolvedor em implementar casos de testes para os aspectos e fica evidente os benefícios fornecidos pelos testes de aspectos. O desenvolvedor implementa os aspectos e, em seguida, pode executar uma bateria de testes sobre estes aspectos, tendo assim, a certeza que não serão introduzidos erros no programa de componentes afetado.

O estudo de caso elaborado na seção 5.3.3.2 também exemplifica a utilidade da ferramenta para o desenvolvedor. O estudo de caso elucida testes sobre as classes e métodos afetados pelo processo de *weaving*, garantindo assim ao desenvolvedor que os aspectos foram corretamente introduzidos no programa de componentes e não alteraram o escopo original destes métodos afetados.

O Quadro 13 relaciona os trabalhos correlatos mencionados na seção 4.2 com este trabalho, identificando características comuns e distintas em cada um dos trabalhos.

Características	**Este trabalho**	**Oliveira (2006)**	**Silva (2005)**	**Ayroso (1998)**
Técnicas e categorias de testes	X			X
Conceitos e explanações sobre POA	X	X	X	
Comparativo entre POO e POA			X	
Análise, projeto e implementações orientada a aspectos	X	X	X	
Análise, projeto e implementações orientada a objetos	X	X	X	
Testes de unidade para POA	X			

Quadro 13 – Características em comum e distintas de cada trabalho

6 CONCLUSÕES

Este trabalho propicia uma visão geral sobre as categorias e técnicas de testes de software e uma introdução ao paradigma da programação orientada a aspectos. Considerando as vantagens da introdução deste paradigma de programação na linguagem Object Pascal, é muito importante o surgimento de tecnologias que propiciem implementar com qualidade sistemas desenvolvidos com a técnica de programação orientada a aspectos.

Os objetivos do trabalho foram atingidos. Foi implementada na ferramenta AOPDelphi (OLIVEIRA, 2006), uma linguagem para definição de testes para aspectos e classes aspectadas, além de um ambiente para facilitar a implementação dos casos de testes. A ferramenta foi estendida para que receba como entrada programas fontes para testes unitários na linguagem específica para este fim. Estes programas fontes sofrem análise léxica, sintática e semântica para posterior geração de código Object Pascal. Estas novas extensões foram integradas no ambiente desenvolvido por Oliveira (2006), possibilitando ao desenvolvedor implementar os testes logo após a implementação dos aspectos.

Porém, a linguagem criada para implementar os testes na ferramenta AOPDelphi (OLIVEIRA, 2006) apresenta limitações como, por exemplo, a não possibilidade de criar variáveis e métodos locais dentro dos métodos de testes o que facilitaria a implementação dos casos de testes unitários.

Por fim, fica evidenciada a importância da ferramenta GALS (GESSER, 2003), a mesma foi fundamental na especificação e implementação da linguagem de definição de testes, sendo esta a responsável pelos analisadores léxico e sintático. O suporte fornecido pelo DUnit para testes unitários em Object Pascal também foi fundamental na especificação da ferramenta, pois é ele quem gerencia e dispara os testes.

6.1 EXTENSÕES

A ferramenta trabalha de forma separada em relação ao ambiente Delphi. Visando uma maior integração com o ambiente Delphi, sugere-se como trabalho futuro, a integração das funcionalidades da ferramenta à IDE do Delphi. Os programas de aspectos, além dos programas de testes poderão ser escritos no próprio ambiente do Delphi.

Tendo em vista as limitações das linguagens de aspectos e de definição de testes, sugere-se também, implementar melhorias nas mesmas. Ambas as linguagens tem limitações quanto à declaração de variáveis. A linguagem de aspectos não permite declarar variáveis nos *advices* e, a linguagem de testes, não permite declarar variáveis nos métodos de testes. Esta deficiência pode ser sanada alterando a gramática das linguagens.

Para finalizar, sugere-se ainda uma maior integração entre testes de aspectos e classes aspectadas no sentido de unir os dois projetos criados. Podem-se eliminar as abas na interface da ferramenta que diferem os testes de aspectos dos testes de classes e, por meio de palavras reservadas ou métodos, especificar apenas uma linguagem de definição de testes para ambos os casos.

REFERÊNCIAS BIBLIOGRÁFICAS

ANEZ, Juanco. **DUNIT**: An Xtreme testing framework for Borland Delphi programs. [Venezuela], 2000. Disponível em: <http://dunit.sourceforge.net>. Acesso em: 30 maio 2007.

AYROSO, Vanessa D. **Avaliação de métodos de teste de software em desenvolvimento de sistemas em banco de dados.** 1998. 86 f. Monografia (Especialização em Tecnologia em Desenvolvimento de Sistemas) – Fundação Universidade Regional de Blumenau, Blumenau.

BARROS, Alexandra. **Aspect oriented programming.** [Recife], 2004. Disponível em: <http://www.cin.ufpe.br/~abab/ppt/aop.ppt>. Acesso em: 11 set. 2007.

CHAVEZ, Christina von F. G.; LUCENA, Carlos J. P. de. **Um enfoque baseado em modelos para o design orientado a aspectos**. 2004. 298 f. Tese (Doutorado em Informática) – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

DELAMARO, Márcio E. MALDONADO, José C. JINO, Mario. **Introdução ao teste de software.** Rio de Janeiro: Elsevier, 2007.

GESSER, Carlos E. **GALS**: gerador de analisadores léxicos e sintáticos. 2003. 150 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis.

HETZEL, William. **Guia completo ao teste de software.** Rio de Janeiro: Campus, 1987.

INTHURN, Cândida. **Qualidade & teste de software.** Florianópolis: Visual Books, 2001.

KICZALES, Gregoret al. **Aspect-oriented programming.** Proceedings of ECOOP'97, Finland: Springer – Verlag, 1997.

KULESZA, Uirá; SANT'ANNA, Claudio; LUCENA, Carlos J. P. **Técnicas de projeto orientado a aspectos.** Uberlândia, 2005. Disponível em: <http://www.sbbd-sbes2005.ufu.br/mini_curso.aspx>. Acesso em: 09 set. 2007.

LEMOS, Otavio A. L. et al. **Teste de unidade para programas orientados a aspectos.** São Paulo, 2004. Disponível em: <http://www.lbd.dcc.ufmg.br:8080/colecoes/sbes/2004/005.pdf>. Acesso em: 11 set. 2007.

NELSON, Torsten. **Apostila do curso de programação orientada a aspectos com AspectJ.** Belo Horizonte, 2005. Disponível em: <http://www.aspectos.org/courses/aulasaop/curso_poa.pdf>. Acesso em: 09 set. 2007.

OLIVEIRA, Edmar S. de. **Protótipo de um *weaver* para pøgramação orientada a aspectos em Delphi.** 2006. 71 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Fundação Universidade Regional de Blumenau, Blumenau.

PAULA FILHO, Wilson de P. **Engenharia de software:** fundamentos, métodos e padrões. Rio de Janeiro: LTC, 2001.

PARRINGTON, Norma; ROPER, Marc. **Understanding software testing.** Chichester: Ellis Horwood, 1989.

PFLEEGER, Shari Lawrence. **Engenharia de software:** teoria e prática. Tradução Dino Franklin. São Paulo: Prentice Hall, 2004.

POL, Martin; TEUNISSEN, Ruud; VEENENDAAL, Erik van. **Software testing:** a guide to the TMap approach. Harlow: Addison Wesley Longman, 2002.

PRESSMAN, Roger S. **Engenharia de software.** Tradução Rosângela Delloso Penteado. São Paulo: Makron Books, 2002.

RESENDE, Antônio M. P.; SILVA, Claudiney C. **Programação orientada a aspectos em Java.** Rio de Janeiro: Brasport, 2005.

ROCHA, Ana Regina C. da; MALDONADO, José C.; WEBER, Kival C. **Qualidade de software.** São Paulo: Prentice Hall, 2001.

SILVA, Kelli A. B. B. da. **Análise comparativa entre programação orientada a objetos e orientada a aspectos.** 2005. 94 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Fundação Universidade Regional de Blumenau, Blumenau.

TIRELO, Fábio. et al. Desenvolvimento de software orientado por aspectos. In: JAI – JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, 23., 2004, Salvador. **Anais**... Belo Horizonte: [s.n.], 2004. p. 3-39.

ZHAO, Jianjun. **Data-flow-based unit testing of aspect-oriented programs.** Japan, 2003. Disponível em: <http://cse.sjtu.edu.cn/~zhao/pub/pdf/compsac03.pdf>. Acesso em: 11 set. 2007.

APÊNDICE A – Manual da linguagem de definição de testes unitários

Como a ferramenta oferece suporte para testes unitários em Object Pascal, a sintaxe da linguagem de definição de testes é semelhante ao próprio Object Pascal. Uma rotina de definição de testes é composta por seu identificador, declarações de *uses,* declarações de variáveis, declaração do *Setup*, declaração do *TearDown* e as declarações dos métodos de testes como mostra o quadro 8.

```
Test Exemplo;

Uses
  classes;

Declaration
 x: String;
End;


Setup
  x:= 'Teste';
EndSetup;


TearDown
  x:= '';
EndTearDown;

Test Exemplo1;
BeginTest
  CheckEquals('Teste', x, 'Erro');
EndTest;

End.
```

Quadro 14 – Exemplo de definição de testes

A palavra reservada `Uses` identifica a lista de classes que devem ser adicionadas ao projeto para não haver erros de compilação como, por exemplo, identificadores não declarados. O seu funcionamento é idêntico ao Object Pascal.

A palavra reservada `Declaration` serve fazer declarações de variáveis necessárias a implementação de um método de teste. A declaração de variáveis segue o formato do Object Pascal. Os tipos das variáveis (*string, integer, boolean* etc) não são consistidos pela ferramenta, pois o desenvolvedor pode necessitar instanciar um objeto dificultando assim a consistência de tipos. As variáveis declaradas neste ponto serão declaradas no escopo de visibilidade privado da classe do Object Pascal.

O método `Setup` tem por função, no DUnit, executar comandos do Object Pascal antes que os métodos de testes sejam disparados. Este método é útil, caso seja necessário, ter objetos instanciados durante a execução dos testes.

O método `TearDown`, ao contrário do `Setup`, tem por função executar comandos do Object Pascal após a execução de todos os métodos de testes. Este método é útil para, por exemplo, liberar a memória utilizada por objetos instanciados.

Todo código inserido entre as palavras reservadas `Setup/EndSetup` e `TearDown/EndTearDown` é código Object Pascal. Este código não é validado sintaticamente pela ferramenta, pois neste caso, seria necessário trabalhar com toda a gramática do Object Pascal.

As declarações do `Uses`, `Declaration`, `Setup` e `TearDown` não são obrigatórias, podendo ficar ausentes.

Por fim, vem a declaração dos métodos de testes unitários. O quadro 9 apresenta a sintaxe para declaração dos métodos.

```
Test IDENTIFICADOR;
BeginTest
  <codigo_teste>
EndTest ;
```

Quadro 15 – Sintaxe da declaração de métodos de testes

A palavra reservada `Test` identifica que este trecho do código é um método de teste. A palavra reservada vem seguida por um identificador, sendo este identificador, o nome do método. O método ainda necessita das palavras reservadas `BeginTest` e `EndTest` para delimitar o código do método. O código inserido entre estas duas palavras reservadas é exclusivamente código em Object Pascal e, seguindo os moldes dos métodos `Setup` e `TearDown`, esse código não é analisado sintaticamente pela ferramenta.

É neste ponto onde efetivamente são implementados os casos de testes unitários. Todos os métodos implementados serão abordados pelo *framework* DUnit, sendo este o responsável por disparar os métodos e verificar o sucesso ou falha do teste. Não há limite para a quantidade de métodos declarados, sendo então possível declarar quantos métodos de testes forem necessários. Estes métodos ficarão no escopo de visibilidade público da classe gerada em Object Pascal.

APÊNDICE B – Classe UClientes

Para exemplificar o estudo de caso apresentado no item 5.3.3.2, no quadro 16 está apresentada a interface da classe `UClientes` e o método que foi afetado pelo aspecto de autenticação.

```
unit UClientes;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, Mask, DBCtrls, ExtCtrls, Buttons, DB;

type
  TFrmClientes = class(TForm)

  private
    lyOperacao : Byte;
    laEstados : Array[0..26] of String[2];
    procedure plLimpaCampos;
    procedure plControlaBotoes(vEstaEmEdicao:Boolean);
    procedure plCarrega_SQLUpdate;
    procedure plExecuta_SqlDelete;
    Function flGetProximoCodigo : Integer;
    Function flGetIndiceUF(vUF:String):Integer;
  public
    NomeTabela : String;
    function flCarrega_SQLInsert: Boolean;
    procedure plCarrega_SQLUpdate;
    procedure plExecuta_SqlDelete;
    procedure plCarregaCampos(vCodigo:Integer);
  end;

var
  FrmClientes: TFrmClientes;

implementation

 uses UDataModule, UConsClientes, IBQuery, UPrincipal;

function TFrmClientes.flCarrega_SQLInsert: Boolean;
begin

// Código inserido por AOPDelphi - Inicio
// [TFrmClientes.flCarrega_SQLInsert] Afetado pelo aspecto AAutenticacao -
Advice AutenticaInsert(Before)
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('OPERACAO').AsInteger  :=  1;     //1=Insert,  2=Update,
3=Delete
```

```
      ParamByName('TABELA').AsString := Self.NomeTabela;
      Open;
      if  Fields[0].AsInteger = 0 then
          Raise  Exception.Create('Você  não  tem  permissão  para  essa
atividade. ' + Chr(13) + 'Entre em contato com o seu superior.');
    end;
// Código inserido por AOPDelphi - Fim
  Result:= False;
  With DMTaskMan.qryClientes_Insert do
    begin
      ParamByName('CL_CODIGO').AsInteger := flGetProximoCodigo;
      ParamByName('CL_DESCRICAO').AsString := edtDescricao.Text;
      ParamByName('CL_END_LOG').AsString := edtLogradouro.Text;
      if  trim(edtNumero.text) <> '' then
          ParamByName('CL_END_NUM').AsInteger:=
StrtoInt(trim(edtNumero.text))
      else
          ParamByName('CL_END_NUM').Value := Null;
      ParamByName('CL_END_COMP').AsString := edtComplemento.Text;
      ParamByName('CL_END_BAIRRO').AsString := edtBairro.Text;
      ParamByName('CL_END_CEP').AsString := edtCep.Text;
      ParamByName('CL_FONE').AsString := edtFone.Text;
      ParamByName('CL_END_CIDADE').AsString := edtCidade.Text;
      ParamByName('CL_END_UF').AsString := cbUF.Text;
      Case cbTipoPessoa.ItemIndex of
         -1 : ParamByName('CL_TIPOPESSOA').Value := Null;
          0 : ParamByName('CL_TIPOPESSOA').AsString := 'F';
          1 : ParamByName('CL_TIPOPESSOA').AsString := 'J';
      end;
      ParamByName('CL_CNPJ_CPF').AsString := edtCnpjCpf.Text;
      ParamByName('CL_CONTATO').AsString := edtContato.Text;
      ParamByName('CL_EMAIL').AsString := edtEmail.Text;
    end;
  Result:= True;
end;
```

Quadro 16 – Interface da classe afetada pelo aspecto autenticação