UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA DE SUPORTE AO PLANEJAMENTO DE TESTE FUNCIONAL DE SOFTWARE A PARTIR DE DIAGRAMAS DE CASOS DE USO

JULIANO BIANCHINI

BLUMENAU
2004

2004/1-19

JULIANO BIANCHINI

FERRAMENTA DE SUPORTE AO PLANEJAMENTO DE TESTE FUNCIONAL DE SOFTWARE A PARTIR DE DIAGRAMAS DE CASOS DE USO

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 Ciência da Computação — Bacharelado.

Prof. Everaldo Artur Grahl - Orientador

BLUMENAU
2004

2004/1-19

FERRAMENTA DE SUPORTE AO PLANEJAMENTO DE TESTE FUNCIONAL DE SOFTWARE A PARTIR DE DIAGRAMAS DE CASOS DE USO

Por

JULIANO BIANCHINI

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

Presidente:	Prof. Everaldo Artur Grahl, FURB
Membro:	Prof. Maurício Capobianco Lopes, FURB
Membro:	Prof. Marcel Hugo, FURB

Blumenau, 02 de junho de 2004

Dedico este trabalho a minha querida esposa Rosangela e ao meu querido avô Ingo Wengrath (in memorian)

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família por acreditar em mim, principalmente, a minha querida esposa pelo apoio e compreensão.

A todos os meus amigos que me ajudaram direta ou indiretamente, principalmente ao Giancarlo Tomazelli e ao André Vinicius Castoldi.

Ao meu orientador, Everaldo Artur Grahl, por todo o apoio e por ter acreditado na conclusão deste trabalho.

RESUMO

Atualmente existem muitas pesquisas sobre a utilização da UML para a geração de casos de testes funcionais. Este trabalho analisou algumas alternativas existentes na literatura sobre o uso de diagrama de casos de uso, um dos diagramas mais usados na UML, e a partir deste estudo foi criada uma ferramenta de suporte ao planejamento de testes funcionais. Inicialmente foi modificada a ferramenta CASE ArgoUML para incluir extensões apropriadas a casos de testes. Posteriormente foi desenvolvida uma ferramenta para permitir a leitura destas extensões e documentar os testes seguindo orientações previstas no padrão IEEE 829 que trata da documentação de testes de software.

Palavras chaves: teste de software; UML; casos de teste; casos de uso; IEEE 829.

ABSTRACT

Currently there are many researches on the use of the UML for the generation of functional test cases. This work analyzed some existing alternatives in literature on the use of use case diagrams, one of the most used diagrams in the UML, and from this study a tool for support functional test planning was created. Initially, the ArgoUML CASE tool was customized to include appropriate extensions for the test case specification. Later, a tool was developed to allow the reading of these extensions and to document the tests cases according with IEEE 829 standard that it deals with the documentation of software tests.

Key-Words: software test; UML; test cases; use cases; IEEE 829.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A contínua demanda por sistemas de computação exige que os profissionais da área de tecnologia da informação estejam preparados para fornecer produtos de software com qualidade. Para tanto, a verificação e validação (V&V) são consideradas atividades fundamentais no desenvolvimento de software. Estas atividades são especialmente valiosas quando aplicadas desde cedo no processo de desenvolvimento, pois erros encontrados nas fases de especificação e projeto são muito mais baratos para serem corrigidos do que quando encontrados em fases mais avançadas (RYSER; GLINZ, 2003, p. 1-2). Entre as atividades de V&V está a atividade de teste de software.

O papel do teste é encontrar erros na implementação de um software e validar os requisitos implementados. Isso significa que o teste tem um papel importantíssimo na verificação e validação de software. Porém, em muitos casos, a preparação e o desenvolvimento dos casos de teste são feitos sem muito planejamento. Além disso, freqüentemente os testes são selecionados de forma aleatória ou *ad-hoc* (momentânea) e os casos de teste são desenvolvidos de uma forma não estruturada e não sistemática. Esta é uma realidade encontrada no desenvolvimento de softwares comerciais, onde há recursos limitados e tempo escasso para o teste.

Apesar de existirem várias técnicas e estratégias aplicadas a teste de software, a prática ainda está distante da teoria (RYSER; GLINZ, 2003, p. 1-2). Alguns problemas encontrados incluem:

a) falta de planejamento dos tempos e custos: na prática, o planejamento dos testes é feito tardiamente no processo, quando ele está quase no final. Desta forma há pressão para manter menor custo e tempo e somado ao fato de que o projeto pode estar atrasado, a preparação e execução do teste são feitas superficialmente, comprometendo a garantia da qualidade do produto de software;

b) falta de documentação: os testes não são devidamente preparados, não há um plano de teste e os testes não são documentados;

c) o teste é a última etapa do processo de desenvolvimento: o desenvolvimento dos casos de teste é feito apenas quando o software está quase pronto. Porém, a

fase de testes deveria começar imediatamente após a especificação ter sido desenvolvida. Desta forma, muitos erros, omissões e inconsistências podem ser encontradas nas fases de análise e projeto. É mais barato corrigir erros nestas fases iniciais do que após a implementação ter sido feita;

d) casos de teste não são gerados de forma sistemática: os testes são escolhidos de forma aleatória e os casos de teste gerados com base no conhecimento do testador e sem procedimento definido.

Entre as abordagens de testes de software encontra-se o teste Funcional também conhecido como teste de Caixa-Preta. As técnicas de teste Funcional derivam os casos de testes a partir da análise da funcionalidade (dados de entrada/saída e especificação) do programa sem levar em consideração a estrutura interna do mesmo.

A *Unified Modeling Language* (UML) é muito usada para modelar sistemas orientados a objetos e, na versão 1.3, é formado por nove diagramas com objetivos e perspectivas diferentes. A utilização dela como modelo sistemático para o desenvolvimento dos casos de teste tem sido estudada e evoluída durante os últimos anos. Um diagrama que tem sido estudado para teste é o caso de uso que é uma forma de capturar as funcionalidades e comportamentos de um sistema na perspectiva do usuário. O caso de uso é usado para ajudar na elicitação e documentação dos requisitos dos usuários. Desta forma, o caso de uso é uma fonte de informações que ajuda não só o desenvolvimento do software, mas também, a atividade de teste de software. Entre as propostas de utilização da UML para teste encontra-se a apresentada por Heumann (2004) e o teste de caso de uso estendido, proposto por Binder (2000, p. 722-731).

A não documentação do processo de software é, conforme já dito anteriormente, um fato nas empresas de desenvolvimento de software. O padrão IEEE 829 tem por objetivo descrever os documentos necessários para apoiar a atividade de teste de software. Os documentos deste padrão descrevem o planejamento, especificação e a geração de relatórios de testes. Este padrão será usado neste trabalho como base para documentação da atividade de testes.

1.1 OBJETIVOS DO TRABALHO

Este trabalho tem por objetivo principal o desenvolvimento de uma ferramenta de suporte ao planejamento do teste funcional de software a partir da utilização de diagramas de casos de uso da UML.

Os objetivos específicos são:

a) análise e aplicação das metodologias de teste de software baseada em casos de uso da UML apresentadas por Heumann (2004) e Binder (2000, p. 722-731);

b) adaptação de uma ferramenta CASE de código aberto para suportar extensões aplicáveis a casos de testes funcionais;

c) incorporação de padrões de documentação de teste de software da IEEE 829 na ferramenta construída.

1.2 ESTRUTURA DO TRABALHO

No segundo capítulo é feita uma revisão bibliográfica para o entendimento do trabalho incluindo temas como qualidade de software, testes de software, UML como modelo para testes de software, Padrão IEEE 829 e a ferramenta CASE ArgoUML.

O terceiro capítulo apresenta os requisitos principais do problema tratado neste trabalho, assim como a especificação do software e a operacionalidade do mesmo demonstrando o seu uso prático.

O quarto capítulo apresenta as conclusões e sugestões de extensão deste trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conceitos, técnicas, modelos, padrões e ferramentas mais relevantes para o entendimento e desenvolvimento deste trabalho.

2.1 QUALIDADE DE SOFTWARE

Antes de efetivamente falar sobre qualidade de software, convém voltar no tempo e analisar os fatos que levaram este tema a ser tão discutido. Nos últimos anos tem-se falado muito em qualidade de software. Isso se intensificou mais depois da tão falada "globalização". A quebra da fronteira de comércio entre os países abriu as portas para a exportação de software. Essa quebra trouxe consigo mais oferta de produtos, maior concorrência e maiores exigências de qualidade. Produzir software apenas pensando em menor custo e prazo de entrega não seria mais o que o mercado queria. A isto, soma-se um fator decisivo: a qualidade do produto de software.

Isso desencadeou uma série de novas iniciativas na área de qualidade e trouxe a tona aquelas já existentes. Iniciativas estas que visam garantir a qualidade do produto de software através da verificação e validação do produto e da maturidade do processo de desenvolvimento.

A qualidade de software pode ser definida como um processo sistemático que focaliza todas as etapas e artefatos produzidos com o objetivo de garantir a conformidade de processos e produtos, prevenindo e eliminando defeitos (BARTIÉ, 2002, p. 16). Isto significa que a qualidade não é uma fase do ciclo de desenvolvimento de software, mas sim parte de todas as fases.

No Brasil, a preocupação com a qualidade de software existe, porém, ainda há muito a ser feito. Os indicadores de qualidade e produtividade gerados pelo Programa Brasileiro da Qualidade e Produtividade (PBQP, 2004) mostram que em 1995 apenas 11% das 177 empresas de software consultadas tinham algum programa de qualidade total, sistema da qualidade ou similar implantado e em 1999 esse percentual aumentou para 26%. A meta para 2001 era alcançar 50%.

Outro fato constatado pela Secretaria de Política de Informática (SEPIN, 2004) é que mais de 50% das empresas conhecem normas de qualidade de software, mas não usam. Mais

de 50% das empresas utilizam práticas de Engenharia de Software para avaliar a qualidade do produto, incluindo diferentes tipos de testes - funcionais, de aceitação, de campo, baseados em erros, de integração e do sistema integrado. A Tabela 1 mostra práticas de Engenharia de Software adotadas na avaliação da qualidade do produto.

Tabela 1 – Práticas de Engenharia de Software adotadas na avaliação da qualidade

Categorias	**Nº de organizações**	**%**
Auditorias	97	22,6
Inspeção formal, Revisão por pares (Peer-review), Walthrough estruturado	70	16,3
Julgamento de especialistas	88	20,5
Levantamento de requisitos de qualidade	78	18,1
Medições da qualidade (Métricas)	75	17,4
Modelos de confiabilidade de software	21	4,9
Prova formal de programas	82	19,1
Segurança do produto final	58	13,5
Testes baseados em erros	236	54,9
Testes de aceitação	246	57,2
Testes de campo	243	56,5
Testes de integração	232	54,0
Testes de unidade	149	34,7
Testes do sistema integrado	222	51,6
Testes estruturais	107	24,9
Testes funcionais	255	59,3
Testes orientados a objetos	91	21,2
Testes para web	135	31,4
Outras	3	0,7
Não adota tais práticas	50	11,6
Base	**430**	**100**

Fonte: SEPIN (2004)

Infelizmente este é o cenário de muitas empresas de software no Brasil. Cenário este que mostra a imaturidade dos processos de desenvolvimento de software. Não basta ter apenas uma etapa que exercite o software à procura de erros, é necessário que se implante um processo que garanta e gerencie o nível de qualidade do produto e do processo de desenvolvimento, para que a qualidade seja uma constante durante o processo.

Este trabalho aborda a qualidade do produto de software através da verificação e validação (teste de software), ou seja, garantir que o produto de software produzido esteja em conformidade com os requisitos e reduzir os riscos do mesmo apresentar efeitos indesejáveis (falhas). Vale lembrar que, conforme mencionado anteriormente, a atividade de teste é apenas uma peça que compõe o quebra-cabeça da garantia da qualidade de software, isto é, há a

necessidade de outros esforços no processo de desenvolvimento para se atingir a qualidade total.

2.2 O PAPEL DO TESTE NA QUALIDADE DO PRODUTO DE SOFTWARE

V&V é o processo de verificação e análise que assegura que o software cumpra com suas especificações e atenda às necessidades dos clientes (SOMMERVILLE, 2003, p. 358). Embora sejam facilmente confundidas, V&V não são a mesma coisa. Pode-se resumir a diferença através de uma pergunta chave:

- *Validação*: estamos construindo o software certo?
- *Verificação*: estamos construindo certo o software?

Em outras palavras, validação é um conjunto de atividades que garante que o software construído corresponde aos requisitos do cliente e verificação é o conjunto de atividades que garante que o software implementa corretamente determinada função (PRESSMAN, 2002, p. 470).

Dentro da V&V pode-se utilizar duas técnicas (SOMMERVILLE, 2003, p. 358):

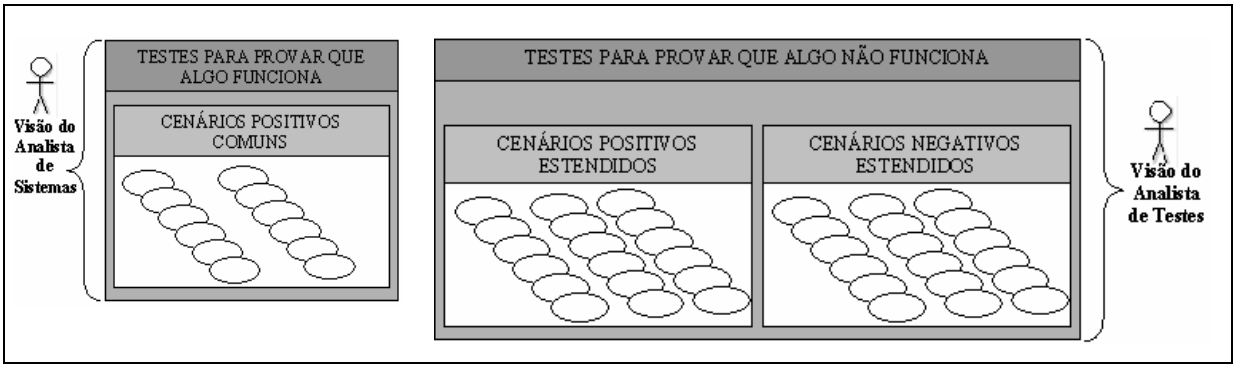
a) *inspeções de software*: analisam e verificam as representações do sistema, como o documento de requisitos, os diagramas do projeto e o código-fonte do programa. As inspeções podem ser aplicadas a todas as fases do processo de desenvolvimento. A inspeção é uma técnica estática, pois não requer que o software seja executado.

b) *teste de software*: envolve a execução do software com dados de teste e a análise das saídas e do comportamento operacional, a fim de verificar se ele está sendo executado conforme esperado. O teste é uma técnica dinâmica, pois trabalha com uma representação executável do sistema.

O teste de software é um processo sistemático e planejado que tem por finalidade a identificação de erros (BARTIÉ, 2002, p. 22). O teste é um elemento importantíssimo da garantia da qualidade do software, pois evita que o cliente se frustre com um software cheio de problemas. As técnicas e metodologias de teste de software têm um papel importantíssimo no processo de teste, isto é, elas fornecem diretrizes sistemáticas para projetar testes que (1) exercitam a lógica interna dos componentes de software e (2) que exercitam os domínios de

entrada e saída do sistema para descobrir erros de funcionalidade, comportamento e desempenho (PRESSMAN, 2002, p. 429).

Myers (1979, p. 5), ao afirmar que a função do teste de software é mostrar a presença de erros (e não a ausência), contribuiu para que o teste de software tomasse um rumo diferente. Na prática, isso significa que mostrar que algo não funciona é mais difícil (exige mais esforço) do que mostrar que funciona. Desenvolvedores de software são, por natureza, pessoas construtivas. O teste de software, ao contrário, exige pessoas com natureza destrutiva. A Figura 1 ilustra visões dos testes por perspectivas diferentes. A visão do analista de sistemas é, geralmente, provar que o software funciona, desta forma seus casos de teste serão limitados a cenários que provem que o sistema está funcionando. Já o analista de teste tem uma percepção mais ampla, isto é, além de testar as funcionalidades, seus casos de teste tentam mostrar falhas no software.

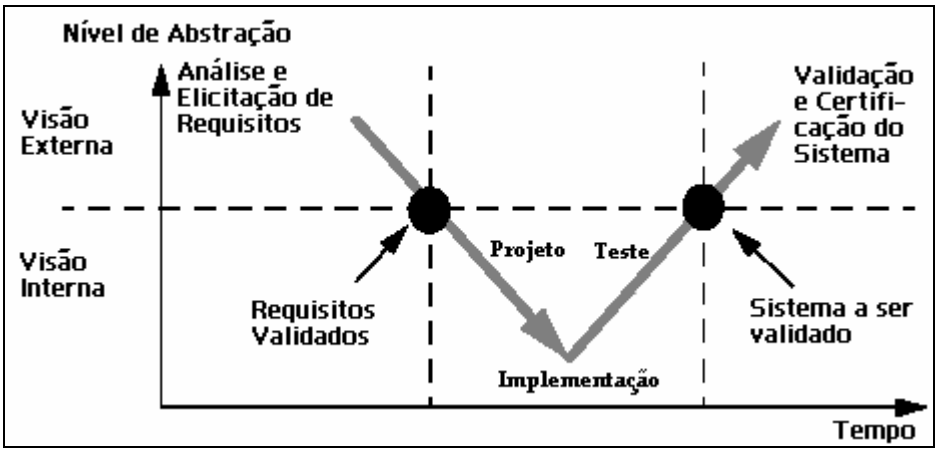


Fonte: Bartié (2002, p. 21)

Figura 1 – Visões dos testes por perspectivas diferentes

As atividades que envolvem a fase de teste são planejamento, projeto de casos de teste, execução e avaliação dos resultados. Elas devem ser conduzidas ao longo de todo o processo de desenvolvimento (ROCHA; MALDONADO; WEBER, 2001, p. 74). Pode-se salientar que a falta da atividade de planejamento é uma das causas dos problemas no desenvolvimento de software. É importante destacar que o processo de desenvolvimento de software passa por níveis de abstração diferentes ao longo do tempo. Desta forma testes em níveis diferentes de abstração devem ser planejados, projetados, executados e avaliados. A figura 2 mostra os diferentes níveis de abstração ao longo do processo de desenvolvimento. Nesta figura é apresentada uma visão interna e externa do processo de desenvolvimento de software ao longo do tempo. O processo começa com a análise e elicitação de requisitos de um ponto de vista mais abstrato (externo) e vai se detalhando até a implementação (visão interna). Assim que há uma implementação pronta, começam os testes, partindo de uma visão mais interna e

detalhada (teste de unidade, integração e sistema) até uma visão mais externa e abstrata (teste de validação e certificação do sistema).



Fonte: Regnell , Runeson e Wohlin (2003, p. 5)
Figura 2 – Diferentes níveis de abstração ao longo do processo de desenvolvimento

Antes dos testes serem efetivamente projetados, alguns princípios deveriam ser levados em conta (DAVIS apud PRESSMAN, 2002, p. 431-432):

- todos os testes devem ser relacionados aos requisitos do cliente: os defeitos mais indesejáveis de um software são aqueles que deixam de satisfazer os requisitos do usuário.
- os testes devem ser planejados muito antes do início do teste: o planejamento dos testes pode começar tão logo o modelo de requisitos esteja definido. A definição dos casos de teste pode começar assim que o modelo do projeto tenha sido consolidado. Sendo assim, todos os testes podem ser planejados e projetados antes do código ter sido gerado.
- o princípio de Pareto se aplica ao teste de software: aplicando o princípio de Pareto, implica que 80% de todos os erros descobertos durante os testes serão, provavelmente, relacionados a 20% de todos os componentes do programa. O grande problema é isolar estes componentes suspeitos e testá-los rigorosamente.
- o teste deve começar pela unidades do sistema até chegar no sistema como um todo: os primeiros testes planejados e executados geralmente concentram-se nos componentes individuais. À medida que o teste progride, os testes vão sendo aplicados na integração entre os componentes e, por fim, no sistema inteiro.
- testar completamente não é possível: a quantidade de permutações de caminhos é excepcionalmente grande. Desta forma, é impossível executar todas as

combinações de caminhos durante o teste. É possível, no entanto, cobrir adequadamente a lógica do programa e garantir que todas as condições no nível de componentes tenham sido exercitadas.

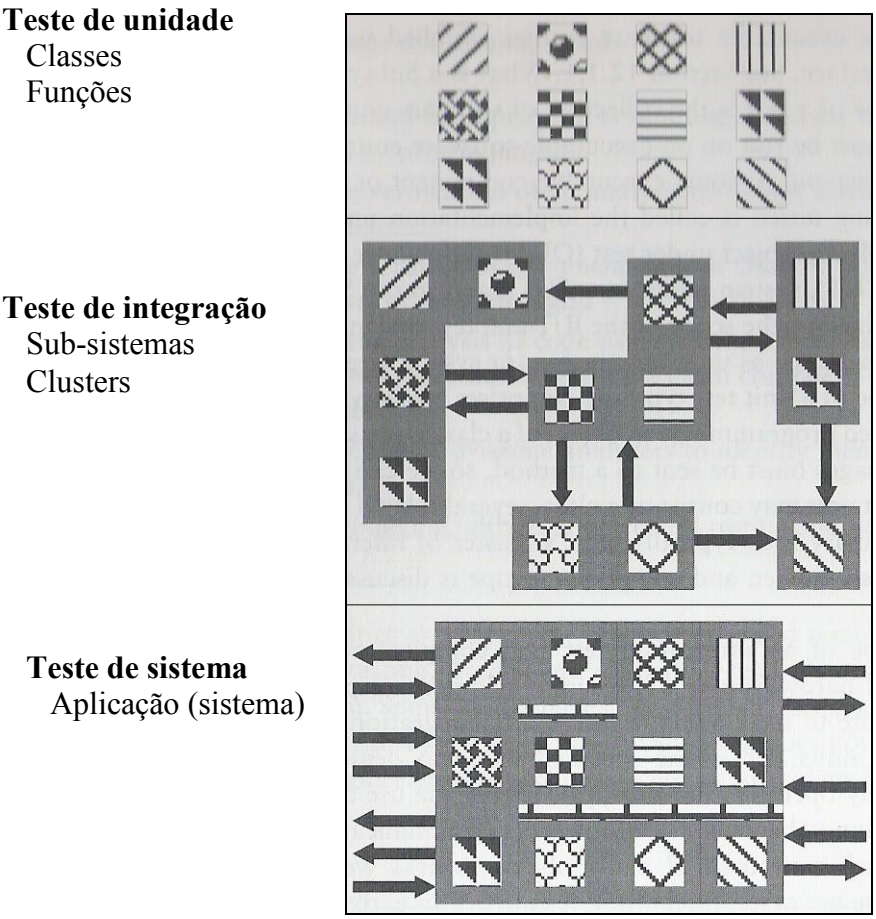
- para ser mais efetivo, o teste deveria ser conduzido por terceiros: como dito anteriormente, a natureza do testador deve ser destrutiva. Por outro lado, pessoas envolvidas no processo de desenvolvimento (construção) tem natureza construtiva. Isto é, quanto menos a pessoa estiver envolvida no processo de desenvolvimento, maior é a probabilidade dela encontrar erros.

O projeto de casos de teste pode ser feito de duas formas principais (PRESSMAN, 2002, p. 435):

- teste de caixa preta: também conhecido como teste funcional, é o teste conduzido na interface do software. Apesar de serem projetados para descobrir erros, são usados para demonstrar que as funções do software estão implementadas, ou seja, que a entrada é adequadamente aceita e a saída é corretamente produzida, e que a integridade das informações externas é mantida. O teste de caixa preta examina algum aspecto fundamental do sistema se preocupando pouco com a estrutura lógica interna do software.
- teste de caixa branca: também conhecido como teste estrutural, se trata de um exame rigoroso da estrutura interna do software. Caminhos lógicos, internos ao software são testados, definindo casos de testes que exercitam conjuntos específicos de condições e/ou ciclos.

Além dos casos de teste poderem ser desenvolvidos de duas formas, estes podem ser aplicados em diferentes escopos ou estágios. O escopo do teste é a coleção de componentes de software a ser verificada. Como os testes devem ser executados sobre uma implementação, o escopo é tipicamente definido para corresponder o componente que será testado. Tradicionalmente, o escopo de teste é divido em (BINDER, 2000, p. 45-46): teste de unidade, teste de integração e teste de sistema. A figura 3 mostra o relacionamento entre os testes de unidade, integração e sistema. Na parte superior desta figura são mostradas unidades de um software. A estas unidades são aplicados testes de unidade. Na parte central da figura é mostrado o agrupamento de unidades (módulos de um software) e a comunicação entre estes agrupamentos. A estes agrupamentos ou módulos e a comunicação entre eles, são aplicados

testes de integração. Por último, a junção dos módulos forma um sistema completo. Ao sistema completo são aplicados testes de sistema.



Fonte: Adaptado de Binder (2000, p. 46)

Figura 3 – Relacionamento entre os testes de unidade, integração e sistema

No teste de unidade (PRESSMAN, 2002, p. 476), testa-se a menor unidade do projeto de software, isto é, o menor componente de software. O teste de unidade é orientado para caixa branca e pode ser conduzido em paralelo para diversas unidades. No contexto de software convencional (desenvolvido com linguagens estruturadas), o teste de unidade deve ser conduzido pelos detalhes algorítmicos e o fluxo dos dados da unidade em teste. Já no contexto da orientação a objetos (OO), o teste de unidade é conduzido pelo teste da classe, onde a classe como um todo deve ser testada. Isto significa que não se pode mais testar uma única operação isoladamente (um método, por exemplo) e sim como parte de uma classe. O teste de classe deve ser conduzido pelas operações encapsuladas nas classes e pelo estado de comportamento da classe.

O objetivo do teste de integração é descobrir erros associados às interfaces entre a unidades do software (ROCHA; MALDONADO; WEBER, 2001, p. 75). No contexto da OO, o teste de integração pode ser feito de duas maneiras (PRESSMAN, 2002, p. 623). Na primeira, um conjunto de classes é integrado para responder a uma entrada ou evento do sistema. Na segunda, o teste começa com as classes que usam poucas (ou nenhuma) classes servidoras. Em seguida, estas classes independentes são integradas com outras classes dependentes e, então, são testadas. Isto se repete até que todo o sistema seja integrado e testado.

O escopo do teste de sistema é testar todo o sistema, levando em consideração componentes de software e hardware, ou seja, o software em seu provável ambiente de produção. Neste estágio são testadas características que estão presentes apenas no sistema como um todo. Entre os testes que são aplicados neste estágio estão testes funcionais (para validar requisitos do usuário), testes de performance (tempo de resposta) e teste de carga ou estresse (desempenho do sistema quando sobrecarregado) (BINDER, 2000, p. 45).

A Tabela 2 mostra a relação entre os estágios de teste e os paradigmas procedimental e OO.

Tabela 2 – Relação entre os estágios de teste e os paradigmas procedimental e OO

	Paradigma	
Estágio de Teste	**Teste Procedimental**	**Teste Orientado a Objetos**
Unidade	Sub-rotina ou função	Métodos das classes
Integração	Duas ou mais unidades	Classe *Cluster* Componente Subclasse
Sistema	Toda a aplicação	Toda a aplicação

Fonte: adaptado de Rocha, Maldonado e Weber (2001, p. 76)

Além deste três estágios de teste, alguns autores (como PRESSMAN, 2002, p. 487) apresentam um quarto estágio chamado de teste de validação ou teste de aceitação. A validação do software é feita por intermédio de uma série de testes de caixa preta que demonstram conformidade com os requisitos. Um plano de teste descreve os testes que devem ser conduzidos e um procedimento de teste define os casos de teste específicos que serão usados para demonstrar a conformidade com os requisitos. Tanto o plano quanto os casos de teste são projetados para garantir que todos os requisitos funcionais sejam satisfeitos, todas as

características comportamentais sejam conseguidas, todos os requisitos funcionais sejam alcançados, entre outros (PRESSMAN, 2002, p. 487).

Depois que cada caso de teste de validação tenha sido executado, uma de duas condições pode acontecer: (1) o software está de acordo com os requisitos e é aceito ou (2) um desvio da especificação é descoberto e é gerada uma lista de deficiências.

2.3 A UML E O TESTE DE SOFTWARE

2.3.1 Introdução a UML

A *Unified Modeling Language* (UML) é uma linguagem para modelagem de estruturas de software. Através da UML é possível visualizar, especificar, construir e documentar artefatos de software (BOOCH; RUMBAUGH; JACOBSON, 2000, p. 13).

A UML (versão 1.3) é composta por nove diagramas: diagrama de casos de uso, diagrama de objetos, diagrama de classes, diagrama de seqüência, diagrama de colaboração, diagrama de gráficos de estados, diagrama de atividades, diagrama de componentes, diagrama de implantação. Cada um dos diagramas possui funcionalidades distintas. A seguir tem-se uma breve explicação de cada diagrama. Mais detalhes sobre a construção de cada diagrama pode ser encontrado em Booch, Rumbaugh e Jacobson (2000):

a) *diagrama de casos de uso*: representa um conjunto de caso de uso e atores e seus relacionamentos. Este diagrama abrange a visão estática de formas de uso do sistema. É importante a utilização deste diagrama principalmente porque através dele é possível organizar e modelar os comportamentos do sistema. Além disso, os casos de uso servem para ajudar a validar a arquitetura e para verificar o sistema a medida que ele evolui durante seu desenvolvimento;

b) *diagrama de objetos*: mostra um conjunto de objetos (instâncias de classes) e seus relacionamentos. Um diagrama de objetos abrange uma visão estática de um conjunto de objetos e seus relacionamentos em determinado ponto no tempo;

c) *diagrama de classes*: é utilizado para fazer a modelagem da visão estática do sistema. Este diagrama serve para mostrar um conjunto de classes, interfaces e colaborações e os seus relacionamentos;

d) *diagrama de seqüência*: mostra a ordenação temporal das mensagens entre objetos. Ou seja, este diagrama mostra a seqüência das mensagens trocadas entre os objetos e qual o tempo de vida de cada mensagem;

e) *diagrama de colaboração*: exibe a organização dos objetos que participam de uma interação. Este diagrama mostra o relacionamento entre determinados objetos e quais as mensagens trocadas entre eles;

f) *diagrama de gráficos de estados*: exibe uma máquina de estados, que é formada por estados, transições, eventos e atividades. Este diagrama apresenta uma visão dinâmica do sistema em questão;

g) *diagrama de atividades*: apresenta uma visão dinâmica de um sistema. Este diagrama é um gráfico de fluxo, onde é mostrado o fluxo de controle de uma atividade para outra;

h) *diagrama de componentes*: exibe as organizações e as dependências existentes em um conjunto de componentes. Este diagrama é empregado para mostrar uma visão estática da implementação de um sistema. Está relacionado principalmente com o diagrama de classes, pois geralmente os componentes mapeados são classes, interfaces ou colaborações;

i) *diagrama de implantação*: exibe a configuração dos nós de processamento em tempo de execução e os componentes que nele existem. Este diagrama é utilizado para modelar uma visão estática da implantação de um sistema.

2.3.2 UML como modelo de teste

A UML é um mecanismo muito eficiente e prático, que traz muitos benefícios às atividades de análise e projeto de software. Utilizá-la na atividade de testes, sem dúvida, aumentaria mais ainda os benefícios ao desenvolvimento de software. Esforços não têm sido poupados para procurar maneiras de se utilizar a UML na atividade de testes. Dentre os trabalhos desenvolvidos tem-se Cavarra (2003) que utiliza os diagramas de classe, objeto e estados, Wittevrongel e Maurer (2004) propõe a geração de testes funcionais a partir de diagramas de seqüência, Ryser (2003) que utiliza casos de uso, Offutt (2003b) que é baseado no diagrama de estados, Offutt (2003a) que utiliza o diagrama de colaboração e Binder (2000)

que apresenta um guia com padrões de teste baseado na UML. O escopo deste trabalho está em utilizar o diagrama de casos de uso para testes de sistema e para validar requisitos, por isso, será dada ênfase no diagrama de casos de uso. Mais especificamente, este trabalho utilizará o padrão proposto por Binder (2000, p.722-731) e Heumann (2004), pois estes se apresentaram mais completos e abrangentes no estudo realizado, além de serem mais aplicáveis ao contexto de testes funcionais.

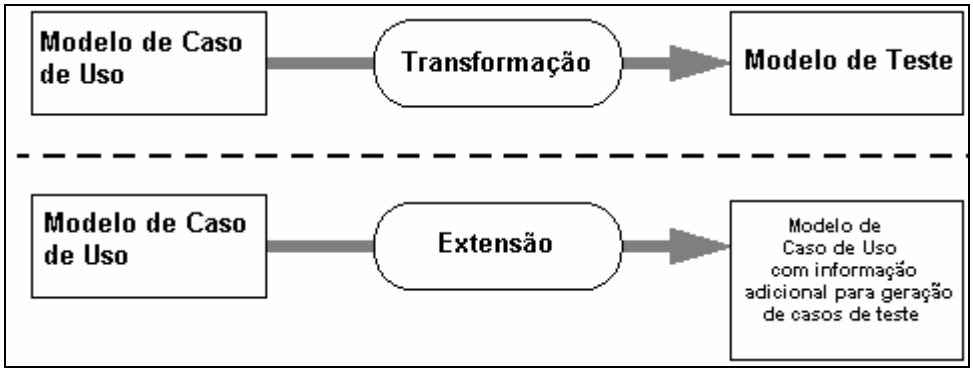
Para maiores informações sobre a utilização da UML no teste de software, vide o anexo A. Neste anexo estão relacionados os diagramas da UML e os padrões de projeto de teste relacionados a cada diagrama, conforme proposto por Binder (2000, p. 272-273).

2.3.3 Diagrama de Casos de Uso

Um sistema especificado através de casos de uso provê muitas das informações necessárias para se testar um sistema, pois ele descreve os requisitos funcionais de um software, isto é, pode-se testar e validar os requisitos de software com base nos casos de uso. Algumas metodologias para se testar um sistema baseando-se em casos de uso serão apresentadas a seguir:

2.3.3.1 Integração da modelagem de casos de uso e testes baseados em uso

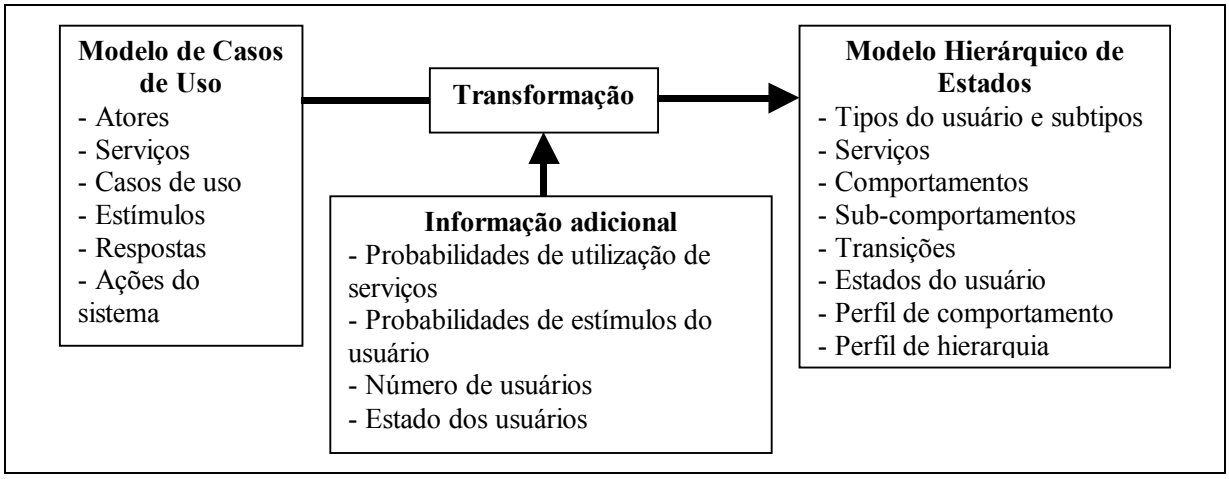
Regnell , Runeson e Wohlin (2003) propõe um modelo de uso como base para as disciplinas de engenharia de requisitos e teste. Além disso investigam a possibilidade de integrar as duas disciplinas. Basicamente, são apresentadas duas estratégias de integração entre as disciplinas: integração através de transformação de modelos e integração através de extensão de modelos. A figura 4 ilustra as duas estratégias.



Fonte: Regnell , Runeson e Wohlin (2003, p. 8)

Figura 4 – Estratégias de integração entre modelagem de requisitos e testes

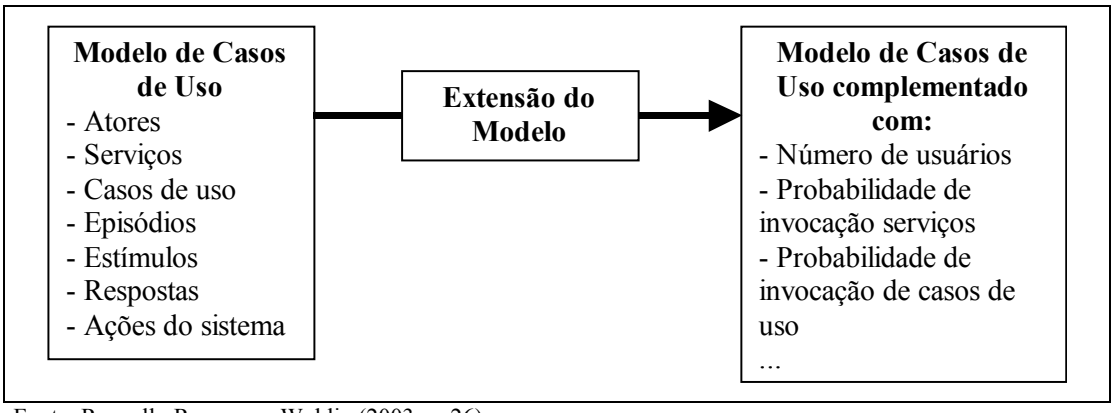
A estratégia de transformação de modelos é baseada na observação de que muitos dos conceitos utilizados na modelagem de casos de uso e teste estático de utilização têm semântica similar. A transformação consiste em adicionar informações de teste aos casos de uso, gerando um modelo hierárquico de estados. A figura 5 ilustra a estratégia de transformação.



Fonte: Regnell , Runeson e Wohlin (2003, p. 26)

Figura 5 – Transformando modelos de caso de uso em modelos hierárquico de estados

A estratégia de extensão do modelo é baseada na idéia de complementar o modelo de casos de uso em qualquer parte em que não haja uma escolha determinística em probabilidades de diferentes escolhas. Se for possível criar semânticas bem definidas de construção de casos de teste diretamente dos casos de uso, será possível economizar esforços na construção dos casos de teste. A figura 6 ilustra a estratégia de extensão dos casos de uso.



Fonte: Regnell , Runeson e Wohlin (2003, p. 26)

Figura 6 – Modelo de caso de uso com informação adicional

A estratégia de transformação tem a vantagem de ser baseada em duas disciplinas maduras e que termina com dois modelos especialmente definidos para seus propósitos. A maior desvantagem de utilizar esta estratégia é a necessidade de se utilizar os conceitos de duas disciplinas e ter de fazer a transformação entre os modelos das disciplinas. Já a estratégia

de extensão tem a vantagem de não precisar um segundo modelo. Ao invés disto, estende o modelo de caso de uso com informações para gerar casos de teste.

2.3.3.2 Geração de casos de teste a partir dos casos de uso

Heumann (2004) apresenta uma metodologia para criar casos de teste a partir dos casos de uso executando três passos:

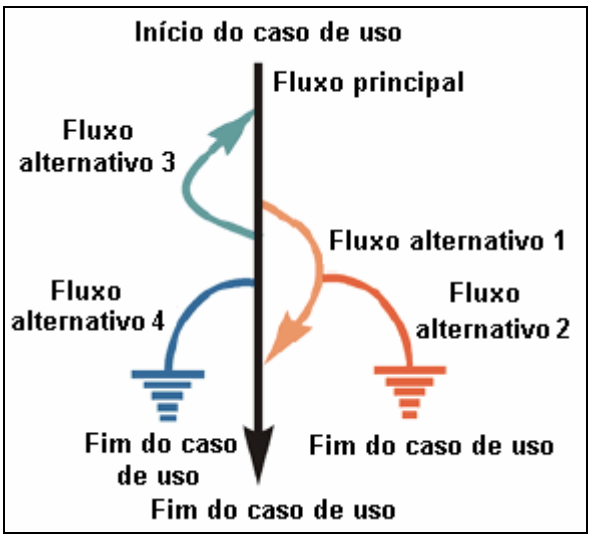
a) para cada caso de uso, gerar uma lista de cenários de casos de uso[1];

b) para cada cenário, identificar, ao menos, um caso de teste e as condições que o farão ser executado; e

c) para cada caso de teste, identificar os dados que serão utilizados para o teste.

No primeiro passo, deve-se ler a descrição textual do caso de uso e identificar o fluxo principal e os fluxos alternativos. Por exemplo, um caso de uso para descrever o uso de um sistema para fazer validação de usuários terá um fluxo principal que é: o usuário digita seu nome e senha corretamente e o sistema vai para a tela principal. Um fluxo alternativo seria: o usuário digita seu nome corretamente e senha inválida e, então, o sistema emite um aviso e permanece na tela de validação. A figura 7 mostra o fluxo principal e os fluxos alternativos para um caso de uso.

[1] Cenário de caso de uso: lista ordenada de interações entre parceiros, usualmente entre um sistema e uma lista de atores externos ao sistema. Pode caracterizar uma seqüência de passos para interação ou uma lista de possíveis interações (RYSER e GLINZ, 2003).



Fonte: Heumann (2004)

Figura 7 – Fluxo principal e os fluxos alternativos para um caso de uso

No segundo passo, os casos de teste para cada cenário devem ser identificados. Para tanto deve-se analisar os cenários, revisar a descrição do caso de uso e então criar o caso de teste. Criar uma tabela que represente valores de entrada e saída pode ser muito útil neste ponto, ajudando na organização e montagem dos casos de teste.

No terceiro e último passo, deve-se revisar e validar os casos de teste para assegurar a meticulosidade e identificar casos de teste redundante ou faltantes. Depois de aprovados, os dados de teste devem ser identificados. Sem os dados de teste, os casos de teste não podem ser executados.

Heumann (2004) afirma que com a metodologia apresentada, desenvolvedores podem simplificar o processo de teste, incrementar a eficiência e ajudar na certeza de se ter uma cobertura completa dos testes.

2.3.3.3 Teste de caso de uso estendido

Binder (2000, p. 722-731) apresenta um padrão chamado de *Teste de Caso de Uso Estendido* (TCUE), que tem por objetivo desenvolver testes de sistema pela modelagem dos requisitos essenciais em forma de casos de uso estendidos. Este padrão procura encontrar falhas como:

a) características de interação (comportamentos da interface com o usuário) indesejáveis;

b) saídas incorretas;

c) erro de execução levando o programa a ser terminado;

d) tempo inadequado de resposta do sistema;

e) funções (requisitos funcionais) omitidas;

f) funções extras (apareceram sem ser requisitadas), etc.

Primeiramente, os casos de uso estendidos devem ser desenvolvidos, sendo que estes devem conter as seguintes informações:

a) um inventário completo das variáveis operacionais;

b) uma especificação completa do domínio de cada variável operacional;

c) os relacionamentos operacionais para cada caso de uso;

d) a freqüência relativa para cada caso de uso (opcional).

Para cada caso de uso estendido deve-se executar os seguintes passos:

a) *identificar as variáveis operacionais*: as variáveis operacionais são todas as variáveis que são explicitamente parte da interface que suporta o caso de uso. Entre elas estão entradas e saídas do sistema, condições do ambiente que resultam em diferentes comportamentos dos atores e estados do sistema;

b) *identificar os domínios das variáveis operacionais*: os domínios são desenvolvidos definindo quais são os valores válidos e inválidos para uma variável;

c) *desenvolver os relacionamentos operacionais*: nesta etapa, é modelado o relacionamento entre as variáveis operacionais que determinam diferentes respostas do sistema. Esta modelagem pode ser feita através de uma tabela de decisão. Quando todas as condições de uma linha da tabela de decisões forem verdadeiras, uma ação esperada é produzida. Cada linha da tabela de decisão é chamada de variante. A tabela deve ser modelada de forma que cada linha seja

exclusiva, isto é, para determinada condição do sistema, apenas uma linha da tabela de decisão represente esta condição;

d) *desenvolver casos de teste*: cada variante da tabela é verdadeira e falsa pelo menos uma vez. Este passo requer dois casos de teste para cada variante: um que represente a variante falsa e outro que represente a variante verdadeira. Os resultados esperados para cada caso de teste são tipicamente desenvolvidos por inspeção, isto é, uma pessoa (provavelmente um analista de testes) observa os valores de entrada dos casos de teste e desenvolve os resultados esperados. Preferencialmente, esta pessoa deve ter grande conhecimento do negócio, principalmente do requisito em teste, e das formas como o sistema em teste deve ser usado.

O desenvolvimento dos casos de uso estendidos deve ser feito assim que os casos de uso foram desenvolvidos e validados. Já os casos de teste devem ser desenvolvidos assim que os casos de uso estendidos foram desenvolvidos e validados e o sistema em teste já tenha passado pelo teste de integração que demonstre que os componentes necessários para suportar o caso de uso estejam operacionais.

Como critério de aceitação e término dos testes, todos os requisitos precisam ser exercitados, pelo menos uma vez, para que se tenha cobertura mínima do sistema em teste. Binder (2000, p. 729) propõe uma métrica, chamada de cobertura de variantes (CV), que pode ajudar a definir a completude do sistema (ver figura 8).

$$CV = \frac{\text{Número de casos de uso implementados}}{\text{Número de casos de uso necessários}} \times \frac{\text{Total de variantes testadas}}{\text{Total de variantes}} \times 100$$

Fonte: Binder (2000, p. 729)
Figura 8 – Fórmula para calcular a cobertura dos testes dos casos de uso estendidos

Além disso, é possível rastrear os casos de teste para cada caso de uso conforme exemplo mostrado na tabela 3:

Tabela 3 – Matriz de rastreabilidade entre casos de uso e casos de teste

	Caso de teste 1	**Caso de teste 2**	**...**	**Caso de teste 9999**
Caso de uso 1	✓			✓
Caso de uso 2		✓		
...				
Caso de uso 9999	✓			✓

Fonte: Binder (2000, p. 729)

Entre as vantagens de utilizar este padrão estão (BINDER, 2000, p. 730):

a) casos de uso podem ser desenvolvidos por analistas e testadores que não possuem experiência no desenvolvimento orientado a objetos;

b) a utilização dos casos de uso está consolidada nos processos de análise e projeto, isso facilita o desenvolvimento dos casos de teste;

c) casos de uso refletem o ponto de vista do usuário (cliente), sendo que o foco dele está voltado para as funcionalidades que estão implementadas ou não e é isso que determinará se o sistema em teste atende ou não suas necessidades;

d) casos de uso estendidos provêem uma forma sistemática de desenvolvimento das informações necessárias para o projeto de teste. A informação deverá ser desenvolvida para testar qualquer caso de uso;

e) se o sistema em teste for desenvolvido a partir de casos de uso ambíguos, inconsistentes ou incompletos, estes logo serão apontados pelos testes.

Por outro lado, entre as desvantagens pode-se apontar (BINDER, 2000, p. 729-730):

a) casos de uso não são usados para especificar, entre outras, performance e tolerâncias a falhas. Portanto é necessário utilizar outras técnicas e metodologias para suprir estas deficiências;

b) a UML define construtores do tipo *extends* e *includes* para os casos de uso. Esta metodologia não está preparada para suportar as dependências entre casos de uso.

Radzius (2004) e Ryser e Glinz (2003) também apresentam metodologias para validação de casos de uso, porém estas não serão abordadas neste trabalho.

2.4 IEEE STD 829-1998 – PADRÃO PARA DOCUMENTAÇÃO DO TESTE DE SOFTWARE

O objetivo do padrão IEEE 829 (INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS, 1998) é descrever os documentos necessários para apoiar a atividade de teste de software. Os documentos descritos neste padrão abrangem o planejamento, especificação e a geração de relatórios de testes.

Um plano de testes deve descrever o escopo, plano de ação, recursos e cronograma da atividade de testes. Ele deve identificar os itens a serem testados, as características a serem testadas, as tarefas de teste a serem executadas, os responsáveis por cada tarefa e os riscos associados ao plano.

Três tipos de documentos fazem parte da especificação de teste:

a) o documento de especificação do projeto, que detalha o plano de ação, identifica as características que serão testadas e os casos de teste associados. Também identifica os casos e os procedimentos de teste, além de especificar o critério de aceitação e rejeição dos testes.

b) o documento de especificação do caso de teste que especifica os valores de entrada e as saídas esperadas. Além disso, identifica restrições nos resultados dos procedimentos de teste.

c) o documento de especificação do procedimento de teste que especifica todos os passos necessários para operar o sistema e exercitar os casos de teste.

Quatro tipos de documentos fazem parte do relatório de testes:

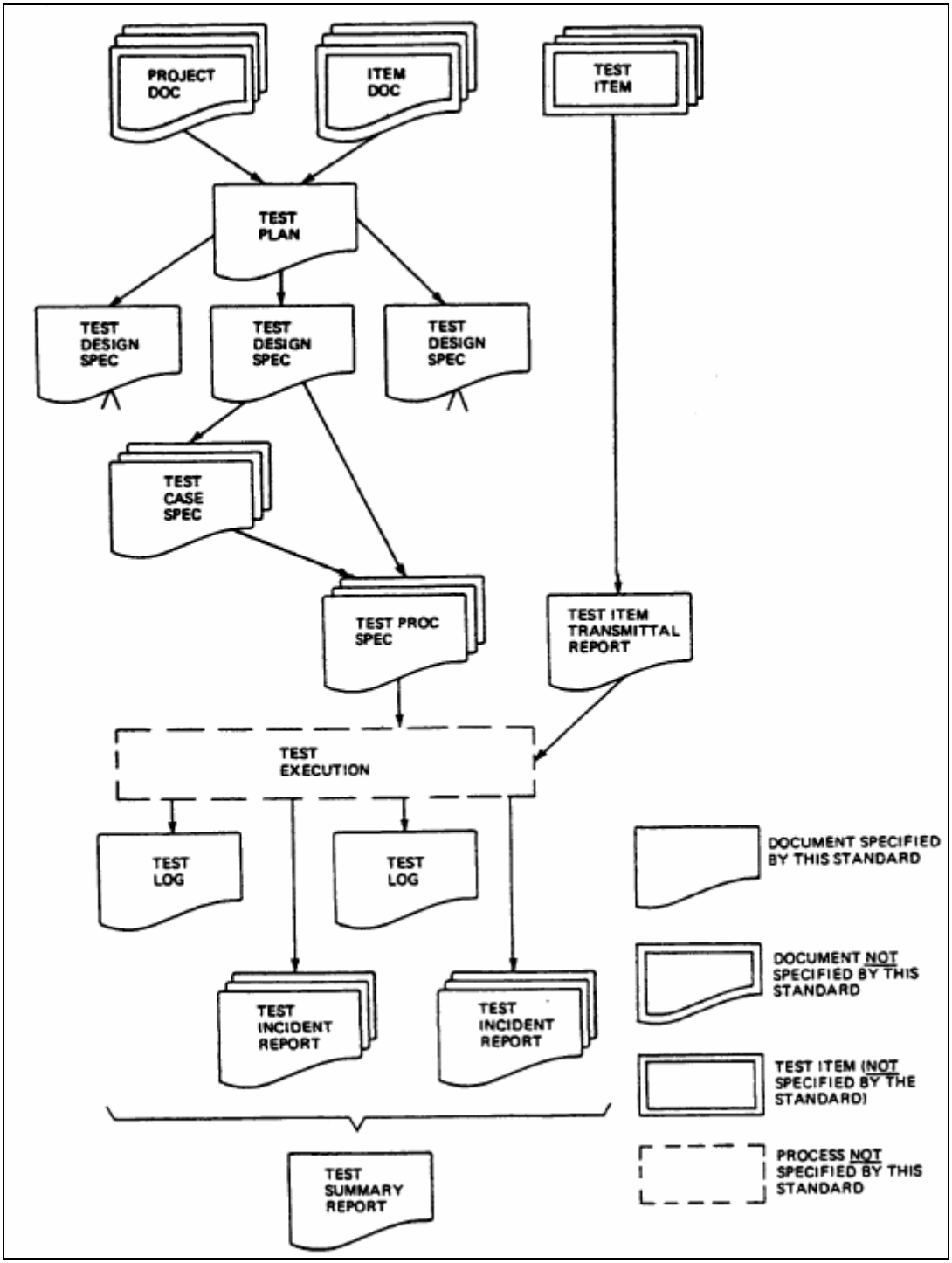
a) o relatório de transição de item de teste que identifica os itens de teste que deverão ser testados. Este documento é transmitido da equipe de desenvolvimento para a equipe de testes.

b) o relatório de *log* de execução (diário de bordo) que é usado pela equipe de teste para gravar o que aconteceu durante a execução dos testes.

c) o relatório de incidentes que descreve qualquer evento, que ocorreu durante a execução dos testes, que exija investigação.

d) o relatório de resumo dos testes que sumariza as atividades de teste associado a um ou mais projetos de teste.

A figura 9 mostra o relacionamento dos documentos de teste com o processo de teste.



Fonte: INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS (1998, p. iv)

Figura 9 – Relacionamento dos documentos de teste com o processo de teste

A seguir é apresentado cada um dos documentos que o padrão IEEE 829 especifica:

2.4.1 Especificação do plano de teste

O objetivo do plano é prescrever o escopo, o plano de ação, os recursos e o cronograma das atividades de teste. Um plano de teste deverá conter a seguinte estrutura:

a) *identificador único do plano*;

b) *introdução*: sumariza os itens de software e as características a serem testadas;

c) *características a serem testadas*: identifica todas características e combinações de características que serão testadas;

d) *características que não serão testadas*: identifica todas características e combinações de características que não serão testadas e qual a razão;

e) *plano de ação*: para cada característica ou conjunto de características deve ser descrito qual é o plano de ação que assegura que as mesmas serão adequadamente testadas. Deve ser especificado as principais atividades, técnicas e ferramentas que serão utilizadas para testar as características;

f) *critério de aceitação e rejeição dos testes*: descreve quais os critérios que serão utilizados para determinar se cada item de teste passou ou falhou;

g) *critério de suspensão e retomada*: descreve quais os critérios que serão utilizados para suspender e reiniciar todos ou parte dos testes;

h) *documentos*: identifica quais os documentos a serem gerados na atividade de teste;

i) *atividades de teste*: descreve a lista de tarefas necessárias para preparar e executar os testes;

j) *necessidades de ambiente*: descreve quais as necessidades de software, hardware e outros, para o ambiente de teste;

k) *responsabilidades*: descreve os grupos responsáveis pelo gerenciamento, projeto, preparação, execução, verificação e solução de problemas dos testes;

l) *mão de obra e treinamento*: descreve as necessidades de treinamento para capacitar as pessoas envolvidas no processo;

m) *cronograma*: descreve o cronograma da atividade de teste;

n) *riscos e contingências*: identifica os riscos do plano de testes e o plano de contingência;

o) *aprovação*: identifica os nomes e funções de todas as pessoas que precisam aprovar o plano.

2.4.2 Especificação do projeto de teste

O objetivo do projeto de teste é refinar o que foi definido no plano de teste e identificar as características que serão testas pelo projeto e seus casos de teste. Um projeto de teste deverá conter a seguinte estrutura:

a) *identificador único do projeto*;

b) *características que serão testadas*: identifica todas características e combinações de características que serão testadas pelo projeto. Para cada característica ou combinação de características deve haver a referência para o requisito associado;

c) *detalhamento do plano de ação*: descreve detalhe do plano de ação descrito no plano de teste. Inclui-se aqui a descrição de técnicas de teste a serem usadas. Além disso, deve incluir um resumo dos atributos comuns a todos os casos de teste;

d) *identificação dos casos de teste*: lista dos casos de teste que fazem parte do projeto;

e) *critérios de aceitação/rejeição*: descreve os critérios usados para determinar se uma característica ou conjunto de características passou ou falhou.

2.4.3 Especificação do caso de teste

O objetivo do caso de teste é definir o caso de teste identificado no projeto de teste. Um caso de teste deverá conter a seguinte estrutura:

a) *identificador único do caso*;

b) *itens do teste*: identifica os itens e características que serão exercitadas pelo caso de teste. Para cada item deve-se considerar os seguinte documentos associados ao item: especificação dos requisitos, especificação do projeto, guia do usuário, guia de operações e guia de instalação;

c) *especificação das entradas*: especifica cada entrada necessária para executar o caso de teste;

d) *especificação das saídas*: especifica todas as saídas e características dos itens de teste;

e) *necessidades de ambiente*: especificas as necessidades de ambiente de software, hardware e outros que são necessários para a execução do caso;

f) *procedimentos especiais*: descreve qualquer restrição especial no procedimento de execução do caso de teste.

g) *dependências entre casos*: lista os casos que devem ser executados antes deste caso.

2.4.4 Especificação do procedimento de teste

O objetivo do procedimento de teste é especificar os passos para execução do caso de teste. A especificação do procedimento de teste deverá conter a seguinte estrutura:

a) *identificador único do procedimento*;

b) *propósito*: descreve o propósito deste procedimento;

c) *requisitos especiais*: identifica qualquer requisito especial que seja necessário para a execução deste procedimento;

d) *passos para execução*: descreve os passos para execução do procedimento.

2.4.5 Relatório de transição de item de teste

O objetivo do relatório de transição de item de teste é identificar os itens que estão sendo enviados para a equipe de testes. Este documento deverá conter a seguinte estrutura:

a) *identificador único do relatório*;

b) *itens transmitidos*: identifica os itens em transmissão, incluindo a versão/revisão e o responsável pela transmissão;

c) *localização*: identifica a localização dos itens em transmissão, devendo ser descrito a mídia que contém os mesmos;

d) *status*: descreve o estado dos itens em transmissão;

e) *aprovação*: descreve os nomes e cargo das pessoas que aprovaram este documento.

2.4.6 *Log* dos testes (diário de bordo)

O objetivo do log é prover um histórico cronológico dos detalhes relevantes da execução dos testes. Este documento deverá conter a seguinte estrutura:

a) *identificador único*;

b) *descrição*: informações relativas a todas as entradas devem ser descritas;

c) *atividades e eventos de entrada*: os eventos e atividades devem ser gravados para que se tenha o histórico da execução. Entre as informações que devem ser incluídas no log estão a execução, o resultado de cada procedimento, informações de ambiente e eventos anormais;

2.4.7 Relatório de incidente

O objetivo do relatório de incidente é documentar qualquer evento que ocorreu durante o processo de teste e que requer investigação. Este documento deverá conter a seguinte estrutura:

a) *identificador único*;

b) *sumário*: resumo do incidente, identificando os itens envolvidos e sua versão/revisão;

c) *descrição do incidente*: descreve o incidente ocorrido;

d) *impacto*: indica qual o impacto do incidente sobre o plano, projeto, procedimento ou caso de teste.

2.4.8 Relatório com resumo dos testes

O objetivo deste relatório é sumarizar os resultados das atividades de teste projetadas e prover uma base para avaliação dos resultados. Este documento deverá conter a seguinte estrutura:

a) *identificador único*;

b) *sumário*: apresenta um resumo dos itens testados;

c) *divergências*: descreve qualquer diferença dos itens testados em relação ao que foi planejado;

d) *avaliação do plano*: apresenta uma avaliação do que foi executado em relação ao que foi planejado;

e) *resumo dos resultados*: sumariza os resultados dos testes. Identifica todos incidentes resolvidos apresentando um resumo das soluções. Além disso, descreve todos os incidentes não resolvidos;

f) *avaliação*: provê uma avaliação geral de todos os itens incluindo suas limitações;

g) *resumo de atividades*: sumariza as atividades e eventos mais importantes. Informações de totais de tempo e recursos utilizados em cada atividade devem ser apresentados;

h) *aprovação*: descreve os nomes e cargos das pessoas que devem aprovar este relatório.

Boa parte deste padrão foi adotado na concepção da ferramenta proposta neste trabalho e será citada futuramente no texto através da indicação da seção específica para um melhor entendimento e relação com o texto descrito.

2.5 A FERRAMENTA CASE ARGOUML

O ArgoUML (TOLKE, 2004) é uma ferramenta CASE para criação de diagramas da UML. Esta ferramenta é gratuita e seu código-fonte pode ser obtido pela internet (endereço: http://argouml.tigris.org). A ferramenta é simples, prática e não exige muito esforço para poder utilizá-la. A partir dos modelos é possível gerar código-fonte em Java. É possível também fazer a engenharia reversa a partir de código-fonte em Java. A Figura 10 mostra a interface da ferramenta. Uma característica interessante é que os dados dos diagramas podem ser exportados no formato *XMI* (*XML Metadata Interchange*), possibilitando assim levar dados do ArgoUML para outras ferramentas.

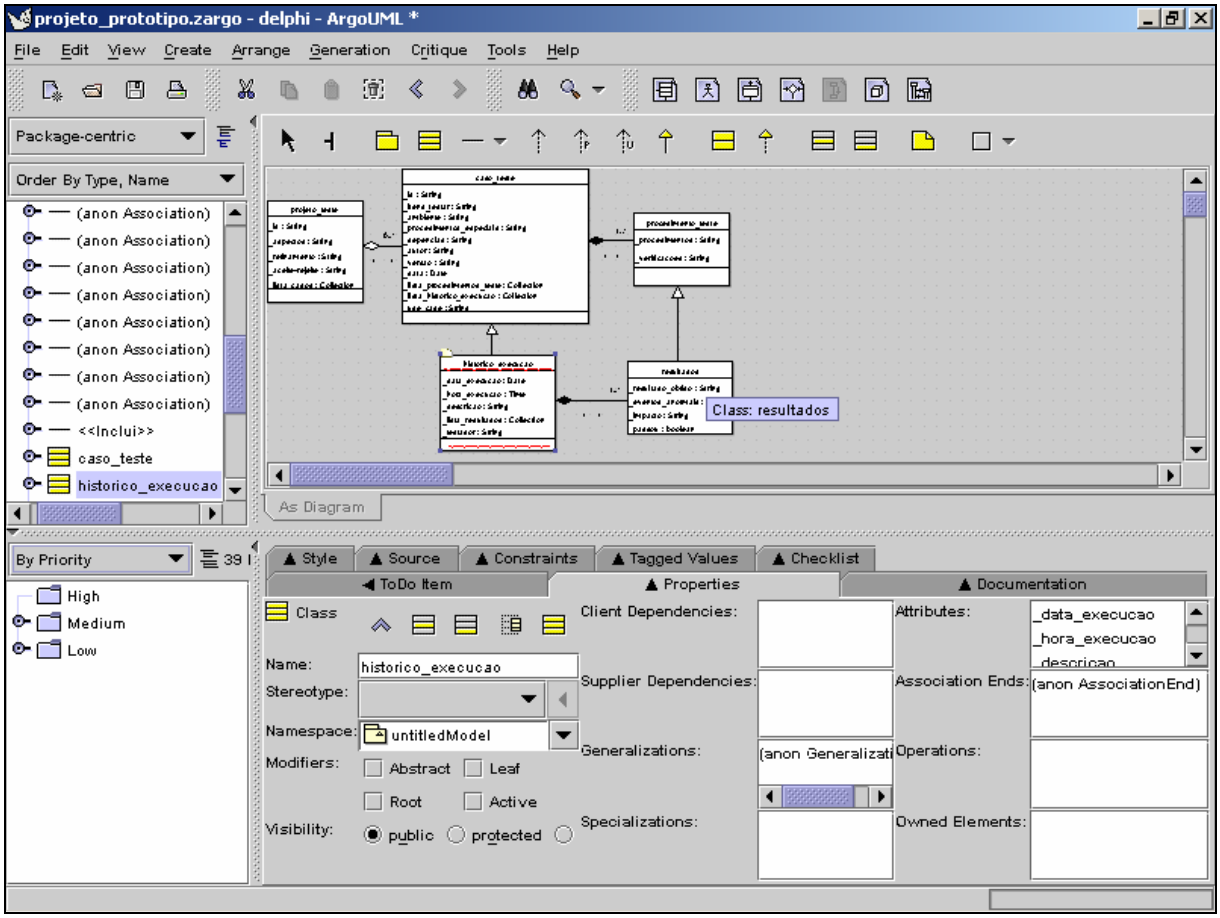


Figura 10 – Interface da ferramenta CASE ArgoUML

O projeto ArgoUML, que é composto por voluntários de todo o mundo, possui atualizações regulares e está aberto a qualquer pessoa que esteja interessada em contribuir na melhoria do mesmo. Entre as características tecnológicas desta ferramenta pode-se citar que ela é implementada em Java e que utiliza diversos outros projetos de código-fonte aberto. O projeto possui documentação de usuário e desenvolvedor, apesar da documentação do desenvolvedor estar defasada em relação ao projeto. Examinando o projeto, pode-se ver na prática a utilização de XML (*eXtensible Markup Language*), padrões de projeto, teste de unidade (utiliza o JUnit), Java e outros conceitos e tecnologias. A Figura 11 mostra os principais pacotes que compõem o projeto.

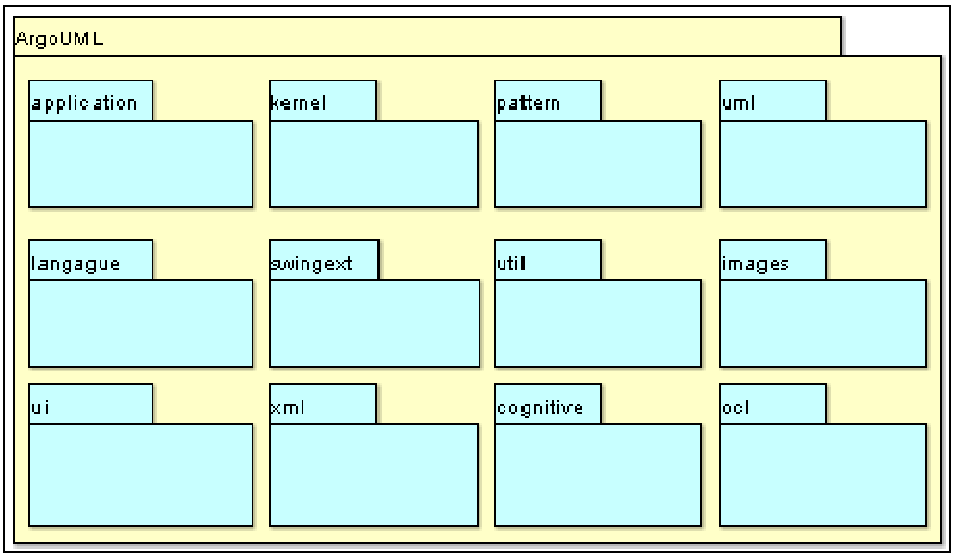


Figura 11 – Principais pacotes do projeto

Segue uma breve explicação sobre cada pacote:

a) *application*: contém a classe principal e é o ponto de partida do ArgoUML. Possui a definição das classes que tem abrangência sobre toda a aplicação;

b) *kernel*: é núcleo que controla toda a ferramenta (controle de eventos, exceção, etc);

c) *pattern*: possui classes que implementam padrões de projeto;

d) *uml* suporta classes que implementam os diagramas da UML;

e) *language*: provê suporte para múltiplas linguagens;

f) *swingext* possui extensões para a API Swing;

g) *util*: possui classes de diversas utilidades como, por exemplo, API para *log* de execução, que é muito útil para depuração;

h) *images*: todas as imagens utilizadas no projeto são armazenadas neste pacote;

i) *ui*: contém muitas das classes relacionadas a interface com o usuário;

j) *xml*: suporta o processamento dos documentos XML;

k) *cognitive*: define os elementos fundamentais do sistema de suporte cognitivo;e

l) *ocl*: contém as classes para suporte a *Object Constraint Languange* (OCL) – linguagem de restrição de objeto.

Fazem parte dos pacotes mencionados, outros projetos de código aberto, tais como:

a) *GEF (Graph Editing Framework)*: provê funções para a edição de gráficos em tempo de execução como, por exemplo, o desenho dos diagramas da UML e seus componentes;

b) *log4j*: provê funções que facilitam a geração de *logs* (arquivos de transação);

c) *SAX Parser Factory*: provê funções para manipulação de arquivos XML;

d) *Novosoft UML Library*: provê funções para manter e manipulador informação de meta-dados da UML.

O projeto ArgoUML está passando por uma reengenharia em busca da simplicidade e legibilidade do código-fonte e da performance da ferramenta. Mais informações sobre o projeto ArgoUML podem ser encontradas em Tolke (2004).

2.6 TRABALHOS CORRELATOS

Na FURB alguns trabalhos de conclusão de curso anteriores já trataram de testes de software. Serão citados três desenvolvidos nos últimos anos: Rosa (1997) apresenta um software de apoio ao testes funcional utilizando técnicas de caixa preta, análise de valor limite e particionamento por equivalência, cuja principal funcionalidade é gerar massa de dados para utilizar nos testes funcionais de software.

Santiago (2002), apresenta uma ferramenta de apoio para testes de programas utilizando teste de validação, cujo objetivo é auxiliar os alunos no aprendizado de programação, utilizando os componentes da biblioteca CLX do Delphi 6.0. Neste trabalho, a ferramenta aplica testes de caixa preta para efetuar a correção dos exercícios submetidos pelos alunos em uma disciplina de programação. Desta forma, ao concluir um exercício proposto pelo professor, o aluno o submeterá à correção. A partir de um conjunto de dados de entrada padrão, a ferramenta irá comparar os resultados alcançados pelo programa do aluno com os resultados esperados, informando ao aluno se o programa atende ou não as especificações do professor.

Sander (2002), apresenta uma ferramenta de suporte ao gerenciamento do teste de software com base nas recomendações existentes na norma ISO/IEC 12207 para o processo de gerência e para as atividades de teste. A ferramenta apresenta um roteiro que orienta o gerente na elaboração de planos de teste com o propósito de padronizar e melhorar o desempenho da atividade de teste.

3 DESENVOLVIMENTO DO TRABALHO

Neste capítulo a ferramenta desenvolvida é descrita, desde o levantamento de requisitos até a descrição de um exemplo de sua utilização.

O desenvolvimento da ferramenta aconteceu em duas etapas distintas. Na primeira etapa foram feitas customizações na ferramenta CASE ArgoUML visando sua adaptação às técnicas adotadas. Na segunda etapa foi desenvolvida a ferramenta de suporte ao planejamento, controle, execução e documentação de testes (aqui denominada de *TestCen*) que utiliza os diagramas estendidos gerados na ferramenta CASE ArgoUML.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA

Conforme já dito anteriormente, a baixa qualidade de software pode ter como fonte algumas das práticas:

a) os testes não são devidamente preparados, não há um plano de teste e os testes não são documentados;

b) o desenvolvimento dos casos de teste é feito depois que o sistema foi desenvolvido. Porém, os casos de teste podem (e deveriam) ser desenvolvidos assim que a especificação foi desenvolvida;

c) os casos de teste não são gerados de forma sistemática, isto é, estes são escolhidos de forma aleatória, com base no conhecimento do testador e sem procedimento definido.

Para gerar um produto de software com qualidade, um processo de qualidade (e teste) precisa ser implementado. Este processo deve utilizar metodologias para planejar e executar os procedimentos de garantia da qualidade, bem como, utilizar ferramentas que suportem o processo através da organização e documentação.

Utilizar uma metodologia para criar casos de teste com base nos cenários de casos de uso é uma forma de garantir que os requisitos do software foram implementados e atendem as necessidades do usuário (cliente). Acrescentando-se a esta metodologia, uma ferramenta que suporte a utilização desta, pode-se reduzir os riscos de insatisfação do usuário por não ter seus requisitos implementados. Neste contexto pensou-se em desenvolver uma ferramenta que

suporte o planejamento, execução e documentação do teste funcional de software com base no padrão IEEE 829.

Os principais requisitos funcionais a serem atendidos pela ferramenta incluem:

a) integração com ferramenta CASE: a ferramenta deve permitir a leitura dos casos de testes criados na ferramenta CASE ArgoUML;

b) planejamento de testes: a ferramenta deve permitir a manutenção dos cadastros necessários ao planejamento de testes incluindo os projetos de testes, casos de testes e respectivos procedimentos de testes;

c) execução dos casos de teste: a ferramenta deve permitir o registro de informações sobre a execução dos casos de testes criados e armazenar o histórico destas execuções;

d) relatório de erros: a ferramenta deve emitir um relatório com os erros encontrados na execução de um caso de teste junto com um resumo do projeto de teste;

e) geração de gráficos: a ferramenta deve gerar gráficos sobre a cobertura de testes realizados e testes executados.

Como principal requisito não-funcional pode-se destacar a aderência ao padrão IEEE 829 que apresenta diversas recomendações sobre a documentação de testes de software.

3.2 ESPECIFICAÇÃO

A seguir são representados o diagrama de casos de uso, com suas descrições, e o diagrama de classes da ferramenta.

3.2.1 Especificação dos casos de uso

Na figura 12 está representado o diagrama de casos de uso da ferramenta. Os atores envolvidos com a ferramenta são: o analista de testes responsável pela especificação dos casos de teste (o que testar e como testar) e o testador responsável pela parte operacional dos testes (execução). O analista de testes deve ter domínio do negócio do sistema em desenvolvimento.

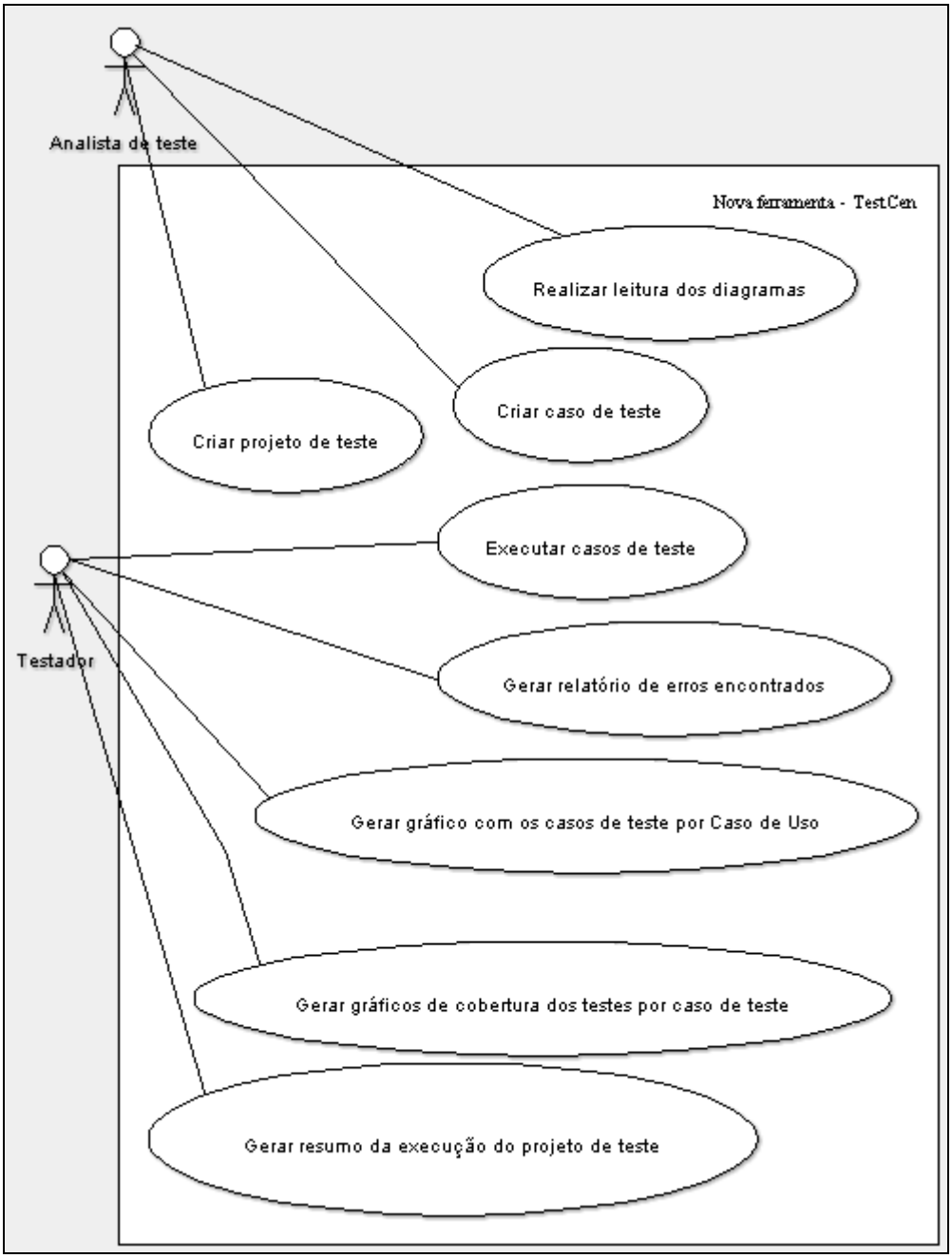


Figura 12 – Casos de uso da ferramenta de suporte ao teste de funcional

A seguir é apresentada uma breve descrição de cada caso de uso. Quando existir alguma relação com o padrão IEEE 829 o número da seção é citado entre parênteses.

a) realizar leitura dos diagramas*:* Este caso de uso tem como pré-condição a criação de diagrama de casos de uso e suas extensões de teste na ferramenta CASE ArgoUML. A partir disto o analista de teste pode exportar os dados de teste para um arquivo de interface (arquivo formato XMI) e importá-los na ferramentas TestCen (figura 13). Os atributos da especificação do caso de teste

devem ser baseados no padrão IEEE 829 para especificação do caso de teste (2.4.3) e especificação do procedimento de teste (2.4.4). Este caso de uso representa a relação com a ferramenta CASE ArgoUML;

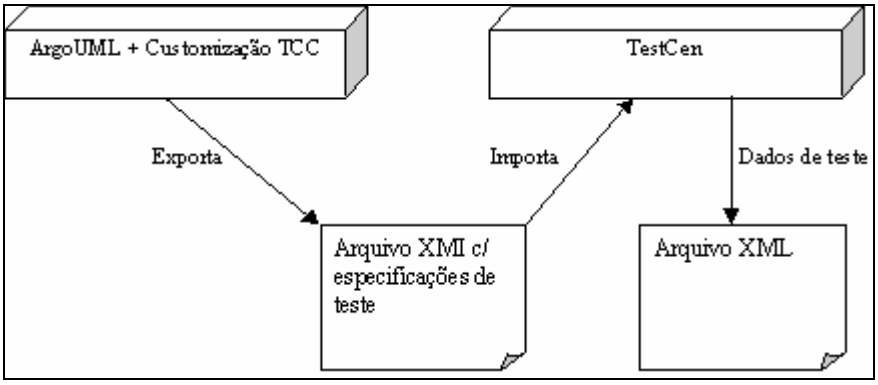


Figura 13 – Fluxo de dados entre as ferramentas

b) criar projeto e caso de teste: o analista de testes cria um projeto de teste e os casos de teste que compõe o projeto. Cada caso de teste pode estar associado a um caso de uso. Os atributos devem ser baseados no padrão IEEE 829 para especificação do projeto de teste (2.4.2), especificação do caso de teste (2.4.3) e especificação do procedimento de teste (2.4.4);

c) executar casos de teste: a partir dos casos de teste planejados, o testador pode escolher um caso, de cada vez, para ser executado. Para cada execução do caso é criado um histórico, com os dados utilizados nos testes e os resultados obtidos, para posterior consulta;

d) gerar relatório de erros encontrados: O testador seleciona um caso de uso e poderá gerar um relatório dos erros encontrados. Os atributos do relatório devem ser baseados no padrão IEEE 829 para relatórios de incidentes (2.4.7);

e) gerar gráfico com casos de teste por caso de uso: O testador poderá selecionar e gerar um gráfico do tipo barra com o número de casos de teste por caso de uso;

f) gerar gráficos de cobertura dos testes: O testador poderá selecionar e gerar um gráfico na tela do tipo pizza com o número de casos de teste executados e não executados em relação ao total de casos de teste criados;

g) gerar resumo da execução do projeto de teste: O testador poderá selecionar e gerar um relatório com sumário do projeto de teste. Os atributos do relatório devem ser baseados no padrão IEEE 829 para relatórios de resumo dos testes (2.4.8).

3.2.2 Diagrama de classes

O diagrama de classe, apresentado na figura 14, representa as classes de negócio da ferramenta TestCen. Os atributos das classes estão baseados no padrão IEEE 829 e os métodos das classes dizem respeito a consulta e manutenção dos valores dos atributos.

A classe *projeto_teste* mantém as informações que dizem respeito a todos os casos de teste, além da lista de casos de teste associados ao projeto. Esta classe representa a especificação do projeto de teste descrito pelo padrão IEEE 829 (ver 2.4.2).

A classe *caso_teste* mantém informações relativas a configuração dos casos de teste e, também a lista de procedimentos de teste. A especificação do caso de teste do padrão IEEE 829 (2.4.3) é representada por esta classe.

A classe *procedimento_teste* guarda informações para a efetiva execução dos testes. Representa a especificação do procedimento de teste do padrão IEEE 829 (2.4.4).

As classes *histórico_execução* e *resultados* guardam dados do histórico de execução dos casos de teste e os resultados obtidos. Estes dados são usados nos relatórios e gráficos definidos nos requisitos. Os relatórios representam o relatório de incidentes (2.4.7) e relatório de resumo dos testes (2.4.8)

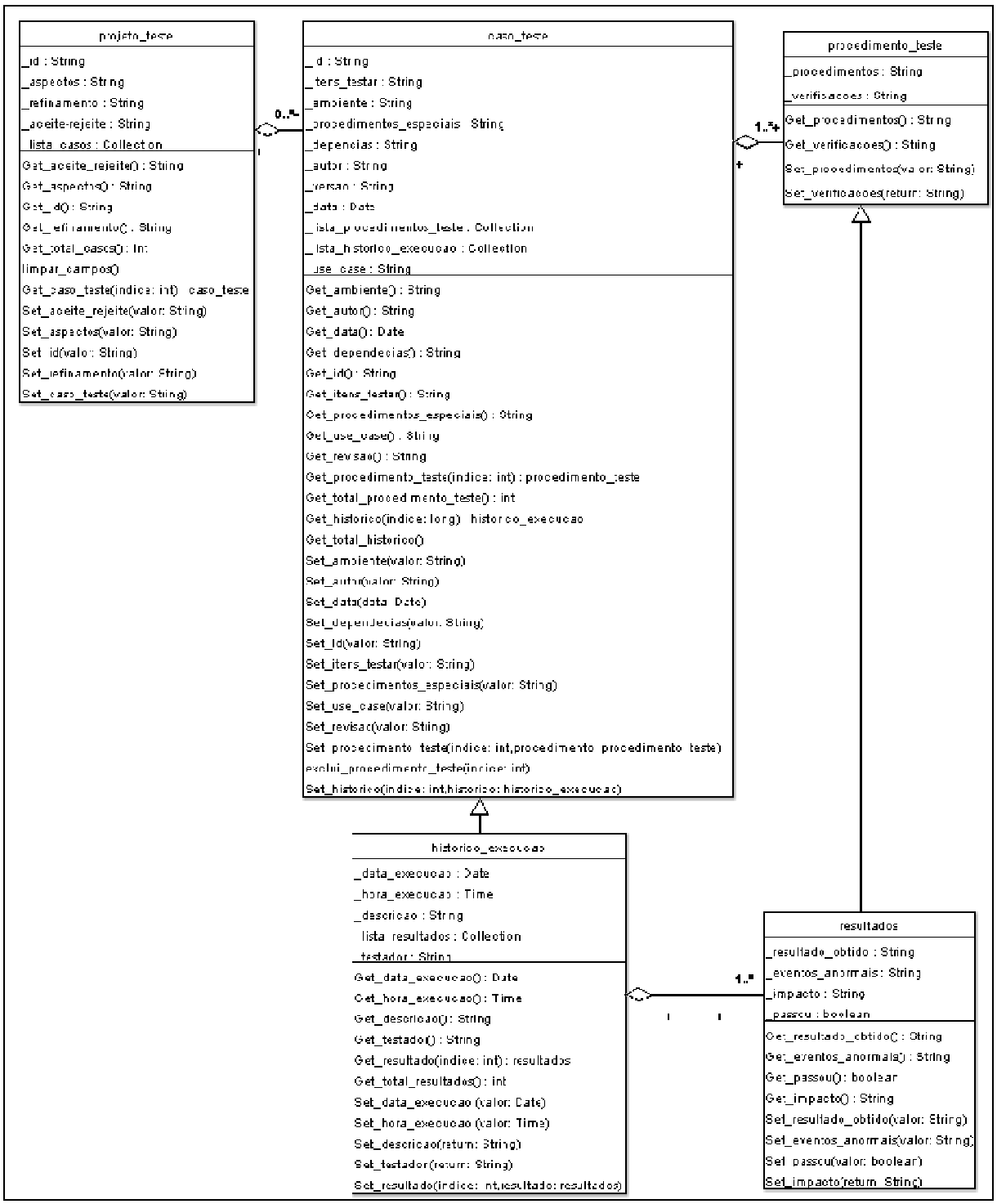


Figura 14 – Classes de persistência da ferramenta TestCen

As classes da customização da ferramenta ArgoUML estão representadas na figura 15. Todas as classes estão dentro do pacote *org.argouml.uml*. A descrição do nome de cada classe é composto pelo nome do sub-pacote, o nome da classe e separado por ponto. Por exemplo, a classe *FigTestCase* encontra-se em *ui*, este por sua vez está em *use_case* e este encontra-se em *diagram*. Mais informações sobre as classes serão apresentadas no item 3.1.1.

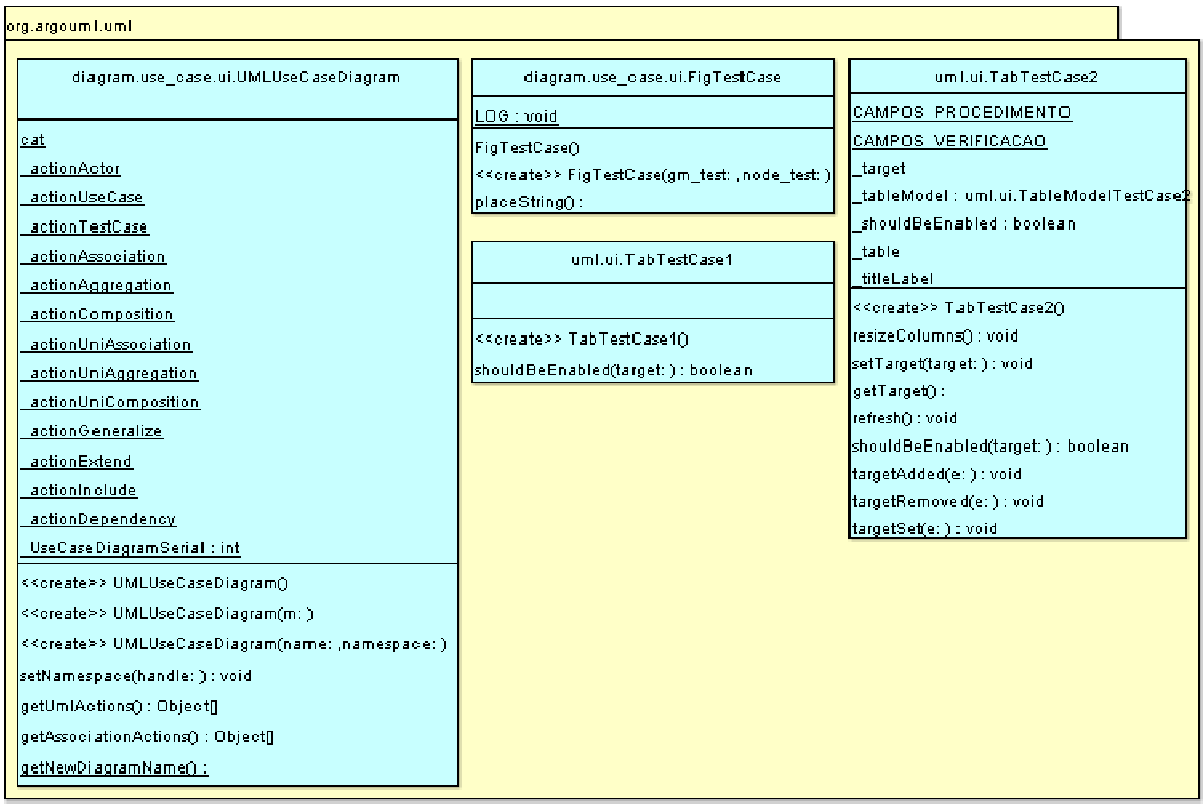


Figura 15 – Classes da customização da ferramenta ArgoUML

3.3 IMPLEMENTAÇÃO

3.3.1 Alterações na ferramenta ArgoUML

As classes criadas para representar os casos de teste foram implementadas no pacote *uml*, conforme mostra a figura 16.

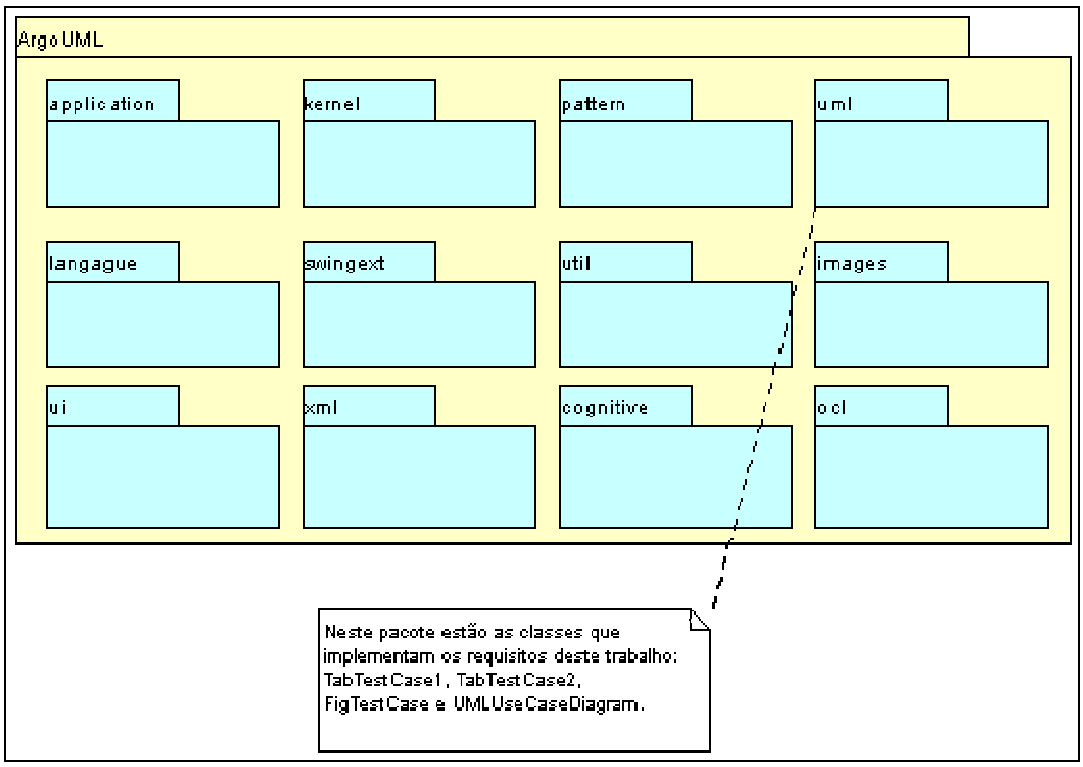


Figura 16 – Classes implementados e/ou alteradas no ArgoUML

A classe *org.argouml.uml.ui.TabTestCase1* mantém as informações do autor, revisão, data, ambiente necessário e procedimentos especiais do caso de teste compatíveis com padrão IEEE 829 visto na seção (2.4.3). Estas informações são persistidas como *tagged values* (mecanismo de extensão da UML, ver BOOCH, RUMBAUGH e JACOBSON 2000, p.74-88) do diagrama de casos de uso. Para criar *tagged values* foi utilizada a classe *UMLModelElementTaggedValueDocument* do ArgoUML, conforme mostra a figura 17.

```
//Adiciona os campos
addField(
         "Autor",
         new UMLTextField2(new UMLModelElementTaggedValueDocument("autor")));

addField(
         "Revisão",
         new UMLTextField2(new UMLModelElementTaggedValueDocument("revisao")));

addField(
         "Data",
         new UMLTextField2(new UMLModelElementTaggedValueDocument("data")));

UMLTextArea2 _ambiente =
    new UMLTextArea2(new UMLModelElementTaggedValueDocument("ambiente"));
```

Figura 17 – Criação de campos com persistência na forma de tagged values

A figura 18 mostra o resultado gerado pela classe *TabTestCase1*.

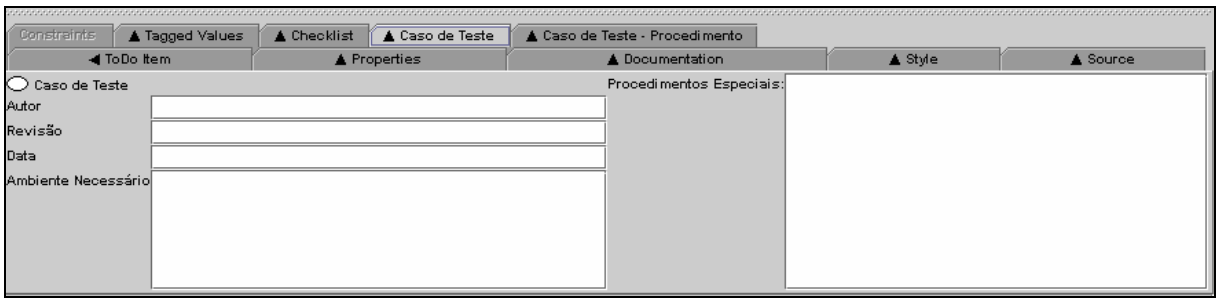


Figura 18 – Guia Caso de Teste, implementada pela classe TabTestCase1

A classe *org.argouml.uml.ui.TabTestCase2* implementa a guia que está representada na figura 19.

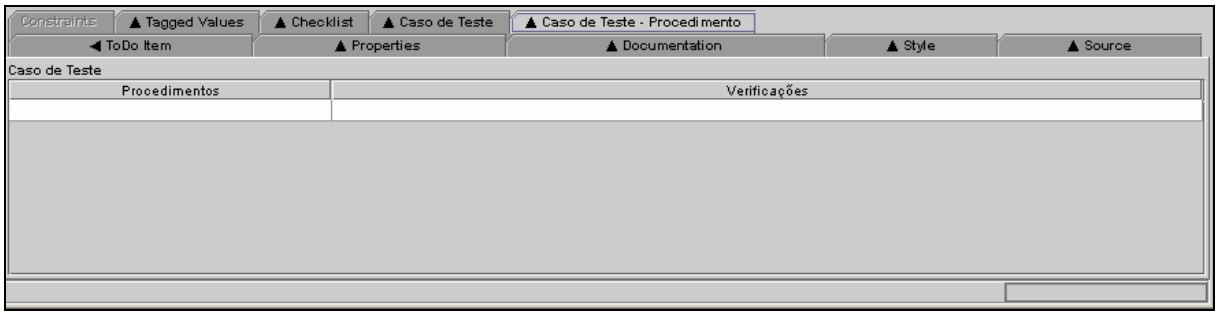


Figura 19 – Guia *Caso de Teste - Procedimento*, implementada pela classe *TabTestCase2*

Nesta classe são mantidos os dados dos procedimentos de teste (entradas e saídas). Estas informações também são persistidas como *tagged values* do diagrama de casos de uso. A classe TableModelTestCase2 implementa uma tabela que recebe os dados digitados e transforma em *tagged values*, conforme mostra a figura 20. As *tags* são formadas pelos prefixos procedimento_ e verificação_ mais o número da linha em que se encontram os dados e cada *value* recebe o valor digitado na tabela.

```
if (tvs.size() <= rowIndex) {
    if (columnIndex == 0){
        ModelFacade.setTaggedValue(_target, "procedimento_"+ rowIndex, (String) aValue);
    }else
    if (columnIndex == 1) {
        ModelFacade.setTaggedValue(_target, "verificacao_"+ rowIndex, (String) aValue);
    }

    mEvent = new TableModelEvent(this, tvs.size(), tvs.size(),
                                 TableModelEvent.ALL_COLUMNS, TableModelEvent.INSERT);

}
```

Figura 20 – Persistência dos procedimentos de teste em forma de tagged values

Mais informações sobre mecanismos de extensão da UML podem ser encontradas em Booch, Rumbaugh e Jacobson (2000, p.74-88).

A classe *org.argouml.uml.diagram.use_case.ui.FigTestCase* implementa o estereótipo [2] que representa o caso de teste (ver figura 21). Este estereótipo tem como base o elemento caso de uso. Esta classe é herdeira da classe *FigUseCase* (classe que representa o caso de uso) e a ela foi dado o nome de estereótipo *Caso de Teste*. A figura 22 mostra parte do código que implementa a classe *FigTestCase* e o estereótipo *Caso de Teste*.

Figura 21 – Estereótipo de Caso de Teste

```
// Cria estereótipo Caso de Teste
Object newStereo = ExtensionMechanismsFactory.getFactory().buildStereotype(
                                        /*(MModelElement)*/ (Object)node_test, "Caso de Teste");
TargetManager.getInstance().setTarget(newStereo);
```

Figura 22 – Trecho do código que implementa o estereótipo de Caso de Teste

A classe *org.argouml.uml.diagram.use_case.ui.UMLUseCaseDiagram* é do projeto original. Ela foi customizada, para possibilitar a inserção de casos de teste no diagrama de casos de uso (ver figura 23).

Figura 23 – Barra de ferramentas customizada para permitir a inclusão de casos de teste

3.3.2 Ferramenta de planejamento e execução de testes (TestCen)

Esta ferramenta foi desenvolvida utilizando a linguagem Object Pascal. A figura 24 mostra a tela principal da ferramenta em tempo de projeto. Para a implementação da interface com o usuário foram utilizados componentes visuais do Delphi, tais como TForm, TTreeView, TPageControl, TMainMenu, TMemo e outros.

Do lado esquerdo da figura 24 está o componente TTreeView que mostra um projeto de teste e seus casos de teste em forma de árvore. No primeiro nível da árvore está o projeto

[2] Estereótipo: mecanismo de extensão da UML, cuja a função é estender o vocabulário da UML através da criação de novos tipos de elementos que são derivados de outros já existentes.

de teste e no segundo nível estão os casos de teste associados ao projeto de teste. A árvore é montada em tempo de execução a partir das instâncias das classes projeto_teste e caso_teste.

Do lado direito da figura 24 é utilizado o componente TPageControl, que possui três guias. A primeira guia (Informações do Projeto) mostra as informações do projeto de teste. Esta guia aparece apenas se, no componente TTreeView, é selecionado o objeto projeto_teste. A segunda e terceira (Informações do Caso de Teste e Casos de Teste – Procedimentos) mostram os dados de um caso de teste. Ao selecionar um caso de teste no TTreeView, os dados do objeto correspondente são mostrados nestas guias. Para mostrar os dados foram utilizados os componentes TEdit, TMemo e TStringGrid.

Na parte superior da figura 24 está o menu de opções e, abaixo deste, está a barra de ferramentas com atalhos para itens do menu de opções.

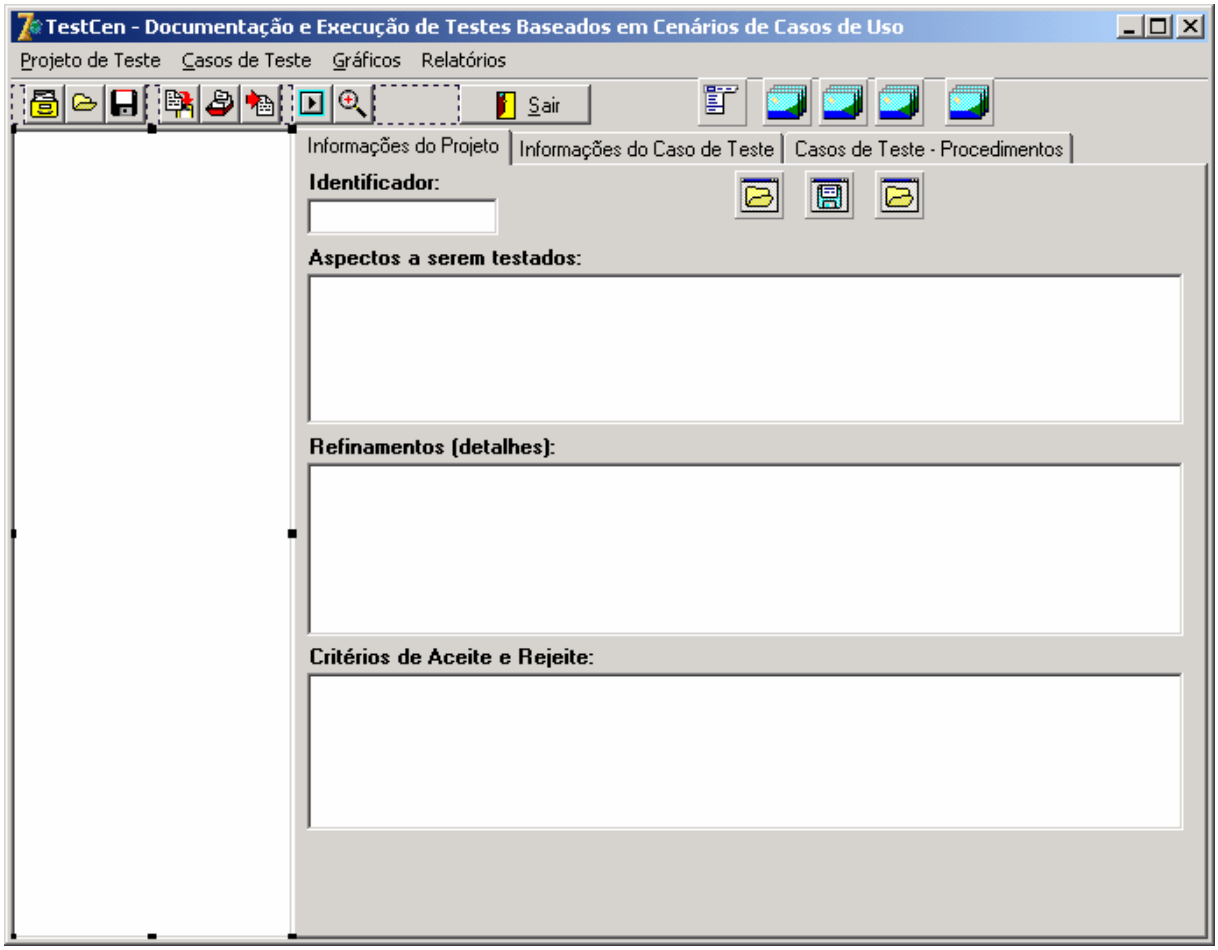


Figura 24 – Imagem da tela principal da ferramenta TestCen

No menu Projeto de Teste, estão opções para criar, fechar, abrir e salvar um projeto de teste. No menu Casos de Teste, estão as opções para incluir, excluir, importar e executar casos

de teste. Além disso, pode-se visualizar o histórico de execuções de um caso de teste e exibir, em forma de árvore, os casos de teste por caso de uso.

Já no menu Gráficos, existem as opções de gráfico de casos de teste por caso de uso, que mostra a quantidade de casos de teste por caso de uso, e gráfico de cobertura de casos de teste, que mostra o número de casos de teste testados e não testados em relação ao total de casos de teste.

No último menu, Relatórios, existem as opções para imprimir os dados de um caso de teste onde aconteceram erros na execução e imprimir o resumo do projeto de teste.

A camada de persistência é formada por instâncias das classes apresentadas na figura 14. Os dados mantidos nos campos da interface com o usuário são gravados nos objetos da camada de persistência. Os dados dos objetos são persistidos em disco no formato XML através da opção salvar do menu Projeto de Teste ou no botão salvar da barra de ferramentas. A figura 25 mostra um exemplo da estrutura do arquivo XML de um projeto de teste.

```
- <projeto_teste>
    <id>Projeto1</id>
    <aspectos>Descreva aqui os itens a serem testados e as características e/ou conjunto de características
      que serão objeto deste projeto de teste.</aspectos>
    <refinamento>Descreva aqui detalhes do que foi definido no planejamento macro de teste. Por exemplo,
      especifique técnicas que serão utilizadas para auxiliar nos testes.</refinamento>
    <aceite_rejeite>Descreva aqui os critérios que serão utilizados para determinar quando os testes serão
      aceitados ou rejeitados.</aceite_rejeite>
  - <caso_teste>
      <id>Novo Caso</id>
      <itens>Descreva aqui os itens e características que serão testadas. Os seguintes documentos deveriam
        ser mencionados aqui: documento de requisitos, especificação de projeto, guias de usuário, guias de
        operação e guias de instalação.</itens>
      <ambiente>Descreva aqui as características e configurações de software e hardware necessárias para
        execução do teste. Aqui podem ser incluídas descrições do sistema operacional necessário,
        compiladores, simuladores e outras ferramentas de teste. Também podem ser descritas outras
        necessidades como, por exemplo, pessoas com treinamento especial.</ambiente>
      <prod_especiais>Descreva aqui qualquer restrições especial sobre o procedimento de execução deste
        caso de teste</prod_especiais>
      <dependecias>Descreva aqui os identificadores dos casos de teste que devem ser executados antes de
        executar este caso.</dependecias>
      <autor>Autor desta especificação</autor>
      <revisao>1.0</revisao>
      <data>16/06/2004</data>
      <use_case>Identificador do caso de uso</use_case>
    - <procedimento_teste>
        <procedimentos>Descreva aqui os procedimentos de execução do teste, incluindo dados de
          entrada.</procedimentos>
        <verificacoes>Descreva aqui as verificações que devem ser feitas, incluindo tempo de resposta,
          saídas em relatório e na interface com o usuário, etc.</verificacoes>
      </procedimento_teste>
    </caso_teste>
  + <caso_teste>
  </projeto_teste>
```

Figura 25 – Estrutura do arquivo XML de um projeto de teste de exemplo

Na classe TfrmTestCen (herdeira da classe TForm), foram criados métodos para implementar as regras de negócio da ferramenta. Destaca-se, entre os métodos, o método *importa_XMI* que faz a importação dos dados, exportados do ArgoUML, para a ferramenta. O apêndice B mostra o código de implementação da importação. Para auxiliar na leitura do arquivo XMI, foi utilizado assistente *XML Data Binding* do próprio Delphi. Baseado em um arquivo XMI, criado com o ArgoUML, o assistente criou classes e interfaces para ler e processar os dados do arquivo. O método *importa_XMI* implementa as regras de quais dados devem ser importados e onde devem ser gravados dentro da ferramenta TestCen.

Para implementação dos gráficos, foi utilizada a classe TChart e para gerar os relatórios foi utilizada a classe TQuickRep.

3.3.3 TÉCNICAS E FERRAMENTAS UTILIZADAS

Para a modelagem foi utilizada a UML com a ferramenta CASE ArgoUML. Para a customização da ferramenta ArgoUML foi utilizada a IDE (*Integrated Development Environment* – ambiente integrado de desenvolvimento) NetBeans juntamente com o pacote de desenvolvimento Java™ 2 *SDK Standard Edition* versão 1.4.2. Para a ferramenta de planejamento e execução de testes foi utilizada a IDE Delphi 7.

Para o desenvolvimento de todo o trabalho foram usadas técnicas de análise e projeto de software orientado a objetos com UML (BEZZERA, 2002) e técnicas de teste funcional de software, além de conceitos de arquitetura de software e XML.

3.3.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Para facilitar o entendimento da ferramenta desenvolvida, foi realizado um conjunto de testes para um sistema fictício de controle de uma biblioteca. Neste sistema pode-se cadastrar o acervo, alunos, bibliotecários, bem como controlar os empréstimos e devoluções de livros. Entre os casos de uso obtidos, tem-se um caso principal que é o empréstimo de livros, que é mostrado na figura 26. A tabela 4 descreve o caso de uso *emprestar livro*.

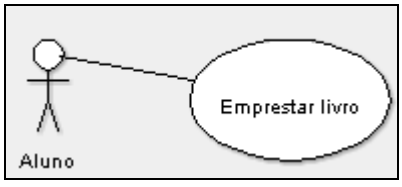


Figura 26 – Caso de uso - empréstimo de livros

Inicialmente o Analista precisa criar os casos de uso na ferramenta CASE ArgoUML. Aplicando as metodologias de teste baseado em caso de uso, apresentadas por Binder (2000) e Heumann (2004), foram desenvolvidos manualmente os casos (cenários) de teste apresentados na figura 27. Estes casos (identificados pelo estereótipo <<*Caso de Teste*>>) foram especificados, no diagrama de casos de uso, utilizando a modificação aplicada na ferramenta CASE ArgoUML.

Tabela 4 – Descrição do caso de uso de empréstimo de livros

Caso de uso emprestar livro	
Identificador	CU001
Ator primário	Aluno
Ator secundário	Bibliotecário
Pré-condições	O aluno deve estar cadastrado no sistema de acadêmicos e ter um cadastro e senha na biblioteca.
Fluxo Principal	O aluno informa os livros a serem emprestados. A bibliotecária faz a leitura do código de barras de cada livro e o sistema registra a data e hora do empréstimo, o código do livro e qual a data de devolução do mesmo.
Fluxo Alternativo 1	O aluno tenta emprestar um livro, porém existem livros emprestados cuja data de entrega já venceu. O sistema não deve permitir o empréstimo.
Fluxo Alternativo 2	O aluno tenta emprestar um livro, mas existem multas não pagas de atraso na entrega de livros. O sistema não deve permitir o empréstimo.
Fluxo Alternativo 3	O aluno tenta emprestar um livro, mas o número máximo de empréstimos já excedeu. O sistema não deve permitir o empréstimo.
Pós-condições	Para o fluxo principal, o sistema deverá ter registrado com sucesso o empréstimo do livro. Para os fluxos alternativos o sistema deverá emitir mensagens de erro e/ou alerta.

A criação dos casos de teste começou com a análise da descrição textual do caso de uso, onde foi encontrado o fluxo principal e os fluxos alternativos. Neste caso, os fluxos estavam explicitamente descritos, mas poderiam não estar. Em seguida foram identificadas as variáveis operacionais, isto é, o código do livro, o código do aluno, a data de empréstimo, as multas não pagas, os livros em atraso e o número de livros já emprestados. Para auxiliar na criação dos casos de teste, foi criada uma tabela (temporária) contendo as variáveis

operacionais e seus valores para teste. A tabela 5 mostra as variáveis operacionais para o caso de teste do fluxo principal (*Testar empréstimo de livros*).

Tabela 5 – Variáveis operacionais e valores para teste

Código do aluno	**Código do livro**	**Data e hora do empréstimo**	**Data da devolução**
001	005	Data e hora do momento do empréstimo	Data do momento de empréstimo + 7 dias

Por último, foram desenvolvidos os casos de teste, incluindo os procedimentos para execução dos testes. Estes foram desenvolvidos com base nas descrições dos casos de uso e nas variáveis operacionais identificadas. A tabela 6 apresenta a descrição do caso de teste do fluxo principal (*Testar empréstimo de livros*).

Tabela 6 – Procedimento de teste para fluxo principal -*Testar empréstimo de livros*

Procedimentos	**Verificações**
Na tela de validação do aluno, digitar como código do aluno **001** e senha **abc123** (já previamente cadastrados).	O sistema deverá ir para a tela de empréstimo de livros.
Informar como código do livro a ser emprestado **005**.	O sistema deverá trazer as informações do livro: nome autor, descrição, editora e n° da edição.
Confirmar a empréstimo do livro para o aluno **001**.	O sistema deverá mostrar uma mensagem de confirmação do empréstimo, seguido de uma lista de livros emprestados pelo aluno, onde para o livro 005 a data e hora do empréstimo deve ser a data e hora do momento de empréstimo e a data de devolução deve ser a data do momento de empréstimo + 7 dias.

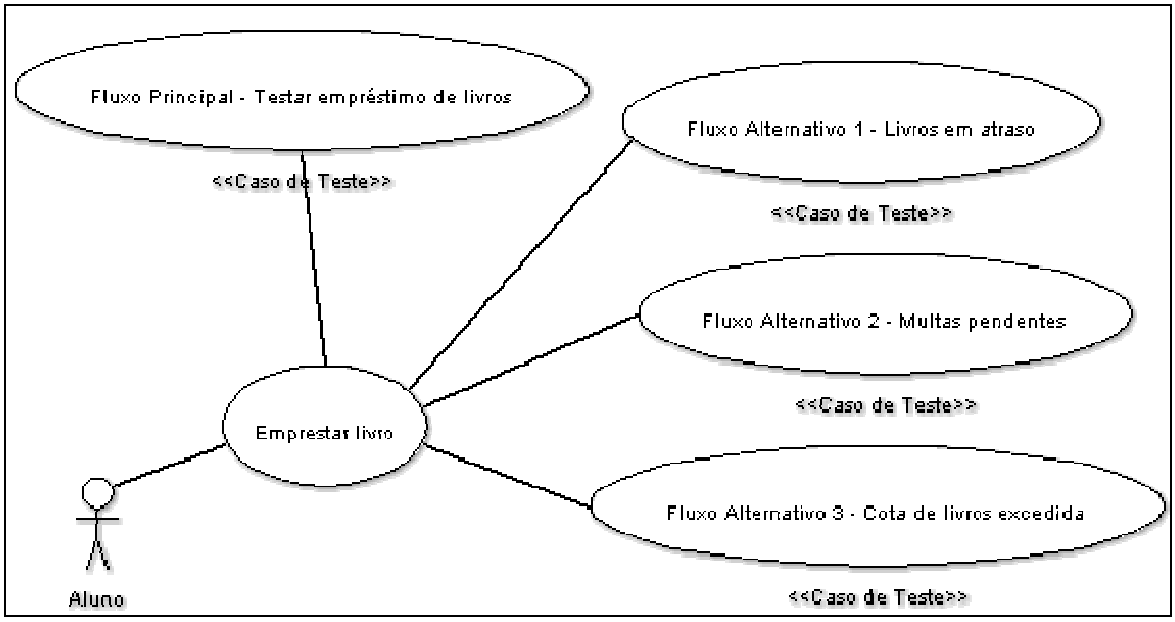


Figura 27 – Testes de cenário para o caso de uso *Emprestar livro*

As figuras 28 e 29 mostram as especificações do caso de teste fluxo principal (*Testar empréstimo de livros*) na customização feita no ArgoUML. A figura 29 descreve os procedimentos para execução do teste. Na tabela 6 foram descritos os procedimentos, desta forma, bastou apenas copiar os procedimentos do caso de teste para o ArgoUML.

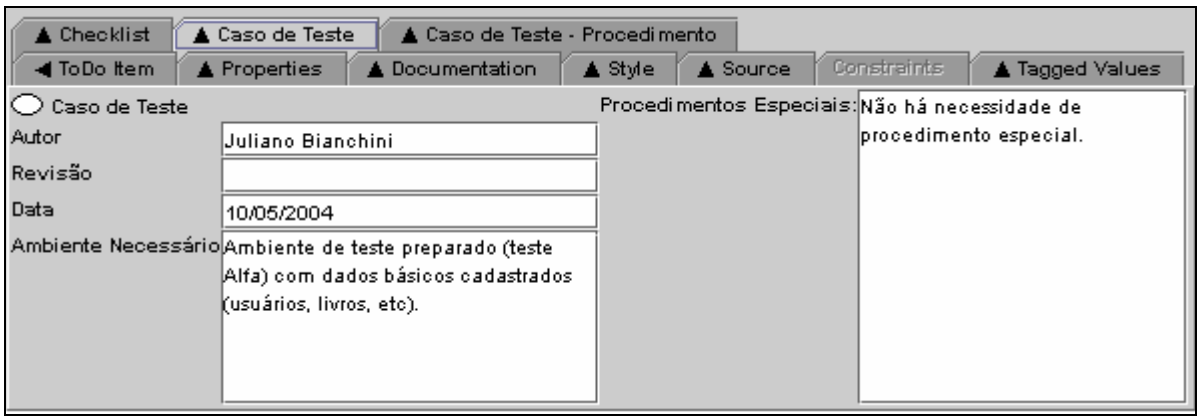


Figura 28 – Especificação do caso de teste fluxo principal (*Testar empréstimo de livros*)

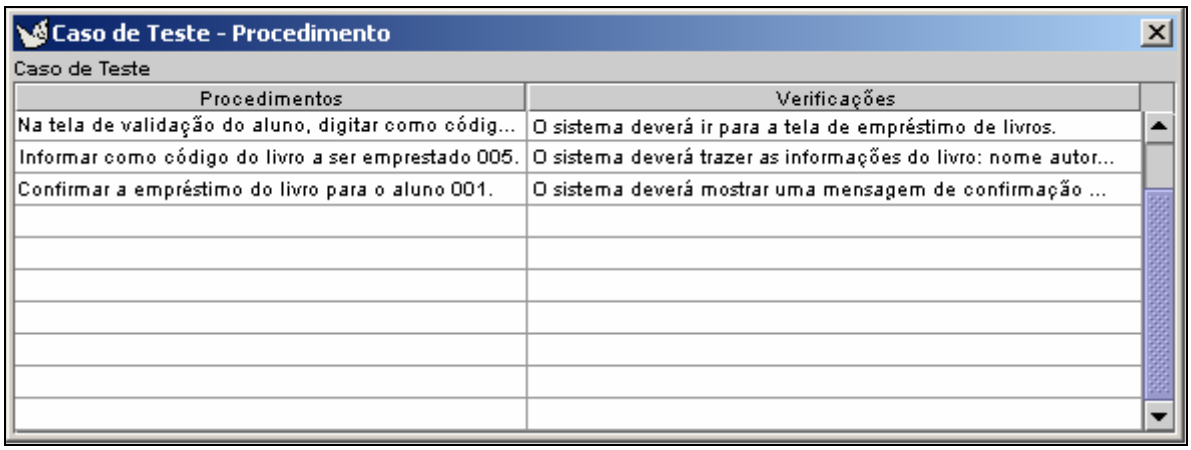


Figura 29 – Procedimentos do caso de teste principal (*Testar empréstimo de livros*)

3.3.4.1 Especificação dos casos de teste na ferramenta TestCen

Após os casos de teste terem sido especificados no ArgoUML, as informações podem ser exportadas para um arquivo *XMI* (XML Metadata Interchange), através do menu *Tools*, opção *Export as XMI*. Este arquivo pode ser importado pela ferramenta TestCen, através do menu *Casos de Teste* opção *Importar XMI (ArgoUML)* (ver figura 30). Vale lembrar que antes de importar os casos de teste, é necessário criar um projeto de teste.

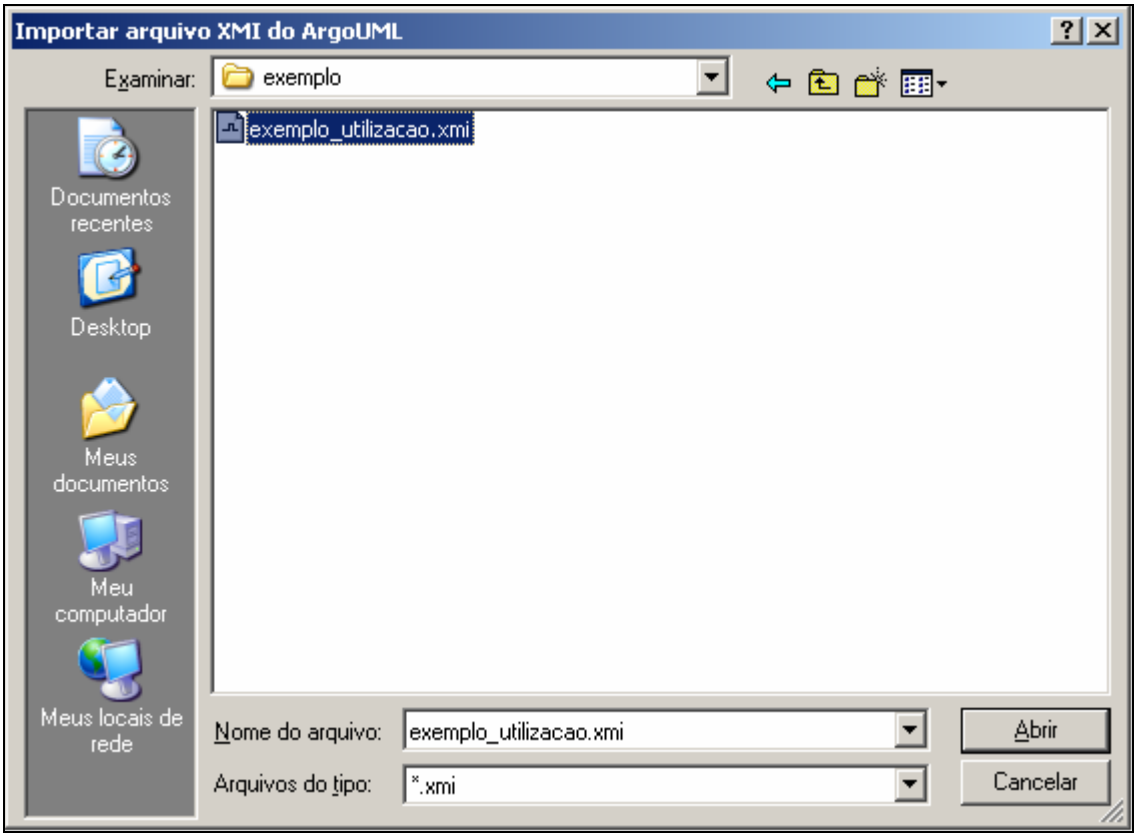


Figura 30 – Importação do arquivo XMI do ArgoUML

O próximo passo é complementar as especificações dos casos de teste, caso as informações importadas estejam incompletas. Neste exemplo as informações de teste foram importadas, mas elas poderiam ser digitadas, pois a ferramenta permite a digitação ou importação das especificações de teste. A figura 31 ilustra a especificação dos casos de teste.

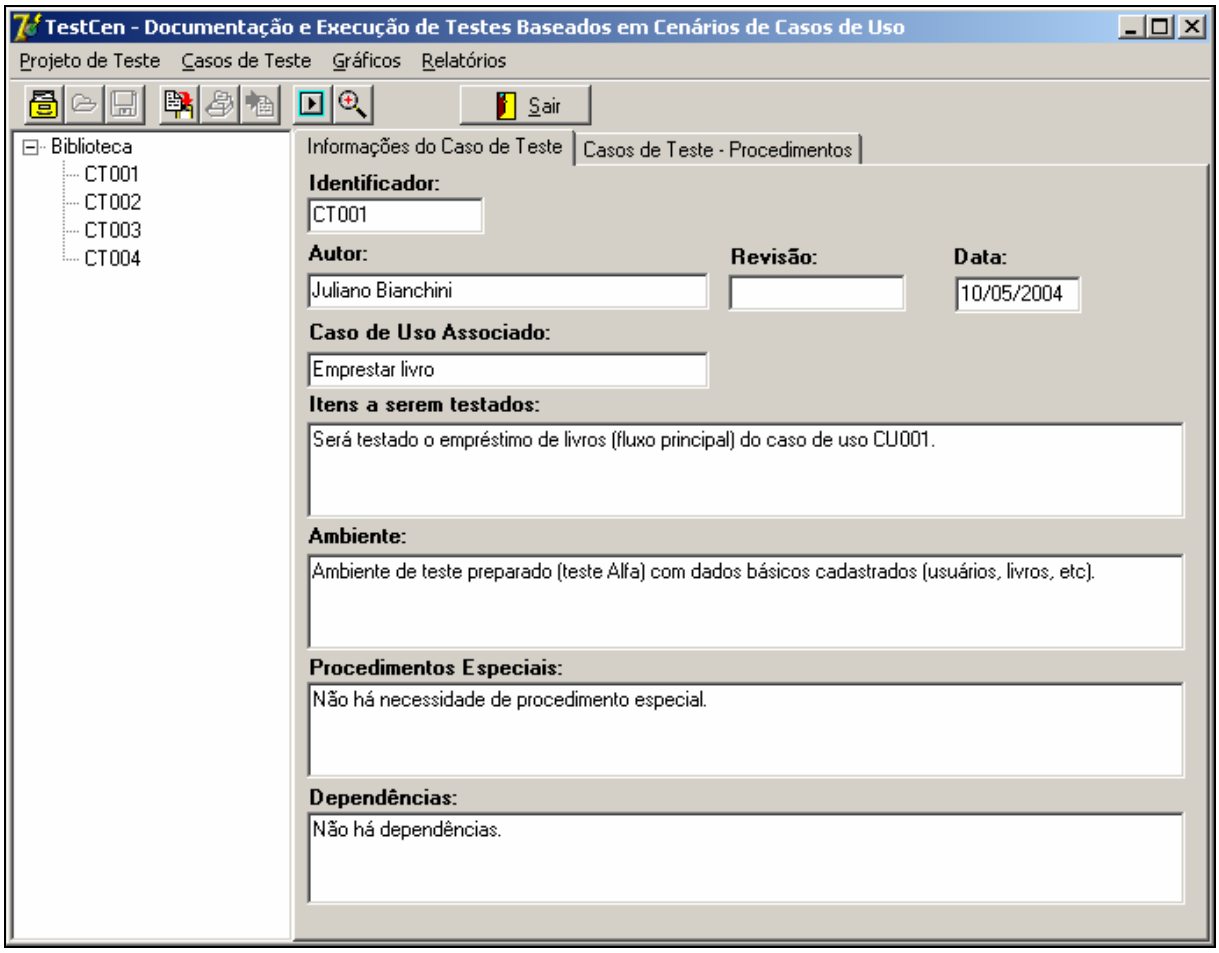


Figura 31 – Parte da especificação dos casos de teste

Depois de especificados os casos de teste, cada um deles pode ser executado, manualmente, através do menu *Casos de Teste* opção *Executar*. Cada execução pode gerar um histórico para posterior consulta. A figura 32 representa a tela de execução de um caso de teste. Nesta tela podem ser inseridas informações sobre a execução do caso, assim como, para cada procedimento é registrado se o mesmo foi executado com sucesso ou não, quais foram os resultados obtidos, se houve algum evento anormal e qual o impacto gerado por um erro encontrado.

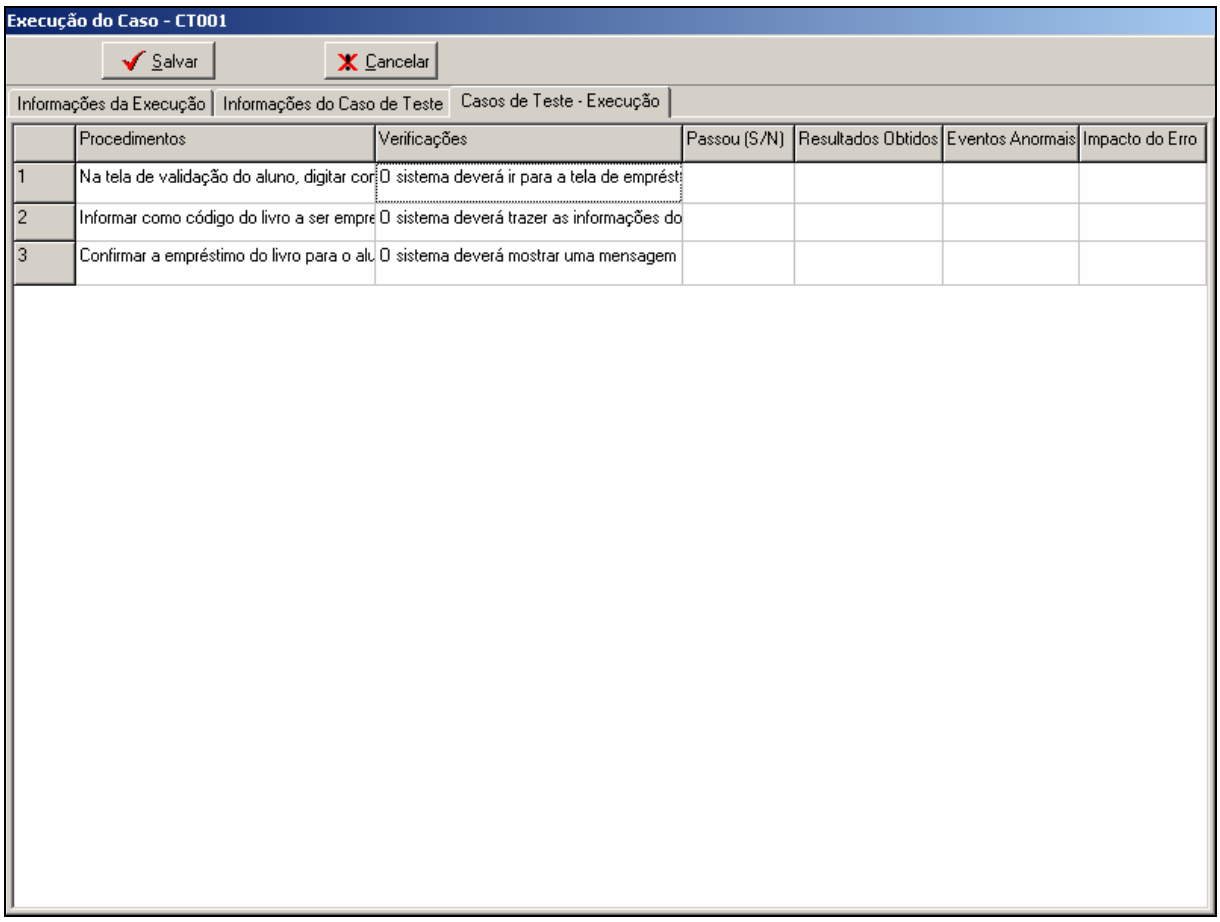


Figura 32 – Execução do caso de teste CT001

Ainda no menu *Casos de Teste*, existe a opção *Exibir Casos de Teste por Caso de Uso* que mostra a lista de casos de teste para cada caso de uso, conforme mostra a figura 33. Esta opção facilita no rastreamento dos casos de teste por caso de uso.

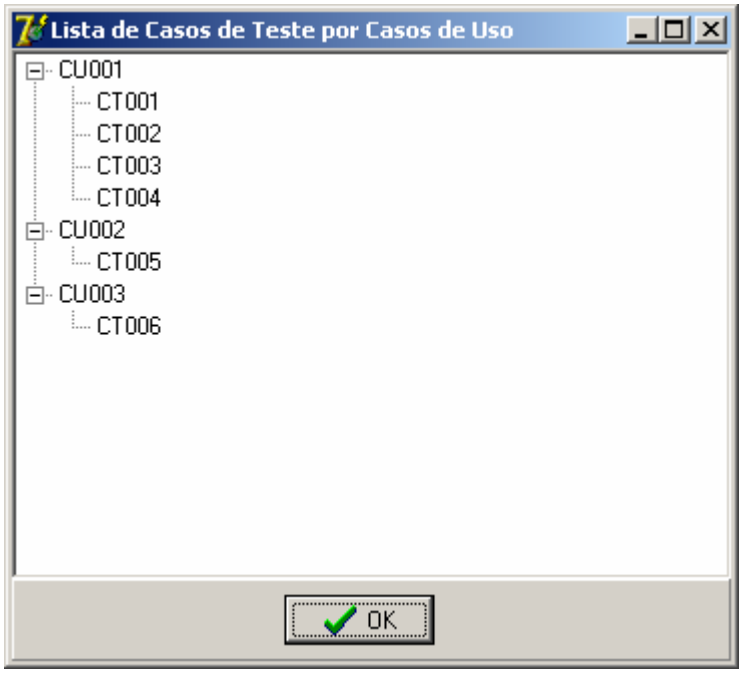


Figura 33 – Lista de casos de teste por casos de uso

No menu *Gráficos* pode-se visualizar gráficos da quantidade de casos de teste por casos de uso e a cobertura de casos de teste. A figura 34 mostra um gráfico com a quantidade de casos de teste associada a cada caso de uso. Neste gráfico os casos de teste são agrupados por caso de uso, possibilitando uma visualização dos casos de uso que possuem casos de teste implementados.

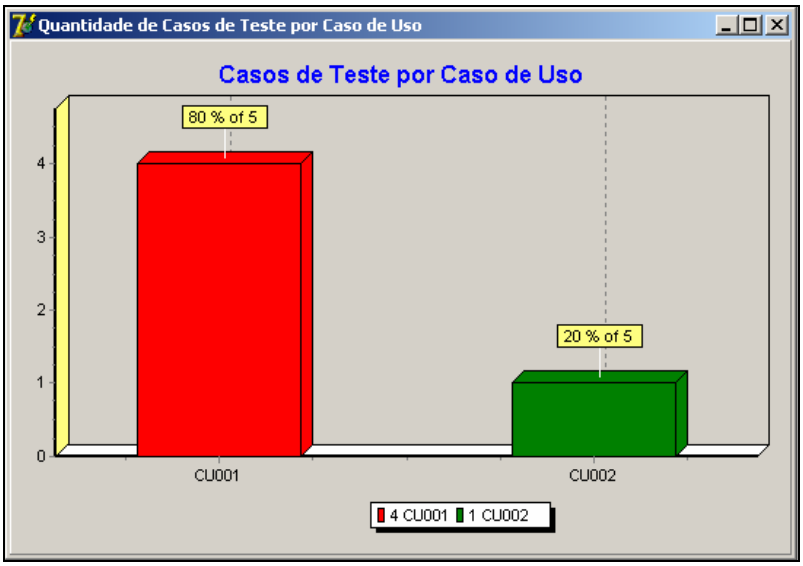


Figura 34 – Gráfico de casos de teste por caso de uso

No menu *Relatórios* podem ser impressos os relatórios de erros por caso de teste e o resumo do projeto. No apêndice A é apresentado o relatório de erros por caso de teste. Neste

relatório, são mostrados os *logs* de execução onde houve algum erro, para o caso de teste selecionado. Isto é, dado um caso de teste que já foi executado, pelo menos uma vez, o relatório mostra apenas as execuções onde foi encontrado algum erro.

3.4 RESULTADOS E DISCUSSÃO

A utilização das metodologias apresentadas por Binder (2000, p. 722-731) e Heumann (2004) proporcionaram uma criação mais fácil e ágil dos casos de teste. A utilização das ferramentas facilitou e organizou todo o processo de teste. Além disso, contribuiu para a re-execução dos testes, isto é, no momento em que o software foi alterado todos os testes executados anteriormente foram executados da mesma forma que foram executados nas vezes anteriores.

A utilização dos casos de uso no teste é facilitada a medida que a especificação de requisitos e a especificação dos casos de uso é feita de forma detalhada. Isto é, a elicitação de requisitos e a análise estão intimamente ligadas ao teste e validação de requisitos.

A figura 35 mostra a cobertura das ferramentas implementadas em relação aos documentos descritos no padrão IEEE 829. Dentre os documentos especificados por este padrão apenas a especificação do plano de teste (2.4.1) e o relatório de transição de item de teste não foram implementados. O primeiro não foi coberto por ser um documento de planejamento global dos testes, onde são descritas as atividades, ferramentas e estratégias que serão utilizadas na fase de teste de um software. Este documento é criado juntamente com planejamento de um projeto de software e neste ponto não se tem uma visão detalhada dos requisitos. O segundo não foi coberto por ser um documento de comunicação entre a área de desenvolvimento e a área de testes.

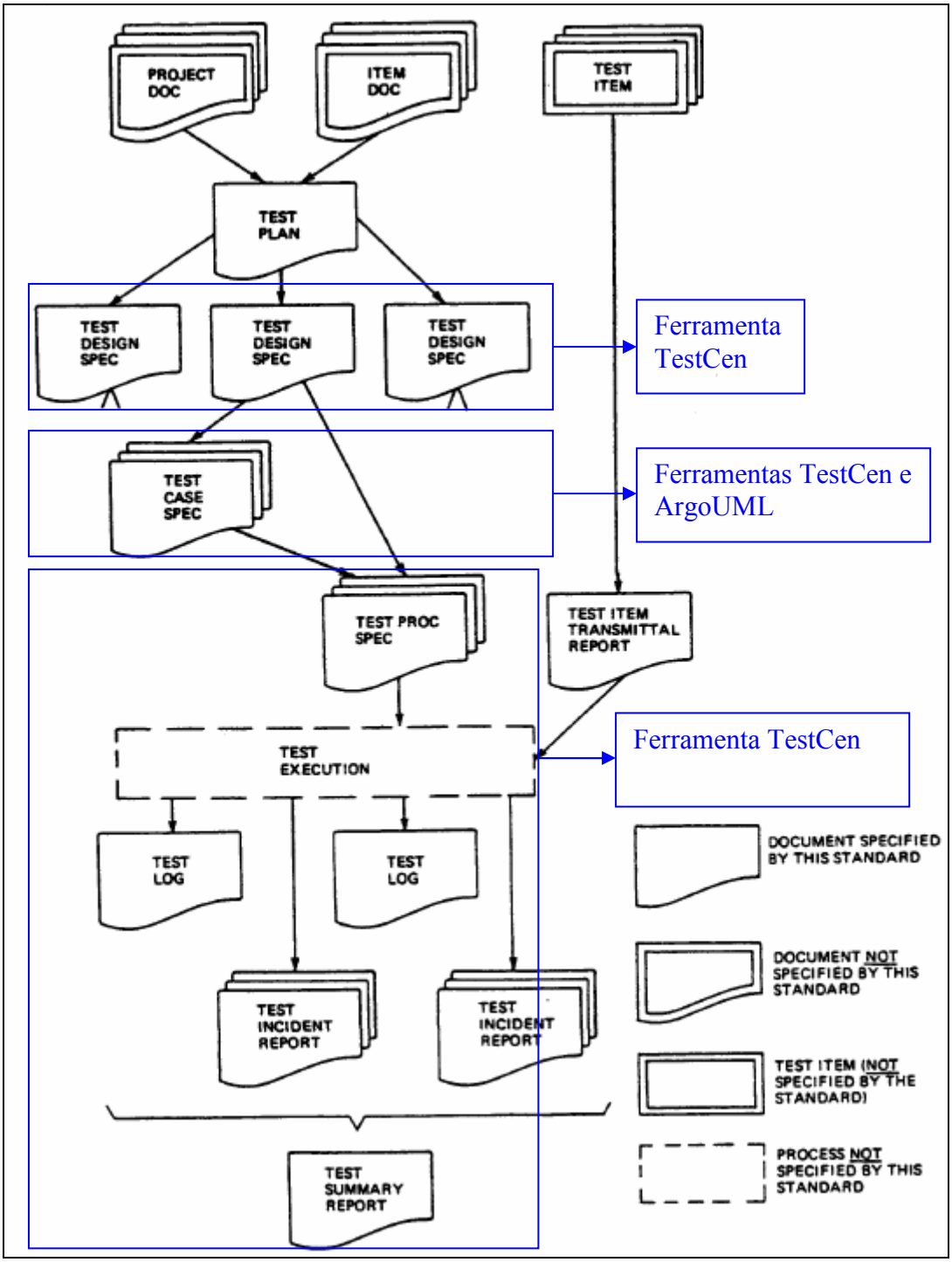


Figura 35 – Documentos cobertos pelas ferramentas implementadas

4 CONCLUSÕES

A utilização das metodologias e ferramentas apresentadas neste trabalho podem auxiliar as equipes de teste a verificar e validar os requisitos de software de forma simples e organizada.

Os objetivos definidos foram alcançados. Uma ferramenta de suporte ao planejamento do teste funcional de software a partir dos diagramas de casos de uso foi desenvolvida. As metodologias de teste baseada em casos de uso da UML, apresentadas por Heumann (2004) e Binder (2000, p. 722-731), foram aplicadas e facilitaram a criação de casos de teste. A ferramenta CASE ArgoUML foi adaptada para incorporar extensões a casos de testes. Isto foi possível com a utilização do mecanismo de criação de novos estereótipos disponível na ferramenta CASE. Além disso a ferramenta desenvolvida utilizou boa parte dos padrões existentes na IEEE 829 e que facilitaram e organizaram o processo de planejamento e execução dos testes.

É importante salientar que o trabalho apresentado apenas supre parte do que a atividade de teste deve contemplar. Isso significa que o teste não se limita apenas a testar e validar os requisitos funcionais. Outros testes, como teste de performance, de carga, de unidade e integração, devem ser executados durante a atividade de testes e estes também devem ser feitos de forma sistemática e organizada .

Outro aspecto a ser considerado é o conjunto de testes elaborados para a validação das ferramentas. Os testes ficaram restritos ao orientando e seu orientador.

Futuramente pretende-se disponibilizar a ferramenta nas disciplinas de engenharia de software da FURB visando facilitar o entendimento de conceitos sobre testes de software a partir da UML.

4.1 EXTENSÕES

Como extensão, um mecanismo de gravação e reprodução de entradas de dados em interfaces gráficas com o usuário poderia ser implementado. O mecanismo de gravação seria responsável em gravar qualquer interação do usuário (clique do mouse ou digitação pelo teclado) com a interface gráfica do software em teste. O resultado da gravação seria um arquivo relacionando os dados e os componentes da interface onde estes dados foram

aplicados. O mecanismo de reprodução teria como entrada o arquivo gerado pelo gravador e então reproduziria a interação que o usuário teve. Além disso, o arquivo (chamado de *script*) poderia ser alterado manualmente e a este poderiam ser acrescentados comandos de interação e controle de fluxo. Desta forma a mesma interação com a interface poderia ser feita com dados diferentes.

Outra sugestão seria o estudo de técnicas de inteligência artificial como, por exemplo, processamento de linguagem natural aplicada a automatização do processo de testes baseados em casos de uso.

REFERÊNCIAS BIBLIOGRÁFICAS

BARTIÉ, Alexandre. **Garantia da qualidade de software**: adquirindo maturidade organizacional. Rio de Janeiro: Elsevier, 2002.

BEZERRA, Eduardo. **Princípios de análise e projeto de sistemas com UML**. Rio de Janeiro: Campus, 2002.

BINDER, Robert V. **Testing object-oriented systems:** models, patterns and tools. Addison-Wesley, 2000.

BOOCH, Grady; RUMBAUGH, James. JACOBSON, Ivar. **UML, guia do usuário**. Rio de Janeiro: Campus, 2000.

CAVARRA, Alessandra et al. **Using UML for automatic test generation**. Disponível em: <http://www.haifa.il.ibm.com/projects/verification/mdt/papers/uml4testgen.pdf>. Acesso em: 08 set. 2003.

HEUMANN, Jim. **Generating test cases from use cases**. Disponível em: <http://therationaledge.com/content/jun_01/m_cases_jh.html>. Acesso em: 20 jan. 2004.

INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **IEEE Std 829-1998**: IEEE standard for software test documentation. Nova York, 1998.

OFFUTT, Jeff; ABDURAZIK, Aynur. **Using UML collaboration diagrams for static checking and test generation**. Disponível em: <http://www.cs.cmu.edu/afs/cs/project/able/ftp/uml00/uml00.pdf>. Acesso em: 08 set. 2003a.

OFFUTT, Jeff; ABDURAZIK, Aynur. **Generating tests from UML specifications**. Disponível em: <http://www.isse.gmu.edu/faculty/ofut/rsrch/papers/uml99.pdf>. Acesso em: 08 set. 2003b.

MYERS, Glenford J. **The art of software testing**. New York: J. Wileyr, 1979.

PBQP. **Indicadores e metas da qualidade e produtividade em software**. Disponível em: <http://www.mct.gov.br/Temas/info/Dsi/PBQP/Indic.htm>. Acesso em: 15 abr. 2004.

PRESSMAN, Roger S. **Engenharia de software**. 5. ed. Rio de Janeiro: McGraw-Hill, 2002.

RADZIUS, Eimutis S. **A methodology for the planning of a scenario based test program**. Disponível em: <http://www.usecasemaps.org/pub/test_planning_SBT.pdf>. Acesso em: 2 mai. 2004

REGNELL, Björn; RUNESON, Per; WOHLIN, Claes. **Towards integration of use case modelling and usage-based testing**. Disponível em: <http://www.coba.usf.edu/departments/isds/nistp/Optimization/resources/papers/Towards%20Integration%20of%20Use%20Case%20Modelling.pdf>. Acesso em: 20 out. 2003.

ROCHA, Ana R. Cavalcanti da; MALDONADO, José C; WEBER, Kival C. **Qualidade de software**: teoria e prática. São Paulo: Prentice Hall, 2001.

ROSA, Edson. **Software de apoio a etapa de testes, utilizando técnicas de caixa preta**. 1997. 67 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

RYSER, Johannes; GLINZ, Martin. **A practical approach to validating and testing software systems using scenarios**. Disponível em: <http://www.ifi.unizh.ch/groups/req/ftp/papers/QWE99_ScenarioBasedTesting.pdf>. Acesso em: 01 set. 2003.

SANDER, Izabela. **Sistema de gerenciamento do teste de software baseado na norma ISSO/IEC 12207**. 2002. 64 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SANTIAGO, Denise. **Ferramenta para teste de programas utilizando componentes da biblioteca CLX**. 2002. 57 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

SEPIN. **Qualidade dos produtos de software**. Disponível em: <http://www.mct.gov.br/Temas/info/Dsi/Quali2001/QualiProutosSW2001.htm>. Acesso em: 15 abr. 2004.

SOMMERVILLE, Ian. **Engenharia de software**. 6. ed. São Paulo: Addison Wesley, 2003.

TOLKE, Linus. **ArgoUML Project**. Disponível em: <http://argouml.tigris.org>. Acesso em: 15 abr. 2004.

WITTEVRONGEL, Jeremiah; MAURER, Frank. **Using UML to partially automate generation of scenario-based test drivers**. Disponível em: <http://sern.ucalgary.ca/~milos/papers/2001/WittevrongelMaurer.pdf>. Acesso em: 03 mai. 2004.

APÊNDICE A – Relatório de erros por caso de teste

RELATÓRIO DE ERROS POR CASO DE TESTE

*** INFORMAÇÕES DO LOG ****

Data Execução: 18/ 6/2004

Hora Execução: 08:35:55

Descrição: Teste da versão Beta 0.5.

Testador: Juliano Bianchini

*** INFORMAÇÕES DO CASO DE TESTE ****

Identificador: CT001

Caso de Uso: Emprestar livro

Autor: Juliano Bianchini

Revisão:

Data: 10/05/2004

Ambiente: Ambiente de teste preparado (teste Alfa) com dados básicos cadastrados (usuários, livros, etc).

Itens a testar: Será testado o empréstimo de livros (fluxo principal) do caso de uso CU001.

Procedimentos especiais: Não há necessidade de procedimento especial.

Dependências: Não há dependências.

*** INFORMAÇÕES DOS RESULTADOS ****

Procedimento: Na tela de validação do aluno, digitar como código do aluno 001 e senha abc123 (já previamente cadastrados).

Verificação: O sistema deverá ir para a tela de empréstimo de livros.

Resultados Obtidos: Após confirmar o código do aluno e senha o sistema apresenta uma tela com erro fatal.

Passou: Não

Impacto: Teste parado até a correção do erro.

Eventos Anormais: Erro de conexão com o Banco de Dados.

*** SOLUÇÃO DO PROBLEMA ****

Descrição da Solução do Problema:

1

APÊNDICE B – Código fonte da importação do arquivo XMI

```
//Importa arquivo XMI
procedure TfrmTestCen.importa_XMI;
var
  i, i1, i2, i3, i4, i5 : integer;
  docXMI: IXMLXMIType;
  contentType: IXMLXMIcontentType;
  modelo: IXMLModel_ManagementModelType;
  ownedElement: IXMLFoundationCoreNamespaceownedElementType;
  listaEstereotipo: IXMLFoundationExtension_MechanismsStereotypeTypeList;
  estereotipo: IXMLFoundationExtension_MechanismsStereotypeType;
  LUseCase: IXMLBehavioral_ElementsUse_CasesUseCaseTypeList;
  UseCase: IXMLBehavioral_ElementsUse_CasesUseCaseType;
  useCaseEstereotipo:
IXMLFoundationExtension_MechanismsStereotypeextendedElementType;
  listaTaggedValue: IXMLFoundationCoreModelElementtaggedValueType;
  listaAssociacoes: IXMLFoundationCoreAssociationTypeList;
  associacao: IXMLFoundationCoreAssociationType;
  conexao: IXMLFoundationCoreAssociationconnectionType;

  tempCasoTeste: caso_teste;
  tempProcedimento: procedimento_teste;

begin

  //Abre arquivo XMI para importação
  docXMI:= LoadXMI(odImportaXMI.FileName);
  contentType:= docXMI.XMIcontent;
  modelo:= contentType.Model_ManagementModel;
  ownedElement:= modelo.FoundationCoreNamespaceownedElement;
  ListaEstereotipo:= ownedElement.FoundationExtension_MechanismsStereotype;

  //Busca estereótipos de caso de teste
  for i:= 0 to ListaEstereotipo.Count - 1 do
  begin
    estereotipo := ListaEstereotipo.Items[i];
    if estereotipo.FoundationCoreModelElementname = 'Caso de Teste' then
    begin
      useCaseEstereotipo:=
estereotipo.FoundationExtension_MechanismsStereotypeextendedElement;
      break;
    end
    else
      estereotipo:= nil; //endif
  end; //endfor
```

```
  if estereotipo <> nil then
  begin
    //Inicializa treeview

    tvTestCen.Items.item[0].Selected := true;
    // busca lista de casos de uso
    LUseCase:= ownedElement.Behavioral_ElementsUse_CasesUseCase;

    for i:=0 to LUseCase.Count -1 do
    begin
      //Pega caso de uso da vez
      UseCase:= LUseCase.Items[i];


      //Verifica se é um caso de teste - se o caso de uso possui o estereótipo
de caso de teste
      for i1:=0  to  useCaseEstereotipo.Count - 1 do
      begin
        if UseCase.Xmiid =
useCaseEstereotipo.FoundationCoreModelElement[i1].Xmiidref then
        begin
          //Importa caso de teste
          tempCasoTeste := caso_teste.Create;
          tempCasoTeste.Set_id(UseCase.FoundationCoreModelElementname);

          //Busca Associações com os casos de uso
          listaAssociacoes:= ownedElement.FoundationCoreAssociation;
          for i4:= 0 to (listaAssociacoes.Count - 1) do
          begin
            associacao:= listaAssociacoes.Items[i4];
            conexao:= associacao.FoundationCoreAssociationconnection;
            for i5:=0 to (LUseCase.Count - 1) do
            begin
              //Verifica se existe conexão
              if  ((LUseCase.Items[i5].Xmiid =
conexao.FoundationCoreAssociationEnd[0].FoundationCoreAssociationEndtype.Found
ationCoreClassifier.Xmiidref) or
                  (LUseCase.Items[i5].Xmiid =
conexao.FoundationCoreAssociationEnd[1].FoundationCoreAssociationEndtype.Found
ationCoreClassifier.Xmiidref)) and

((conexao.FoundationCoreAssociationEnd[1].FoundationCoreAssociationEndtype.Fou
ndationCoreClassifier.Xmiidref = UseCase.Xmiid) or

(conexao.FoundationCoreAssociationEnd[0].FoundationCoreAssociationEndtype.Foun
dationCoreClassifier.Xmiidref = UseCase.Xmiid)) then
              begin

tempCasoTeste.Set_use_case(LUseCase.Items[i5].FoundationCoreModelElementname);
                break;
              end;  //if
            end; // for i5
          end; // for i4

          //Lista de tagged values
          listaTaggedValue := UseCase.FoundationCoreModelElementtaggedValue;
          for i2:= 0 to (listaTaggedValue.Count - 1) do
          begin
```

```
if
(listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].FoundationExte
nsion_MechanismsTaggedValuetag = 'autor') then

tempCasoTeste.Set_autor(listaTaggedValue.FoundationExtension_MechanismsTaggedV
alue[i2].FoundationExtension_MechanismsTaggedValuevalue);

            if
(listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].FoundationExte
nsion_MechanismsTaggedValuetag = 'revisao') then

tempCasoTeste.Set_revisao(listaTaggedValue.FoundationExtension_MechanismsTagge
dValue[i2].FoundationExtension_MechanismsTaggedValuevalue);

            if
(listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].FoundationExte
nsion_MechanismsTaggedValuetag = 'data') then

tempCasoTeste.Set_data(listaTaggedValue.FoundationExtension_MechanismsTaggedVa
lue[i2].FoundationExtension_MechanismsTaggedValuevalue);

            if
(listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].FoundationExte
nsion_MechanismsTaggedValuetag = 'ambiente') then

tempCasoTeste.Set_ambiente(listaTaggedValue.FoundationExtension_MechanismsTagg
edValue[i2].FoundationExtension_MechanismsTaggedValuevalue);

            if
(listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].FoundationExte
nsion_MechanismsTaggedValuetag = 'procedimentos_especiais') then

tempCasoTeste.Set_procedimentos_especiais(listaTaggedValue.FoundationExtension
_MechanismsTaggedValue[i2].FoundationExtension_MechanismsTaggedValuevalue);

            //Busca procedimentos
            if (PosEx('procedimento_',
listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].FoundationExten
sion_MechanismsTaggedValuetag) > 0) then
            begin
              //Busca o índice do procedimento de teste
              i3:=
StrToInt(Copy(listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].F
oundationExtension_MechanismsTaggedValuetag, 14,2));

              if tempCasoTeste.Get_procedimento_teste(i3) = nil then
              begin
                tempProcedimento:= procedimento_teste.Create;

                //Valoriza procedimento de teste

tempProcedimento.Set_procedimentos(listaTaggedValue.FoundationExtension_Mechan
ismsTaggedValue[i2].FoundationExtension_MechanismsTaggedValuevalue);

                //Atribui procedimento de teste ao caso de teste
                tempCasoTeste.Set_procedimento_teste(i3, tempProcedimento);
              end
              else
                procedimento_teste
(tempCasoTeste.Get_procedimento_teste(i3)).Set_procedimentos(listaTaggedValue.
FoundationExtension_MechanismsTaggedValue[i2].FoundationExtension_MechanismsTa
ggedValuevalue);
            end; //If PosEx
```

```
            //Busca verificacoes
            if (PosEx('verificacao_',
listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].FoundationExten
sion_MechanismsTaggedValuetag) > 0) then
            begin
              //Busca o índice do procedimento de teste
              i3:=
StrToInt(Copy(listaTaggedValue.FoundationExtension_MechanismsTaggedValue[i2].F
oundationExtension_MechanismsTaggedValuetag, 13,2));

              if tempCasoTeste.Get_procedimento_teste(i3) = nil then
              begin
                tempProcedimento:= procedimento_teste.Create;

                //Valoriza procedimento de teste

tempProcedimento.Set_verificacoes(listaTaggedValue.FoundationExtension_Mechani
smsTaggedValue[i2].FoundationExtension_MechanismsTaggedValuevalue);

                //Atribui procedimento de teste ao caso de teste
                tempCasoTeste.Set_procedimento_teste(i3, tempProcedimento);
              end
              else
                procedimento_teste
(tempCasoTeste.Get_procedimento_teste(i3)).Set_verificacoes(listaTaggedValue.F
oundationExtension_MechanismsTaggedValue[i2].FoundationExtension_MechanismsTag
gedValuevalue);
            end;  //If PosEx
          end; //endfor

           //Insere Caso de teste no Projeto
           projTeste.Set_caso_teste(projTeste.Get_total_casos, tempCasoTeste);

           //Adiciona caso ao treeview
           tvTestCen.Items.AddChild(TestCenTreeNode, tempCasoTeste.Get_id);

        end; //endif
      end; //endfor

    end; // endfor

    //Expande treeview
    tvTestCen.FullExpand;
  end; //endif
end;  //fim importação
```

ANEXO A – Diagramas da UML e padrões de projeto de teste aplicáveis a cada diagrama

Padrão de projeto de teste		Caso de uso	Classe	Seqüência	Atividade	Estados	Colaboração	Componente	Implantação
Escopo de método	Partição de categoria	☆		☆	☆		☆		
	Função combinacional	☆		☆	✧				
	Função recursiva			☆			☆		
	Mensagem polimórfica		☆	☆			☆		
Escopo de classe	Limites invariáveis	☆✧	☆✧	☆✧			☆		
	Classe não modal			✧					
	Classe modal					★			
	Classe quase-modal					✧			
Integração de escopo de classe	Small pop								
	Ciclo Alfa-omega		☆✧						

Classe nivelada	Servidor polimórfico		✧	✧			✯		
	Hierarquia modal		✧	✧		★	✧		
Componentes reutilizáveis	Classe abstrata		✯✧						
	Classe genérica		✯✧						
	Framework novo	✯✧	✯✧	✧	✧	✧	✧	✧	✧
	Framework popular	✯✧	✯✧	✧	✧	✧	✧	✧	✧
Subsistema	Associações de classe		★						
	Cenário round-trip			★	✯	✯	✧		
	Exceção controlada			✧	✧		✧		
	Máquina de estados				✯	★			
Integração	Big bang	✧		✧			✧	✧	✧
	Bottom-up						✧	✧	✧

	Top-down			✧			✧	✧	✧
	Colaborações			✧			★	✧	✧
	Backbone	✧		✧				✧	✧
	Linhas	✧		✧				✧	✧
	Cliente/Servidor	✧		✧				✧	✧
	Serviços distribuídos	✧		✧				✧	✧
	Alta freqüência	✧		✧				✧	✧
Escopo de aplicação	Casos de uso estendido	★						✧	✧
	Cobertura em CRUD	★						✧	✧
	Testes alocados por perfil	★						✧	
Teste de	Re-testar todos							✧	✧

regressão	Re-testar casos de uso de risco	✧		✧				✧	✧
	Re-testar por perfil	✧						✧	✧
	Re-testar código alterado	✧						✧	✧
	Re-testar dentro da parede de fogo						✧	✧	✧

Fonte: adaptado de Binder (2000, p. 272-273)

Legenda: ★ O padrão está explicitamente baseado neste diagrama, ☆ O padrão pode ser útil para testar elementos deste diagrama, ✧ O diagrama pode ser útil quando estiver utilizando este padrão