Testabilidade e planejamento de testes de software

Neste artigo, uma rotina de testes de software é proposta com o objetivo de mostrar como a sua realização colabora para o progresso do desenvolvimento de software.

Por que eu devo ler este artigo:Neste artigo, uma rotina de testes de software é proposta com o objetivo de mostrar como a sua realização colabora para o progresso do desenvolvimento de software. Os resultados obtidos apontam para a criação de um produto final que esteja de acordo com os requisitos identificados e seja livre da maioria dos erros encontrados. O ponto principal do estudo é a abordagem da etapa dos testes de unidade, que, por serem executados em paralelo ao desenvolvimento do sistema, têm um papel fundamental para que se alcance a testabilidade do software.

Autores: Guilherme Lourenção Tempesta e Elisamara de Oliveira

A área de testes de software vem se tornando cada vez mais importante para que o desenvolvimento alcance um alto nível de qualidade e confiabilidade. Um software que apresenta erros e falhas após a sua entrega para o usuário final fatalmente não irá obter uma boa aceitação. Além disso, os desenvolvedores acabam gastando muito tempo para a correção de erros muitas vezes complexos que poderiam ter sido identificados na fase inicial. Tudo isso pode aumentar significativamente o custo do desenvolvimento. O ideal é que os testes de software sejam parte essencial desde o planejamento do projeto, passando por todas suas etapas e indo até a finalização e entrega do software totalmente funcional.

Estratégias e estruturas de testes adequadas devem ser criadas em cada etapa do desenvolvimento para que os testes sejam realizados no software. O objetivo é procurar os possíveis erros de forma automatizada, quando viável, facilitando o trabalho do programador ou responsável pela execução dos testes e preparando o software para que esses sejam facilmente elaborados e efetuados.

No entanto, a utilização apropriada das técnicas de testes de software ainda não é uma prática muito difundida entre as equipes de desenvolvimento. Muitos não preparam rotinas de testes principalmente por não compreenderem a sua importância. A realização dos testes contribui claramente para o aumento da qualidade do programa e resulta na satisfação do usuário final e na conformidade com os requisitos que foram identificados no projeto.

Os testes de software e a garantia da qualidade

A principal finalidade de se aplicar os fundamentos da engenharia de software no desenvolvimento é a criação de softwares dentro dos prazos e custos planejados e, sobretudo, com alta qualidade. A

qualidade de software é definida pela norma ISO/IEC 9126 como sendo a capacidade de um produto ou serviço apresentar funcionalidades e características que atendam totalmente às necessidades específicas ou implícitas dos usuários. A garantia da qualidade reúne processos que definem como alcançar a qualidade do software e como a equipe de desenvolvimento deverá se comportar para satisfazer o nível de qualidade exigido.

Verificação e Validação (V&V) é um dos processos de garantia de qualidade do software, e abrange várias atividades, entre elas o teste de software. A verificação deve garantir a consistência interna do produto, ou seja, certificar que suas especificações estejam sendo atendidas. Já a validação utiliza as referências externas do usuário final para assegurar que o produto construído satisfaz suas necessidades e expectativas.

Os testes de software são conjuntos de funções e atividades que são executadas com o objetivo de encontrar erros cometidos na construção de um software. Além de buscar e descobrir falhas no software, os testes têm como meta comprovar que o produto atende aos requisitos e que está em conformidade com suas especificações. É importante citar que apesar dos testes contribuírem significativamente para a confiabilidade do software, eles não podem assegurar a qualidade de um produto, pois alcançar a qualidade de um software depende de outros fatores.

Estratégias de testes de software

Para executar os testes de software é preciso, inicialmente, definir uma estratégia de teste, fornecendo um roteiro que indica o caminho a ser seguido e definindo como e quando os passos referentes aos testes serão executados, bem como o tempo, esforço e recursos necessários. Uma estratégia de teste bem definida deve incluir o planejamento dos testes, o projeto dos casos de teste, a execução dos testes e, por fim, a coleta e avaliação de resultados.

O planejamento dos testes deve ser feito para garantir que se obtenham os resultados esperados, decidindo onde os erros podem ser encontrados e projetando os testes mais eficientes para localizá-los. Uma das maneiras de se criar um plano de teste é baseá-lo em riscos, ou seja, em uma parte do código em que a probabilidade de ocorrência de uma falha seja maior ou em um módulo específico onde o impacto do erro poderia comprometer o funcionamento do produto.

Um caso de teste deve documentar um teste com o objetivo de provar uma exigência. Deve haver pelo menos um caso de teste por requisito, mas algumas vezes são utilizados vários casos de testes para provar um único requisito. Também se pode utilizar o mesmo caso de teste em muitas situações a fim de verificar por completo uma determinada exigência.

A criação de projetos de casos de teste eficazes é um dos tópicos mais importantes relacionados aos testes de software, pois técnicas eficientes de casos de teste, utilizadas em conjunto, possibilitam a identificação da maioria dos erros. Entre essas técnicas estão:

- **Teste de caixa-preta:** seu objetivo é descobrir situações em que o software não se comporta de acordo com as suas especificações, sem se preocupar com os seus aspectos estruturais. Deve ser derivada de uma série de condições de entrada para que o programa seja testado com foco em seus requisitos funcionais, sendo que essas entradas podem ser válidas ou não;

- **Teste de caixa-branca:** testa cada caminho no software pelo menos uma vez utilizando técnicas específicas para executar decisões lógicas e ciclos em seus limites operacionais, assegurando, assim, a validade de suas estruturas de dados.

Nota: o teste de caixa-cinza também pode ser considerado uma técnica intermediária. Assim como o teste de caixa-preta, tem como foco dados de entrada e saída, porém utiliza o conhecimento relacionado às estruturas internas do programa como apoio para o desenvolvimento dos casos de teste.

Fases e etapas do teste

A atividade de teste de software pode ser dividida em duas fases fundamentais, que são o teste de componentes, que é executado pelo próprio desenvolvedor e tem como meta encontrar erros testando componentes individuais do software (funções, objetos, etc.), e o teste de sistema, que tem o objetivo de fazer a integração dos componentes do software e a verificação dos requisitos, sendo esse executado por uma equipe de testes independente.

Os testes de software também podem ser relacionados com a espiral que ilustra o processo de software dividindo-os em quatro etapas sequenciais:

- **Teste de unidade:** o teste unitário é focado nos menores blocos ou unidades de um sistema e tem o objetivo de facilitar a depuração e apresentar um paralelismo no processo de teste, executando o teste em vários módulos simultaneamente. É considerado como um auxílio para a codificação do software, podendo ser projetado até mesmo antes do código fonte, pois, através de informações do projeto, podem ser estabelecidos casos de teste que devem ser ligados a um conjunto de resultados aguardados. Os testes de unidade instituem um escopo restrito no qual se focalizam as estruturas de dados e a lógica interna de processamento;
- **Teste de integração:** ao integrar os componentes ou unidades, deve ser verificado se trabalham corretamente em conjunto, com os dados exatos (entrada e saída) e no tempo esperado. Devido à complexidade de algumas interações, a localização dos erros pode se tornar um problema, que pode ser resolvido com a utilização de uma abordagem incremental, isto é, integrando um conjunto mínimo de componentes e testando-os até que todos estejam devidamente integrados;
- **Teste de validação:** logo após o término dos testes de unidade e integração, se inicia o teste de validação, que consiste em um conjunto de testes que deve comprovar a conformidade com os requisitos e se as características e especificações do produto estão de acordo com o esperado, tendo como foco as operações que são visíveis ao usuário final;
- **Teste de sistema:** o software é somente uma das partes de um sistema computacional e o teste de sistema reúne uma série de testes que tem o objetivo de exercitar todas as partes do sistema, obtendo a garantia de que todos seus elementos funcionam adequadamente. Existem alguns testes específicos que fazem parte do teste de sistema, sendo os mais utilizados e eficientes os seguintes: recuperação, segurança, esforço, desempenho e disponibilidade.

Testes automatizados

A utilização de ferramentas automatizadas é essencial para que a atividade de testes tenha um alto grau de confiabilidade e que não esteja limitada somente a softwares com baixa complexidade.

Testes automatizados contribuem para a redução do custo de teste, além de apresentar uma grande quantidade de recursos e alternativas para a sua implementação e execução em um programa. Para sistemas de grande porte, em que o custo de teste é relativamente alto, é importante a utilização do *workbench*, um conjunto de *frameworks* de teste que auxilia o processo e emprega ferramentas como gerenciadores de teste, geradores de dados, comparação dos resultados apresentados com os de testes anteriores, entre outras.

Não é necessário fazer um alto investimento para utilizar ferramentas de automação, tendo em vista que existem alguns *frameworks* gratuitos.

Testabilidade de software

A testabilidade de software é um conjunto de características apresentadas por um software que faz com que ele se torne fácil de ser avaliado. As características desejadas para que se desenvolva um software com alta testabilidade são:

- **Operabilidade:** implementação do software visando a qualidade e evitando qualquer impedimento na execução de algum teste;
- **Observabilidade:** entradas e saídas de fácil visualização e identificação, além de acessibilidade ao código-fonte;
- **Controlabilidade:** maior controle possível das variáveis e estados do software;
- **Decomponibilidade:** modularização do software em unidades que possam ser tratadas e testadas de forma independente;
- **Simplicidade:** construção de uma codificação padronizada e simples;
- **Estabilidade:** as alterações realizadas no software não devem tornar nenhum teste já existente inválido; devem ser pontuais e controladas. Além disso o software deve apresentar boa capacidade de recuperação de falhas;
- **Compreensibilidade:** a arquitetura do software é de fácil compreensão assim como sua documentação, que deve ser organizada e precisa.

Utilizar o conceito de testabilidade no ciclo de vida de desenvolvimento de software é fundamental para que a maioria dos defeitos sejam encontrados e corrigidos antes que o produto final seja entregue. Sendo assim, um software testável torna muito mais fácil a execução dos planos de testes e aumenta consideravelmente as chances de se alcançar a satisfação do usuário.

Hands on: Fazendo o planejamento dos testes

O software utilizado como modelo para os testes é um protótipo que consiste em um cadastro básico de alunos e disciplinas com controle simplificado de notas. Primeiramente deve ser definida a estratégia de testes que será utilizada para que seja feito o planejamento dos testes, considerando que a estratégia escolhida deve ser a mais adequada e eficaz. Serão apontados os objetivos que deverão ser alcançados pelos testes, os recursos que serão utilizados para possibilitar a execução dos testes, quais tipos de testes serão efetuados e quais as fases que serão contempladas.

Para o exemplo aqui apresentado, a estratégia de testes desenvolvida descreve o escopo, as etapas e tipos de testes envolvidos e as ferramentas utilizadas para a execução dos testes:

- **Escopo da estratégia de testes de software:** a estratégia deve contribuir para se criar um software que apresente alta testabilidade, encontrar o maior número possível de erros no software e garantir que ele esteja de acordo com os requisitos.
- **Etapas e tipos de teste de software:** as fases ou etapas do teste abordadas neste projeto são, principalmente, os testes de unidade e os de integração. Também é feita uma abordagem dos testes de validação e sistema, porém com menor ênfase. Alguns tipos ou técnicas utilizados foram o teste caixa-branca, responsável pelos testes estruturais, e o de caixa-preta, usado para avaliações funcionais. Além de testes manuais, também foram utilizadas avaliações automatizadas, empregando algumas ferramentas de apoio para a automação dos testes, como *frameworks* que podem ser utilizados no ambiente de desenvolvimento e softwares específicos para a execução de diversos tipos testes;
- **Ferramentas utilizadas:** o protótipo do software de cadastro foi criado na plataforma de desenvolvimento Embarcadero Delphi, que utiliza a linguagem de programação Object Pascal. O Delphi possui um *framework* específico para criar e executar os testes de software, o DUnit, que utiliza conceitos de programação orientada a objetos (POO) e é usado principalmente para a criação de classes que executam os testes de unidade em paralelo com o desenvolvimento do sistema. Além do *framework* DUnit do Delphi, foi utilizado o software TestComplete, que além de executar os testes de unidade, ainda realiza outros tipos de testes, que podem ser aplicados em todas as etapas. Para criar os casos de uso de teste, utilizou-se o software Visual Paradigm, especializado na criação de diagramas UML. Para realizar o monitoramento do acesso ao banco de dados na etapa de testes de sistema, utilizamos o software Sinática Monitor.

Preparando o ambiente

Após definir a estratégia de testes e fazer todo o planejamento, o próximo passo é a criação de um ambiente organizado para que os testes sejam executados com total eficiência e confiabilidade.

Para os testes de unidade, o ambiente é a própria IDE de desenvolvimento, ou seja, a plataforma Delphi. A única particularidade é a utilização do *framework* DUnit, que é nativo nas versões mais atuais do Delphi e não necessita de instalação para a criação das classes específicas que são utilizadas na realização dos testes. A função do DUnit é, basicamente, executar os testes comparando os resultados obtidos com os resultados esperados.

Para a execução das demais etapas, a preparação do ambiente consiste na instalação do software TestComplete, para desenvolver testes automatizados, do software Visual Paradigm, para criar os casos de uso de testes e, por fim, do Sinática Monitor para monitorar o acesso ao banco de dados.

Criando os casos de teste

Os casos de teste desenvolvidos mostram um roteiro a ser seguido para que as avaliações sejam realizadas de acordo com as entradas efetuadas no sistema e suas respectivas saídas ou resultados esperados. Foram utilizados diagramas de caso de uso da UML já que eles refletem as diferentes situações que acontecem na utilização do sistema, como: cadastrar um novo aluno, consultar ou alterar o cadastro de um determinado aluno, calcular a média das notas de uma disciplina, etc.

A **Figura 1** representa alguns dos casos de uso utilizados referentes às rotinas de busca e alteração do cadastro de um aluno.

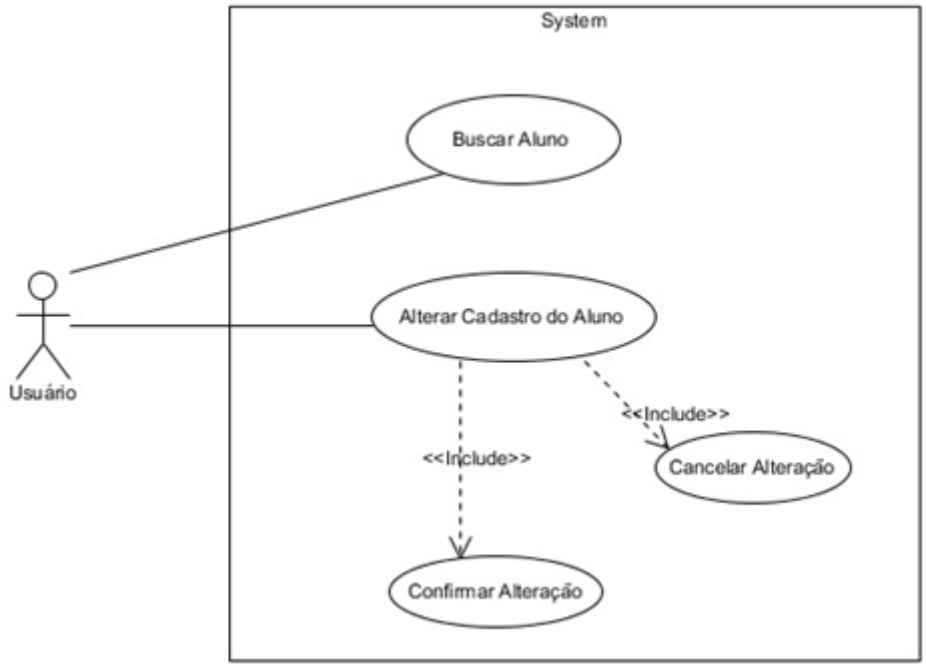

Figura 1. Diagrama de Caso de Uso – Busca e alteração do cadastro de aluno

A partir do diagrama de caso de uso apresentado, foi criado o caso de teste correspondente mostrando os dados de entrada, que são os procedimentos realizados, e os dados de saída, que são os resultados esperados. Além disso, o caso de teste descreve alguns critérios especiais referentes à validação dos dados.

O caso de teste correspondente ao caso de uso para alterar o cadastro de um aluno é mostrado na **Tabela 1**.

Caso de Teste: Alteração do cadastro de um aluno	
Condição para o teste: Abrir o formulário de cadastro de alunos	
Procedimentos Realizados	**Resultados Esperados**
1. Clicar no botão *Buscar*.	Abrir a lista de alunos cadastrados.
2. Selecionar o aluno.	Fechar a lista de alunos e carregar os dados do aluno selecionado na tela.
3. Clicar no botão *Alterar*	Habilitar a edição dos dados do aluno.
4. Após a alteração, clicar em *Confirmar*.	Salvar as alterações efetuadas no banco de dados.
5. Repetir os passos de 1 a 3, e após a alteração, clicar em *Cancelar*.	Descartar as alterações e limpar os campos da tela de cadastro.
Critérios Especiais	**Resultados Esperados**
1. Informar CPF.	Aceitar somente caracteres numéricos e fazer a validação do CPF.
2. Informar Data de Nascimento.	Aceitar somente uma data válida.
3. Deixar de preencher algum campo obrigatório	Não confirmar a alteração e exibir mensagem de

e clicar em *Confirmar*.	advertência.

Tabela 1. Caso de teste – Alteração do cadastro de aluno.

De acordo com a norma IEEE 829, um documento específico para o caso de teste elaborado deve ser criado e a estrutura do documento deve conter o identificador, os itens de teste, as especificações de entrada e de saída e o ambiente necessário, além de possíveis exigências especiais e interdependências existentes.

Após o planejamento, documentação e criação dos casos de uso e dos casos de teste correspondentes, os testes foram aplicados no protótipo de acordo com o que foi idealizado e seguindo a sequência planejada na estratégia.

Testes de unidade

Inicialmente foram realizados os testes de unidade em cada procedimento (ou método) do código fonte do software, sendo que procedimento corresponde à menor unidade a ser testada no sistema. Nessa etapa foram executados tanto testes manuais quanto testes automatizados, realizando avaliações estruturais e funcionais no sistema.

Os testes estruturais executados utilizam a técnica de caixa-branca, que analisa o comportamento lógico de uma unidade do software. Na técnica de caixa-branca é feito o teste do caminho básico, onde se utiliza um grafo de fluxo para descrever o conjunto de todos os caminhos possíveis a serem percorridos na estrutura de uma determinada unidade.

O procedimento referente à operação de exclusão de um registro no cadastro de alunos pode ser observado na **Listagem 1.**

Listagem 1. Procedimento para excluir um registro do cadastro de alunos.

```
01 procedure TFrm_CadastroAluno.Btn_ExcluirClick(Sender: TObject);
01 begin
01   if Edt_Nome.Text <> EmptyStr then
02   begin
02     gOperacao := opExclui;
03     if TControlaForm.ConfirmaMsg then
04     begin
04       oAluno := TAluno.Create;
04       oAluno.Matricula := StrToInt(Lbl_Matricula.Caption);
04       TAlunoPersistencia.Exclui(oAluno);
04       FreeAndNil(oAluno);
04     end;
05     else
05       gOperacao := opLivre;
06     TControlaForm.TravaDestravaForm(Self, gOperacao);
06     TControlaForm.LimpaForm(Self);
```

```
06   end;
07   else
07     ShowMessage('Selecione um registro para ser excluído.');
08 end;
```

A numeração exibida à esquerda de cada linha de código do procedimento o decompõe em blocos, sendo que esses se baseiam nas condições verificadas no decorrer da execução do procedimento. No bloco 01, por exemplo, é verificado se o conteúdo de um componente é diferente de vazio: se for verdadeiro, segue para o bloco 02, senão vai para o bloco 07.

Dessa forma, compreendem-se claramente os caminhos percorridos dentro da unidade que está sendo testada, possibilitando a criação de um grafo de fluxo, onde cada bloco identificado no código-fonte corresponde a um nó do grafo (veja a **Figura 2**).

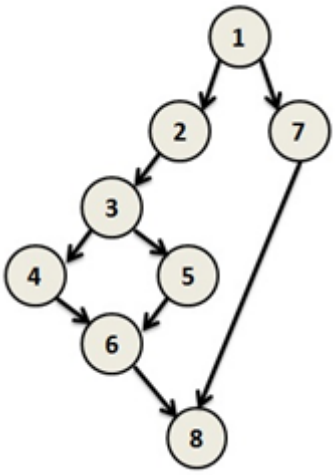

Figura 2. Grafo de fluxo do procedimento para excluir um registro do cadastro de alunos.

Agora é feito o teste do caminho independente, que identifica cada caminho da estrutura que segue para uma nova condição. No grafo exibido na **Figura 2** foram encontrados três caminhos independentes, que são os seguintes:

- Caminho 01: 1, 7, 8;
- Caminho 02: 1, 2, 3, 4, 6, 8;
- Caminho 03: 1, 2, 3, 5, 6, 8.

O próximo passo a ser feito é calcular a complexidade ciclomática. Esse cálculo tem a função de definir o limite superior para a quantidade de testes que precisam ser efetuados, garantindo a execução de todas as condições e comandos pelo menos uma vez. Para encontrar a complexidade ciclomática V(G) do grafo de fluxo G, foi utilizado o cálculo V(G) = E – N + 2, sendo que E representa a quantidade de arestas e N é a quantidade de nós do grafo. Veja as contas a seguir:

V(G) = E – N + 2

V(G) = 9 – 8 + 2

V(G) = 3

O resultado do cálculo é igual a 3, o que indica que três testes devem ser planejados para a execução dos caminhos do grafo de fluxo referente ao procedimento de software apresentado. Neste caso, devido à baixa complexidade da unidade em questão, os testes planejados foram executados de forma manual. Esses testes avaliaram cada condição, percorrendo cada caminho possível no procedimento. O resultado atendeu às expectativas e não apresentou falhas.

Após essa etapa foram executados os testes funcionais utilizando a técnica de testes de caixa-preta, que tem como foco os aspectos funcionais do software, como a comparação dos resultados apresentados (os dados de saída são comparados com os resultados esperados obtidos através de dados de entrada). Ao contrário dos testes de caixa-branca, em que o foco é a estrutura interna do programa, o caixa-preta tem o objetivo de avaliar o comportamento externo.

Foi utilizado o *framework* DUnit do Delphi para a realização dos testes de caixa-preta de forma automatizada, empregando as classes do sistema e seus procedimentos, que são as menores unidades do sistema.

Para a realização de cada teste foi criada uma classe específica, apresentando os valores de entrada e os resultados esperados na saída. O DUnit executa todos os testes com base nos valores indicados e os compara com os resultados obtidos através do procedimento *CheckEqualsString*. Se os valores apresentados correspondem aos resultados esperados, o teste indica que a unidade do software está funcionando corretamente. Caso contrário, retorna uma mensagem de erro informando que o valor obtido após os testes é diferente do valor esperado.

Uma das classes testadas corresponde ao registro de notas dos alunos, que contém o método utilizado para retornar o *status* de aprovação. O teste de unidade criado para essa classe recebe os valores de entrada referentes a três notas de um aluno em uma determinada disciplina, denominadas Nota1, Nota2 e Nota3, o número de faltas, que são Faltas1, Faltas2 e Faltas3, e retorna como dados de saída a média das notas, o total de faltas e o resultado final de aprovação ou reprovação.

O segmento de código mostrado a seguir é uma das linhas do método denominado *TestExibeResultadoFinalTestTNotas*:

```
CheckEqualsString('Aprovado',TNotas.ExibeResultadoFinal(70,25));
```

O procedimento *CheckEqualsString* tem como resultado esperado um valor do tipo *string*, nesse caso “Aprovado”, e como dados de entrada a média final das notas, 70, e o número total de faltas, 25. Os testes foram executados e o resultado exibido na interface do DUnit indicou que não houve nenhum erro.

Porém, foi forçada a ocorrência de um erro com a alteração do método *ExibeResultadoFinal* da classe *TNotas* alterando o valor da média para um valor menor que 70. Com isso, o resultado esperado, “Aprovado”, não corresponde mais ao valor apresentado na execução do teste (vide **Figura 3**).

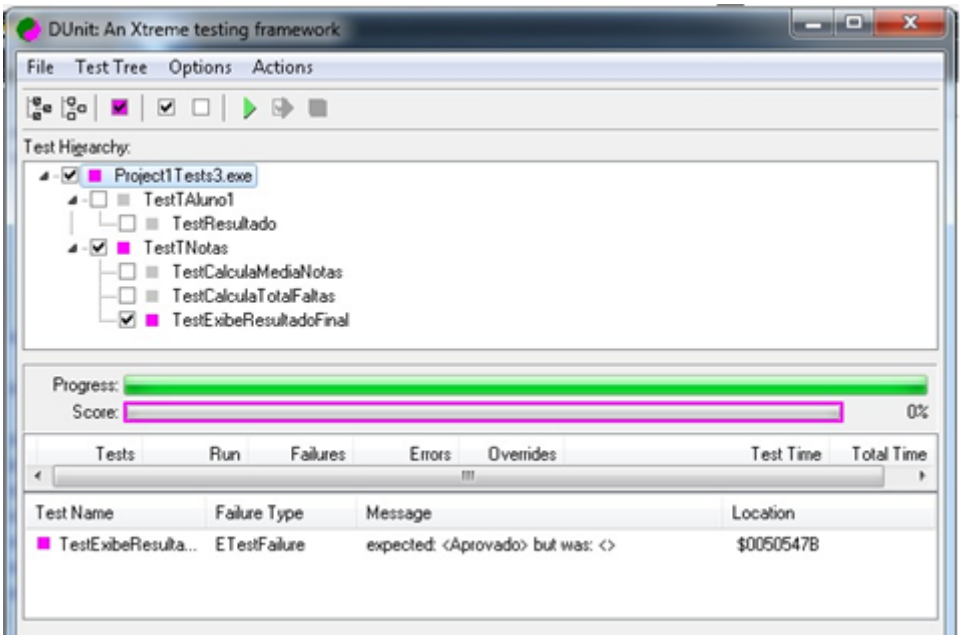

Figura 3. Resultado do teste de unidade utilizando o DUnit com erro proposital.

A mensagem exibida pelo DUnit mostra com bastante clareza que o erro ocorrido no método *ExibeResultadoFinal* está no trecho do código em que o resultado esperado seria “Aprovado”, restando ao responsável efetuar a correção do código fonte.

Após o término de todos os testes planejados para a fase de testes de unidade, pode-se considerar esta etapa como concluída, devendo-se prosseguir para a etapa seguinte, que é avaliar a integração dos componentes.

Testes de integração

Com os testes de unidade finalizados, foram realizados os testes de integração, que consistem na realização de avaliações nas unidades depois de integradas utilizando uma abordagem incremental, ou seja, construindo a estrutura do sistema e avaliando a cada incremento adicionado. Além disso, pode ser feita a busca de possíveis erros relacionados à criação das interfaces do software.

A estratégia escolhida para o teste de integração foi o teste baseado em sequência de execução que faz a integração apenas das unidades necessárias para executar determinada tarefa no software. A sequência de execução utilizada deve fazer a integração dos métodos *Pesquisa* e *Altera* da classe *TAlunoPersistencia*, que é a classe a que pertencem as operações de acesso ao banco de dados do sistema.

Os testes foram feitos manualmente executando a interface do software. Os tipos de testes escolhidos para essa etapa foram os funcionais pois a estrutura dos métodos de classe já foi avaliada individualmente na etapa anterior. A estratégia de testes funcionais aplicada foi a caixa-preta, utilizando dados de entrada e dados de saída como resultado esperado.

Um dos casos de testes criados utiliza, inicialmente, o método *Pesquisa*, que retorna uma lista de objetos da classe *TAluno*. Um determinado objeto da lista é selecionado e o mesmo serve de parâmetro para que o método *Altera* seja acessado. O método *Altera* efetua as alterações no objeto selecionado pelo método *Pesquisa* e depois grava essas alterações no banco de dados.

O resultado esperado nesse teste é que o parâmetro passado para o método *Altera* seja realmente o objeto que foi selecionado utilizando o método *Pesquisa*. Assim, caso ocorra alguma falha no método *Pesquisa*, o método *Altera* não será acessado ou então o parâmetro passado para ele não será o correto. A **Figura 4** exibe o teste efetuado conforme o resultado esperado.

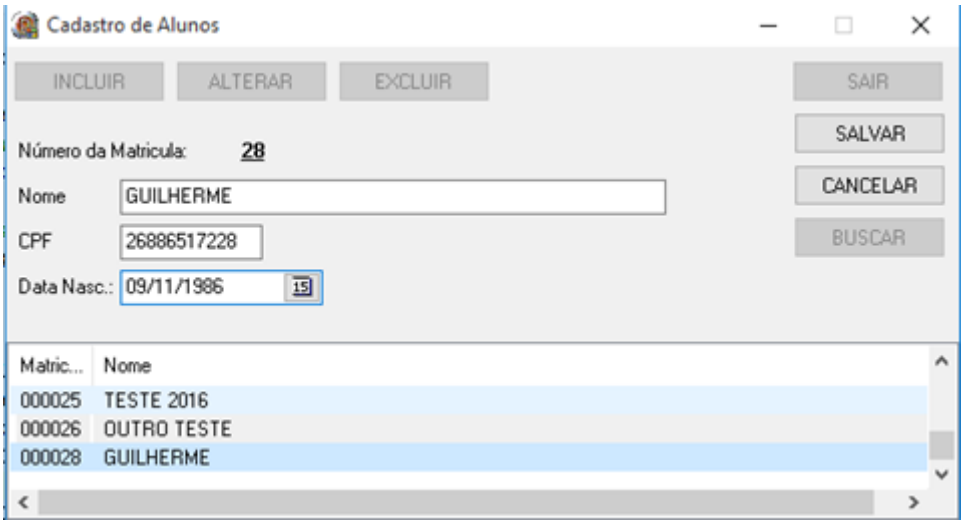

Figura 4. Resultado do teste de integração entre os métodos Pesquisa e Altera.

O resultado obtido após todos os testes de integração indica que a integração das unidades, que neste caso específico são os métodos da classe *TAluno*, está correta e não apresentou qualquer falha, estando o sistema integrado corretamente.

Testes de validação

Nesta etapa é verificado se as especificações do software são totalmente cumpridas, ou seja, se o sistema realmente faz o que se propõe a fazer. O método selecionado para o teste de validação é o baseado em cenário, que procura erros que possam ser encontrados na utilização do sistema pelo usuário final, através de casos de uso que simulam operações ou tarefas por ele executadas.

Durante o planejamento, foram criados diagramas UML de caso de uso que resultaram em alguns casos de teste utilizados na fase de validação. O caso de uso em questão se refere à tarefa de lançamento das notas e das faltas de um determinado aluno no sistema, conforme exibido na **Figura 5**.

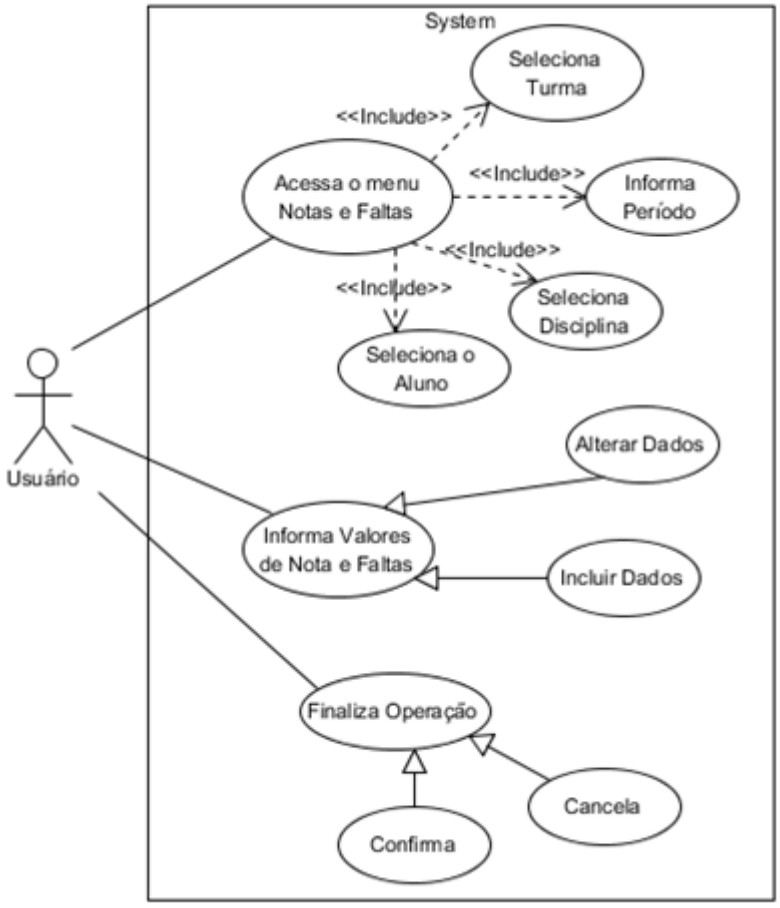

Figura 5. Caso de uso para lançamento de notas e faltas.

O caso de uso exibido na **Figura 5** foi utilizado para apoiar a criação do caso de teste apresentado na **Tabela 2**.

Caso de Teste: Lançamentos de notas e faltas	
Condição para o teste: Abrir o formulário Notas e Faltas	
Procedimentos Realizados	**Resultados Esperados**
1. Acessar o menu Notas e Faltas.	Abrir a tela Notas e Faltas.
2. Selecionar uma turma	Habilitar os campos Período e Disciplina
3. Informar o período de referência.	Não prosseguir se for um período inválido.
4. Selecionar uma disciplina.	Habilitar o campo Aluno.
5. Selecionar um aluno.	Se houver dados cadastrados, habilita a alteração, senão faz a inclusão dos valores para Nota e Faltas.
6. Informar os valores para Nota e Faltas.	Aceitar somente dados do tipo decimal e inteiro, respectivamente, senão exibe advertência.
7. Confirmar a alteração ou inclusão, clicando em salvar.	Gravar os dados no banco e limpar os campos da tela.
8. Repetir os passos de 2 a 6 e descartar o lançamento clicando no botão cancelar.	Não gravar nenhum dado no banco e limpar os campos da tela.

9. Clicar no botão Sair.	Fechar a tela de Notas e Faltas

Tabela 2. Caso de teste utilizado na etapa de teste de validação — lançamento das notas e das faltas de aluno.

O cenário exibido é o correto para que a tarefa de lançamento de notas e faltas no sistema seja executada com sucesso. Neste cenário é identificado que o usuário tem a necessidade de incluir uma nova nota ou editar uma já existente no sistema. Além disso, o usuário tem como opções confirmar ou cancelar o que foi realizado.

O teste de validação criado sobre esse cenário deve avaliar as operações de inclusão e alteração com o objetivo de buscar algum erro causado em alguma dessas tarefas que possa fazer com que as especificações não sejam cumpridas.

Neste caso, os testes foram realizados executando a aplicação manualmente e os resultados obtidos mostraram que as operações de alteração e inclusão de notas atendem aos requisitos identificados no projeto do sistema.

Testes de sistema

Os testes de sistema, neste caso, compreendem as avaliações de esforço e de desempenho. Os testes de esforço são realizados executando a aplicação com o objetivo de forçá-la a apresentar algum erro relacionado à memória, processamento ou outro elemento em especial.

Um dos casos de teste relacionados à avaliação de esforço tem o objetivo de incluir um grande número de registros no banco de dados e depois utilizar a opção de pesquisa para listar os dados inseridos. Ao fazer uma busca excessiva de dados em disco, o sistema pode forçar um processamento e gasto de memória além do esperado.

O software TestComplete foi utilizado para a criação dos *scripts* para executar as operações repetidamente. O script definido é apresentado na **Listagem 2**.

Listagem 2. Script do TestComplete para o teste de esforço.

```
procedure Test1;
    var project1 : OleVariant;
    var frm_CadastroAluno : OleVariant;
    var edit : OleVariant;
    var i: integer;
  begin
    i := 0;
    project1 := Aliases.Project1;
    frm_CadastroAluno := project1.Frm_CadastroAluno;
    while i < 100 do
    begin
      frm_CadastroAluno.Btn_Incluir.ClickButton;
```

```
    edit := frm_CadastroAluno.Edt_Nome;
    edit.Click(26, 12);
    edit.wText := RandomString(10);
    edit.Keys('[Tab]');
    edit := frm_CadastroAluno.Edt_CPF;
    edit.wText := geraCPF;
    edit.Keys('[Tab]');
    frm_CadastroAluno.Dt_Nasc.Keys('31121990');
    frm_CadastroAluno.Btn_Salvar.ClickButton;
    project1.dlgConfirm.DirectUIHWND.CtrlNotifySink.btnYes.ClickButton;
    frm_CadastroAluno.Btn_BuscarLista.ClickButton;
    Inc(i);
  end;
```

O *script* do TestComplete insere 100 registros no banco de dados automaticamente, sendo que, para o campo *Nome* é gerado de forma aleatória um valor do tipo *string*, para o campo *CPF* é utilizada a função *geraCPF*, que calcula e retorna um valor válido, e para o campo *DtNasc* é informada a data 31/12/1990. Após a inserção de cada registro, é feita a busca de todos os dados da tabela através do clique no componente *Btn_BuscarLista* a fim de forçar o sistema a processar um comando SQL em uma tabela do banco de dados que possui uma grande quantidade de registros.

O resultado apresentado após os testes de esforço indicou que o comportamento do sistema está em conformidade com o que era previsto, pois as tarefas programadas foram executadas corretamente sem que houvesse nenhuma alteração anormal no consumo de memória e no uso de CPU.

Complementando os testes de esforço, também foram executadas avaliações de desempenho. Na prática, esses dois testes se completam, pois, executando a aplicação com o máximo de esforço, fatalmente o seu desempenho também será avaliado, sendo que além do programa cumprir todos os requisitos que foram definidos, ele precisa apresentar um desempenho que não interfira ou prejudique o bom funcionamento de nenhuma operação.

Tendo como base o teste de esforço, as tarefas de acesso ao banco de dados demandam um processamento mais robusto para a sua execução, desse modo, foi necessário que se realizasse um monitoramento específico das operações relacionadas ao banco de dados do sistema. Conforme especificado anteriormente, a ferramenta utilizada no monitoramento dos *scripts* executados no banco foi o Sinática Monitor, que acompanha o desempenho de tarefas específicas, como a execução de um comando SQL em uma determinada tabela, retornando mensagens de advertência e gerando *logs* de erros como perda de conexão ou comando muito lento. A ocorrência desses problemas tem grandes chances de causar algum dano maior ao sistema como um todo.

Após registrar o banco de dados do sistema no software Sinática Monitor, inicia-se o monitoramento e qualquer operação executada no banco passa a ser avaliada. Além disso, identifica-se todas as

conexões, transações, comandos e processos do banco de dados, listando suas informações detalhadas e gerando gráficos em tempo real.

Um recurso importante é a aba Alarmes, mostrada na **Figura 6**, que exibe informações como os *logs* e advertências referentes aos recursos do banco e aos comandos executados que são considerados lentos.

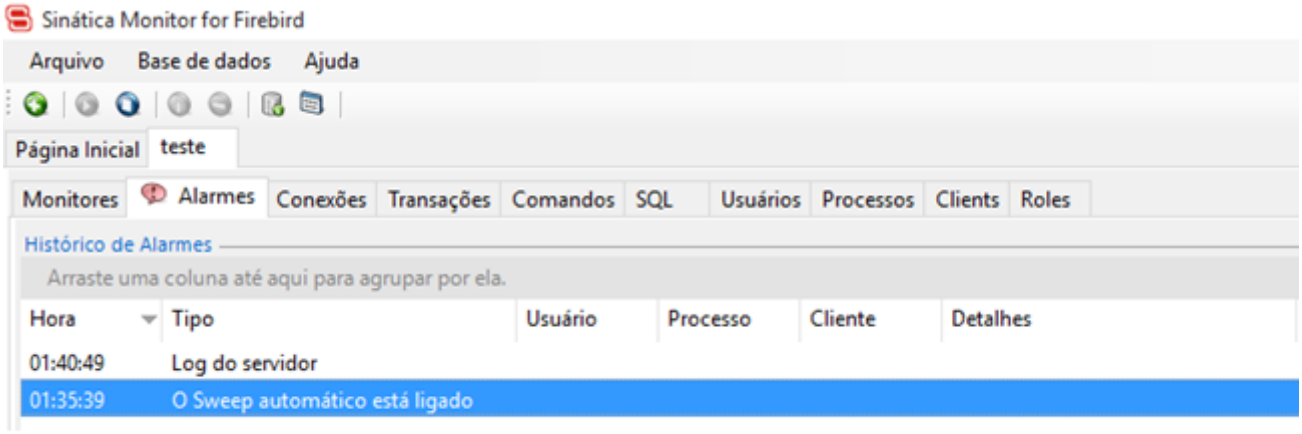

Figura 6. Resultado do teste de performance através do Sinática Monitor.

Durante o teste de performance, não foi registrado nenhum comando lento que poderia ser otimizado para aprimorar a performance do sistema. Os únicos alarmes listados são o log de conexão com o servidor, que é apenas informativo, e o aviso do Sweep automático (**BOX 1**).

BOX 1. O que é Sweep?

O Sweep é um processo de varredura feito pelo Firebird em uma base de dados, que consiste em percorrer todos os registros de cada uma de suas tabelas e realocar os espaços não mais utilizados pelo SGDB e que são ocupados pelas cópias temporárias de registros atualizados ou excluídos.

Ao confirmar uma transação (commit), as cópias temporárias dos registros são descartadas automaticamente através do processo de *Garbage Collection*, porém, ao abortar uma transação (rollback), ou seja, encerrá-la desfazendo todas as operações a ela associadas, as cópias continuam no banco consumindo espaço em memória e, com o seu acúmulo, causando queda de performance no sistema.

Diferentemente do *Garbage Collection* automático, o *Sweep* é capaz de eliminar também o lixo gerado pelas transações abortadas. Por padrão, o Firebird dispara o *Sweep* automaticamente a cada 20.000 transações, entretanto, essa variável pode ser alterada, assim como também é possível desabilitar a sua execução automática conforme a necessidade do sistema.

O *Sweep* pode ser automático, porém, se o banco apresentar um volume de dados relativamente alto, é aconselhável desativar o comando que o realiza automaticamente para evitar que o Firebird o dispare em paralelo com outra operação, podendo tornar o sistema lento durante o processo. Sendo assim, para impedir que o desempenho do sistema possa ser comprometido, o *Sweep* deverá ser executado manualmente quando houver necessidade.

Ao finalizar a última etapa de testes de forma satisfatória, o produto final deverá estar inteiramente de acordo com as especificações definidas no escopo do projeto, além de apresentar total consistência, confiabilidade e, principalmente, melhor qualidade.

O planejamento de testes focado no alcance da testabilidade, feito desde o início do ciclo de vida do desenvolvimento de software, gera resultados exigindo um menor esforço. Esses resultados mostram onde o sistema apresenta falhas na codificação e se há discordância com seus requisitos.

Ao corrigir os erros antes da entrega do produto final, o software terá um ganho importante de qualidade e estará livre de erros que comprometeriam sua utilização e aceitação.

- http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=8077
- http://luizcamargo.com.br/arquivos/NBR%20ISO_IEC%209126-1.pdf
- http://www.nickjenkins.net/prose/testingPrimer.pdf
- CANTU, Carlos H. **Firebird Essencial**. Rio de Janeiro: Editora Ciência Moderna Ltda.
- MALDONADO, José C.; et al. **Introdução ao Teste de Software.** São Carlos: Instituto de Ciências Matemáticas e de Computação, 2004.
- MYERS, Glenford J. **The Art of Software Testing.** New Jersey: John Wiley & Sons Inc., 2004.
- PRESSMAN, Roger S. **Engenharia de Software – Uma Abordagem Profissional**. 7 ed. Porto Alegre: AMGH Editora, 2011.
- RÄTZMANN, Manfred; DE YOUNG, Clinton. **Galileo Computing – Software Testing and Internationalization**. Salt Lake City: Lemoine International Inc., 2003.
- SOMMERVILLE, Ian. **Engenharia de Software**. 8 ed. São Paulo: Pearson Addison-Wesley, 2007.

Saiba mais sobre Testes de Software;)

- Processo de teste ágil x tradicional
- Desvendando o processo de teste de software