Heurísticas para Identificação da Ordem de Integração das Classes em Testes Aplicados a Software Orientado a Objetos

Gladys Machado Pereira Santos Lima
Diretoria de Administração da Marinha - Marinha do Brasil
Ilha das Cobras – Ed. Alte Gastão Mota – 3º andar CEP 20.091-000 – Rio de Janeiro
gladys@dadm.mar.mil.br
Fax: +55 21 32342052

Arilo Cláudio Dias Neto
acdn@cos.ufrj.br

Guilherme Horta Travassos
ght@cos.ufrj.br

Universidade Federal do Rio de Janeiro – COPPE/Sistemas
Caixa Postal 68.511 – CEP 21.941-972
Rio de Janeiro – RJ – Brasil

Resumo

Os ciclos de dependência entre componentes (classes) representam um problema prático para identificar a ordem de integração em software orientado a objetos. Abordagens clássicas de integração ascendente ou descendente (ou sua combinação) tornam-se menos aplicáveis devido à sua característica acíclica. As estratégias de teste de integração OO tratam das quebras destes ciclos, responsáveis diretas pela necessidade da implementação de *stubs*. O propósito das estratégias é reduzir o esforço de teste, minimizando o número de *stubs* produzidos. Este artigo apresenta uma estratégia aplicada diretamente em um nível alto de abstração de projeto OO – diagrama de classes UML – permitindo reduzir esforços extras de construção de diagramas adicionais ao projeto, empregados em outras estratégias pesquisadas na literatura. É apresentada a ferramenta FAROL que implementa a estratégia proposta.

Palavras-chave: Teste de Software Orientado a Objetos, Teste de Integração, Engenharia de Software Experimental.

1. Introdução

O teste de software é aplicado para assegurar que o software funcione corretamente como especificado nos requisitos. De uma maneira geral, um teste bem sucedido é aquele em que os engenheiros de software fazem com que o software falhe durante sua execução, o que permite identificar problemas presentes nele, levando os desenvolvedores a encontrar os defeitos correspondentes antes da entrega do produto final [1].

Uma das maneiras de se prevenir ou detectar defeitos cometidos ao longo do processo de desenvolvimento do software é a definição de diferentes níveis de testes em relação às diversas fases do processo de desenvolvimento [2]. O teste de integração, escopo deste trabalho, representa um conjunto de atividades com intenção de descobrir defeitos na estrutura do software definida durante a fase de projeto [3], comumente associados com as interfaces dos módulos (componentes) que compõem a arquitetura.

No contexto do desenvolvimento orientado a objetos (OO), a dificuldade do entendimento da execução do software OO traz um novo desafio, pois as estratégias tradicionais de teste (acíclicas) não são capazes de lidar com algumas características da arquitetura OO, como, por exemplo, o enlace (dependências) entre os componentes (representados pelas classes), tornando não trivial a tarefa de identificação da ordem de integração e o teste dos componentes [4]. É preciso rever o problema da integração para software OO.

Algumas estratégias para testes de integração (*SIT – Strategy for Integration Testing*) vêm sendo propostas para resolver a ordem de integração das classes em projetos de software OO [5, 6, 7, 8]. Estas estratégias buscam resolver as dependências cíclicas minimizando o esforço de teste, medido pelo número de *stubs*. Os *stubs* são módulos construídos para apoiar os testes e simulam o comportamento dos módulos ainda não prontos (não integrados), representando esforço adicional de desenvolvimento que precisa ser minimizado. As estratégias estudadas, no entanto, necessitam construir artefatos adicionais (aos já existentes num projeto) para auxiliarem na sua aplicação, o que reflete em um esforço extra.

O propósito desta estratégia é apoiar engenheiros de software durante a tarefa de identificar a ordem de integração das classes em testes de software OO com menor esforço de teste. Ao ser aplicada diretamente em um artefato de nível alto de abstração do projeto (diagrama de classes UML), a estratégia deve reduzir custos adicionais com o desenvolvimento de artefatos extras de projeto e deve contribuir para um aumento de produtividade, quando define, também, a ordem mais adequada de construção das classes, em virtude do paradigma OO permitir o uso da mesma semântica (classes, por exemplo) no desenvolvimento e na implementação.

Neste sentido, este artigo descreve a estratégia proposta, composta de um conjunto de heurísticas e de um processo de aplicação. E conhecendo a necessidade de avaliar de forma mais abrangente o processo de aplicação das heurísticas, foram planejados e executados estudos experimentais durante o desenvolvimento deste trabalho. Resultados obtidos mostraram que pelo menos 80% dos engenheiros de software ou gerentes de projetos participantes dos estudos conseguem identificar uma ordem de integração das classes com esforço de teste mínimo usando as heurísticas. A análise destes resultados permitiu caracterizar a viabilidade e efetividade da estratégia, como mostrado também neste artigo.

As próximas seções são organizadas da seguinte forma: na seção 2 são descritas as heurísticas e seu processo de aplicação. Na seção 3 são apresentados os estudos experimentais para caracterização da estratégia de integração proposta. A ferramenta FAROL, que automatiza o processo das heurísticas, é apresentada na seção 4. Finalmente, na seção 5 são apresentadas as conclusões e sugestões de trabalhos futuros.

2. As Heurísticas e o Processo de Aplicação

O diagrama de classes UML retrata a visão estática de um projeto OO e descreve um conjunto de classes e seus relacionamentos (associação, generalização, dependência e realização) [9, 10]. Com base neste conhecimento e com o propósito de empregar informações retratadas e disponíveis durante o projeto do software, o objetivo inicial foi estabelecer, usando a semântica definida pela UML (o significado dos símbolos e suas interpretações por si e em conjunto com outros), características determinantes que justifiquem a realização dos testes de integração de classes anteriormente a outras. Estas características formalizam os critérios de precedências entre classes, que são: herança; assinatura de método de classe; agregação; navegabilidade; classe associação; dependência; e cardinalidade, mostradas na Tabela 1. Para quantificar as relações de precedências entre as classes foram definidas duas propriedades: o Fator de Influência (FI) e o Fator de Integração Tardia (FIT). O *fator de influência* representa o número de classes que precisam ser testadas depois da classe em análise ser testada, sendo o número de classes sobre as quais a classe em análise tem precedência. O *fator de integração tardia* tem a intenção de capturar a idéia do tempo de integração das classes. O FIT indica o momento quando a classe deve ser considerada para integração e teste, em relação às demais e é calculado pela soma dos valores dos Fatores de Influência (FI) das classes com precedência direta sobre a classe em análise. As heurísticas são detalhadamente apresentadas em [11].

Tabela 1 – Heurísticas: Critérios de Precedência.

Critério	**Precedência**
Herança	1. Superclasses concretas → testar primeiro as superclasses. 2. Superclasses abstratas → testar primeiro a subclasse com menor dependência.
Assinatura de método da classe	3. Classe *servidora* testada antes da classe *cliente* (que assinou o método).
Agregação	4. Classe *parte* testada antes da classe *todo*.
Navegabilidade	5. A classe que tornou-se atributo deve ser testada primeiro. 6. Na associação a navegabilidade deve ser analisada nos dois sentidos.
Classe Associação	7. As classes que deram origem a associação devem ser testadas primeiro.
Dependência	8. A classe *fornecedora* deve ser testada antes da classe *consumidora*.
Cardinalidade	9. A classe com cardinalidade opcional deve ser testada antes de outra classe com cardinalidade opcional.

Estas heurísticas são resultantes do aperfeiçoamento da proposta de Travassos *et al*. [12] e Travassos e Oliveira [13][1]. Entretanto, estudos permitiram identificar algumas situações especiais, não definidas inicialmente pelos pesquisadores: *Fator de Influência Nulo*; *Iterações sem Fator de Integração Tardia Nulo*; e o *Deadlock (classes com mesmo FIT)*. A busca para tratamento destas situações especiais guiou a definição do processo para

[1] Estudo da disciplina Laboratórios de Engenharia de Software, COPPE, UFRJ, podendo ser obtido por consulta ao autor ght@cos.ufrj.br .

aplicação das heurísticas. Este processo é composto por processos menores, conforme apresentado na Figura 1, utilizando a *notação para modelar processos de software* [14].

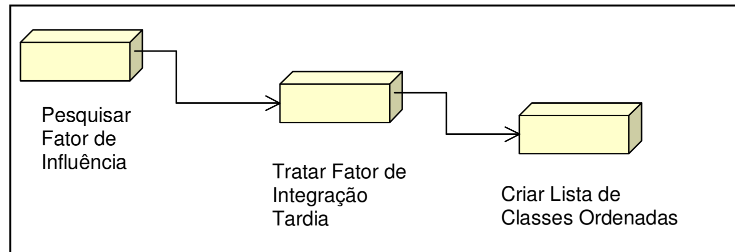

Figura 1 – Processo de Aplicação das Heurísticas.

O propósito do processo *Pesquisar Fator de Influência* é calcular o valor do fator de influência (FI) para todas as classes existentes no modelo, separar aquelas classes com FI nulo, gerando uma lista de classes dependentes (LCD), e criar a lista de classes não ordenadas (LCNO) para o próximo processo, como apresentado na Figura 2.

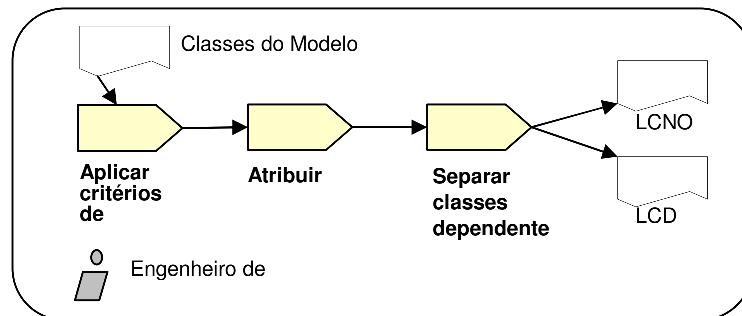

Figura 2 – Processo de *Pesquisar Fator de Influência*.

O processo *Tratar Fator de Integração Tardia*, apresentado na Figura 3, é responsável por gerar a lista de classes ordenadas para teste de integração (LCOTI) a partir das classes da lista LCNO. Enquanto existirem classes a serem ordenadas, uma nova iteração será realizada para "Calcular FIT" e selecionar "classes com FIT nulo" ou "classes com menor FIT", sendo então retiradas da LCNO e inseridas na LCOTI. Antes de iniciar uma nova iteração, serão reduzidas as influências das classes recém ordenadas.

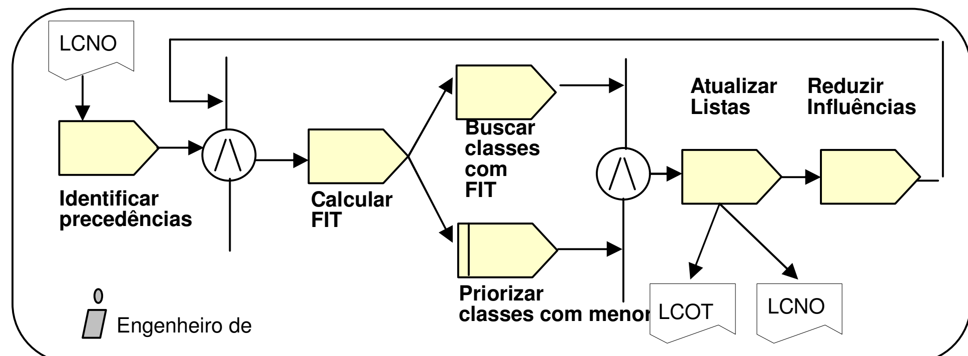

Figura 3 – Processo de *Tratar Fator de Integração Tardia*.

A atividade composta "Priorizar classes com menor FIT", detalhada na Figura 4, é responsável pelo tratamento do *deadlock*, onde a prioridade é estabelecida pelas atividades.

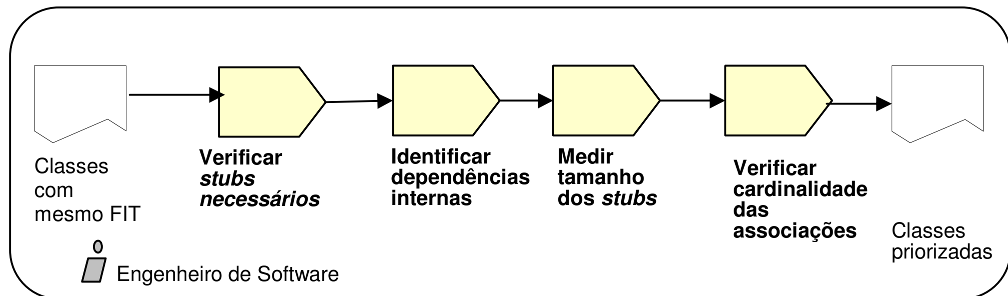

Figura 4 – Atividade “Priorizar classes com menor FIT” detalhada.

O processo *Criar Lista de Classes Ordenada* é responsável por criar a lista final de classes ordenadas para teste de integração (LCOTI). A ordem final das classes é obtida pela atividade “Inserir LCD”, do processo *Tratar Fator de Influência*, ao final da LCOTI, do processo *Tratar Fator de Integração Tardia*, como apresentado na Figura 5.

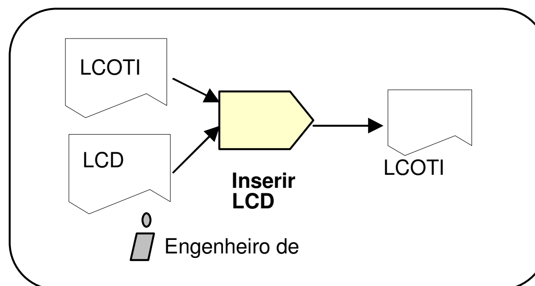

Figura 5 – Processo de *Criar Lista de Classes Ordenada*.

Para avaliar de forma mais abrangente o processo de aplicação das heurísticas em diferentes situações de projeto e tentar, minimamente, ampliar os primeiros resultados encontrados [11], evitando qualquer viés ou risco de aplicação em apenas um modelo, identificamos a necessidade de elaborar os estudos experimentais visando à caracterização destas heurísticas. Esses estudos são discutidos na próxima seção.

3. Estudos de Caso

Segundo Pfleeger [15], é preciso planejar estratégias para ajudar a lidar com a imperfeição do conhecimento e as incertezas dos modelos e medidas que são utilizadas por engenheiros de software. Um desafio seria entender as chances que, sob certas condições, uma ferramenta ou técnica em particular conduzirá para o aprimoramento do software. Neste sentido, a experimentação é tida como determinante na identificação da efetividade de técnicas aplicadas à engenharia de software [16].

3.1 Definição dos Objetivos

Nesta pesquisa, experimentação foi utilizada como um modo sistemático, disciplinado e controlado para avaliar a viabilidade da estratégia de teste de integração proposta, reduzindo as incertezas durante as diversas atividades e fornecendo alguns indícios para melhor compreensão e evolução do trabalho.

Os estudos de caso foram elaborados em conformidade com a metodologia proposta por Wohlin *et al.* [17] para processos de experimentação, utilizada pelo *Grupo de Engenharia de Software Experimental* da COPPE/UFRJ, e composta pelas fases: definição; planejamento; operação; análise e interpretação; e apresentação e empacotamento, conforme Travassos *et al*. [18]. Os quatro estudos são apresentados a seguir:

- *Usabilidade das Heurísticas*. Com objetivo global de caracterizar a usabilidade das heurísticas na identificação da ordem de integração de classes (diagrama de classes UML), comparando o esforço de teste desta estratégia com o esforço de teste gerado por outra estratégia, proposta por Briand *et al*. [19].
- *Procedimentos Pré-existentes*. Para conhecer outros procedimentos (*ad-hoc*) utilizados pelos participantes do estudo na identificação da ordem de integração. Utilizado para comparar o esforço de teste destes procedimentos com o esforço de teste sobre o mesmo modelo quando aplicando as heurísticas propostas. Usado como pré-teste do estudo da efetividade das heurísticas.
- *Autotreinamento nas Heurísticas*. Com o objetivo global de medir o aprendizado das heurísticas pelos participantes, comparando seus resultados (esforços de teste) com o do pesquisador. Utilizado como tratamento do estudo *Efetividade das Heurísticas*.
- *Efetividade das Heurísticas*. Pós-teste para avaliar os resultados dos participantes aplicando as heurísticas, comparativamente aos resultados obtidos no pré-teste (*Procedimentos Pré-existentes*).

3.2 Preparação dos Estudos

A preparação para a execução do primeiro estudo de caso constitui-se da seleção do modelo. A escolha foi baseada em dois critérios: ser um modelo conhecido e para o qual tenham sido anteriormente publicados a aplicação da estratégia de Briand, Labiche e Wang e seus resultados. O modelo selecionado foi o ATM (*Automated Teller Machine*) [19], conforme apresentado na Figura 6.

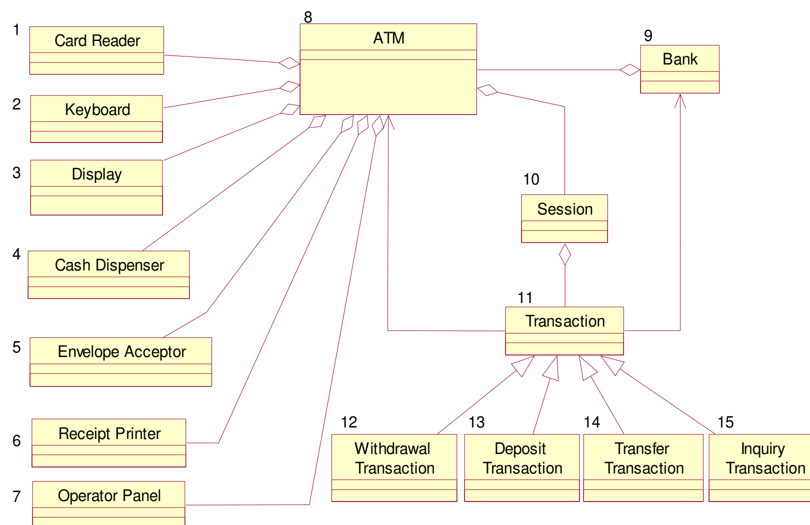

Figura 6 – Modelo ATM: Diagrama de Classes.

Nos demais estudos, a preparação constituiu-se da seleção e caracterização dos participantes; e da preparação do *Autotreinamento*. O *Autotreinamento* é uma apresentação (slides em formato PowerPoint) contendo o processo de aplicação das heurísticas e que permite o aprendizado da estratégia a cada participante, individualmente, e sem interações com outros participantes ou com o pesquisador.

A seleção dos participantes foi limitada em dois grupos, compostos por: alunos do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ (Grupo 1 – Academia) e engenheiros de software ou gerentes de projetos do Centro de Análises de Sistemas Navais (CASNAV/Marinha do Brasil) (Grupo 2 – Indústria). Os participantes da indústria e da academia possuem média de experiência prática em desenvolvimento de software de 10.7 e 3.18 anos, respectivamente. O conhecimento dos conceitos de orientação a objetos e UML foi registrado numa escala de 1 a 5, onde o grau 5 identifica o participante que usou os conceitos em mais de um projeto na indústria, conforme apresentado na Figura 7.

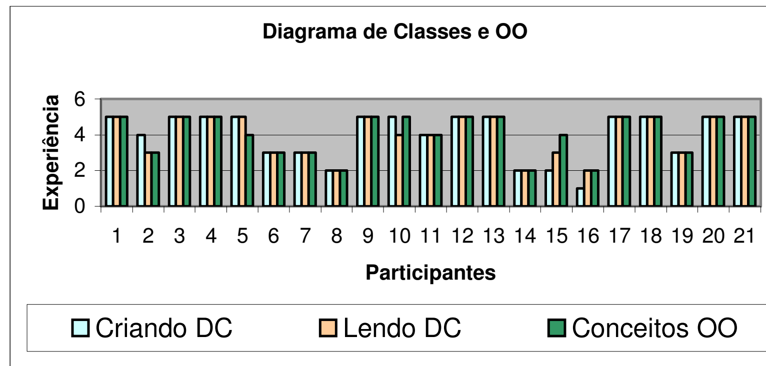

Figura 7 – Conhecimento OO dos Participantes.

3.3 Resultados e Lições Aprendidas

A estratégia proposta por Briand *et al*.[8] obteve resultados melhores que outras estratégias: Kung *et al*.[5]; Tai e Daniels [6]; e Le Traon *et al*.[7]. Portanto, o menor esforço de teste encontrado com processo de aplicação das heurísticas comparado ao resultado da proposta de Briand, Labiche e Wang, como apresentado na Tabela 2, foi fundamental para caracterizar a viabilidade da estratégia proposta e para a continuidade do estudo das heurísticas.

Tabela 2 – Resultado do estudo *Viabilidade das Heurísticas*.

Proposta	Briand – Labiche - Wang	Heurísticas
Ordem de Integração	1, 2, 3, 4, 5, 6, 7, 11, 10, 12, 13, 14, 15, 8, 9	1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10, 12, 13, 14, 15
Stubs	A classe 11 é testada com *stubs* das classes 8 e 9	A classe 8 é testada com *stub* da classe 10
Esforço de Teste	2 *stubs*	1 *stub*

Os estudos *Procedimentos Pré-Existentes (pré-teste)* e *Efetividade das Heurísticas (pós-teste)* contribuíram para a análise da efetividade da estratégia de teste de integração proposta. A Figura 8 apresenta os esforços de teste obtidos com a ordem de integração identificada pelos participantes por meio de procedimentos *ad-hoc* (pré-teste) e com o processo de aplicação das heurísticas (pós-teste). Significativo aumento dos engenheiros de software que obtêm, como resultado, o esforço de teste mínimo (17 *stubs*) para o modelo, como apresentado na Figura 9.

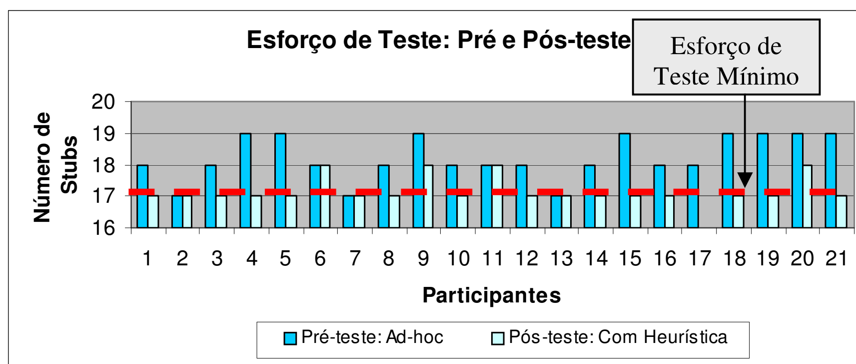

Figura 8 – Resultado dos estudos *Procedimentos Pré-Existentes* e *Efetividade das Heurísticas*.

Os indícios da efetividade dos estudos anteriores são reforçados com a análise do estudo *Autotreinamento nas Heurísticas.* 95% e 85% dos participantes obtiveram o esforço de teste mínimo esperado para os modelos.

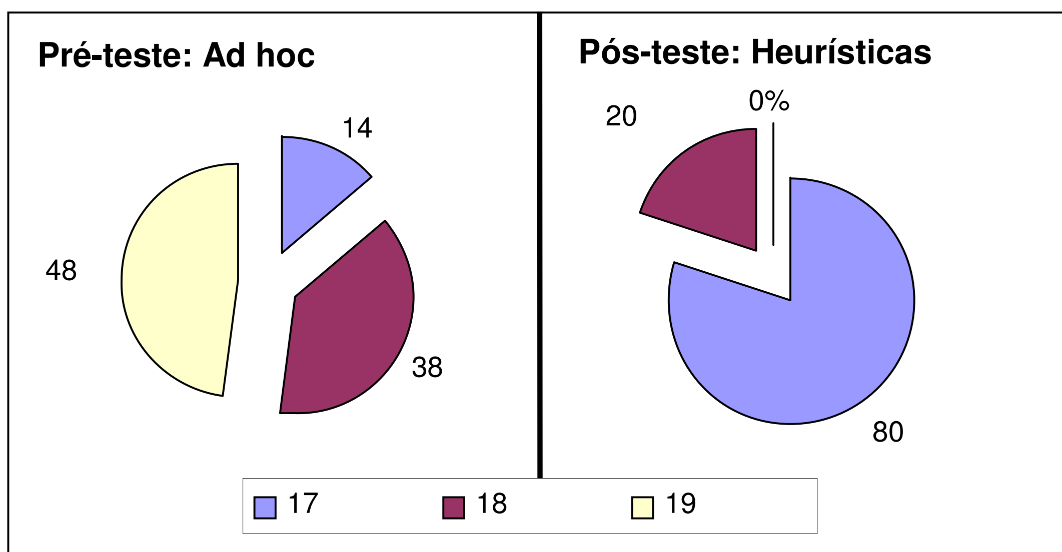

Figura 9 – Comparando Resultados.

O esforço (tempo em minutos) para identificação da ordem de integração também foi medido durante os estudos, considerando o grupo do participante: academia ou indústria. A diminuição dos coeficientes de variações do tempo, no pós-teste, apresentados na Tabela 3, indica uma menor dispersão dos tempos, representando uma nova contribuição do processo de aplicação das heurísticas.

Tabela 3 – Esforço de Identificação – Pré e Pós-Teste.

Estudo de Caso	Esforço de identificação	Academia	Indústria
Procedimentos Pré-Existentes (Ad-hoc)	Média	20,91	15,78
	Desvio Padrão	7,11	10,57
	Coeficiente de Variação	33,98	66,98
Efetividade das Heurísticas	Média	30,91	39,78
	Desvio Padrão	9,21	20,61
	Coeficiente de Variação	29,81	51,81

Outras informações relevantes foram obtidas com análises qualitativas dos estudos de caso, como:

- O uso dos conceitos OO e da semântica da UML como fatores de decisão durante o procedimento *ad-hoc* de identificação da ordem de integração pelos participantes 2, 7 e 13, que apresentaram esforço de teste mínimo durante o estudo *Procedimentos Pré-Existentes (pré-teste)*;
- Ausência de relação entre desempenho do participante nos estudos com algum fator de caracterização (formação acadêmica ou experiência prática em software, por exemplo);
- A intenção de utilização da estratégia em projetos futuros pelos participantes; e
- Erros de cálculo (não de aplicação da estratégia) por dois participantes durante o estudo *Efetividade das Heurísticas,* provavelmente ocorrido devido ao maior número de classes do modelo.

Outra contribuição dos estudos experimentais, para esta pesquisa, foi apontada pela análise de suas próprias limitações permitindo identificar a necessidade do planejamento de novos estudos para caracterizar a aderência da estratégia proposta em outros contextos, como a viabilidade em projetos de maior escala (grande número de classes).

4. A Ferramenta FAROL

O interesse apontado pelos participantes do estudo em utilizar a estratégia proposta em trabalhos futuros, aliado ao grande número de iterações no *processo de aplicação das*, despertou a necessidade de automatizar a estratégia.

A ferramenta FAROL[2] foi definida e implementada com este objetivo. Como apresentado na Figura 12, a tela principal da ferramenta FAROL é composta de cinco áreas: (A) Menu Principal e Barra de Ferramentas; (B) Painel do Modelo de Classes; (C) Painel de Seqüência de Ordenação; (D) Lista Ordenada de Classes; e (E) Esforço de Teste e Modificação da Lista Ordenada.

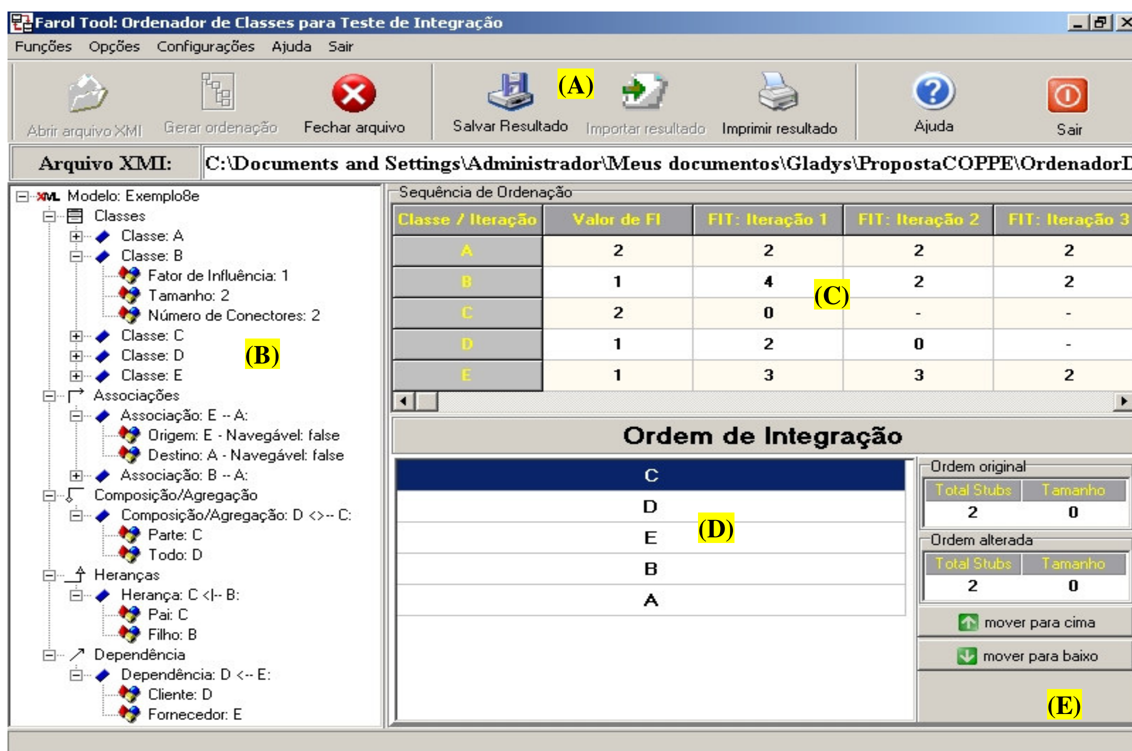

Figura 12 – FAROL: Tela Principal.

A grande variedade de ferramentas CASE para representação de diagrama de classes (UML) no mercado foi responsável pela decisão de construir uma ferramenta independente que utilizasse a estratégia para identificar a ordem de integração das classes. A solução no intercâmbio de dados entre as ferramentas CASE e a FAROL foi adotar o XMI (*XML Metadata Interchange*), padrão XML que provê um formato para descrição de elementos do modelo UML e disponível para muitas ferramentas UML, diretamente ou com auxílio de *plug-ins*. A FAROL é constituída pelos componentes: *Tradutor XMI*; *Ordenador*; *Avaliador*; e *de Exteriorização*, conforme visualizado na Figura 13.

O componente *Tradutor XMI* é responsável pela tradução do arquivo XMI de entrada (exportado da ferramenta CASE UML), reconhecendo os elementos relevantes para as heurísticas. As informações são divididas de acordo com os elementos do diagrama original (classes, associações, composição / agregação, heranças e dependências) e é representado em uma estrutura de árvore no Painel de Modelo de Classes.

Outra tarefa do componente *Tradutor XMI* é gerar a matriz de precedências. Esta matriz bidimensional (NxN) guarda as informações de precedências estabelecidas pelas *heurísticas* entre as N classes de um modelo. Nesta matriz, a marcação de uma célula (X,Y) com o valor 1 (um) indica que a classe X tem precedência sobre a classe Y;

[2] FAROL é um termo empregado na navegação: construção erguida na costa e em cuja parte superior há uma luz com características especiais, para servir de guia ou ponto de referência para os navegantes.

caso contrário, seu valor será 0 (zero). Desta forma, numa coluna estão representadas todas classes que têm precedência sobre a classe representada na coluna.

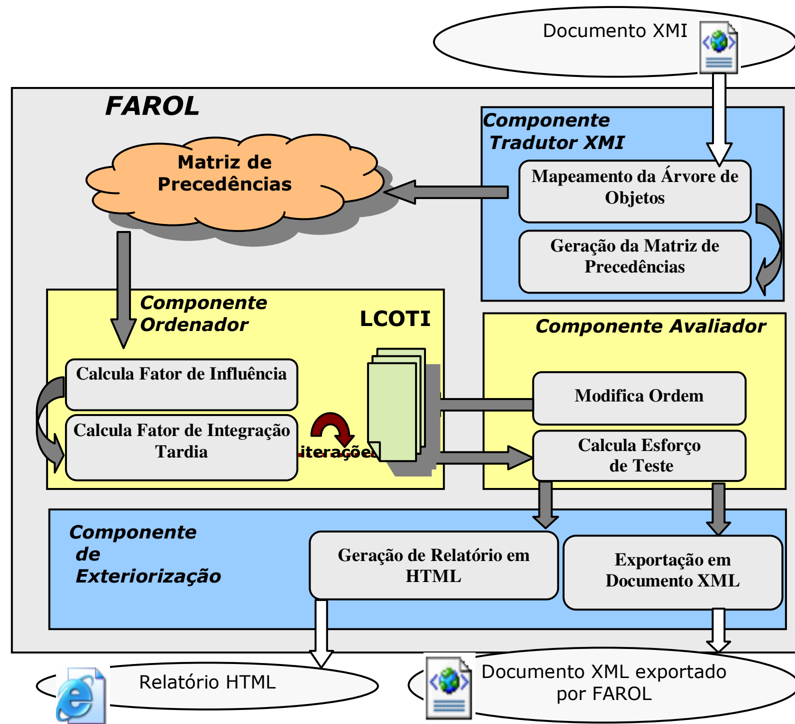

Figura 13. Arquitetura de Apoio ao Processo de Aplicação das Heurísticas

A Tabela 4 apresenta a matriz de precedências referente ao diagrama da Figura 14. Na Tabela 4, a precedência da classe A sobre as classes B e E está representada pelos valores 1 encontrados nas células (A,B) e (A,E), respectivamente. Outra leitura obtida da matriz é a precedência das classes A e C sobre a classe B diretamente pelos valores 1 nas células (A,B) e (C,B) da coluna B.

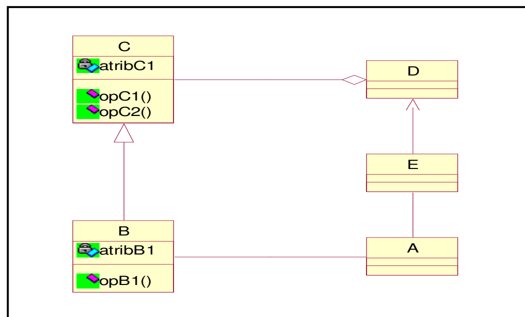

Figura 14 – Exemplo: Diagrama de Classes.

Tabela 4 – Matriz de precedências do exemplo.

Classes	A	B	C	D	E
A	0	1	0	0	1
B	1	0	0	0	0
C	0	1	0	1	0
D	0	0	0	0	1
E	1	0	0	0	0

A implementação do *processo de aplicação das heurísticas* é responsabilidade do componente *Ordenador* que determinará a Lista de Classes Ordenadas para Teste de Integração (LCOTI), disponível por meio da função “GerarOrdenação”. Os valores de FI e FIT são calculados a partir das informações da matriz de precedências:

- **Fator de Influência.** O FI de uma classe consiste na quantidade de colunas marcadas na matriz de precedências para a linha da classe. Por exemplo, o FI da classe C na Tabela 4 é 2, pois as células (C, B) e (C, D) estão marcadas. O mesmo é calculado para as demais classes:

$FI(C_i) = \sum_{j=1}^{N} MatInf(C_i, C_j)$, ou seja, o FI da classe C_i é o somatório das colunas para a classe C_i.

- **Fator de Integração Tardia.** O FIT para uma classe consiste no somatório do FI das classes marcadas na matriz de precedências, na coluna referente à classe em questão. Por exemplo, o FIT da classe B na Tabela 4 é 4, pois na coluna da classe B estão marcadas as classes A (FI=2) e C (FI=2), logo 2+2=4. A cada iteração, somente as classes ainda não ordenadas são consideradas no cálculo do FIT:

$FIT(C_i) = \sum_{j=1}^{N} MatInf(C_j, C_i) * FI(C_j)$, ou seja, o FIT da classe C_i é o somatório de cada linha da matriz de precedências multiplicado pelo valor de FI da classe referente a cada linha.

O componente *Avaliador* calcula o número total de *stubs* necessários e o tamanho total dos *stubs* a serem construídos (número de atributos mais número de métodos, total dos *stubs* identificados) para a LCOTI. Os valores de FI e FIT calculados e a LCOTI identificada para o modelo da Figura 14 são apresentados no Painel de Seqüência de Ordenação. A FAROL também permite ao engenheiro de software conhecer novos custos – esforço de teste (total de *stubs* e tamanho) – no caso de necessitar alterar a ordem original identificada. Esta facilidade foi implementada por meio dos botões de "Mover para cima" ou "Mover para baixo" a ordem da classe selecionada (ver Figura 12).

5. Conclusões e Trabalhos Futuros

No contexto dos estudos experimentais realizados, pode-se dizer que a estratégia definida atingiu o propósito estabelecido de identificar a ordem de integração de classes por meio de um conjunto de heurísticas e de um processo que, aplicada diretamente a um diagrama de classes UML do projeto, minimiza o esforço de teste (número de *stubs*), caracterizando a viabilidade e efetividade da estratégia de teste de integração proposta.

Os resultados obtidos demonstraram a necessidade de implementação da ferramenta FAROL que, a partir das informações contidas num arquivo XMI exportado de uma ferramenta CASE de modelagem, gera a ordem de integração das classes, automatizando o processo de aplicação das heurísticas.

Entendimentos adicionais podem ser obtidos por meio da análise das limitações estabelecidas neste trabalho. O esforço de teste medido considerou somente o número de *stubs* específicos envolvidos, assumindo que cada *stub* requer o mesmo esforço para ser criado. Então, a minimização do esforço de criação de *stubs* seria o mesmo que a minimização do número de *stubs*. Esta assunção poderia ser relaxada se fosse associado um valor de complexidade para criação dos *stubs*. Outra limitação trata a validade das conclusões obtidas. Estas não podem ser estendidas para projetos de maior escala, pois os resultados apresentados referem-se a modelos de pequenos projetos (máximo de 14 classes).

Algumas perspectivas de trabalhos futuros são vislumbradas, como: o planejamento de novos estudos de caso para estender a estratégia para tratar as outras situações (complexidade dos *stubs* e integração entre subsistemas, por exemplo). Além destes, acreditamos que outro desafio importante seria entender a aplicabilidade das heurísticas em paradigmas de desenvolvimento alternativos, como em sistemas multi-agentes (*Multi-Agent Systems – MAS*) [20].

Agradecimentos

À Marinha do Brasil pela oportunidade de participação no Curso de Mestrado em Engenharia de Sistemas e Computação do Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia - COPPE / UFRJ. Agradecemos à FAPEM pelo apoio fornecido. Este trabalho foi realizado no contexto do projeto CNPq – 475407/2003-2.

Referências Bibliográficas

[1] ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C., **Qualidade de Software: Teoria e Prática**, Prentice Hall, 2001.

[2] MYERS, G., **The Art of Software Testing**, John Wiley & Sons, 1979.

[3] BRIAND, L.C.; FENG, J.; LABICHE, Y., **Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders**, Carleton University, Technical Report SCE-02-03, Version 3, October, 2002. Disponível em http://squall.sce.carleton.ca/people/briand/pubs.html#2002 em 03 de julho de 2006.

[4] BINDER, R.V., **Testing object-oriented systems: models, patterns, and tool,** Addison-Wesley, 2000.

[5] KUNG, D., GAO, J., HSIA, P., TOYOSHYMA, Y., **Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs**, Journal of Object-Oriented Programming, vol. 8 (2), pp 51-65, 1995.

[6] TAI, K.C; DANIELS, F.J, **Test Order for Inter-Class Integration Testing of Object-Oriented Software**. 21st International Computer Software and Applications Conference, pp 602-607, August, 1997.

[7] TRAON, Y.L.; JÉRON, T.; JÉZÉQUEL, J.; e MOREL, P., **Efficient Object-Oriented Integration and Regression Testing**, IEEE Transactions Reliability, Vol. 49, no. 1, Page(s): 12-25, 0018-9529/00, 2000.

[8] BRIAND, L.C.; LABICHE, Y.; YIHONG, W, **An investigation of graph-based class integration test order strategies**, IEEE Transactions on Software Engineering, 0098-5589/03, Vol. 29, Issue: 7, Page(s): 594 -607, July, 2003.

[9] BOOCH, G., RUMBAUCH, J., JACOBSON, I., **UML – Guia do Usuário**, Editora Campus, 2000.

[10] FURLAN, J.D., **Modelagem de Objetos através da UML**, Makron Books**,** São Paulo, 1998.

[11] LIMA, G.M.P.S.; TRAVASSOS, G.H., **Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes**, SBQS 2004, pp 221-233, Brasília-DF, Maio, 2004.

[12] TRAVASSOS, G.H.; WERNER, C.; VASCONCELOS, F.M., **Testing Complex Object Oriented Models: a Practical Approach,** II International Congress on Informational Engineering, Buenos Aires, Argentina, 1995.

[13] OLIVEIRA, H.; TRAVASSOS, G.H., **Construção de um componente genérico baseado em heurísticas para ordenação das classes em ordem de prioridade de teste de integração,** Estudo da disciplina Laboratórios de Engenharia de Software, COPPE, UFRJ, 2003.

[14] VILLELA, K., **Ambientes de Desenvolvimento de Software Orientados à Organização**, Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, 2004. Disponível em **http://ramses.cos.ufrj.br/taba** (publicações, em 03 de julho de 2006).

[15] PFLEEGER, S.L., **Albert Einstein and Empirical Software Engineering**. IEEE Computer, pp. 32-38, Outubro, 1999.

[16] ZELKOWITZ, M.V., WALLACE, D.R., BINKLEY, D.W., **Experimental Validation of New Software Technology,** Lecture Notes On Empirical Software Engineering, Chapter 6, pp 229-263, World Scientific, 2003.

[17] WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M.C.; REGNELL, B.; WESSLÉN, A., **Experimentation in Software Engineering: an Introduction**. Kluwer Academic Publishers, Massachusetts, 2000.

[18] TRAVASSOS, G.H., GUROV, D., AMARAL, E.A.G.G., **Introdução à Engenharia de Software Experimental**. In: Relatório Técnico ES-590/02-Abril, PESC, COPPE/UFRJ, disponível em http://www.cos.ufrj.br/publicacoes, 2001.

[19] BRIAND, L.C., LABICHE, Y., WANG, Y., **Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles**. 12th ISSRE, Hong Kong, pp 287-296, November 27-30, 2001.

[20] LUCENA, C., **Ongoing Research on the Software Engineering of Multi-Agent Systems**. 18º. SBES, pp 02-09, Outubro, 2004.