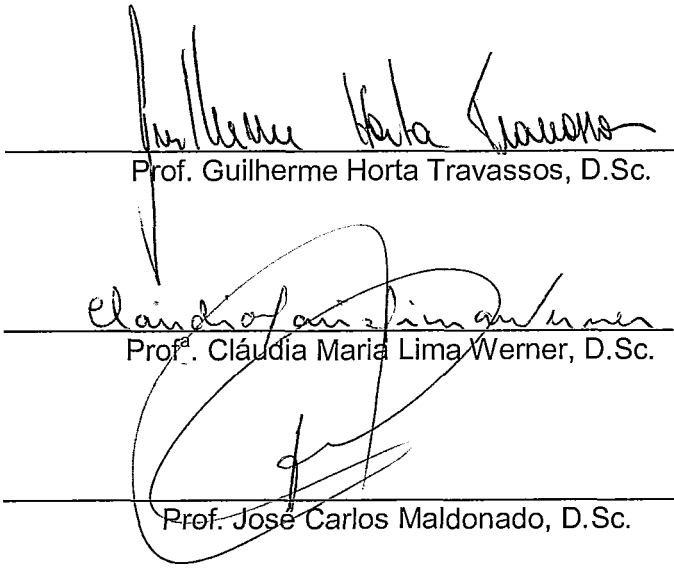
HEURÍSTICAS PARA IDENTIFICAÇÃO DA ORDEM DE INTEGRAÇÃO DE CLASSES EM TESTES APLICADOS A SOFTWARE ORIENTADO A OBJETOS

Gladys Machado Pereira Santos Lima

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Prof. Guilherme Horta Travassos, D.Sc.

Prof^a. Cláudia Maria Lima Werner, D.Sc.

Prof. José Carlos Maldonado, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2005

LIMA, GLADYS MACHADO PEREIRA SANTOS

Heurísticas para Identificação da Ordem de Integração de Classes em Testes Aplicados a Software Orientado a Objetos [Rio de Janeiro] 2005

XV, 148 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2005)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Ordem de Integração de Classes
2. Testes de Integração Orientados a Objetos

I. COPPE/UFRJ II. Título (série)

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

HEURÍSTICAS PARA IDENTIFICAÇÃO DA ORDEM DE INTEGRAÇÀO DE CLASSES EM TESTES APLICADOS A SOFTWARE ORIENTADO A OBJETOS

Gladys Machado Pereira Santos Lima

Março/2005

Orientador: Guilherme Horta Travassos

Programa: Engenharia de Sistemas e Computação

Uma questão crucial quando aplicando teste de integração em software orientado a objetos é decidir a ordem de integração das classes. As classes precisam ser integradas uma de cada vez ou, em alguns casos, em pequenos *clusters* já que a abordagem de integração *big-bang* se demonstra inadequada nesta situação. Conceitos como encapsulamento, herança e polimorfismo adicionam complexidade aos testes, fazendo com que critérios precisem ser estabelecidos para, eventualmente, quebrar a dependência existente entre as classes sem aumentar a complexidade (esforço) do teste.

Neste contexto, esta Tese apresenta: (1) as dificuldades para identificar a ordem de integração e testes das classes em software orientado a objetos; (2) algumas estratégias de integração existentes na literatura, como base de estudo dos requisitos de estratégias; (3) um conjunto de heurísticas e um processo de aplicação para identificar uma seqüência de integração das classes, buscando um esforço mínimo de teste; (4) quatro estudos de caso realizados para análise da efetividade das heurísticas; e (5) FAROL, uma ferramenta que implementa o processo de aplicação das heurísticas propostas.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

HEURISTICS FOR IDENTIFYING THE CLASS INTEGRATION ORDER IN OBJECT-ORIENTED SOFTWARE TESTING

Gladys Machado Pereira Santos Lima

March/2005

Advisor: Guilherme Horta Travassos

Department: System Engineering and computer Science

An important issue when applying object oriented integration testing is concerned with the decision about the classes' integration order. The classes need to be integrated one by one or, in some cases, in small clusters, since big-bang integration approach can not be directly applied. Concepts such as encapsulation, inheritance and polymorphism can increase integration testing complexity, mainly for those situations when developers need to identify a set of criteria (heuristics) to break down the dependences among the classes.

In this context, this Thesis presents: (1) the difficulties for identify a class integration testing order when applying the object oriented software paradigm; (2) some strategies for OO integration testing identified in the literature, base for the requirements to elaborate the heuristics; (3) a set of heuristics and the application process to identify a class integration testing order among at to minimize the test effort; (4) four experimental studies used to evaluate the heuristics feasibility; and (5) FAROL, a case tool that implements the proposed heuristics application process.

Ao Sérgio,

Thadeu e Luanna,

Pelo amor e compreensão.

Agradecimentos

A Deus, que pela infinita bondade e misericórdia, me concedeu saúde e inúmeras graças a fim de que conseguisse chegar ao fim de mais uma etapa.

Ao Prof. Guilherme Travassos, muito mais que um orientador, um incentivador e amigo, por fornecer um "senso de direção" a este trabalho, pelas dicas e conselhos sempre pertinentes e por acreditar no meu trabalho.

Aos membros da banca examinadora, Profª Claudia Werner e Prof. Maldonado, pela generosidade nas críticas a este trabalho e pelas valiosas sugestões. Foi uma honra contar com a avaliação de tão conceituados pesquisadores.

À Profª Ana Regina da Rocha, da área de Engenharia de Software, e demais professores da COPPE/UFRJ, pela contribuição ao meu aprendizado no decorrer do curso.

A minha mãe Edi e ao meu pai Walter (*in memorian*) pelo incentivo, apoio, exemplo, amor e carinho com que me educaram, nunca medindo esforços para me proporcionar o melhor que poderiam em todos os aspectos da vida. Serei sempre grata a vocês.

Ao meu marido Sérgio, por não somente ter compreendido as minhas muitas ausências, mas também pela participação em todas as etapas desta conquista.

À Marly, minha segunda mãe, que, ao cuidar dos meus filhos durante minhas horas de estudos e viagens para simpósios, proporcionou-me tranqüilidade para realização desta dissertação e cujo carinho e apoio incondicionais a mim demonstrados são motivos de minha gratidão e amor eternos.

Ao Hamilton Oliveira pelo entusiasmo e a preocupação em disponibilizar todos os conhecimentos disponíveis sobre seu relatório final do curso de Laboratório de Engenharia de Software, motivação inicial deste trabalho.

Às amigas de mestrado Lucia Nigro e Regiane, pelas muitas horas de estudos compartilhadas e pelas palavras de ânimo nos momentos difíceis.

Ao Arilo Cláudio por aceitar o desafio de implementar a ferramenta e cuja empolgação no aprendizado de um novo conhecimento comprovou seu espírito inovador e interesse pela pesquisa.

Aos amigos do Grupo de Engenharia de Software Experimental da COPPE: Luis Felipe, Rodrigo, Kalinowsky, Leonardo, Marco Antônio, Paula, Ana Cândida, Tayana, Wladmir, Hélio, Rafael, Sômulo, Paulo Sergio, não somente pelas revisões, sugestões e compartilhamento de conhecimentos, mas principalmente pelo companheirismo.

Aos colegas do Projeto Readers, Érika, André, Ubirajara e Walter (USP/São Carlos), Anderson (UFSCar), e aos colegas do CASNAV/Marinha, Suldine, Alencar, Corsino, Muradas, Rosane, Sérgio, Selma, Vera Ulm, Greice, Mozar e Ronaldo, pelo desprendimento e dedicação demonstrados na participação dos estudos experimentais, muitas vezes com o comprometimento de outras atividades pessoais e profissionais.

Aos amigos Vasconcellos e Alencar pelo apoio às minhas demandas administrativas junto ao CASNAV.

A Claudia, Solange, Lúcia, Mercedes, Sônia e demais funcionários do PESC, pela atenção e eficiência no atendimento às minhas solicitações.

A Borland do Brasil pela liberação das licenças dos produtos ao LENS/COPPE/UFRJ.

À Marinha do Brasil, pela oportunidade que me deu, para realizar este trabalho.

Sumário

Índice de Figuras

Índice de Tabelas

Capítulo 1
Introdução

Neste capítulo é apresentada a motivação para a realização deste trabalho, seu propósito principal e a sua organização.

1.1 – Motivação

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 no software, levando os desenvolvedores a encontrar os defeitos correspondentes antes da entrega do software (ROCHA *et al.*, 2001).

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 (MYERS, 1979). O teste de software pode ser aplicado em diferentes níveis: unidade, integração, sistema e aceitação (JACOBSON, 1992). O escopo deste trabalho está concentrado no teste de integração.

O teste de integração representa um conjunto de atividades com intenção de descobrir defeitos na estrutura do software definida durante a fase de projeto. Neste contexto, os defeitos são comumente associados com as interfaces dos módulos (componentes) que compõem a arquitetura do software (BRIAND *et al.*, 2002). Para a realização dos testes de integração, os engenheiros de software geralmente precisam construir módulos específicos para apoiar os testes: *stubs* (pseudo-controlado) ou *drivers* (pseudo-controlador) (PRESSMAN, 2001).

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 componentes (representados pelas classes), responsável pelos ciclos de dependências entre componentes. Conceitos OO como herança, encapsulamento e polimorfismo, que facilitam a representação das soluções durante a modelagem dos sistemas podem,

por si ou combinados, acrescer complexidade no planejamento e na execução dos testes de integração (BROOKS *et al.*, 1995) (KUNG *et al.*, 1995b) (PEZZE e YOUNG, 2004), tornando não trivial a tarefa de identificação da ordem de integração e o teste dos componentes. Entretanto, é importante decidir a ordem de integração, pois esta implica diretamente na determinação dos *stubs* necessários aos testes para simularem o comportamento dos componentes ainda não prontos (não integrados), exercendo simultaneamente os papéis de pseudo-controlador e pseudo-controlado, no contexto OO. Desta forma, reduzir o número de *stubs* significa reduzir o esforço de integração, pois os *stubs* e sua implementação representam esforço adicional de desenvolvimento que precisa ser minimizado.

O teste de integração pode também ser considerado uma atividade de construção, pois o paradigma OO permite o uso da mesma semântica (classes, por exemplo) no desenvolvimento e na implementação. Assim a redução do número de *stubs* conduzirá a um ganho de produtividade, já que a ordem mais adequada de construção das classes também será identificada.

Algumas estratégias para testes de integração vêm sendo propostas para resolver a ordem de integração das classes em projetos de software OO (TAI e DANIELS, 1997) (TRAON *et al.*, 2000) (BRIAND, 2001). Estas estratégias buscam resolver a dependências cíclicas minimizando o esforço de teste (número de *stubs*). Entretanto, necessitam construir artefatos adicionais (aos já existentes num projeto) para auxiliarem na sua aplicação, o que reflete em mais esforço extra.

Neste sentido, este trabalho foi norteado pela busca de uma nova estratégia para identificar a ordem de integração das classes. Porém, diferentemente das estratégias existentes, esta deve ser aplicada em um nível de abstração mais alto do projeto, permitindo antecipar algumas decisões. Permitirá, também, fornecer uma orientação da ordem de construção das classes, diferenciando-se novamente das outras estratégias, aplicadas a projetos prontos (código implementado). Outra motivação deste estudo é permitir aplicar a nova estratégia diretamente em um artefato pronto de projeto (diagrama de classes UML), garantindo a redução do esforço extra com artefatos adicionais. Entretanto, a nova estratégia deve manter ou, na melhor situação, reduzir o esforço de teste mínimo alcançado pelas estratégias existentes, ou seja, deve buscar reduzir o número de *stubs* associados à ordem de integração das classes identificada.

1.2 – Objetivo da Tese

O propósito fundamental deste trabalho é definir um conjunto de heurísticas e um processo para sua aplicação para apoiar engenheiros de software durante a tarefa de identificação da ordem de integração de classes em testes aplicados a software OO. A aplicação das heurísticas diretamente no diagrama de classes UML (*Unified Modeling Language*) deve fornecer uma ordem de integração com o menor esforço de teste possível.

Nesse sentido, a pesquisa inicialmente se direcionou para o estudo de OLIVEIRA e TRAVASSOS (2003) e de estratégias para testes de integração (SIT – *Strategy for Integration Testing*) encontradas na literatura. Uma análise da proposta de OLIVEIRA e TRAVASSOS mostra o uso da semântica da UML para identificar a ordem de integração de classes, entretanto não apresenta tratamento para certas situações especiais (dependências cíclicas, por exemplo), não identificando uma solução geral e com menor esforço de teste, quando comparada às outras estratégias estudadas. Foram definidos novos tratamentos para as situações identificadas e elaborado um novo processo de aplicação das heurísticas, juntamente com um aperfeiçoamento e complemento destas.

Para verificar a efetividade das heurísticas e do processo de aplicação foram planejados alguns estudos de caso em dois ambientes: na academia e na indústria. Participaram destes estudos alunos de mestrado e doutorado da COPPE/UFRJ e USP/São Carlos; um aluno de doutorado da UFSCarlos/São Carlos; e alguns profissionais do Centro de Análises de Sistemas Navais (Marinha do Brasil). Os resultados permitiram avaliar as heurísticas e identificar a necessidade de automatizar o processo de aplicação.

Por fim, foi definida e implementada uma ferramenta para apoiar o engenheiro de software na atividade de identificar a ordem das classes para execução dos testes de integração, denominada FAROL. Devido a grande variedade de ferramentas CASE para representação de diagrama de classes (UML), optou-se pela construção de uma solução independente destas aplicações, utilizando o padrão XMI (*XML Metadata Interchange*) como solução para o intercâmbio de dados entre ferramentas CASE e FAROL.

1.3 – Organização do Texto

Além desta introdução, está tese contém mais cinco capítulos e três anexos.

No Capítulo 2, *Testes de Integração em Software Orientado a Objetos*, é feita uma análise sobre testes de integração, enfatizando as principais dificuldades na identificação da ordem das classes para execução dos testes de integração em software OO. Algumas estratégias existentes são discutidas, criando uma base para o entendimento do problema, sendo também feita uma análise de suas desvantagens.

No Capítulo 3, *Heurísticas para Ordenação de Classes em Testes de Integração Aplicados a Software Orientado a Objetos*, é proposto um conjunto de heurísticas como uma nova estratégia para identificar a ordem de integração e testes das classes em um projeto OO, partindo dos requisitos levantados no Capítulo 2. É apresentado também um processo para controlar a aplicação das heurísticas para garantir que a ordem identificada corresponda a um esforço de teste mínimo.

No Capítulo 4, *Estudos de Caso*, é apresentada uma breve introdução à Engenharia de Software Experimental com a finalidade de justificar sua importância na caracterização da efetividade das heurísticas, propostas no Capítulo 3, por meio de quatro estudos de caso planejados e executados, cujos dados coletados e analisados são apresentados juntamente com as lições aprendidas e conclusões.

No Capítulo 5, *FAROL: Automatização do Processo de Aplicação das Heurísticas*, é apresentado o protótipo desenvolvido para dar apoio às atividades do processo descrito no Capítulo 3 e cuja necessidade foi levantada junto à avaliação qualitativa dos resultados obtidos durante os estudos realizados e apresentados no Capítulo 4.

No Capítulo 6, *Considerações Finais*, são apresentadas as conclusões ressaltando suas contribuições, limitações e perspectivas futuras.

No Anexo A, *Linguagem para Modelagem de Processos*, é apresentada a linguagem de modelagem utilizada para representar o processo de aplicação das heurísticas apresentado no Capítulo 3.

No Anexo B, *2º. Estudo de Caso*, são apresentados o modelo e formulários utilizados durante a execução do estudo apresentado no Capítulo 4.

No Anexo C, *3º. Estudo de Caso*, são apresentados o autotreinamento sobre as heurísticas e formulários utilizados durante a execução do estudo apresentado no Capítulo 4.

Capítulo 2
Testes de Integração em Software Orientado a Objetos

Neste capítulo é descrito e analisado o teste de integração em software orientado a objetos, enfatizando as principais dificuldades na sua aplicação e algumas estratégias existentes para integração e teste de classes.

2.1 – Introdução

A introdução do paradigma orientado a objetos (OO) no contexto de desenvolvimento de software provocou mudanças significativas na forma como os produtos de software são criados e mantidos. As mudanças foram motivadas pela nova perspectiva adotada pelo paradigma OO (ênfase nos objetos), em oposição ao paradigma procedural que utiliza uma abordagem focada na funcionalidade e no fluxo de informação dos sistemas.

Em conseqüência dessa nova visão para o desenvolvimento de software, muitas áreas tiveram que ser revisadas, pois teorias, práticas, modelos e métricas que eram adequados para software convencionais não podem ser aplicados de forma irrestrita quando se desenvolve software OO. Uma dessas áreas é o teste de software.

A atividade de teste de software é uma das atividades de garantia de qualidade de software, sendo uma das atividades de verificação e validação, com o propósito de verificar a presença de defeitos no produto ou artefato, conforme considerado pela norma ISO / IEC 12207 (1995). O termo *defeito* pode ser entendido como a causa de um comportamento incorreto do programa, assim como *engano* e *erro* (ROCHA *et al.*, 2001). O padrão IEEE 610.12 (1990) diferencia os termos da seguinte forma: *defeito* é o passo, processo ou definição de dados incorretos; *engano* corresponde a uma ação humana que produz um resultado incorreto; e *erro* significa a diferença entre o valor obtido e o valor esperado. Neste sentido, o termo *falha* é usado para designar uma conseqüência de um *defeito,* sendo definido como a produção de uma saída incorreta com relação à especificação.

Neste contexto, uma visão mais ampla sobre a atividade de teste de integração no processo de desenvolvimento de software OO vem sendo tratada por vários pesquisadores (BEIZER, 1990) (FEWSTER e GRAHAM, 1999) (McGREGOR e SYKES, 2001) (CRAIG e JASKIEL, 2002) (DUSTIN, 2003) (HUTCHESON, 2003). Vieira (1998) apresentou uma abordagem para apoiar o engenheiro de software na realização de testes de software OO, com a identificação da infra-estrutura necessária e a implementação de algumas facilidades de apoio. BINDER (2000) apresenta uma metodologia para projetar testes para linguagens OO, onde os conceitos de classes, *clusters*, *frameworks* e subsistemas são tratados, considerando os novos problemas para testar software OO, entre eles conceitos fundamentais, como polimorfismo e herança.

Neste capítulo são apresentadas: uma visão geral sobre teste de software e, mais detalhadamente, uma visão sobre testes de integração e as suas especificidades dentro do contexto da orientação a objetos, tendo a particular intenção de ressaltar a dificuldade de determinar a seqüência ou ordem de integração e teste das classes. Algumas estratégias apresentadas por diferentes pesquisadores para tratar este problema foram estudadas, sendo também relatadas ao final deste capítulo, juntamente com algumas considerações sobre suas limitações.

2.2 – Teste de Software

Segundo PERRY (1995), é irreal desenvolver software e não testá-lo. O processo de desenvolvimento perfeito não existe e a probabilidade de existir no futuro é quase nula. Durante o processo de desenvolvimento e manutenção do software são introduzidos defeitos.

Os testes de software são realizados para garantir que os sistemas funcionem apropriadamente sob todas as condições previstas pelos requisitos do software. O propósito dos testes é fazer com que o software falhe, e desta forma, possibilitar a identificação de problemas, levando o desenvolvedor a encontrar defeitos para que estes possam ser corrigidos. Testar é o processo de executar programas com a intenção de encontrar defeitos (MYERS, 1979).

Existe um consenso de que as atividades de teste buscam reduzir os riscos envolvidos no desenvolvimento do software e no seu uso (VIEIRA e TRAVASSOS, 1998b). Testar software é melhorar a qualidade do software (HUTCHESON, 2003). Porém, é necessário planejar, projetar e organizar testes de um software de forma a

garantir sua adequação e viabilidade. A ausência de planejamento das atividades de desenvolvimento é uma das causas da crise do software (PRESSMAN, 2001).

Segundo o padrão IEEE 610.12-1990, o teste é definido como um processo de operar um sistema ou um componente sob certas condições específicas, observando e registrando os resultados e fazer avaliações de alguns aspectos de um sistema ou um componente.

O teste de software é uma das atividades do processo de desenvolvimento de software que é um dos processos fundamentais do ciclo de vida da norma NBR ISO/IEC 12207 (1995). Segundo DUSTIN (2003), o teste para ser mais efetivo precisa estar integrado ao processo de desenvolvimento do software como um todo.

Uma das maneiras de se prevenir ou detectar defeitos cometidos ao longo do processo de desenvolvimento é a definição de diferentes níveis de testes em relação às diversas fases do processo de desenvolvimento (MYERS, 1979).

A seguir apresentamos as fases propostas para divisão dos testes.

2.2.1 – Fases de Testes

A atividade de teste é dividida em fases, tanto na abordagem de desenvolvimento de software procedural quanto na abordagem orientada a objeto e desenvolvida de forma incremental e complementar (ROCHA *et al.*, 2001). Cada fase tem objetivos específicos e limitações (RAKITIN, 2001). Basicamente, existem as fases de teste: de unidade, de integração, de validação e do sistema.

Na primeira fase, cada componente é testado isoladamente dos outros componentes do sistema. Tem o propósito de explorar a menor unidade do projeto – componente ou módulo, procurando identificar erros de lógica e implementação de cada módulo. O teste de unidade verifica se o componente funciona de forma adequada aos tipos de entradas esperadas, a partir do estudo do projeto do componente. Sempre que possível controlado. É fornecido ao componente a ser testado um conjunto de dados predeterminado, sendo observado quais ações e dados são produzidos (PFLEEGER, 2004).

A próxima fase é o teste de integração, onde deve assegurar-se que as interfaces entre os componentes foram definidas e tratadas adequadamente. O teste de integração é o processo de verificar se os componentes do sistema, juntos, trabalham como descrito nas especificações do sistema e do projeto do programa. Segundo PRESSMAN (2001), teste de integração é uma técnica sistemática para construir a estrutura do programa enquanto, ao mesmo tempo, conduz testes para

descobrir erros associados às interfaces. Os objetivos são tornar componentes testados em nível de unidade e construir a estrutura do programa determinada pelo projeto.

Na fase seguinte são verificadas se as especificações estabelecidas no início do processo de desenvolvimento estão de acordo com o software construído, sendo este o teste de validação. E por fim, o teste de sistema, no qual o software e os outros elementos do sistema são testados como um todo.

A Figura 2.1 mostra uma adaptação da proposta de Jacobson para testes orientados a objetos (JACOBSON, 1992).

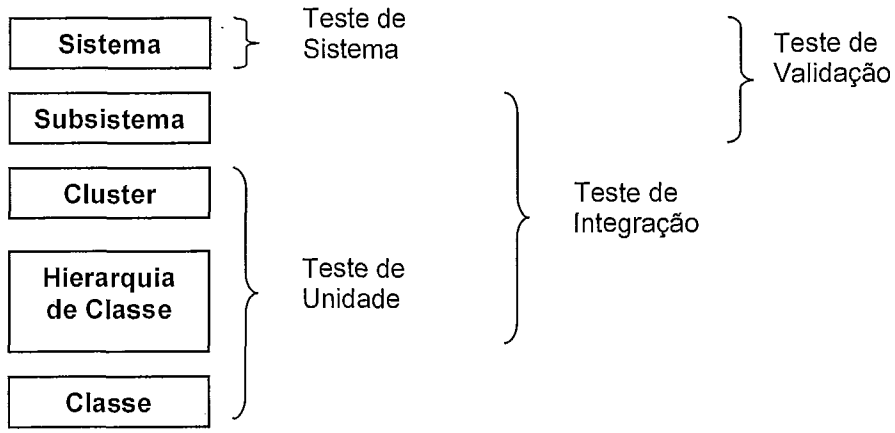


Figura 2.1 – Adaptação para aplicação de testes OO.

Mais detalhes e algumas abordagens sobre os testes de integração são vistas a seguir.

2.3 – Teste de Integração

Quando combinamos unidades, os caminhos possíveis a serem percorridos pelo programa crescem bastante e podem ocorrer falhas impossíveis de serem identificadas quando as unidades são testadas separadamente (VIEIRA, 1998). Por meio do teste de integração será testado se as unidades conseguem trabalhar juntas de forma correta e se comunicam sem problemas.

O teste de integração pode ter sua execução iniciada assim que alguns componentes ficarem prontos. No contexto do ciclo de desenvolvimento, componente pronto significa componente implementado e com os testes de unidade aprovados, ou seja, componentes individuais funcionando corretamente e atingindo os seus objetivos (PFLEEGER, 2004).

Esta fase auxilia a encontrar respostas a alguns questionamentos, como:

- Um componente pode ter efeito adverso sobre o outro?
- Erros de interface, não detectados nos testes de unidades, podem aparecer?
- Como detectar problemas de tempo, em sistemas de tempo real?

Segundo BINDER (2000), uma estratégia de integração deve responder às questões:

- Quais componentes são focados no teste de integração?
- Em que seqüência as interfaces serão exercitadas?
- Qual técnica será empregada para exercitar cada interface?

As estratégias afetam a sincronia (ordem) da integração e a ordenação do código, bem como o custo e a eficácia dos testes. A seguir apresentamos algumas estratégias existentes, juntamente com algumas evidências sobre suas vantagens e desvantagens individuais.

2.3.1 – Estratégia Não-incremental

Conhecida como a abordagem *big-bang* (MYERS, 1979), onde todos os componentes são combinados com antecedência. O sistema inteiro é testado de uma só vez. Geralmente esta abordagem não fornece um resultado positivo, pois vários erros são encontrados, tornando a correção difícil por ser complicado isolar a causa (VIEIRA, 1998). Outra desvantagem, PFLEEGER (2004), seria não poder distinguir facilmente os defeitos da interface de outros tipos de defeitos.

Segundo BINDER (2000), tentar integrar a maioria ou todos os componentes ao mesmo tempo é normalmente problemático. O teste é não sistemático e alguns defeitos de interface podem escapar da detecção.

Considerando que a correção dos erros pode gerar novos erros (PRESSMAN, 2001), a fase dos testes pode se prolongar. Todo esse caos acarreta altos custos e a não garantia da eficácia dos testes, não sendo recomendada para nenhum sistema (PFLEEGER, 2004).

2.3.2 – Estratégia Incremental

A integração incremental é a antítese da abordagem *big-bang* (PRESSMAN, 2001). O sistema é construído e testado em pequenos incrementos, pois desta forma os erros podem ser mais facilmente isolados e corrigidos. Esta vantagem aumenta a possibilidade das interfaces serem testadas completamente e torna viável uma abordagem sistemática de teste.

Segundo PFLEEGER (2004), para sistemas grandes, alguns componentes podem estar na fase de codificação, outros na de teste de unidade, e os conjuntos de componentes podem ser testados juntos.

Esta estratégia vê o sistema como uma hierarquia de componentes, onde os componentes pertencem a camadas distintas do projeto. Existem três abordagens para integrar o sistema maior, a fim de testar o sistema maior: integração ascendente (*bottom-up*); integração descendente (*top-down*); e integração combinada (sanduíche).

Na abordagem ascendente cada componente no nível inferior da hierarquia do sistema é testado individualmente. A seguir serão incluídos nos testes os componentes que chamam os componentes testados anteriormente. Isto se repete até que todos os componentes sejam incluídos no teste. A abordagem descendente é o reverso da ascendente. O componente de nível superior é testado sozinho, em seguida todos os componentes chamados são combinados e testados, sendo reaplicada até que todos os componentes sejam incorporados. A combinação das duas abordagens é a abordagem sanduíche (MYERS, 1979).

Como o software orientado a objetos é organizado como uma coleção cooperativa de objetos, cada um representado como sendo instância de uma classe, e as classes podendo pertencer a hierarquias unidas entre si pela relação de herança ou se relacionando por associação, agregação, uso, instanciação ou metaclasse (BOOCH, 1994), o software orientado a objetos não tem uma estrutura de controle hierárquica (PRESSMAN, 2001). Devido à herança e ao polimorfismo, o controle não é centralizado na classe (unidade) principal.

Exercitar as interações entre as unidades visando determinar se as instâncias das classes se relacionam como o esperado, pode requerer testes de integração mais extensivos e complexos (SOUTER, 2002) (PFLEEGER, 2004). Na seção seguinte, serão discutidas mais detalhadamente as características especiais da orientação a objetos em relação ao teste de integração, buscando entender as novas necessidades e objetivos.

2.4 – Teste de Integração Orientado a Objetos

A dificuldade do entendimento da execução do programa OO traz um novo desafio, pois as estratégias tradicionais de teste não são capazes de lidar totalmente com as questões da orientação a objetos. Os conceitos da orientação a objetos como herança, encapsulamento e polimorfismo, que facilitam a representação das soluções durante a modelagem dos sistemas podem, por si ou combinados, acrescer complexidade no planejamento e na execução dos testes de integração (BROOKS *et al.*, 1995) (KUNG *et al.*, 1995b) (PEZZE e YOUNG, 2004). A necessidade de testes é muito maior, uma vez que os erros podem "se esconder" nos níveis de hierarquia e estarem relacionados ao estado de um objeto ao invés de uma seqüência de passos específica.

GRAHAM (1993) ressalta que o polimorfismo (atributo positivo no projeto OO) ao esconder a complexidade do código pode tornar os testes OO mais difíceis, não sendo possível identificar o código executado quando se utilizando uma função polimórfica. Outro aspecto OO revelado é que a tendência de objetos pequenos pode transferir a complexidade para a interface do componente, o que levará também a teste de integração mais extensivo (PFLEEGER, 2004).

Segundo VIEIRA (1998), outra característica OO que pode aumentar a complexidade dos testes é a herança. O mecanismo de herança conduz a menos código, mas a quantidade de testes não é necessariamente menor. Classes não-instanciáveis (classes abstratas e classes genéricas) não são testadas diretamente. Quando testamos a comunicação entre objetos – testar a interface e o protocolo de mensagens – pode ocorrer problemas em relação à ordenação dos métodos, como, por exemplo, para garantir que um método não é invocado em um objeto após o método destruidor ter sido chamado.

As estratégias incrementais (ascendente, descendente ou combinada), apresentadas anteriormente, são baseadas em grafos (ou árvores) para tratar integrações sem dependências cíclicas e isto torna difícil a aplicação direta destas estratégias em sistemas OO (BARBEY e STROHMEIER, 1994). Os diversos relacionamentos existentes entre classes num programa OO implicam, inevitavelmente, que classes dependam uma das outras (LABICHE *et al*, 2000), o que ocasiona os ciclos de dependências entre componentes, freqüentes em sistemas OO.

Alguns pesquisadores apresentam abordagens que tratam as dependências entre classes para determinar, então, qual a ordem de integração das unidades (BINDER, 1994) (SOUTER, 2002). A decisão fundamental para o teste de integração

seria identificar a ordem que as unidades serão integradas e testadas (TAI e DANIELS, 1997) (TRAON *et al.*, 2000) (BRIAND, 2001).

Antes de apresentarmos novos modelos e estratégias que considerem a alta conectividade entre os componentes de projetos orientados a objetos, mostraremos a seguir alguns conceitos empregados durante o teste de integração: análise de dependência; *stubs*; e esforço de teste.

2.4.1 – Análise de Dependência

Segundo BINDER (2000), os componentes, tipicamente, dependem uns dos outros de diversas maneiras. As dependências são necessárias para implementar colaborações e conseguir a separação de alguns interesses. Algumas dependências são acidentais ou efeitos colaterais de uma implementação, linguagem ou ambiente. As dependências no escopo das classes e *clusters* resultam de alguns mecanismos como: composição e agregação, herança, variáveis globais, uso de objetos como parâmetros de mensagens, entre outros.

Como pode ser observado no diagrama de classes de um sistema de controle de vendas, mostrado na Figura 2.2, existe um ciclo de dependência entre as classes: *Cliente*; *Pedido*; e *CondiçãoEntrega*. Neste exemplo, para testarmos a classe *Cliente* é necessário que as classes *Pedido* e *CondiçãoEntrega* já tenham sido testadas, mas ambas apresentam dependência de *Cliente*, sendo, então, formado um ciclo de dependência. Outro ciclo encontrado no exemplo é formado por *Cliente*; *Pedido*; e *CondiçãoPagamento*.

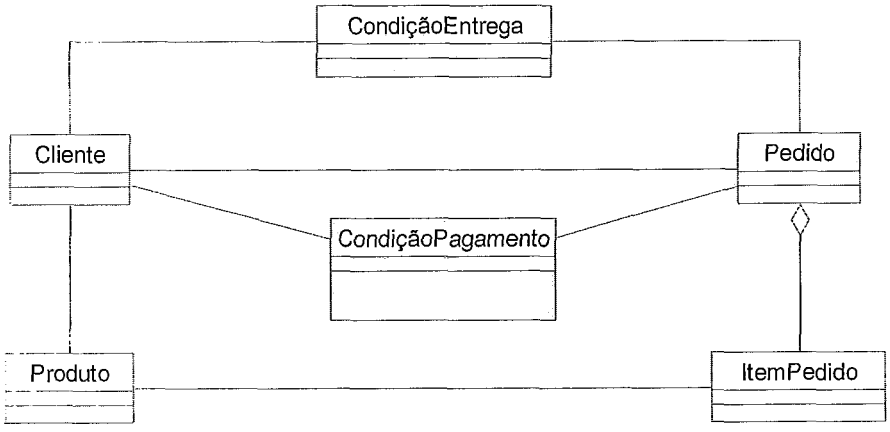


Figura 2.2 – Diagrama de Classe: Sistema de Controle de Vendas.

De maneira semelhante, as dependências componentes ocorrem entre componentes no escopo de um sistema ou de um subsistema.

Os componentes dependem explicitamente dos outros quando eles trocam mensagens, executam chamadas a procedimentos remotos (RPC – *Remote Procedure Calls*) ou usam serviços de comunicação entre processos. Existem, também, as dependências implícitas (por exemplo: comunicações de persistência e restrições de tempo) que não são foco dos testes de integração. As dependências explícitas entre componentes correspondem às interfaces que necessitam ser exercitadas.

2.4.2 – *Stubs* e Esforço de Teste

Num processo de desenvolvimento de software, principalmente para grandes sistemas, os componentes podem estar em fases distintas do seu ciclo de desenvolvimento. Enquanto alguns podem estar prontos para executar testes de integração, outros podem ainda estar em fase de codificação ou em fase de testes individuais (testes de unidades).

Segundo PFLEEGER (2004), independentemente da estratégia escolhida, cada componente é integrado para teste somente uma vez. Além disto, em nenhum momento, um componente deve ser modificado para simplificar o teste. Assim, para dar prosseguimento aos testes quando existirem componentes necessários à integração ainda não codificados e não testados é preciso que sejam criados os *stubs* de testes. Um *stub* é usado como fachada para simular o comportamento de um componente real (BEIZER, 1990) (BRIAND et al., 2001). *Stub* é a implementação parcial de um componente, com a implementação mínima de método (do *stub*) que permita controle e observação durante o teste (BINDER, 2000).

Se um componente A usa um ou mais serviços de outro componente B, dizemos que A "depende" de B. Num processo incremental, quando A for integrado, se B ainda não tiver sido, precisaremos simular os serviços de B. Como *stubs* são pedaços de software que devem ser construídos para simular partes de software que ainda não foram desenvolvidas ou que ainda não foram testadas, mas necessários para testar componentes dependentes deles, precisaremos de um *stub* de B para testar A.

Existem dois tipos de *stubs*: específico ou realístico (TRAON *et al.*, 2000), como mostrado na Figura 2.3. O *stub* específico simula os serviços de um componente para uso de somente um cliente. O *stub* é específico para o componente chamador. O

stub realístico simula todos os serviços do componente original. No caso de um componente chamador usar serviços de diversos outros componentes deverão ser escritos *stubs* específicos para cada um dos serviços solicitados.

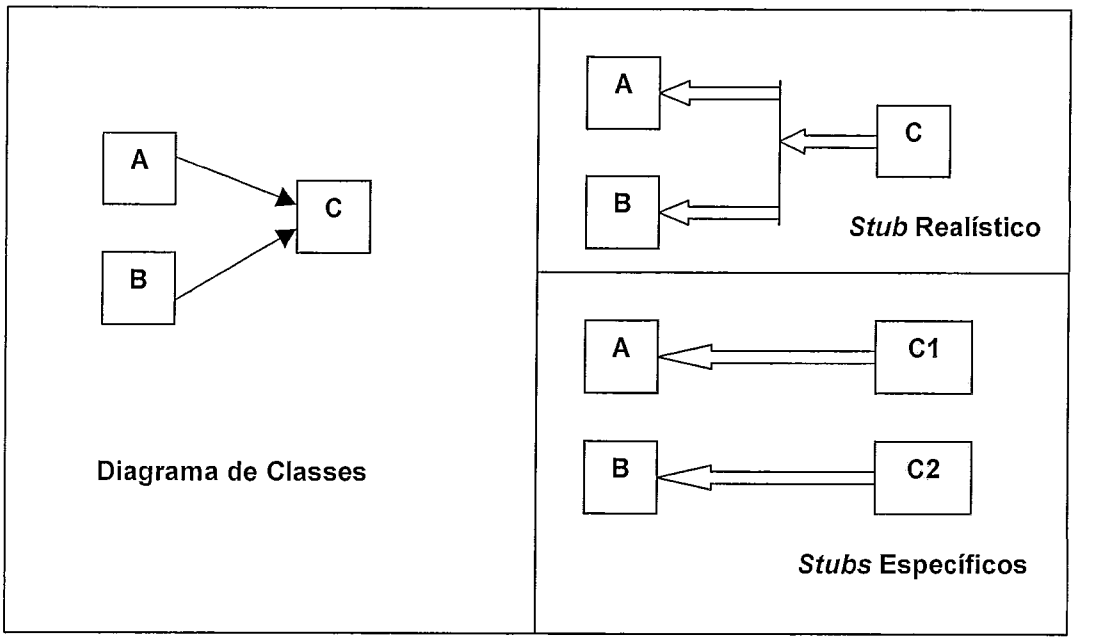


Figura 2.3 – Stubs Específicos e Realísticos (HANH et al., 2001).

Como o *stub* não é um componente real e nem será entregue com o produto final, então, devemos nos preocupar com o número de *stubs* criados. O *stub* representa despesa indireta (PRESSMAN, 2001). Se considerarmos que a complexidade dos *stubs* é a mesma, o esforço do teste pode ser medido pelo número de *stubs* necessários ao teste. Minimizar o esforço de teste é o mesmo que minimizar o número de *stubs* necessários na continuidade dos testes.

Quando as dependências entre componentes não criarem ciclos de dependências não serão necessários *stubs*. Entretanto, a diferença no tratamento (identificação e critério de quebra) dos ciclos de dependência pelas estratégias de teste de integração é fundamental para distinguir o esforço de teste destas estratégias. A ordem de integração dos componentes determinada pela estratégia afeta o número de *stubs* necessários e, conseqüentemente, o esforço de teste. A seguir serão apresentadas algumas estratégias que buscam a redução do esforço de teste por meio da minimização do número de stubs.

2.5 – Estratégias de Testes de Integração

Existem algumas estratégias propostas para identificação da ordem das classes para execução do teste de integração (em inglês, *Strategy for Integration Testing* – SIT) empregando a análise dos ciclos de dependências entre classes com o propósito de minimizar o número de *stubs* necessários nas quebras das dependências, minimizando o esforço de teste.

Com foco no processo utilizado, serão apresentadas as seguintes abordagens: Ordem de Teste de KUNG, GAO, HSIA e TOYOSHYMA; Ordem de Teste de TAI e DANIELS; Ordem de Teste de TRAON, JÈRON, JEZEQUEL e MOREL; Ordem de Teste de BRIAND, LABICHE e WANG; e Ordem de Teste de OLIVEIRA e TRAVASSOS.

2.5.1 – Ordem de Teste de KUNG, GAO, HSIA e TOYOSHYMA.

O algoritmo para encontrar a ordem do teste de integração definido por KUNG *et al.* (1995a) é baseado em um diagrama de relações entre objetos (em inglês, *Object Relation Diagram* – ORD). Um ORD é um grafo onde os vértices representam as classes de um programa e os ramos, os relacionamentos de herança (*Inheritance* – I), agregação (*Aggregation* – Ag) e associação (*Association* – As) (CHUNG *et al.*, 1997). Para quaisquer duas classes A e B:

- Uma seta de A para B (nomeada I) indica que A é herança de B;
- Uma seta de A para B (nomeada Ag) indica que A é uma agregação de B (A contém um ou mais objetos de B); e
- Uma seta de A para B (nomeada As) indica que A se associa com B.

Um ORD captura somente dependências estáticas entre classes (em tempo de compilação). Na Figura 2.4, o ORD é composto de quatro classes (A, B, C e D), onde C é herança de A, A é uma agregação de componentes B e B se associa com D.

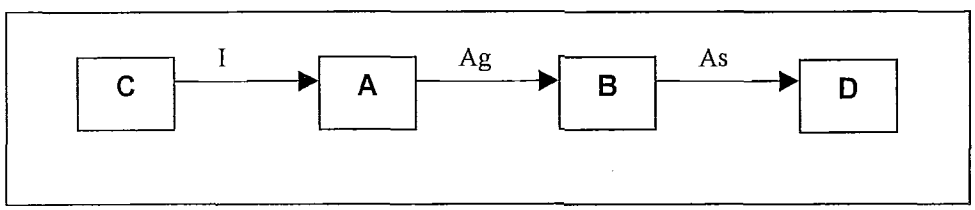


Figura 2.4 – Exemplo 1 de ORD.

KUNG *et al.* (1995a) utilizam o conceito de classe *firewall*. A classe *firewall* CFW(X) para a classe X são todas as classes afetadas por alterações em X, sendo testadas novamente no caso de mudanças em X. CFW(X) é composta por: classes heranças de X, classes componentes de X (agregação) e classes associadas a X.

Após a construção do ORD e de determinar a classe *firewall* de cada classe, KUNG *et al.* propõem que as classes independentes sejam testadas antes das classes dependentes (com base nos relacionamentos), com o propósito de minimizar o número de *stubs*. A estratégia propõe quebrar os ciclos de dependência por meio da identificação dos SCC (*Strongly Connected Component*) e remover as associações até não existirem mais ciclos com SCC. Baseado neste princípio, no caso de ORD acíclico, o algoritmo de ordenação produz uma ordenação topológica.

Usando o exemplo de KUNG *et al.* (Figura 2.5) teríamos: A e D sendo testadas primeiramente em qualquer ordem; e E, F e H testadas após as outras cinco. As respectivas classes *firewall* são mostradas na Tabela 2.1.

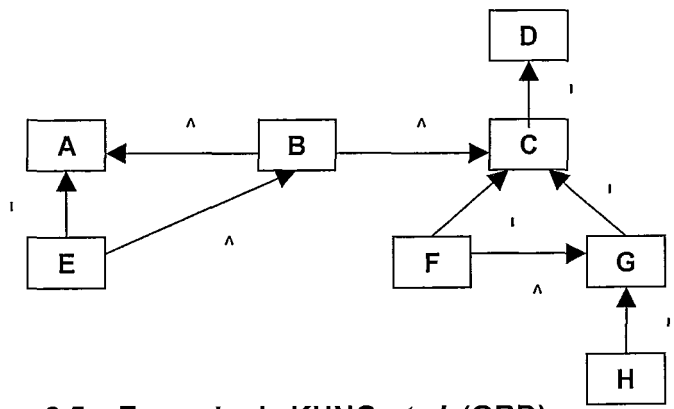


Figura 2.5 – Exemplo de KUNG *et al.* (ORD).

Classes X	CFW(X)
A	B, E
B	E
C	B, E, F, G, H
D	B, C, E, F, G, H
E	-
F	-
G	F, H
H	-

Tabela 2.1 – Classes *Firewall* do exemplo de KUNG *et al.*

O grande problema nesta proposta é não fornecer soluções precisas para diagramas que contêm ciclos de dependência, muito comuns em diagramas de classes. A escolha da associação para quebra um ciclo de dependência será aleatória quando existir mais de uma associação candidata.

2.5.2 – Ordem de Teste de TAI e DANIELS

A proposta de TAI e DANIELS (1999) também emprega diagramas ORD para encontrar a ordem de integração, sendo guiada por dois números: o *nível maior* e o *nível menor*.

O *nível maior* é baseado somente nas dependências de herança e agregação, onde o *nível maior* 1 representa classes independentes. O *nível menor* é baseado em dependências de associação, onde podem aparecer os ciclos de dependência que precisam ser quebrados para aplicar o processo de ordenação topológica.

A Tabela 2.2 apresenta os valores do *nível maior* e do *nível menor* calculados para os objetos do Exemplo 2 de ORD, apresentado na Figura 2.6.

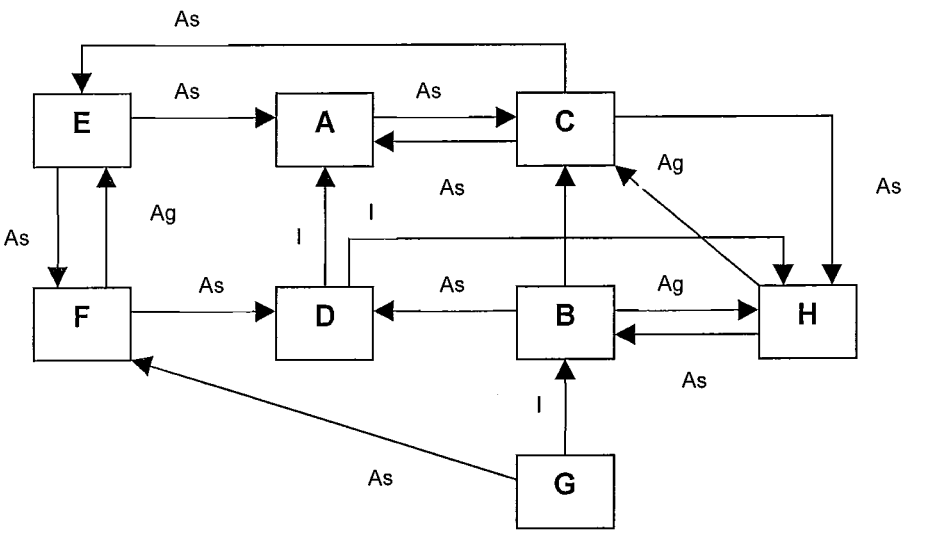


Figura 2.6 – Exemplo 2 de ORD.

Classe	Nível Maior	Nível Menor
A	1	1
B	3	2
C	1	3
D	3	1
E	1	2
F	2	1
G	4	1
H	2	1

Tabela 2.2 – Valores de nível maior e nível menor do Exemplo 2.

A quebra das dependências no *nível menor* está baseada numa função peso de cada nó (número de dependências que chegam mais as que saem do nó). As quebras ocorrem onde existam associações de maiores pesos e que sejam entre classes que possuam valores de *nível maior* distintos, no sentido do menor valor para o maior valor do *nível maior* (associação ascendente). Associações descendentes que ocorrem entre classes com *nível maior* distintos, mas no sentido do maior valor para o menor valor de *nível maior* não são quebradas. A Figura 2.7 apresenta as associações que são quebradas (tracejadas em azul) para o Exemplo 2 de ORD.

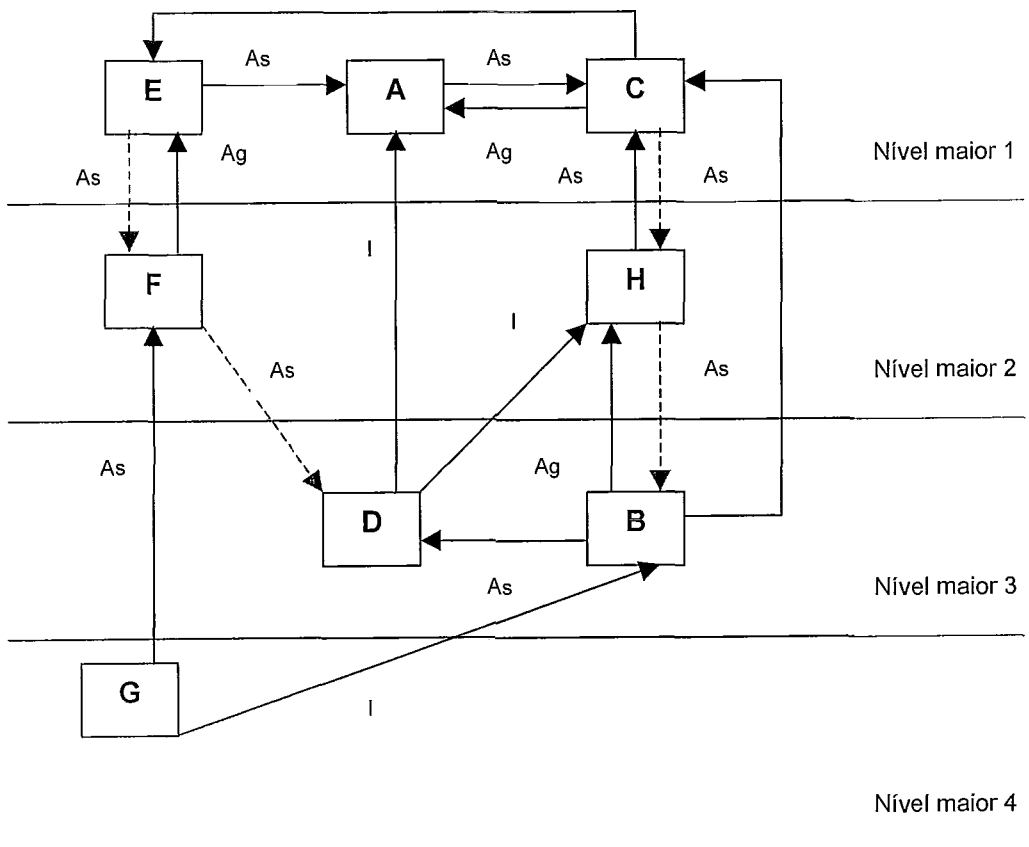


Figura 2.7 – Associações quebradas pelo critério de TAI e DANIELS.

Poderá ocorrer da heurística ser aplicada em associações que não estejam envolvidas em ciclos de dependências. Isto acarretará na criação de um *stub* desnecessário, levando a uma solução do problema com esforço de teste maior que o mínimo.

2.5.3 – Ordem de Teste de TRAON, JÈRON, JEZEQUEL e MOREL.

LE TRAON *et al.* (2000) apresentam uma metodologia para planejar teste de integração a partir de um modelo OO, que empregam grafos de dependências – TDG (*Test Dependency Graph*), como base para ordenação das classes (minimizando o número de *stubs*). Um TDG pode ser extraído a partir de um modelo de projeto (como, diagramas de classes UML para Software Orientado a Objetos), conforme algumas regras.

Num TDG, os vértices representam os componentes (classes ou métodos, dependendo do detalhamento do projeto) e as setas mostram a dependência de teste entre componentes. Existem três níveis de dependência de teste: classe para classe; método para classe; e método para método. O nível classe para classe pode ser deduzido de um diagrama de classes UML, onde as classes são representadas pelos vértices e as setas representam as dependências entre as classes, onde uma seta de A para B significa que "A é dependente de B para teste". Se possível B deve ser testado antes de A.

O mapeamento de alguns relacionamentos existentes num diagrama de classes (UML) para um grafo TDG, conforme as seguintes regras:

- Se B é herança de A, então B é dependente de A para teste, no TDG deverá existir uma seta de B para A nomeada de I (Figura 2.8a);

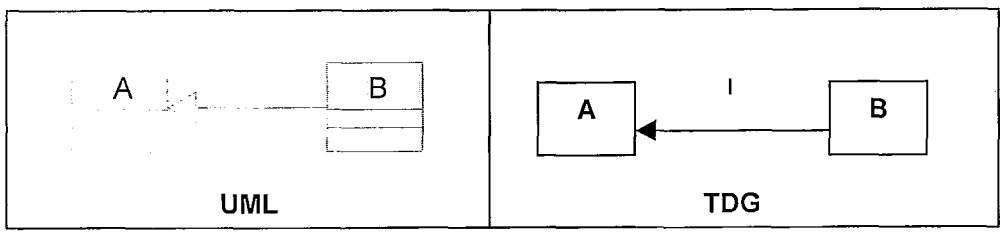


Figura 2.8a – Mapeando herança de UML para TDG.

- Se A é associada ou dependente de B, então A é dependente de B para teste, no TDG deverá existir uma seta de A para B nomeada As (Figura 2.8b); e

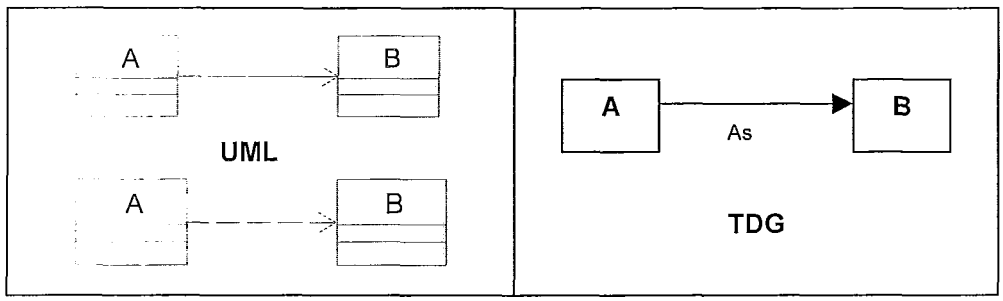


Figura 2.8b – Mapeando associação e dependência de UML para TDG.

- Se A é uma composição (ou agregação) de B, então A é dependente de B para teste, no TDG deverá existir uma seta de A para B nomeada de Ag (Figura 2.8c).

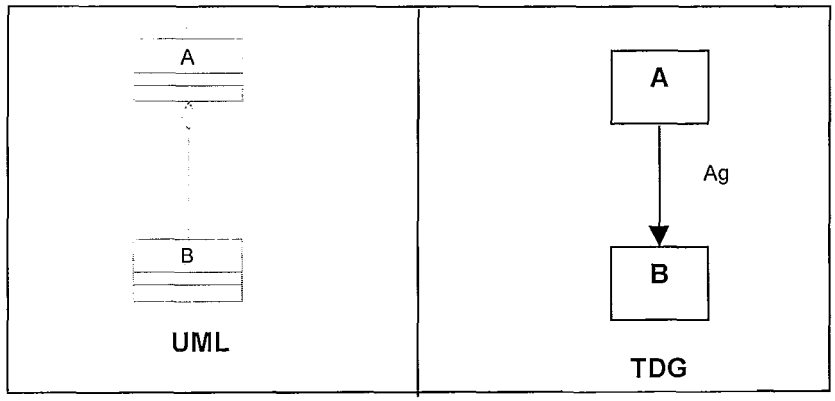


Figura 2.8c – Mapeando agregação de UML para TDG.

Como podemos observar, um ORD (utilizado nas propostas de KUNG *et al.* e TAI e DANIELS) é um tipo simplificado de TDG.

A estratégia de LE TRAON *et al.* apresentada está dividida em dois casos:

- Grafos sem ciclos de dependência – a estratégia seria começar os testes dos descendentes para os ancestrais no grafo; e
- Grafos com ciclos de dependência – a estratégia seria decompor o TDG em SCC (*Strongly Connected Component*), ou seja, um grafo em sub-grafos. A partir daí seria aplicado o algoritmo de Tarjan[1] recursivamente.

Para cada classe raiz de um SCC é calculado um peso baseado no número de dependências existentes (agregações, heranças e associações). As quebras das dependências podem ocorrer não exclusivamente em relacionamentos de associação.

[1] Algoritmo de Tarjan é baseado no algoritmo de busca em profundidade em grafos (árvores) (TARJAN, 1972).

A desvantagem desta proposta é a dependência do resultado do algoritmo de duas decisões arbitradas:

1. escolha da classe inicial (raiz) para começar a busca; e
2. escolha entre classes filhas com mesmo peso.

Portanto, trata-se de um algoritmo não determinístico. Assim sendo, a ordem obtida para integração das classes, bem como o correspondente número de *stubs*, depende das escolhas realizadas, podendo existir mais de um resultado para um mesmo diagrama de classes.

2.5.4 – Ordem de Teste de BRIAND, LABICHE e WANG.

Segundo BRIAND *et al.* (2001), esta proposta é semelhante à de LE TRAON *et al.* (2000). Esta estratégia está baseada em chamadas recursivas ao algoritmo de TARJAN (1972), entretanto emprega o conceito de peso usado por TAI e DANIELS (1997) modificado para caracterizar associações, calculando o número mínimo de ciclos nos quais a associação esteja envolvida num SCC.

A Figura 2.9 apresenta o algoritmo empregado na estratégia de quebra dos ciclos por BRIAND *et al.*, baseado na determinação do SCC.

```
procedure ClassLevel(OperatingGraph)
begin
    Determine the SCCs in the OperatingGraph.  // Tarjan's algorithm
    Order these SCCs (topological sorting).
    Assign level numbers to SCCs
    for each SCC do    // determine an order in each SCC
    begin
        if the SCC contains more than one class then
        begin
            Identify the edge with maximum weight in the SCC.
            NewGraph = the SCC without maximum weight edge.
            ClassLevel(NewGraph)      // recursive call
        end
    end
end
```

Figura 2.9 – Algoritmo de Quebra dos Ciclos (BRIAND *et al.*, 2001).

Desta forma, a característica não determinística no uso do algoritmo de TARJAN encontrada na estratégia de LE TRAON *et al.* (2000) desaparece na estratégia de BRIAND *et al.* Não há necessidade de escolha da classe inicial (o algoritmo é aplicado ao grafo que representa todo o diagrama de classes) e foi estabelecido um critério no caso de empate dos pesos em outros níveis: remover uma

e somente uma associação para quebrar a dependência (ao invés de remover todas as dependências como proposto por LE TRAON *et al.*).

2.5.5 – Ordem de Teste de OLIVEIRA e TRAVASSOS.

A proposta de OLIVEIRA e TRAVASSOS (2003) diferencia-se das anteriores por ser aplicada diretamente num diagrama de classes (UML), sendo guiada por duas propriedades: *Fator de Influência* (FI) e *Fator de Integração Tardia* (FIT).

FI indica o número de classes diretamente proporcional ao número de classes que precisam ser integradas após a classe em questão. FIT é obtido pela soma de todos os FI das classes que precisam ser integradas antes da classe em questão. A ordem das classes é estabelecida pela aplicação das regras:

(1) selecionar a classe com FIT zero; e

(2) recalcular os valores de FIT, reduzindo a influência da classe selecionada.

Aplicando as regras para o exemplo de OLIVEIRA e TRAVASSOS (2003) (Figura 2.10) seriam necessárias três iterações do cálculo de FIT (FIT_1, FIT_2 e FIT_3) para identificar a ordem de integração (Tabela 2.3), selecionando:

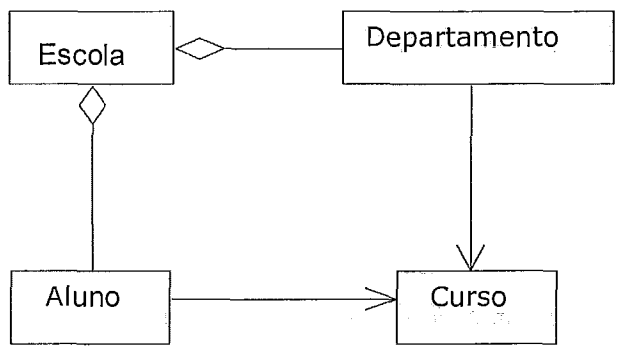


Figura 2.10 – Exemplo 3: Diagrama de Classes UML.

- a classe *Curso* (FIT_1 igual a zero) na primeira iteração.
- as classes *Departamento* e *Aluno* (FIT_2 igual a zero) na segunda iteração. O valor de FIT_2 destas classes é igual a FIT_1 menos FI de *Curso* (redução da influência de *Curso*).
- a classe *Escola* (FIT_3 igual a zero) na terceira iteração. O valor de FIT_3 da classe *Escola* é igual a FIT_2 menos FI de *Departamento* e *Aluno* (redução da influência das classes *Departamento* e *Aluno*).

Classe	FI	FIT_1	FIT_2	FIT_3
Escola	0	2	2	0
Departamento	1	2	0	
Aluno	1	2	0	
Curso	2	0		

Tabela 2.3 – FI e FIT para o Exemplo3.

O problema nesta proposta é não apresentar solução para diagramas que contêm ciclos de dependência entre classes, ou seja, para modelos onde não exista pelo menos uma classe com FIT nulo, por iteração.

2.5.6 – Analisando as Estratégias.

As quatro primeiras estratégias apresentadas anteriormente não são diretamente aplicadas sobre diagramas de classes da UML, artefato desenvolvido e utilizado durante outras fases do ciclo de desenvolvimento do software. Este fato reflete num esforço / custo adicional para engenheiros de software ou gerentes de projeto quando de sua aplicação na tarefa de identificação da ordem de integração e teste das classes: a transformação do diagrama de classes da UML para uma representação particular (ORD ou TDG) para aplicação da estratégia / ferramenta.

As estratégias de KUNG *et al.* (1995a), devido ao emprego de decisão aleatória para a quebra de ciclos de dependências com mais de uma associação candidata; TAI e DANIELS (1999), por permitir quebras onde não existam ciclos de dependências; e OLIVEIRA e TRAVASSOS (2003), por não apresentarem solução para ciclos de dependências, deixam de representar uma solução ampla e aplicável a qualquer diagrama de classe.

As estratégias de LE TRAON *et al.* (2000) e BRIAND *et al.* (2001) empregam o mesmo processo para identificar SCC, apresentando a mesma desvantagem da complexidade de tempo (não linear). Entretanto, a estratégia de LE TRAON *et al.* ainda apresenta outra desvantagem de não tratar os pontos onde podem ocorrer decisões arbitrárias: escolha da raiz inicial e escolha entre classes filhas com mesmo peso, conforme pode ser observado nos resultados apresentados na Tabela 2.4 com a aplicação das estratégias para o Exemplo 2 da Figura 2.6.

	TAI e DANIELS	LE TRAON *et al.*				BRIAND *et al.*
		Raiz H			Raiz G	
		Raiz A	Raiz E	Raiz F		
Classes que precisam de *stubs*	C, F, H, D, B	H, A, F	H, E, A	H, F, A	B, C, F	C, F, H, B
No. classes	5	3	3	3	3	4
Stubs	C, F, H, D, B	3 de H, 3 de A, 1 de F	3 de H, 2 de E, 1 de A	3 de H, 1 de F, 2 de A	1 de B, 2 de C, 1 de F	C, F, H, B
Esforço de Teste	5	7	6	6	4	4

Tabela 2.4 – Exemplo 2: Resultados das estratégias (BRIAND *et al.*, 2001).

2.6 – Considerações Finais

Neste capítulo foi discutido o teste de integração para software OO e foram apresentados, também, alguns complicadores na utilização das abordagens conhecidas para o paradigma procedural no contexto do processo de desenvolvimento OO. Um dos mais importantes é o ciclo de dependências entre componentes e que envolve a necessidade de criação de *stubs* para simular os serviços ainda não integrados durante o processo do teste de integração.

Dentro do novo enfoque de teste de integração para software OO, foram apresentadas algumas estratégias existentes com o propósito de identificar a ordem de integração dos componentes. Estas estratégias buscam a minimização do esforço de teste por meio da minimização do número de *stubs* necessários a serem criados.

Entretanto, com exceção da estratégia proposta por OLIVEIRA e TRAVASSOS (2003) que usa diagrama de classes UML, as demais estratégias partem da necessidade de novos artefatos de projeto – ORD ou TDG – que são grafos de dependências entre os objetos, representado um custo adicional para sua criação e manutenção.

A possibilidade de empregar uma estratégia diretamente aplicada num alto nível de abstração de um projeto de software OO norteou este trabalho na busca de um tratamento para quebra dos ciclos de dependências, complementando a proposta inicial de OLIVEIRA e TRAVASSOS e ampliando sua aplicação a qualquer modelo. Como resultado serão apresentados, no Capítulo 3: a formalização dos critérios de precedências entre classes; e a definição de um processo de aplicação das heurísticas

que permite identificar uma ordem de integração com menor esforço de teste, além da redução dos custos adicionais com novos artefatos.

Capítulo 3
Heurísticas para Ordenação de Classes em Testes de Integração Aplicados a Software Orientado a Objetos

Neste capítulo é proposta uma estratégia para identificar a ordem de integração e teste das classes em um projeto de software orientado a objetos, sendo apresentado um processo para controlar a aplicação das heurísticas formalizadas.

3.1 – Introdução

Uma das considerações a respeito da identificação da ordem de integração e testes em um projeto de software é como minimizar o esforço de teste, medido pelo número de *stubs* necessários para a execução dos testes de integração (KUNG *et al.*, 1995) (HANH *et al.*, 2001). Esta característica é compartilhada pelos diversos tipos de software, entretanto, quando consideramos a arquitetura orientada a objetos (OO), a dificuldade está concentrada, principalmente, nas diferentes relações (dependências) entre os diversos componentes (representados pelas classes) (GRAHAM, 1993) (BROOKS *et al.*, 1995) (KUNG *et al.*, 1995b) (VIEIRA, 1998) que pode acrescer complexidade no planejamento e na execução dos testes de integração (discutida no Capítulo 2). Portanto, uma consideração bastante importante para a execução dos testes de integração aplicados a software OO passa a ser a ordem de integração das classes que implica diretamente na necessidade de implementação dos *stubs*, determinando conseqüentemente o esforço de teste.

Conforme apresentado no Capítulo 2, algumas abordagens para ordenar classes para teste de integração OO foram desenvolvidas (TAI e DANIELS, 1997) (TRAON *et al.*, 2000) (BRIAND *et al.*, 2001). Entretanto, estas abordagens não propõem soluções diretamente aplicadas a representações usuais de projeto de software (abstração de alto nível), como, por exemplo, o diagrama de classes UML (*Unified Modeling Language,* em inglês), o que requer um custo extra na construção de outros diagramas

adicionais (por exemplo: ORD – *Object Relation Order* ou TDG – *Test Dependency Graph*).

Baseado neste contexto, será formalizado na Seção 3.2 o conjunto de heurísticas aplicadas diretamente ao modelo de classes de um projeto de software OO – diagrama de classes UML – para identificação da ordem das classes durante a execução dos testes de integração. Estas heurísticas são resultantes da evolução da proposta inicial da Ordem de Teste de OLIVEIRA e TRAVASSOS (TRAVASSOS *et al.*, 1995) (OLIVEIRA e TRAVASSOS, 2003), apresentada no Capítulo 2, e foram aperfeiçoadas durante os diversos estudos efetuados em modelos de projetos para analisar e tratar os ciclos de dependências entre classes e a efetividade obtida. Na Seção 3.3 são discutidas as limitações da proposta inicial, sendo identificadas algumas situações especiais não definidas. O processo de aplicação das heurísticas proposto, que resulta em esforço de teste mínimo, contemplando o tratamento para as situações especiais, está descrito na Seção 3.4. As considerações finais deste capítulo estão na Seção 3.5.

3.2 – Heurísticas para identificar a ordem de integração de classes

A UML é uma abordagem de notação, muito utilizada para descrever soluções orientadas a objetos, podendo ser utilizada para visualizar, especificar ou documentar os artefatos do projeto (PFLEGEER, 2004). Os diagramas da UML capturam as duas visões do software: a visão estática e a visão dinâmica, além das restrições e da formalização. 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) e a extensibilidade (restrições, valores identificados por *tags* e estereótipos) (HARMON e WATSON, 1997) (McGREGOR e SYKES, 2001).

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, descritos a seguir.

3.2.1 – Critérios de Precedência

3.2.1.1 – Herança

A herança é a situação na qual as subclasses herdam as características, atributos e comportamento (métodos) de outra classe, referida como superclasse (BOOCH, 1994) (HARMON e WATSON, 1997).

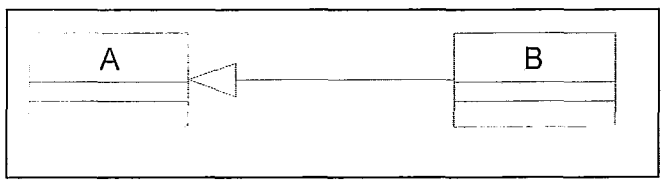


Figura 3.1 – Exemplo de Herança Concreta em UML.

Critério de precedência 1: no caso de superclasses concretas, deve-se testar primeiro as superclasses, para garantir a correta implementação de suas características. No exemplo da Figura 3.1, segundo o critério de precedência estabelecido, a superclasse A deve ser testada antes da subclasse B, ou seja, a classe A tem precedência sobre a classe B.

Classes abstratas são escritas com a expectativa de que suas subclasses podem adicionar atributos ou subscrever métodos (BOOCH, 1994). Portanto, uma operação definida na classe abstrata pode ter comportamento diferente numa subclasse (JACOBSON, 1992).

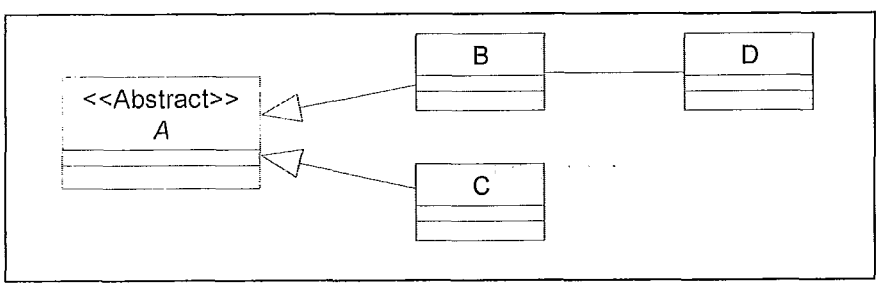


Figura 3.2 – Exemplo de Herança Abstrata em UML.

Critério de precedência 2: no caso da superclasse ser abstrata, deve-se testar primeiro a subclasse com menor dependência. Esta dependência é medida pelo número de relacionamento de todas as subclasses. No exemplo da Figura 3.2, segundo o critério de precedência estabelecido, a subclasse C (não possui relacionamentos) deve ser testada antes da subclasse B (possui um relacionamento), ou seja, a classe C tem precedência sobre a classe B.

3.2.1.2 – Assinatura de Método da Classe

Se uma classe cliente usa serviços de uma classe servidora, é necessário avaliar primeiro se os serviços foram corretamente implementados.

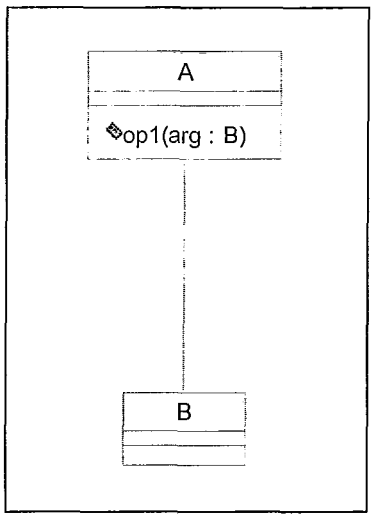


Figura 3.3 – Exemplo de Assinatura de método da Classe em UML.

Critério de precedência 3: a classe servidora (que possui o método a ser assinado) deve ser testada antes da classe cliente (que assinou o método). No exemplo da Figura 3.3, a classe servidora B deve ser testada antes da classe cliente A, ou seja, a classe B tem precedência sobre a classe A.

3.2.1.3 – Agregação

Uma agregação refere-se a um relacionamento *parte-todo* (HARMON e WATSON, 1997), sugerindo que algumas classes são contidas por outra classe. As agregações simples e compostas são consideradas no mesmo nível de abstração: a classe *todo* depende dos serviços fornecidos pelas *partes*.

Critério de precedência 4: as classes *parte* devem ser testadas antes da classe *todo*. No exemplo da Figura 3.4, a classe *parte* B deve ser testada antes da classe *todo* A, ou seja, a classe B tem precedência sobre a classe A.

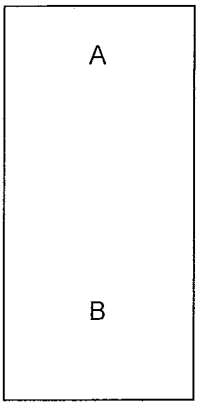


Figura 3.4 – Exemplo de Agregação em UML.

3.2.1.4 – Navegabilidade

A navegabilidade indica que uma classe torna-se atributo de outra, indicada pela direção da seta, indicando uma dependência. A navegabilidade pode ser usada para indicar qual classe deve ser testada primeira.

Critério de precedência 5: a classe que tornou-se atributo deve ser testada primeiro. No exemplo da Figura 3.5, a classe B deve ser testada antes da classe A, ou seja, a classe B tem precedência sobre a classe A.

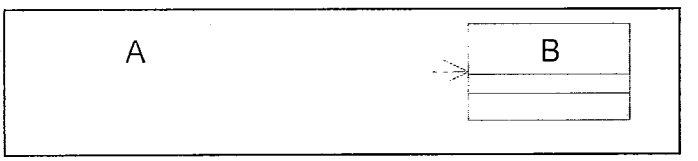


Figura 3.5 – Exemplo de Navegabilidade em UML.

A associação denota dependência mútua, onde as duas classes colaboram para realizar um serviço. Uma associação é naturalmente considerada com navegabilidade bidirecional (BOOCH, 1994) (HARMON e WATSON, 1997). Em outras palavras, uma associação implica que um objeto de uma classe envia mensagens para operações associadas com objetos da segunda classe e vice-versa.

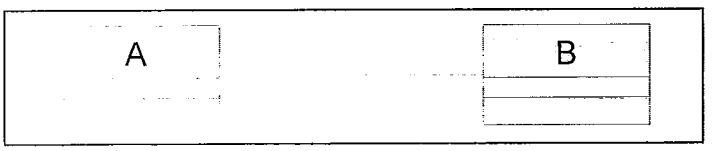


Figura 3.6 – Exemplo de Associação em UML.

Critério de precedência 6: na associação, a navegabilidade deverá ser analisada nos dois sentidos (Figura 3.6), ou seja, o critério de precedência da navegabilidade deverá ser considerado a combinação de duas navegabilidades simples: de uma

classe para outra e também no sentido da outra para a primeira. Quando a classe a ser analisada for a classe A, como mostrado na Figura 3.7, a classe B terá precedência. Quando a classe analisada for a classe B, como mostrado na Figura 3.8, a classe A terá precedência.

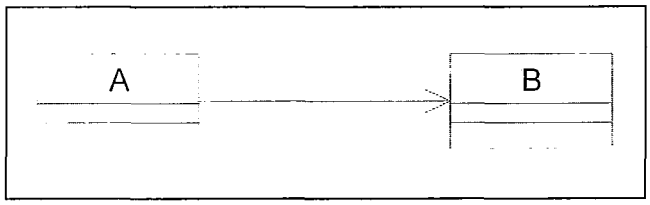


Figura 3.7 – Precedência da classe B sobre a classe A.

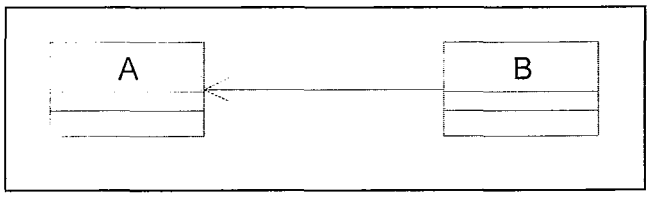


Figura 3.8 – Precedência da classe A sobre a classe B.

3.2.1.5 – Classe Associação

Algumas vezes uma associação pode ser vista como um objeto, com seus próprios atributos e operações, sendo mais usual quando ocorre uma associação entre muitos objetos. Os atributos e operações da classe associação são características da própria associação (HARMON e WATSON, 1997).

Critério de precedência 7: as classes que deram origem à classe associação devem ser testadas primeiro. No exemplo da Figura 3.9, as classes A e C devem ser testadas antes da classe B, ou seja, as classes A e C têm precedência sobre a classe B.

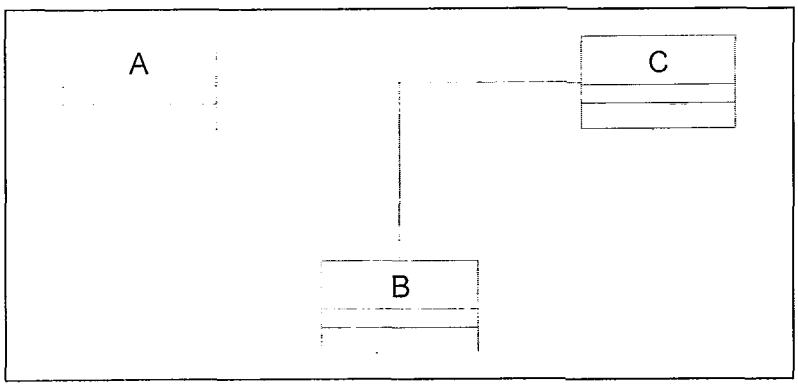


Figura 3.9 – Exemplo de Classe Associação em UML.

3.2.1.6 – Dependência

A dependência indica a ocorrência de um relacionamento semântico entre dois ou mais elementos do modelo. Uma classe consumidora é dependente de alguns serviços da classe fornecedora, não havendo, entretanto, uma dependência estrutural interna com a classe fornecedora.

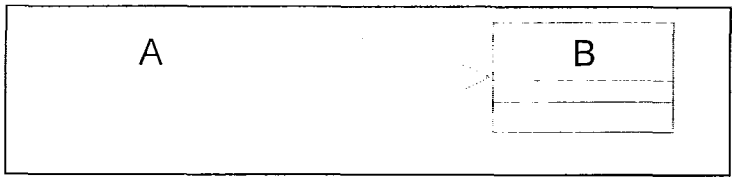


Figura 3.10 – Exemplo de Dependência em UML.

Critério de precedência 8: a classe fornecedora num relacionamento de dependência deve ser testada antes da classe consumidora. No caso do exemplo da Figura 3.10, a classe fornecedora B deve ser testada antes da classe consumidora A, ou seja, a classe fornecedora B tem precedência sobre a classe consumidora A.

3.2.1.7 – Cardinalidade

A cardinalidade (ou multiplicidade) indica quantos objetos de uma classe podem se relacionar com outro objeto de outra classe (HARMON e WATSON, 1997). Em geral a multiplicidade indica os limites, inferior e superior, de participação de um objeto. As multiplicidades mais comuns são: 1 (exatamente um); 0..1 (opcional); 1..N (um-para-muitos); dentre outras. A Figura 3.11 mostra algumas notações empregadas para representar multiplicidade de associações.

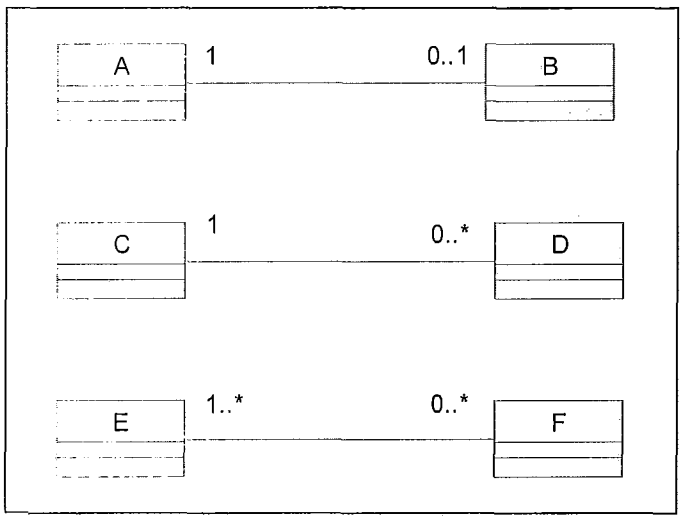


Figura 3.11 – Notações para multiplicidade de associação.

Critério de precedência 9: num relacionamento de associação com representação de cardinalidade, a classe com cardinalidade não opcional (um ou um-para-muitos) deve ser testada antes de outra classe com cardinalidade opcional (zero ou zero-para-muitos). Na Figura 3.11, as classes A, C e E devem ser testadas antes das classes B, D e F, ou seja, as classes A, C e E têm precedência sobre as classes B, D e F.

3.2.2 – Fatores de Influência e Integração Tardia

As propriedades Fator de Influência (FI) e Fator de Integração Tardia (FIT) são redefinidas a partir do conhecimento dos *critérios de precedências* entre classes para auxiliar a estabelecer a ordem entre as diversas classes de um diagrama de classes UML e apresentadas a seguir.

3.2.2.1 – Fator de Influência (FI)

O Fator de Influência (FI) é utilizado para quantificar a relação de precedência entre as classes, representando o número de classes que precisam ser testadas depois da classe em análise ser testada, ou seja, o número de classes sobre as quais a classe em análise tem precedência. Deve ser definido considerando apenas as precedências diretas da classe em análise.

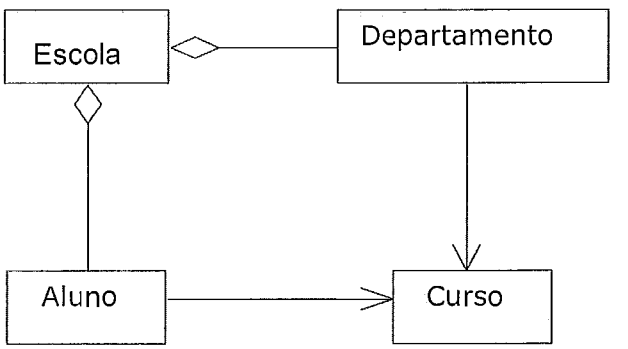


Figura 3.12 – Exemplo 1: Diagrama de Classes UML.

Os valores de FI, apresentados na Tabela 3.1, foram calculados para as classes do diagrama apresentado na Figura 3.12 e estabelecidos a partir da seguinte análise:

- a classe *Escola* não tem precedência sobre nenhuma outra classe;
- a classe *Departamento* tem precedência sobre uma classe, *Escola* – critério da agregação;

- a classe *Aluno* tem precedência sobre uma classe, *Escola* – critério da agregação; e
- a classe *Curso* tem precedência sobre duas classes: *Departamento* e *Aluno* – critério da navegabilidade.

Classe	FI
Escola	0
Departamento	1
Aluno	1
Curso	2

Tabela 3.1 – Fatores de Influência para o Exemplo1.

3.2.2.2 – Fator de Integração Tardia (FIT)

O Fator de Integração Tardia (FIT) tem a intenção de capturar a idéia do tempo de integração das classes. O FIT indica o momento em que a classe deve ser considerada para integração e teste, em relação às demais.

O FIT é calculado pela soma dos valores dos Fatores de Influência (FI) das classes com precedência direta sobre a classe em análise. Quanto maior o valor de FIT, mais tarde a classe deverá ser testada, pois maior será a dependência desta classe em relação às demais.

Os valores de FIT, apresentados na Tabela 3.2, foram calculados para as classes do diagrama apresentado na Figura 3.12 e estabelecidos a partir da seguinte análise:

- as classes *Departamento* (FI = 1) e *Aluno* (FI = 1) têm precedência sobre a classe *Escola*;
- a classe *Curso* (FI = 2) tem precedência sobre a classe *Departamento*;
- a classe *Curso* (FI = 2) tem precedência sobre a classe *Aluno*; e
- nenhuma classe tem precedência sobre a classe *Curso*.

Classe	FIT
Escola	2
Departamento	2
Aluno	2
Curso	0

Tabela 3.2 – FIT (1ª. Iteração) para o Exemplo1.

3.3 – Processo Inicial das Heurísticas e suas Limitações

3.3.1 – Processo Inicial de OLVEIRA e TRAVASSOS

A proposta inicial de Ordem de Teste de OLIVEIRA e TRAVASSOS, apresentada no Capítulo 2, está baseada no Fator de Integração Tardia. Quanto maior o FIT, mais tarde a classe deve ser integrada. A ordem das classes é estabelecida por meio da aplicação de duas regras:

(1) selecionar a classe com FIT zero; e
(2) recalcular os valores de FIT, reduzindo a influência da classe selecionada.

Como previsto por OLIVEIRA e TRAVASSOS (2003), este processo é eficiente para modelos onde exista pelo menos uma classe com FIT nulo por iteração, o que significa a ausência de forte acoplamento entre as classes. Entretanto, os ciclos de dependência são comuns em software reais (TRAON *et al.*, 2000) (BRIAND *et al.*, 2001).

O diagrama de classes UML apresentado na Figura 3.13 contém ciclos de dependências entre classes e será empregado para mostrar as limitações da proposta inicial de OLIVEIRA e TRAVASSOS (2003). Este diagrama é equivalente ao Exemplo 2 de ORD (Figura 2.6) do Capítulo 2, sendo obtido por meio do mapeamento entre ORD e diagramas de classes UML, proposto por BRIAND *et al.* (2001).

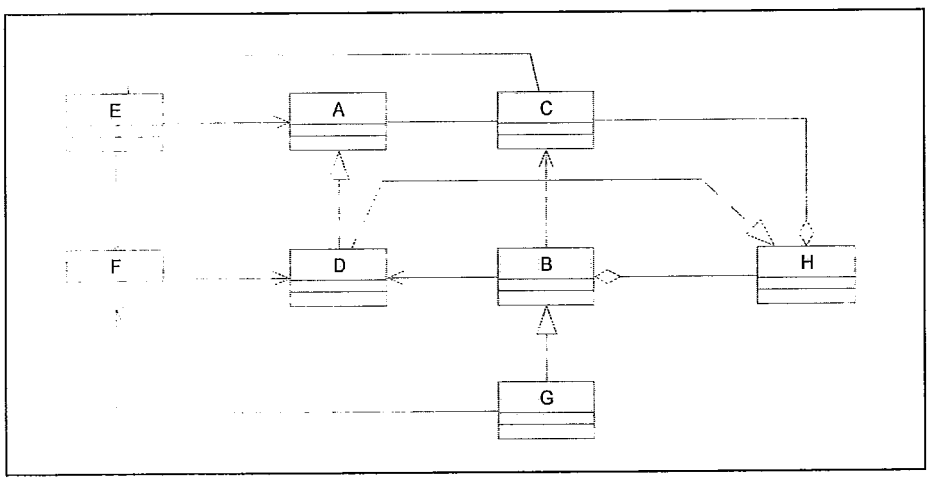


Figura 3.13 – Exemplo 2: Diagrama de Classes UML.

3.3.2 – Identificando Situações Especiais

Na Tabela 3.3 são apresentados os fatores de influência (FI) para as classes do Exemplo 2 (Figura 3.13). Os valores de FI calculados são baseados na seguinte análise do diagrama de classes:

- a classe A tem precedência sobre as classes: C (critério da navegabilidade), D (critério da herança) e E (critério da navegabilidade);
- a classe B tem precedência sobre a classe G (critério da herança);
- a classe C tem precedência sobre as classes: A (critério da navegabilidade), B (critério da navegabilidade) e H (critério da agregação);
- a classe D tem precedência sobre as classes: B e F (critério da navegabilidade, em ambas);
- a classe E tem precedência sobre as classes: C (critério da navegabilidade) e F (critério da agregação);
- a classe F tem precedência sobre a classe G (critério da navegabilidade);
- a classe G não tem precedência sobre nenhuma classe; e
- a classe H tem precedência sobre as classes: B (critério da agregação) e D (critério da herança).

Classe	FI
A	3
B	1
C	3
D	2
E	2
F	1
G	0
H	2

Tabela 3.3 – Fatores de Influência para o Exemplo 2.

Classe	FI	FIT_1
A	3	3
B	1	7
C	3	5
D	2	5
E	2	3
F	1	4
G	0	**2**
H	2	3

Tabela 3.4 – Fatores de Integração Tardia para o Exemplo 2.

Na Tabela 3.4 são apresentados os fatores de integração tardia (FIT) para as classes do Exemplo 2 (Figura 3.13). Os valores de FIT calculados são baseados na seguinte análise do diagrama de classes:

- a classe C (FI = 3) tem precedência sobre a classe A;
- as classes C (FI = 3), D (FI = 2) e H (FI = 2) têm precedência sobre a classe B;
- as classes A (FI = 3) e E (FI = 2) têm precedência sobre a classe C;
- as classes A (FI = 3) e H (FI = 2) têm precedência sobre a classe D;
- a classe A (FI = 3) tem precedência sobre a classe E;
- as classes E (FI = 2) e D (FI = 2) têm precedência sobre a classe F;
- as classes D (FI = 1) e F (FI = 1) têm precedência sobre a classe G; e
- a classe C (FI = 3) tem precedência sobre a classe H.

A primeira classe selecionada é a classe G, com menor FIT ($FIT_1 = 2$) e como esta não tem precedência sobre as demais, então, os valores de FIT_2 (segunda iteração) são iguais aos de FIT_1 (primeira iteração), como apresentado na Tabela 3.5.

Classe	FI	FIT_1	FIT_2
A	3	3	3
B	1	7	7
C	3	5	5
D	2	5	5
E	2	3	3
F	1	4	4
G	0	2	-
H	2	3	3

Tabela 3.5 – Valores de FIT: 2ª. iteração para o Exemplo 2.

Como a definição inicial não trata a situação de mais de uma classe atender ao critério de menor FIT, existem na 2ª. iteração três opções de escolha: a classe A ($FIT_2 = 3$), a classe E ($FIT_2 = 3$) ou a classe H ($FIT_2 = 3$). As Tabelas 3.6, 3.7 e 3.8 apresentam os cálculos para cada uma das possíveis escolhas separadamente.

Classe	FI	FIT_1	FIT_2	FIT_3	FIT_4	FIT_5	FIT_5	FIT_5
A	3	3	**3**	-	-	-	-	-
B	1	7	7	7	7	4	2	**0**
C	3	5	5	2	**0**	-	-	-
D	2	5	5	2	2	2	**0**	-
E	2	3	3	**1**	-	-	-	-
F	1	4	4	4	2	2	2	**0**
G	0	**2**	-	-	-	-	-	-
H	2	3	3	3	3	**0**	-	-

Tabela 3.6 – Valores de FIT: escolha de A na 2ª. iteração .

Classe	FI	FIT_1	FIT_2	FIT_3	FIT_4	FIT_5	FIT_5
A	3	3	3	3	**0**	-	-
B	1	7	7	7	4	2	**0**
C	3	5	5	**0**	-	-	-
D	2	5	5	2	2	**0**	-
E	2	3	**3**	-	-	-	-
F	1	4	4	2	2	2	**0**
G	0	**2**	-	-	-	-	-
H	2	3	3	3	**0**	-	-

Tabela 3.7 – Valores de FIT: escolha de E na 2ª. iteração .

Classe	FI	FIT_1	FIT_2	FIT_3
A	3	3	3	**3**
B	1	7	7	5
C	3	5	5	5
D	2	5	5	**3**
E	2	3	3	**3**
F	1	4	4	4
G	0	**2**	-	-
H	2	3	**3**	-

Tabela 3.8 – Valores de FIT: escolha de H na 2ª. iteração .

Como pode ser observado na Tabela 3.8, na 3ª. iteração, quando a escolha foi pela classe H na 2ª. iteração, ocorre outro ponto de incerteza deste processo. A Tabela 3.9 apresenta todos os resultados possíveis usando o processo inicial das heurísticas.

Ordem de Integração	Esforço de Teste
G, A, E, C, H, D, B, F	3
G, E, C, A, H, D, B, F	4
G, H, A, D, E, C, F, B	4
G, H, D, F, A, E, C, B	6
G, H, D, F, B, A, E, C	7
G, H, D, F, B, E, A, C	8
G, H, D, F, E, A, B, C	8

Tabela 3.9 – Exemplo 2 – Possíveis Resultados do Processo Inicial.

3.3.3 – Tratamento das Situações Especiais

Baseado na análise da aplicação do processo inicial de aplicação das heurísticas, bem como das soluções encontradas com este processo (Tabelas 3.6, 3.7 e 3.8) e dos correspondentes esforços de teste calculados, conforme apresentados na Tabela 3.9, é possível apontar:

- que a falta de definição de um critério para tratamento de FIT iguais numa mesma iteração não define uma solução única (ordem das classes), ou seja, existem diversas soluções com esforço de teste diferentes, sendo algumas maiores do que 4 (quatro), esforço de teste estabelecido pela estratégia de BRIAND (2001) para o Exemplo 2;
- que iniciar os testes por classes sem precedências sobre outras implicará em um esforço de teste no mínimo igual ao número de classes das quais ela for dependente. Para o Exemplo 2, iniciar os testes de integração pela classe G implica em construir, no mínimo, dois *stubs*: um para a classe F e outro para a classe B.

Com o propósito de permitir a obtenção de uma solução única para identificar a ordem de integração das classes que demande um esforço de teste mínimo, os tratamentos descritos a seguir foram estabelecidos para apoiar as situações especiais identificadas.

3.3.3.1 – Análise do Fator de Influência Nulo

O valor de FI nulo é bastante significativo, pois representa uma classe de um modelo sem precedência sobre as demais. Estas classes são referenciadas como Classes Dependentes.

A fim de reduzir o esforço de teste de um modelo, as classes dependentes não devem ser selecionadas para teste de integração antes que as demais classes já tenham sido testadas. Não é preciso calcular valores de FIT para estas, as mesmas devem ser excluídas temporariamente do processo.

Desta forma, para o Exemplo 2, a classe G é uma classe dependente, pois possui FI nulo, como apresentado na Tabela 3.3. A classe G deve ser excluída do processo. A Tabela 3.10 apresenta as classes e os valores de FIT a serem considerados no restante do processo.

Classe	FI	FIT_1
A	3	3
B	1	7
C	3	5
D	2	5
E	2	3
F	1	4
H	2	3

Tabela 3.10 – Exemplo 2: análise do FI nulo.

3.3.3.2 – Iterações sem Fator de Integração Tardia Nulo

A idéia básica de OLIVEIRA e TRAVASSOS (2003) é selecionar, a cada iteração de cálculo de FIT, a(s) classe(s) com FIT nulo. Entretanto, pode ocorrer em algumas iterações que não exista classe com valor nulo de FIT a ser selecionada. Esta situação representa classes com forte ligação com outras classes. Nestas iterações, deve ser selecionada a(s) classe(s) que possuir(em) o menor valor calculado de FIT.

Para o Exemplo 2, não existe classe com valor de FIT_1 nulo. Então, as classes selecionadas devem ter o menor FIT_1 calculado ($FIT_1 = 3$), ou seja, as classes: A, E e H, como apresentado na Tabela 3.10.

3.3.3.3 – Tratamento de *Deadlock* (Classes com mesmo FIT)

Estas situações representam ciclos de dependência, apresentados no Capítulo 2, entre as classes com o mesmo valor de FIT (*deadlock*). Nestes casos, a quebra da dependência implicará na implementação de um ou mais *stubs* para as classes destino daquela escolhida para dar continuidade ao teste de integração.

Todas as classes com menor valor de FIT, numa mesma iteração, deverão ser selecionadas simultaneamente. Entretanto, a ordem a ser estabelecida entre estas classes deve considerar os critérios abaixo estabelecidos na ordem apresentada.

- Primeiro Critério. As classes (com mesmo valor de FIT) devem ser selecionadas em ordem crescente do número de *stubs* específicos necessários para a correspondente integração.
- Segundo Critério. Se existir alguma classe, dentre estas, que contribua para reduzir o número de *stubs* específicos para integrar e testar alguma das outras, indicando uma dependência interna no ciclo, esta deve ser escolhida.
- Terceiro Critério. As classes (com mesmo valor de FIT e com mesmo número de *stubs* específicos) devem ser selecionadas em ordem crescente de tamanho, medido pela soma do número de atributos e de métodos (LORENZ e KIDD, 1994).
- Quarto Critério. No caso de não ser possível ordenar as classes com os critérios anteriores, é preciso verificar se existe alguma associação com multiplicidade obrigatória entre as classes e selecioná-la primeiro.

Na primeira iteração de FIT, apresentado na Tabela 3.10, as classes: A, E e H apresentam o menor valor de FIT, igual a três, indicando que as mesmas devem ser selecionadas simultaneamente para iniciar o processo de integração e teste. A identificação da ordem entre as três classes é feita utilizando-se o primeiro e segundo critérios para tratamento do *deadlock*. Pelo primeiro critério, todas as classes selecionadas precisam de um *stub* para ser testada: a classe A precisa de um *stub* específico de C; a classe E precisa de um *stub* específico de A; e a classe H precisa de um *stub* específico de C. Pelo segundo critério, a classe E não precisa de *stub* se a classe A for testada primeiro.

3.4 – Processo de Aplicação das Heurísticas

O processo de aplicação das heurísticas definido a seguir foi elaborado usando a notação de modelagem de processos proposta por VILLELA (2004) e descrita no Anexo A.

O processo de aplicação das heurísticas é utilizado para gerar a lista de classes ordenadas para teste de integração (LCOTI) de um modelo, representado pelo diagrama de classes UML, como orientação da ordem necessária para execução dos

testes de integração destes componentes, num paradigma orientado a objetos, com o propósito de obter um esforço do teste mínimo.

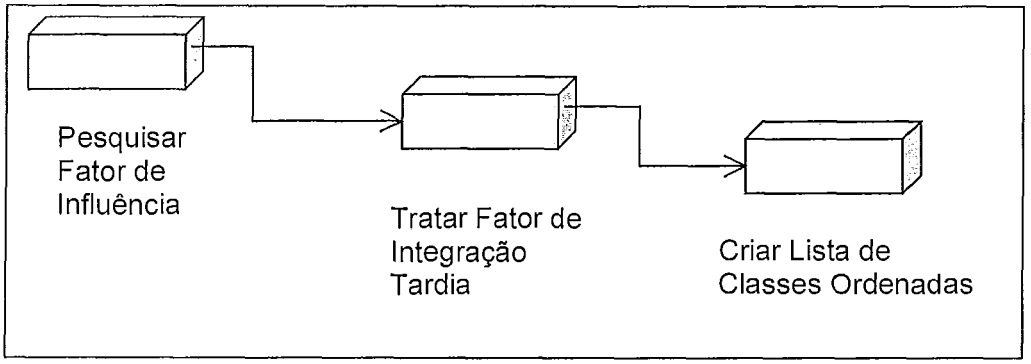


Figura 3.14 – Processo de Aplicação das Heurísticas.

A seguir são listadas as atividades e sub-atividades do processo de aplicação das heurísticas proposto, que é composto pelos processos menores: *Pesquisar Fator de Influência*; *Tratar Fator de Integração Tardia*; e *Criar Lista de Classes Ordenadas*, conforme apresentado na Figura 3.14.

3.4.1 – Pesquisar Fator de Influência

O propósito deste processo 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.

A Figura 3.15 apresenta o modelo deste processo, composto pelas seguintes atividades:

(1) Aplicar critérios de precedência. Identificar as precedências sobre a classe em análise, baseado nos critérios de precedência. Esta atividade deverá ser executada para todas as classes do modelo.

(2) Atribuir FI. Associar o número de classes com precedência sobre a classe em análise ao valor do fator de influência da classe. Esta atividade deverá ser executada para todas as classes do modelo.

(3) Separar classes dependentes. Identificar e separar as classes com fator de influência nulo (FI=0), gerando as listas de classes dependentes (LCD). A lista de classes não ordenadas (LCNO) é criada contendo as demais classes do modelo.

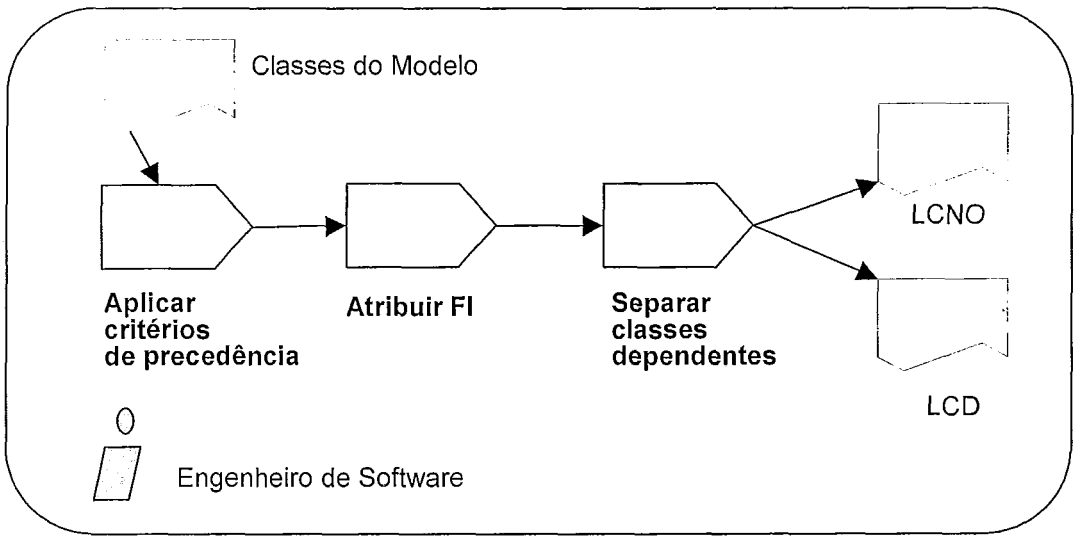


Figura 3.15 – Processo Pesquisar Fator de Influência.

3.4.2 – Tratar Fator de Integração Tardia

O propósito deste processo é ordenar as classes da lista LCNO. A cada iteração, classes com FIT nulo ou menor FIT são selecionadas, como apresentado na Figura 3.16.

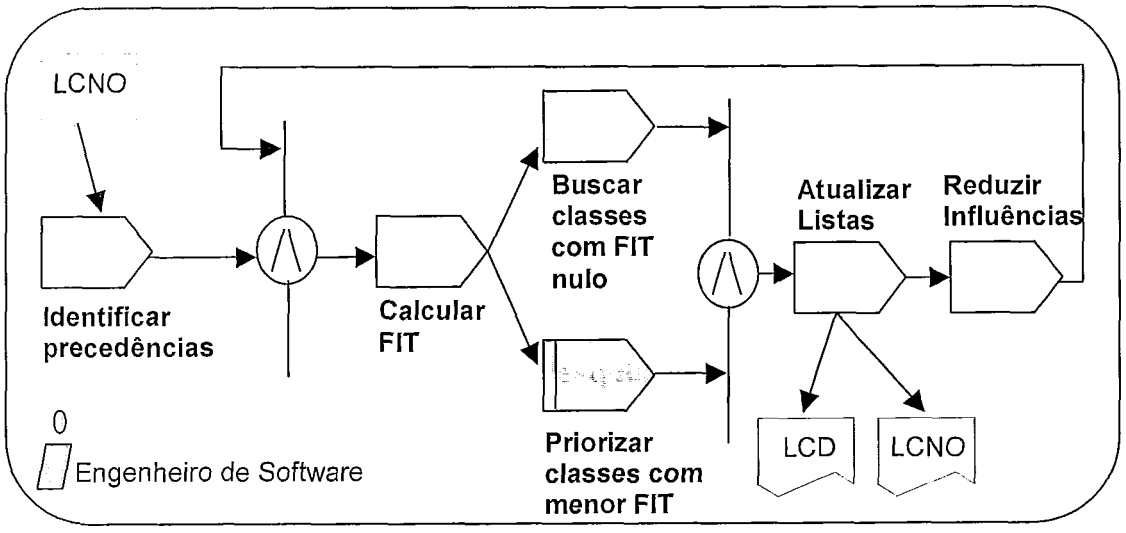


Figura 3.16 – Processo Tratar Fator de Integração Tardia.

O processo é composto pelas atividades:

(1) Identificar precedências: identificar classes que tenham precedência sobre a classe em análise. Esta atividade deverá ser executada para todas as classes do modelo.

(2) Calcular FIT: Somar o valor do FI das classes com precedência sobre a classe em análise. Esta atividade deverá ser executada para todas as

classes do modelo na primeira iteração (FIT1). A cada iteração *i* de FIT*i* serão retiradas as classes selecionadas na iteração anterior (*i* – 1).

(3) Buscar classes com FIT nulo: selecionar as classes com FIT*i* nulo para a LCOTI, a cada iteração *i*.

(4) Priorizar classes com menor FIT: se não existirem classes com FIT*i* nulo, selecionar classes com menor FIT*i* e ordenar classes usando o tratamento de *deadlock*. Esta atividade é composta pelas sub-atividades descritas abaixo e representadas na Figura 3.17:

- Verificar *stubs* necessários: ordenar as classes pelo número de *stubs* específicos. A seleção começa pelas classes com menor número de *stubs*.
- Identificar dependências internas: para classes com mesmo número de *stubs*, testar primeiro aquelas classes que ao serem testadas reduzem o número de *stubs* das outras.
- Medir tamanho dos *stubs*: para classes com mesmo número de *stubs*, selecionar por ordem crescente de tamanho (somar número de atributos mais número de métodos).
- Verificar a cardinalidade das associações: caso exista alguma com opcionalidade, utilizar como critério de desempate no caso de permanecer indefinido pelos critérios anteriores. Caso não seja possível, estabelecer uma ordem aleatória.

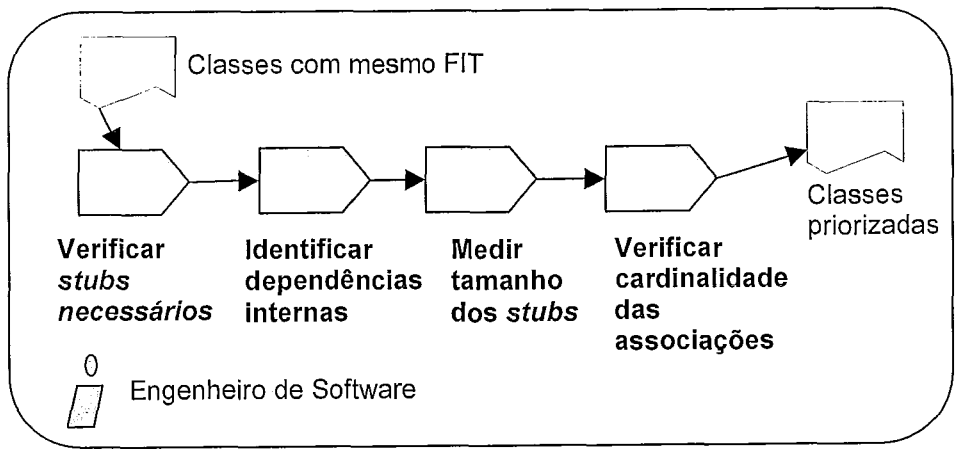


Figura 3.17. Atividade "Priorizar classes com menor FIT" detalhada.

(5) Atualizar listas LCNTI e LCNO: incluir as classes selecionadas na atividade 3 ou 4 na lista ordenada de classes para o teste de integração (LCOTI) e retirá-las da LCNO.

(6) Reduzir influências: a cada iteração novos valores de FIT$_{i+1}$ são recalculados para as classes da LCNO. O novo valor será obtido subtraindo do valor de FIT$_i$ atual do valor de FI das classes selecionadas, se elas tiverem precedência sobre a classe em análise.

3.4.3 – Criar Lista de Classes Ordenadas

O propósito deste processo é 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 3.18.

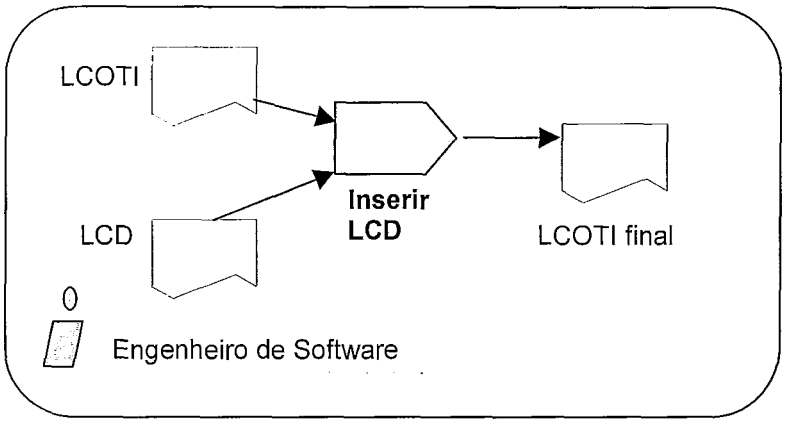


Figura 3.18 – Processo Criar Lista Ordenada.

Aplicando estes processos ao diagrama de classes do Exemplo 2 (Figura 3.13) teremos os seguintes resultados:

- do processo *Pesquisar Fator de Influência:* a lista de classes não ordenadas LCNO igual a {A, B, C, D, E, F, H,}; os valores de *fatores de influência* calculados e apresentados na Tabela 3.11; e a lista de classes dependentes LCD igual a {G}, devido ao FI nulo.

Classes	FI	FIT$_1$	FIT$_2$	FIT$_3$
A	3	**3**		
B	1	7	5	**0**
C	3	5	**0**	
D	2	5	**0**	
E	2	**3**		
F	1	4	2	**0**
H	2	**3**		

Tabela 3.11 – Exemplo 2: análise do FI nulo.

- do processo *Tratar Fator de Integração Tardia*: a lista de classes ordenadas LCOTI={A, E, H, D, C, F, B} . As classes A, E e H são selecionadas na primeira iteração e ordenadas pelo primeiro e segundo critérios de *deadlock* (explicados na seção 3.3.3.3); as classes D e C são selecionadas na segunda iteração com valores de FIT nulos; e as classes F e B são selecionadas na terceira iteração com valores de FIT nulos, conforme demonstrado na Tabela 3.11.
- do processo *Gerar Lista de Classes Ordenadas*: a lista LCOTI final igual a {A, E, H, D, C, F, B, G}, com a inserção da lista LCD ao final da lista LCOTI obtida no final do processo *Tratar Fator de Integração Tardia.*

Conforme discutido no Capítulo 2, o esforço de teste e os *stubs* específicos necessários para integrar e testar as classes do modelo na ordem identificada pelo processo de aplicação das heurísticas é determinado pelas dependências existentes entre as classes. Se uma classe X usa um ou mais serviços de outra classe Y, dizemos que X depende de Y. Num processo incremental, quando X for integrado, se Y ainda não tiver sido, precisaremos de um *stub* de Y para X, representado por (Y, X). Para a ordem de integração das classes {A, E, H, D, C, F, B, G} o processo incremental dos testes de integração será:

- Testar a classe A. A depende de C e a classe C ainda não foi integrada, portanto será necessária a criação de um *stub* específico de C para A, (C, A).
- Testar a classe E. E depende de A, mas A já está integrada, portanto não será necessária a criação de *stubs*.
- Testar a classe H. H depende de C e a classe C ainda não foi integrada, portanto será necessária a criação de um *stub* específico de C para H, (C, H).
- Testar a classe D. D depende de A e H, mas A e H já estão integradas, portanto não será necessária a criação de *stubs*.
- Testar a classe C. C depende de A e E, mas A e E já estão integradas, portanto não será necessária a criação de *stubs*.
- Testar a classe F. F depende de D e E, mas D e E já estão integradas, portanto não será necessária a criação de *stubs*.
- Testar a classe B. B depende de C, D e H, mas C, D e H já estão integradas, portanto não será necessária a criação de *stubs*.

- Testar a classe G. G depende de B e F, mas B e F já estão integradas, portanto não será necessária a criação de *stubs*.

Portanto, para a execução dos testes de integração com a ordem das classes identificada pelo processo de aplicação das heurísticas é necessário implementar apenas dois *stubs* específicos da classe C: (C, A) e (C, H). O esforço de teste, medido pelo número de *stubs* necessários ao teste, neste modelo é igual a dois.

Para analisar a efetividade do processo de aplicação das heurísticas foram utilizados os resultados obtidos pelas estratégias discutidas no capítulo 2. Como apresentado na Tabela 3.12, o esforço de teste igual a dois, obtido com o processo de aplicação das heurísticas, foi inferior aos valores obtidos pelas outras estratégias.

	TAI e DANIELS	**TRAON *et al.* (Raiz G)**	**BRIAND *et al.***	**HEURÍSTICAS**
Classes que precisam de *stubs*	C, F, H, D, B	B, C, F	C, F, H, B	C
No. classes	5	3	4	1
Stubs Específicos	C, F, H, D, B	1 de B, 2 de C, 1 de F	C, F, H, B	2 de C
Esforço de Teste	5	4	4	2

Tabela 3.12 – Comparação do resultado com outras estratégias.

Outros dois resultados importantes obtidos com o processo de aplicação das heurísticas neste exemplo são:

- a redução do número de classes que precisam de *stubs*. Devendo ser avaliada a possibilidade de implementar apenas um *stub* realístico para a classe C do exemplo; e
- a redução do número de iterações a serem calculadas durante o processo. Neste exemplo, foram apenas 3 iterações para ordenar 8 classes.

3.5 – Considerações Finais

Neste capítulo foi formalizado o conjunto de heurísticas para identificação da ordem de integração das classes em um projeto de software orientado a objetos a partir de seu diagrama de classes UML. O foco principal é a minimização do esforço de teste, medido pelo número de *stubs* específicos a serem implementados e necessários para execução dos testes, comparativamente ao esforço de outras estratégias existentes e discutidas no capítulo 2.

Entretanto, as heurísticas isoladamente não permitem a garantia do esforço de teste mínimo. Para suprir esta necessidade foi elaborado um processo de aplicação das heurísticas, também apresentado neste capítulo, com o propósito de controlar as atividades envolvidas, garantindo desta forma que a ordem das classes identificada represente o menor esforço de teste.

Cabe ressaltar que a possibilidade de aplicação das heurísticas diretamente num artefato de alto nível de abstração de projeto, diagrama de classes UML, permite apoiar o engenheiro de software durante a atividade de planejamento da construção do projeto. A ordem identificada para condução dos testes de integração das classes, além de minimizar o esforço de testes, pode sugerir uma ordem de implementação e testes de unidades das classes.

Os valores encontrados utilizando o processo definido para aplicação das heurísticas no modelo deste capítulo sugerem uma redução significativa (cinqüenta por cento do esforço de teste). Para evitar o viés da comparação com um único modelo foram executados estudos experimentais, cujo planejamento, execução, resultados e análises serão apresentados no próximo capítulo.

Capítulo 4
Estudos de Caso

Este capítulo apresenta uma breve introdução à Engenharia de Software Experimental e os Estudos de Caso efetuados para avaliar o conjunto de heurísticas e o processo de aplicação, apresentados no capítulo 3 – Heurísticas para identificação da ordem do teste de integração de classes em software orientado a objetos.

4.1 – Introdução

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 (PFLEEGER, 1999). 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 determinação da efetividade de técnicas aplicadas à engenharia de software (ZELKOWITZ, 2003).

Experimentação é o centro do processo científico. Ela oferece um modo sistemático, disciplinado, computável e controlado para avaliação de novos métodos, técnicas, linguagens e ferramentas. Estes não deveriam ser apenas sugeridos, publicados ou apresentados para venda sem experimentação e avaliação (AMARAL, 2003).

No campo da Engenharia de Software, a utilização de métodos experimentais pode trazer alguns benefícios (TICHY, 1998) (JURISTO e MORENO, 2000) (TRAVASSOS *et al.*, 2001):

- Ajudar a construir uma base confiável de conhecimento e desta forma reduzir a incerteza sobre quais teorias, métodos e ferramentas são adequados;
- Poder levar a novas compreensões sobre áreas atualmente pesquisadas;
- Conduzir a áreas desconhecidas, onde a engenharia de software progride lentamente;
- Acelerar o progresso por meio da rápida eliminação de abordagens não fundamentadas, suposições errôneas e modismos, ajudando a orientar a engenharia e a teoria para direções promissoras;

- Ajudar no processo de formação da Engenharia de Software como ciência, por meio do crescimento do número de trabalhos científicos com uma avaliação experimental significativa.

Segundo WOHLIN *et al.* (2000) os métodos experimentais são classificados em:

- **Pesquisa de campo:** é, geralmente, uma investigação executada em retrospecto quando, por exemplo, uma ferramenta ou técnica tem sido utilizada em uma empresa e pretende-se avaliá-la sob algum aspecto (AMARAL, 2003). Algumas formas para coleta de dados como entrevistas e questionários são bastante conhecidas e utilizadas.
- **Estudos de caso:** são usados para monitorar projetos, atividades ou exercícios objetivando rastrear um atributo específico ou estabelecer relacionamentos entre diferentes atributos sem um controle muito formal sobre as atividades relacionadas ao método experimental.
- **Experimentos:** é uma atividade com o propósito de descobrir algo desconhecido ou de testar uma hipótese envolvendo uma investigação de coleta de dados e de execução de uma análise para determinar o significado dos dados (BASILI *et al.*, 1999). São apropriados para confirmar teorias, o conhecimento convencional, explorar os relacionamentos, avaliar a predição dos modelos ou validar as medidas. As principais características dos experimentos estão no controle total sobre o processo e variáveis e na possibilidade de ser repetido.

A escolha de qual método de pesquisa experimental utilizar depende dos: pré-requisitos da investigação, propósito, disponibilidade de recursos e de como se pretende analisar os dados coletados (WOHLIN *et al.*, 2000).

Segundo SHULL *et al.* (2001) a primeira fase da metodologia experimental para introduzir processos de software tem o propósito de prover informações que justifiquem a continuidade de um trabalho proposto e, neste momento, o novo processo poderá ser avaliado para verificar sua efetividade.

Estes pontos estimularam a definição e planejamento de quatro estudos de caso para avaliar o conjunto de heurísticas e seu respectivo processo de aplicação apresentados no Capítulo 3, que são descritos na ordem cronológica de suas respectivas execuções:

- Viabilidade das Heurísticas;
- Procedimentos Existentes (pré-teste);

- Autotreinamento nas Heurísticas; e
- Efetividade das Heurísticas (pós-teste).

4.2 – 1º. Estudo de Caso: Viabilidade das Heurísticas

4.2.1 – Definição dos Objetivos

4.2.1.1 – Objetivo Global

Caracterizar a viabilidade das heurísticas para identificação da ordem das classes (diagrama de classes UML) para execução dos testes de integração de um modelo OO. Para tal será feita uma comparação entre o processo de aplicação das heurísticas e a estratégia proposta por BRIAND *et al.* (2001).

Nesta etapa pretende-se avaliar se os resultados obtidos justificam os recursos a serem despendidos com a continuidade do estudo.

4.2.1.2 – Objetivo da Medição

Tendo como base que a estratégia proposta por BRIAND *et al.* apresentou resultados melhores que outras estratégias estudadas por estes, investigar e comparar o esforço de teste obtido com o processo de aplicação das heurísticas quando aplicado ao mesmo modelo empregado por BRIAND *et al.* para caracterizar o desempenho.

4.2.1.3 – Objetivo do Estudo

Analisar o processo de aplicação das heurísticas para identificação da ordem das classes para execução de testes de integração OO
Com o propósito de caracterizar
Com respeito à aderência das heurísticas no que tange o esforço de teste
Do ponto de vista do pesquisador
No contexto do diagrama de classes do exemplo ATM (BRIAND, 2001).

4.2.1.4 – Questão

Q: A aplicação do processo das heurísticas para identificação da ordem das classes para execução dos testes de integração encontra uma ordem cujo esforço de teste

é menor do que o esforço de teste necessário para a ordem estabelecida quando utilizando a estratégia de BRIAND *et al.* para o mesmo modelo?

Métrica: Esforço de teste necessário para execução do teste de integração.

4.2.2 – Planejamento do Estudo

4.2.2.1 – Definição das Hipóteses

Hipótese nula (H0): o esforço de teste obtido com a utilização do processo de aplicação das heurísticas não se altera quando comparado ao esforço de teste encontrado por BRIAND *et al.* durante a aplicação da estratégia destes?

S_{Briand}: número de *stubs* específicos – estratégia BRIAND *et al.*; e

$S_{Heurísticas}$: número de *stubs* específicos – processo de aplicação das heurísticas.

H0: $S_{Briand} - S_{Heurísticas} = \varnothing$

Hipótese alternativa (H1): as heurísticas reduzem o esforço de teste se comparadas ao esforço de teste da estratégia de BRIAND *et al.*

H1: $S_{Briand} - S_{Heurísticas} > \varnothing$

4.2.2.2 – Seleção do contexto

O modelo ATM (*Automated Teller Machine*), Figura 4.1, é o mesmo empregado por BRIAND *et al.* (2001), contendo ciclos de dependências e três tipos de dependências. As classes foram numeradas, conforme Tabela 4.1, para facilitar a descrição do processo.

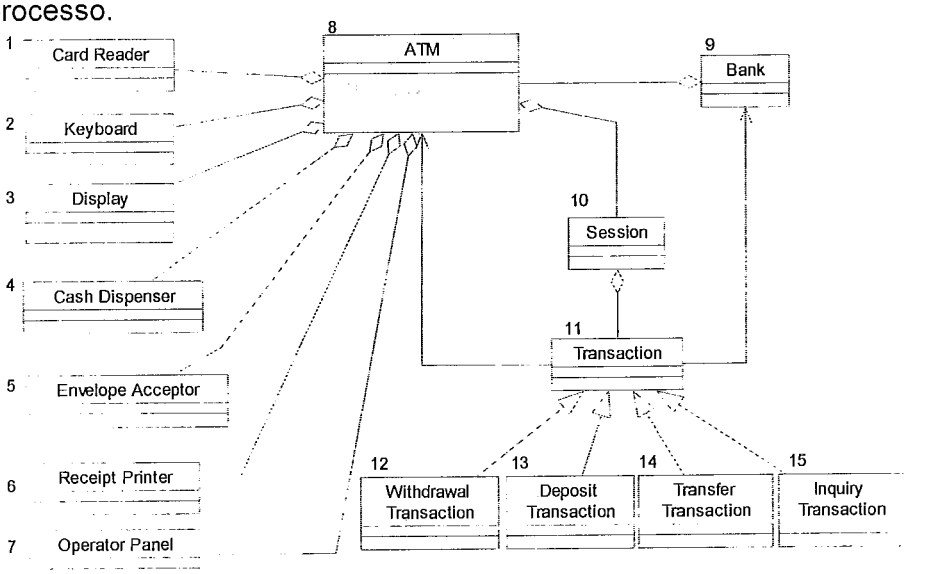


Figura 4.1 – Modelo ATM: Diagrama de Classes.

O processo utilizado é *off-line* porque as heurísticas não são aplicadas durante um ciclo de desenvolvimento real do projeto de um software.

Classes	Identificador
Card Reader	1
Keyboard	2
Display	3
Cash Dispenser	4
Envelope Acceptor	5
Receipt Printer	6
Operator Panel	7
ATM	8
Bank	9
Session	10
Transaction	11
Withdrawal Transaction	12
Deposit Transaction	13
Transfer Transaction	14
Inquiry Transaction	15

Tabela 4.1 – Modelo ATM: Identificadores das Classes.

4.2.2.3 – Seleção dos indivíduos

O único participante deste estudo é o próprio pesquisador, devido ao seu conhecimento do processo de aplicação das heurísticas.

4.2.2.4 – Variáveis

As variáveis independentes são:

1. O processo de aplicação das heurísticas utilizado pelo pesquisador.
2. A estratégia de ordenação de BRIAND *et al.* (2001).

As variáveis dependentes são:

1. A lista ordenada de classes para execução dos testes de integração.
2. O esforço de teste (número de *stubs* específicos).

4.2.3 – Operação do Estudo

O processo *Pesquisar Fator de Influência* é iniciado pela atividade "Aplicar critérios de precedência". Observa-se que as classes 1, 2, 3, 4, 5, 6, 7 e 10 têm precedência sobre a classe 8 (critério da agregação). A classe 8 tem precedência

sobre as classes: 9 (critério da agregação) e 11 (critério da navegabilidade unidirecional). A classe 9 tem precedência sobre a classe 11. A classe 11 tem precedência sobre a classe 10 (critério da agregação) e sobre as classes 12, 13, 14 e 15 (critério da herança). As classes 12, 13, 14 e 15 não têm precedência sobre outras classes.

Portanto, existem cinco classes que devem ser integradas após a classe 11, assim sendo o valor cinco deve ser associado ao fator de influência (FI) da classe 11 durante a atividade "Atribuir FI". Os valores de FI apresentados na Tabela 4.2 são resultantes da execução da atividade descrita anteriormente para todas as classes do Modelo ATM, analogamente ao demonstrado para a classe 11. O resultado da atividade "Separar classes totalmente dependentes" é a lista LCD = {12, 13, 14, 15}, formada pelas classes com FI nulos.

O processo *Tratar Fator de Integração Tardia* para a lista LCNO = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11} inicia-se com os valores mostrados pela coluna FIT_1 da Tabela 4.3. As classes 1, 2, 3, 4, 5 e 6 que possuem FIT_1 nulo serão incluídas na lista LCOTI. Antes de passar para a próxima iteração, é necessário executar a atividade "Reduzir o fator de influência das classes ordenadas" para estas classes. Na segunda iteração (FIT_2), não existem classes cujo valor de FIT seja nulo. Então, a próxima atividade a ser executada será "Priorizar as classes com menor FIT", sendo selecionada a classe 8 com FIT_2 igual a um. Novamente é executada a atividade "Reduzir o fator de influência das classes ordenadas". Na terceira, quarta e quinta iterações são selecionadas as classes 9, 11 e 10, respectivamente.

Classe ID.	FI
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	2
9	1
10	1
11	5
12	0
13	0
14	0
15	0

Tabela 4.2 – Modelo ATM: Fatores de Influência.

Classe ID.	FI	FIT_1	FIT_2	FIT_3	FIT_4	FIT_5
1	1	0	-	-	-	-
2	1	0	-	-	-	-
3	1	0	-	-	-	-
4	1	0	-	-	-	-
5	1	0	-	-	-	-
6	1	0	-	-	-	-
7	1	0	-	-	-	-
8	2	8	1	-	-	-
9	1	2	2	0	-	-
10	1	5	5	5	5	0
11	5	3	3	1	0	-

Tabela 4.3 – Modelo ATM: Fatores de Integração Tardia.

A lista LCOTI = {1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10, 12, 13, 14, 15} é estabelecida com a inclusão da lista LCD ao final mesma, durante o processo de *Gerar Lista Ordenada de Classes*.

4.2.4 – Análise do Resultado

A lista de classes ordenadas para o teste de integração, o correspondente esforço de teste para os dois tratamentos: BRIAND *et al.* (2001) e o processo de aplicação das heurísticas são apresentados na Tabela 4.4.

Proposta	BRIAND *et al.*	Heurísticas
Lista Ordenada	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	11 é testada com *stubs* de 8 e 9	8 é testada com *stub* de 10
Esforço de Teste (No. de stubs)	(S_{Briand}) 2	($S_{Heurísticas}$) 1

Tabela 4.4 – Modelo ATM: Resultados.

A partir da Tabela 4.4 pode-se perceber que a hipótese nula (H0) foi refutada uma vez que o emprego do processo de aplicação das heurísticas produz um esforço

de teste menor. Por conseqüência, a hipótese alternativa 1 (H1) na qual temos que **H1:** $S_{Briand} - S_{Heurísticas} > \varnothing$ foi confirmada.

4.2.5 – Lições Aprendidas

O processo foi avaliado somente no sentido de produzir resultados práticos (esforço de teste menor que outro procedimento). Permite concluir que o processo de aplicação das heurísticas pode atingir o objetivo para o qual as heurísticas foram estabelecidas e que os riscos envolvidos com a aplicabilidade diminuíram, fornecendo informação que justifique a continuidade do trabalho.

Foi identificada a necessidade de elaborar outros estudos de caso para avaliar de forma mais abrangente a aplicação das heurísticas em diferentes situações de projeto e tentar, minimamente, ampliar a abrangência dos resultados encontrados, evitando qualquer viés ou risco de aplicação em apenas um modelo.

4.3 – 2º. Estudo de Caso: Procedimentos Existentes (pré-teste)

4.3.1 – Definição dos Objetivos

4.3.1.1 – Objetivo Global

Caracterizar a viabilidade de utilização das heurísticas para identificação da ordem das classes para execução dos testes de integração de um modelo. Para tal será feita uma comparação entre processo de aplicação das heurísticas e processos *ad-hoc* em um mesmo modelo de software da área financeira, representado pelo diagrama de classes UML.

4.3.1.2 – Objetivo da Medição

Tendo como base o esforço de teste para aplicação do processo das heurísticas pelo pesquisador apresentou melhores resultados que a estratégia proposta por BRIAND *et al.* (2001), caracterizar se outro processo *ad-hoc* diminui o esforço de teste aplicado ao mesmo modelo de software da área financeira empregado pelo pesquisador para comprovar a efetividade das heurísticas.

4.3.1.3 – Objetivo do Estudo

Analisar procedimentos *ad-hoc* para identificação da ordem das classes para execução de testes de integração OO

Com o propósito de caracterizar

Com respeito à aderência das heurísticas no que tange ao esforço de teste e ao esforço de identificação

Do ponto de vista de gerentes de software e engenheiros de software

No contexto de um modelo real de software da área financeira (modelo 0), desenvolvido segundo o paradigma da orientação a objetos e representado por um diagrama de classes UML.

4.3.1.4 – Questões

Q1: A aplicação de algum procedimento *ad-hoc* para identificação da ordem das classes para execução dos testes de integração encontra uma ordem cujo esforço de teste é menor do que o esforço de teste necessário para a ordem estabelecida pelo pesquisador utilizando as heurísticas no mesmo modelo 0?

Métrica: Esforço de teste necessário para execução do teste de integração.

Q2: A aplicação de algum procedimento *ad-hoc* para identificação da ordem das classes para execução dos testes de integração encontra esforço de identificação (tempo em minutos) menor do que o esforço necessário para a ordem estabelecida pelo pesquisador utilizando as heurísticas no mesmo modelo 0?

Métrica: Esforço de identificação necessário para identificação de uma ordem.

4.3.2 – Planejamento do Estudo

4.3.2.1 – Seleção do Contexto

Todos os participantes recebem o diagrama de classes UML do Modelo 0 e um formulário de resposta, onde é feita a ordenação das classes para execução dos testes de integração destas, bem como do tempo gasto (em minutos) para conclusão desta tarefa.

O modelo real da área financeira é apresentado no Anexo B.1. Portanto, trata-se de um contexto específico. O processo utilizado é *off-line* porque as heurísticas não são aplicadas durante um ciclo de desenvolvimento real do projeto de um software.

4.3.2.2 – Definição das Hipóteses

4.3.2.2.1 – Em relação ao esforço de teste

Hipótese nula (H0): o esforço de teste obtido com a utilização de processos *ad-hoc* não se altera quando comparado ao esforço de teste encontrado pelo pesquisador com a aplicação do processo das heurísticas no mesmo modelo 0?

$S0(P)_{ad\text{-}hoc}$: número de *stubs* específicos – procedimentos *ad-hoc* – participantes; e
$S0(G)_{heurísticas}$: número de *stubs* específicos – heurísticas – pesquisador.
H0: $S0(P)_{ad\text{-}hoc} - S0(G)_{heurísticas} = \varnothing$

Hipótese alternativa (H1): as heurísticas reduzem o esforço de teste se comparadas ao esforço de teste de outros procedimentos *ad-hoc* quando aplicados no mesmo modelo.
H1: $S0(P)_{ad\text{-}hoc} - S0(G)_{heurísticas} > \varnothing$

4.3.2.2.2 – Em relação ao Tempo de Identificação

Hipótese nula (H0): o tempo de identificação obtido com a utilização de processos *ad-hoc* não se altera quando comparado ao tempo de identificação encontrado pelo pesquisador com a aplicação do processo das heurísticas no mesmo modelo 0?
$T0(P)_{ad\text{-}hoc}$: esforço de identificação (tempo em minutos) – procedimentos *ad-hoc* – participantes; e
$T0(G)_{heurísticas}$: esforço de identificação (tempo em minutos) – heurísticas – pesquisador.
H0: $T0(P)_{ad\text{-}hoc} - T0(G)_{heurísticas} = \varnothing$

Hipótese alternativa (H1): as heurísticas reduzem o tempo de identificação, se comparadas ao tempo de identificação de outros procedimentos *ad-hoc* quando aplicados no mesmo modelo 0.
H1: $T0(P)_{ad\text{-}hoc} - T0(G)_{heurísticas} > \varnothing$

4.3.2.3 – Seleção dos Indivíduos

A seleção dos participantes para este estudo será limitada em dois grupos, compostos por: alunos do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ (Grupo 1) e gerentes de projetos e engenheiros de software do Centro de Análises de Sistemas Navais da Marinha do Brasil (Grupo 2). Desta forma, a escolha dos indivíduos será baseada em princípios não probabilísticos onde a população será determinada por conveniência.

4.3.2.4 – Variáveis

As variáveis independentes são:

1. O processo de aplicação das heurísticas utilizado pelo pesquisador.
2. Os procedimentos *ad-hoc* utilizados pelos participantes.

As variáveis dependentes são:

1. A lista ordenada de classes para execução dos testes de integração.
2. O esforço de teste (número de *stubs* específicos).
3. O esforço de identificação (tempo em minutos).

4.3.2.5 – Validade do Estudo

4.3.2.5.1 – Validade Interna

Durante a avaliação da validade interna uma maior atenção deve ser prestada aos participantes, ou seja, à seleção da população, à maneira da divisão nas classes, ao modo da aplicação dos tratamentos e aos aspectos sociais (AMARAL, 2003). Neste estudo os participantes foram divididos em dois grupos, conforme sua origem: acadêmica (11 participantes) ou indústria (10 participantes), com o propósito de identificar algumas variações de procedimento ou desempenho.

Para evitar que a troca de informações entre os participantes pudesse influenciar o resultado do estudo foram tomadas duas medidas: a explícita solicitação aos participantes para não trocarem informação durante a execução do estudo e a presença do pesquisador durante a execução do mesmo.

4.3.2.5.2 – Validade Externa

Validade externa mede a capacidade do estudo de refletir o mesmo comportamento em outros grupos de participantes e profissionais da indústria, ou seja, em outros grupos além daquele em que o estudo foi aplicado. O maior problema em relação à validade externa do estudo seria a falta de interesse dos participantes no estudo. Alguns indivíduos podem realizar o estudo de forma descomprometida, sem um interesse real na realização do projeto em menor esforço de teste ou tempo de identificação, como aconteceria na indústria durante um projeto real. As técnicas foram utilizadas fora do ambiente de desenvolvimento de software, em um diagrama de classes pronto, portanto, não é possível generalizar os resultados obtidos na indústria de forma mais ampla.

4.3.2.5.3 – Validade de Construção

A validade de construção considera os relacionamentos entre a teoria e a observação, ou seja, se o tratamento reflete bem a causa e o resultado reflete bem o efeito (WOHLIN, 2000). Neste estudo é empregado o mesmo modelo tanto na aplicação do processo das heurísticas pelo pesquisador quanto nos procedimentos *ad-hoc* pelos participantes, garantindo que o tratamento reflete a causa. Os resultados observados, esforço de teste e tempo de identificação, refletem o comportamento estudado.

4.3.2.5.4 – Validade de Conclusão

Foi empregado o mesmo diagrama de classes pelo pesquisador e pelos participantes, sendo possível acreditar que os esforços de teste medidos refletem o resultado dos procedimentos e/ou técnicas empregados individualmente. Outro aspecto considerado foi o cálculo do esforço de teste, realizado pelo pesquisador, com os mesmos critérios, para todos os participantes.

4.3.3 – Operação do Estudo

4.3.3.1 – Preparação

Nesta etapa os participantes preencheram o Formulário de Consentimento e o Formulário de Caracterização (formação acadêmica e experiência profissional), apresentados nos Anexos B.2 e B.3.

4.3.3.2 – Pesquisador: Aplicação do Processo das Heurísticas

O processo de *Pesquisar o Fator de Influência* é iniciado pela atividade "Aplicar critérios de precedência". Observa-se que a classe J, por exemplo, tem precedência sobre as classes: I (pelo critério da navegabilidade); e N (pelo critério da agregação). Portanto, existem duas classes que devem ser integradas após a classe J, assim sendo, o valor dois deve ser associado ao *fator de influência* (FI) da classe J durante a atividade "Atribuir FI". Os valores de FI apresentados na Tabela 4.5 são resultantes da execução das atividades descritas anteriormente para todas as classes do modelo, analogamente ao demonstrado para a classe J.

Classe	FI	FIT_1
A	2	8
B	2	6
C	4	12
D	3	9
E	3	7
F	3	13
G	1	3
H	2	7
I	3	10
J	2	3
K	2	8
L	2	6
M	1	6
N	6	16

Tabela 4.5 – Modelo 0: Valores de FI e FIT.

Pode-se observar que não existem classes totalmente dependentes (com fatores de influência nulos) a serem separadas. Portanto, o resultado da atividade "Separar classes totalmente dependentes" é uma lista LCD vazia.

A seguir é iniciado o processo *Tratar Fator de Integração Tardia* por meio da atividade de "Identificar precedências", onde a classe J, por exemplo, depende somente da classe I para ser integrada. Durante a atividade "Calcular FIT", o valor do FI da classe I será atribuído ao FIT da classe J. O resultado do FIT_1 (primeira iteração) calculado para todas as classes do modelo é mostrado na Tabela 4.5.

Como pode ser observado, nesta iteração (FIT_1), não existem classes cujo valor de FIT seja nulo. Então, a próxima atividade a ser executada será "Priorizar as classes não folha com menor FIT". São elas: G e J, com FIT_1 igual a 3. Durante esta atividade, a classe J será priorizada em relação à classe G, devido seu tamanho ser maior do

que o tamanho da classe G. A lista LCOTI é atualizada com {J, G} e a lista LCNO = {A, B, C, D, E, F, H, I, K, L, M, N} durante a atividade "Atualizar listas".

Antes de passar para a próxima iteração, é necessário executar a atividade "Reduzir o fator de influência das classes ordenadas" para as classes E, I e N.

Classe	FI	FIT_2	FIT_3	FIT_4	FIT_5	FIT_6
A	2	8	6	-	-	-
B	2	6	-	-	-	-
C	4	12	8	5	-	-
D	3	9	6	-	-	-
E	3	6	-	-	-	-
F	3	13	10	10	6	6
G	1	-	-	-	-	-
H	2	7	7	7	3	-
I	3	8	8	8	8	6
J	2	-	-	-	-	-
K	2	8	6	-	-	-
L	2	6	-	-	-	-
M	1	6	-	-	-	-
N	6	14	13	6	6	6

Tabela 4.6 – Modelo 0: Valores de FIT para as demais iterações.

Os resultados de FIT para a segunda iteração da atividade "Calcular FIT", são apresentados na coluna FIT_2 da Tabela 4.6. Novamente, é executada a atividade "Priorizar classes não folha", com o menor FIT igual a 6. Destaca-se, neste ponto, a necessidade de somente um *stub* específico para testar M comparativamente aos dois *stubs* necessários para testar B, E e L. As listas atualizadas seriam LCOTI = { J, G , M, B, E, L} e LCNO= {A, C, D, F, H, I, K, N}. Sendo novamente reduzido o fator de influência das classes ordenadas para as classes A, C, D, F, K e N.

O processo de *Tratar Fator de Integração Tardia* se repete até a sexta iteração, quando as últimas classes são ordenadas. A lista de classes ordenadas para teste de integração seria LCOTI={ J, G, M, B, E, L, K, D, A, C, H, N, F, I}, após o processo *Gerar Lista Ordenada de Classes*.

Para a lista de classes ordenadas para execução do teste de integração obtida, termos as seguintes métricas: $\mathbf{S0(G)_{heurísticas}}$ igual a 17 (número de *stubs* específicos encontrados pelo pesquisador com a aplicação das heurísticas) e $\mathbf{T0(G)_{heurísticas}}$ (tempo de identificação (em minutos), aplicando heurísticas, pelo pesquisador) igual a 11.

4.3.3.3 – Participantes: Procedimentos *Ad-hoc*

Após a identificação da ordem das classes para execução dos testes de integração pelos participantes, apresentadas nos formulários de respostas (Anexo B.4), foram obtidos: $\mathbf{S0(P)_{ad\text{-}hoc}}$ (número de *stubs* específicos encontrados por procedimentos *ad-hoc* aplicados pelos participantes) e $\mathbf{T0(P)_{ad\text{-}hoc}}$ (esforço de identificação – tempo em minutos - procedimentos *ad-hoc,* aplicados pelos participantes) apresentados nas Figuras 4.2 e 4.3, onde os resultados obtidos pelo pesquisador na seção 4.3.3.2 estão marcados na cor vermelha.

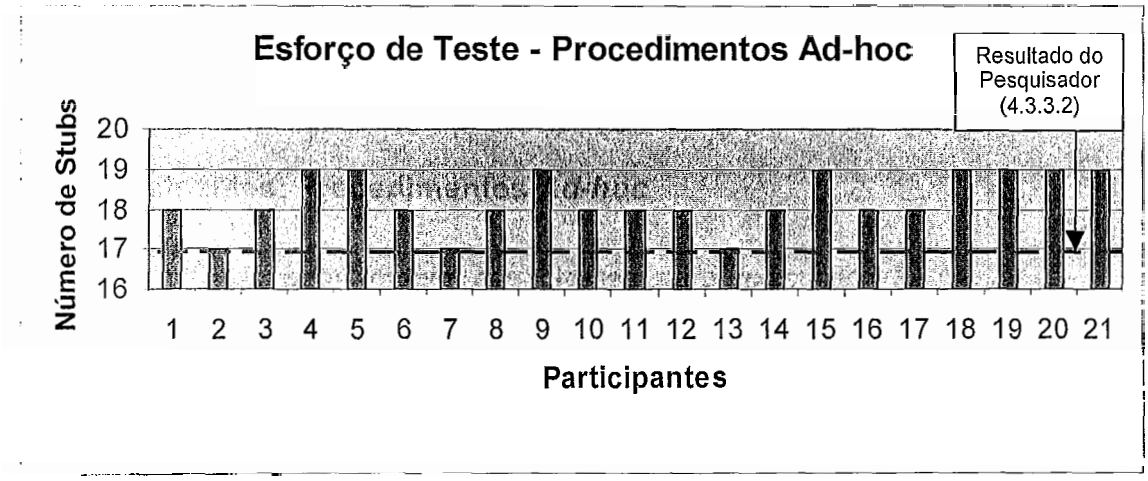


Figura 4.2 – Modelo 0: Esforço de Teste dos Procedimentos *ad-hoc* dos participantes.

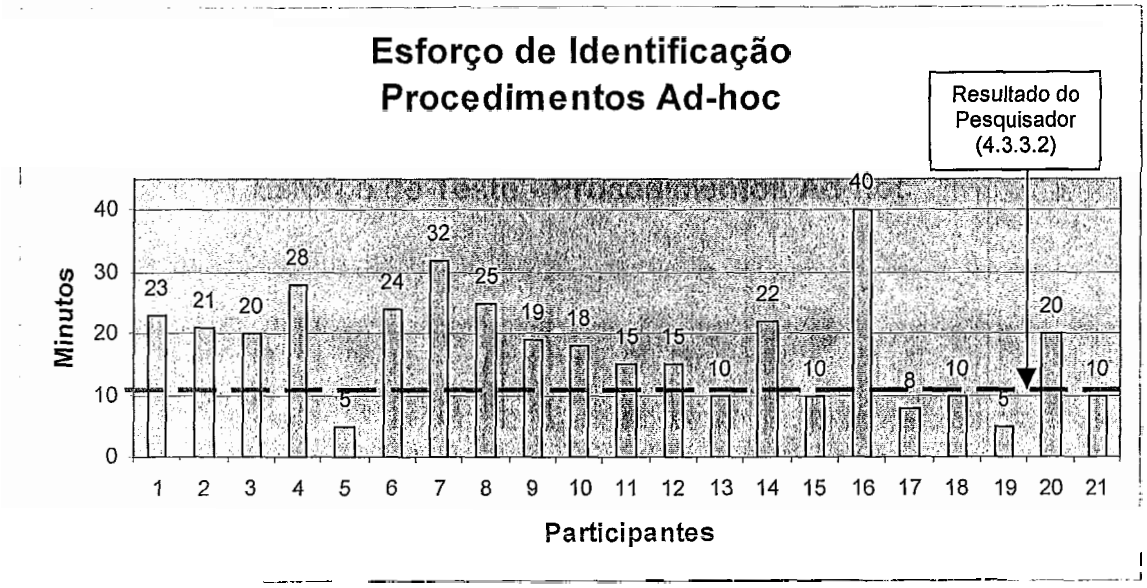


Figura 4.3 – Modelo 0: Esforço de Identificação dos Procedimentos *ad-hoc* dos Participantes.

4.3.4 – Análise dos Resultados

4.3.4.1 – Caracterização dos Participantes

A partir dos formulários de caracterização foi possível identificar que 66,67% dos participantes possuem cursos de especialização ou mestrado, como apresentado na Figura 4.4. Os participantes da indústria – Grupo 2 – possuem média de experiência prática em software (média de 10,7 anos – desvio padrão de 6.96) maior que os participantes da academia – Grupo 1 – (média de 3,18 anos – desvio padrão de 2,82), como mostrado na Figura 4.5. Os participantes 9 (Grupo 1) e 14 (Grupo 2) declararam não ter experiência prática.

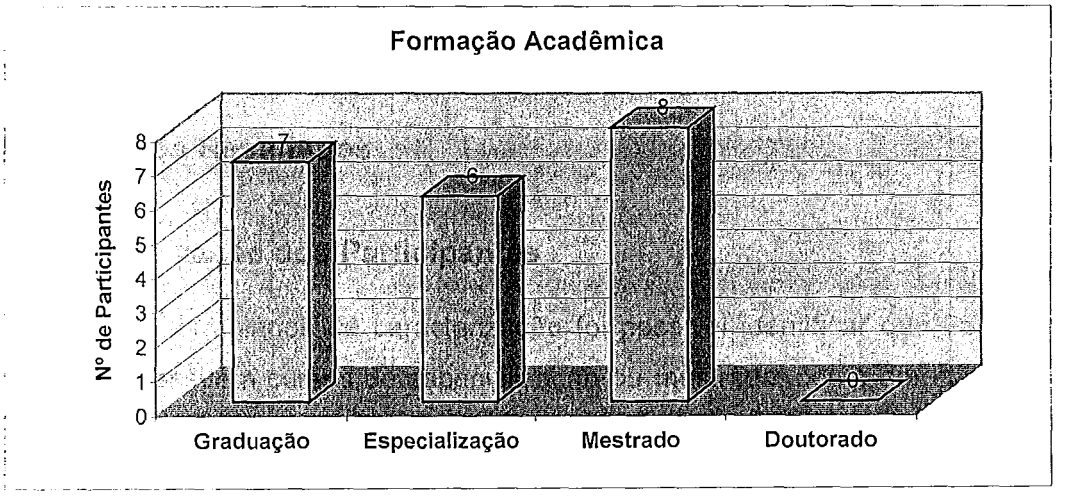


Figura 4.4 – Participantes: Formação Acadêmica.

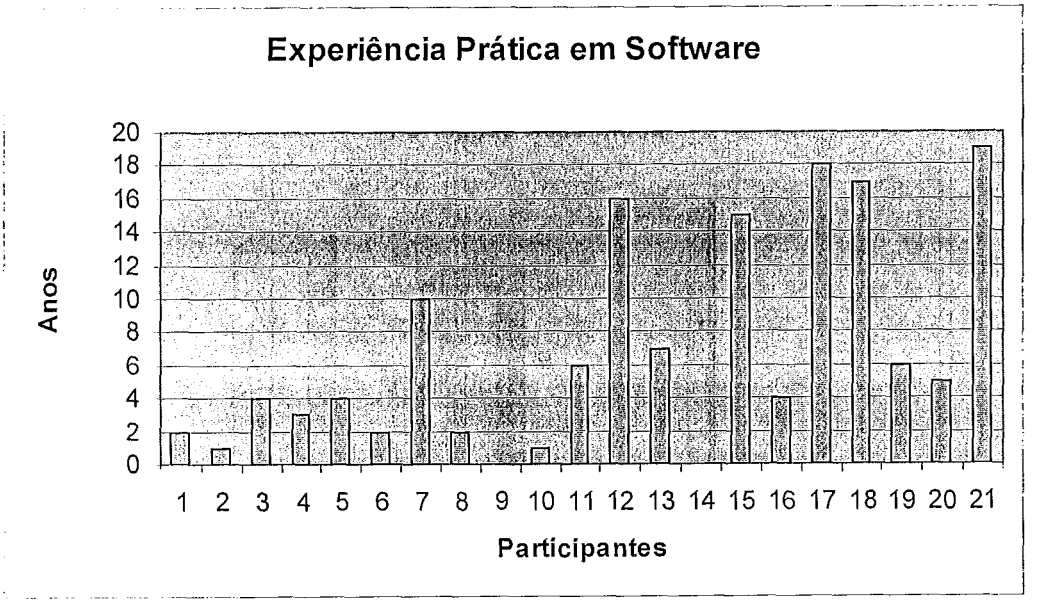


Figura 4.5 – Participantes: Experiência Prática em Software.

Considerando que a graduação da experiência para a Figura 4.6 foi de 1 a 5 e que a informação do participante com grau 5 indica que o mesmo usou o conceito em mais de um projeto na indústria, pode-se caracterizar os participantes deste estudo com bom conhecimento dos conceitos de orientação a objetos e UML.

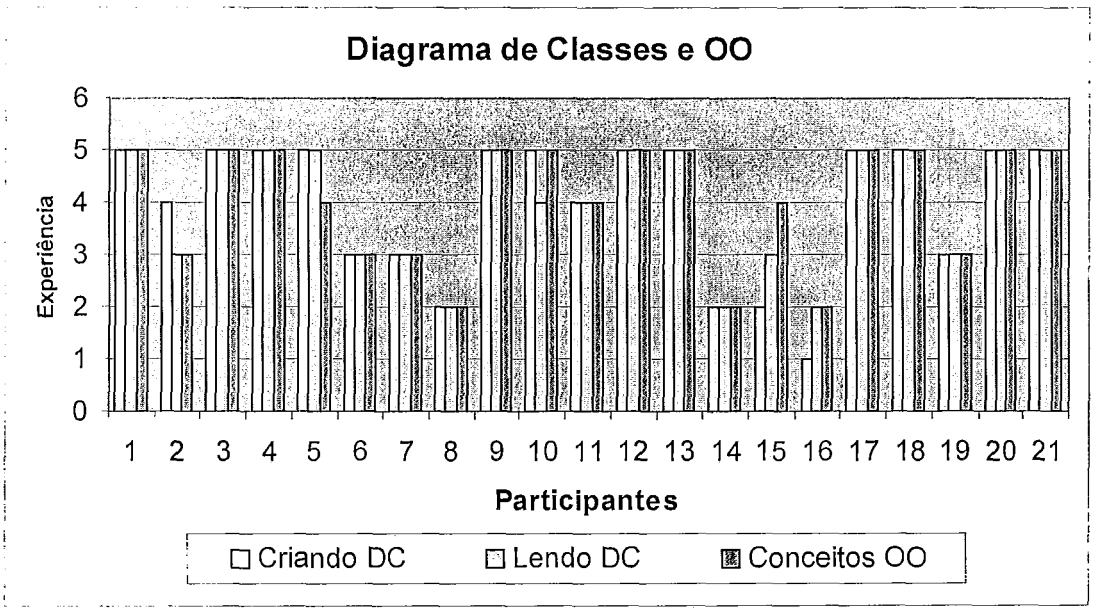


Figura 4.6 – Participantes: Conhecimento no Paradigma Orientado a Objetos.

4.3.4.2 – Teste das Hipóteses

O esforço de teste (mediana) e o esforço de identificação (médio e desvio padrão) calculados usando *Paired T-Test* (amostras emparelhadas) são apresentados na Tabela 4.7. A partir destes resultados pode-se perceber que:

- a hipótese nula (H0) – relativa ao esforço de teste – foi refutada uma vez que o emprego do processo de aplicação das heurísticas produz um esforço de teste menor. Por conseqüência, a hipótese alternativa 1 (H1) na qual temos que **H1: $S0(P)_{ad\text{-}hoc} - S0(G)_{heurísticas} > \varnothing$** foi confirmada.
- a hipótese nula (H0) – relativa ao esforço de identificação – foi refutada uma vez que o emprego do processo de aplicação das heurísticas produz um esforço de identificação menor. Por conseqüência, a hipótese alternativa 1 (H1) na qual temos que **H1: $T0(P)_{ad\text{-}hoc} - T0(G)_{heurísticas} > \varnothing$** foi confirmada.

Wilcoxon Signed Test	Número de Casos
$S0(P)_{ad\text{-}hoc} > S0(G)_{heurísticas}$	18
$S0(P)_{ad\text{-}hoc} = S0(G)_{heurísticas}$	0
$S0(P)_{ad\text{-}hoc} < S0(G)_{heurísticas}$	3
p-value	0,000

Tabela 4.7a – Resultados do 2º. Estudo de Caso: Esforço de Teste

	$T0(P)_{ad\text{-}hoc}$ (minutos)	$T0(G)_{heurísticas}$ (minutos)
Média	18,10	9
Desvio Padrão	9,044	-

Tabela 4.7b – Resultados do 2º. Estudo de Caso: Esforço de Identificação.

Outra análise dos tempos de identificação pode ser feita, considerando separadamente os valores dos Grupos 1 e 2 isolados, como mostrados na Tabela 4.8. Os participantes da academia (Grupo 1) tem um esforço médio de identificação maior do que o esforço médio de identificação dos participantes da indústria (Grupo 2). Entretanto, o coeficiente de variação do Grupo 1 é aproximadamente 50% menor do que o do Grupo 2.

Grupo	T0(P) Média	T0(P) Desvio Padrão	Coeficiente de Variação
Academia (1)	20,91	7,11	33,98
Indústria (2)	15,00	10,26	68,42

Tabela 4.8 – 2º. Estudo de Caso: Tempo de Identificação por Grupo.

4.3.4.3 – Análise Qualitativa

O questionário de avaliação (Anexo B.5) preenchido após a operação do estudo permitiu observar que nenhum dos participantes informou utilizar uma técnica conhecida e/ou publicada anteriormente ao estudo, ressaltando a ausência da adoção pelos participantes de um procedimento semelhante ao processo de aplicação das heurísticas.

Identificado que 62% dos participantes recorreram aos conceitos de orientação a objetos na tentativa de identificar alguma precedência entre as classes. Somente 23% destes participantes (3) conseguiram encontrar resultado semelhante ao do processo de aplicação das heurísticas. Entretanto, não foi possível identificar um padrão ou procedimento de uso dos conceitos OO para os participantes. Nada se pode concluir, pois o fato pode ter advindo de sorte durante os processos de decisão ou mesmo de alguma coincidência para o modelo escolhido.

4.3.5 – Lições Aprendidas

O estudo foi avaliado somente no sentido de identificar outros procedimentos que produzissem melhores resultados práticos do que aos resultados apresentados pelo processo de aplicação das heurísticas (esforço de teste menor com menor tempo de identificação). A Figura 4.7 mostra que 86% (18) dos participantes obtiveram um esforço de teste superior ao esforço de teste do pesquisador com o processo de aplicação das heurísticas.

Permite concluir que no contexto do estudo não foi identificado outro procedimento com esforço de teste menor do que o encontrado com o processo de aplicação das heurísticas, tanto para o Grupo 1 (academia) quanto para o Grupo 2 (indústria).

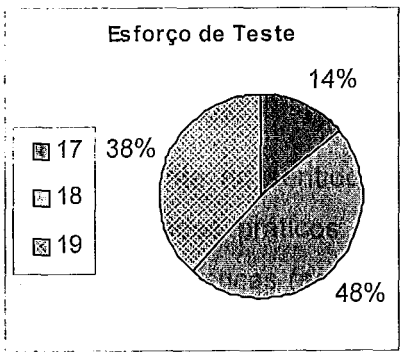


Figura 4.7 – 2º. Estudo de Caso: Esforço de Teste com Procedimentos *Ad-hoc*.

Foi identificada a necessidade de elaborar outro estudo de caso para avaliar o processo de aplicação das heurísticas pelos próprios participantes, em diferentes modelos, e tentar, minimamente, ampliar a abrangência dos resultados encontrados, evitando qualquer viés ou risco de aplicação apenas pelo pesquisador.

4.4 – 3º. Estudo de Caso: Autotreinamento nas Heurísticas

4.4.1 – Definição dos Objetivos

4.4.1.1 – Objetivo Global

Caracterizar a efetividade de utilização das heurísticas para identificação da ordem das classes para execução dos testes de integração de um modelo OO. Para tal será feita uma comparação entre processo de aplicação das heurísticas pelos participantes em dois modelos, após um autotreinamento no processo de aplicação das heurísticas.

4.4.1.2 – Objetivo da Medição

Tendo como base que o esforço de teste para aplicação do processo das heurísticas pelo pesquisador apresentou melhores resultados que outros procedimentos *ad-hoc*, caracterizar se após um autotreinamento dos participantes sobre o processo de aplicação das heurísticas, os esforços de teste calculados pelos participantes mantêm o esforço de teste calculado pelo pesquisador para os mesmos modelos empregados.

4.4.1.3 – Objetivo do Estudo

Analisar o processo de aplicação das heurísticas para identificação da ordem das classes para execução de testes de integração OO

Com o propósito de caracterizar

Com respeito à aderência das heurísticas no que tange ao esforço de teste e ao esforço de identificação

Do ponto de vista de gerentes de software e engenheiros de software

No contexto dos modelos 1 e 2 , desenvolvidos segundo o paradigma da orientação a objetos e representados por diagramas de classes UML, após um autotreinamento sobre o processo de aplicação das heurísticas.

4.4.1.4 – Questões

Q1: O aprendizado sobre o processo de aplicação das heurísticas pelos participantes por meio de um autotreinamento permite aos participantes estabelecerem esforços de teste iguais ao pesquisador?

Métrica: Esforço de teste necessário para execução do teste de integração dos modelos.

Q2: O que acontece com o tempo de identificação para os modelos 1 e 2 quando os participantes utilizam o processo de aplicação das heurísticas para identificar a ordem entre as classes para execução dos testes de integração dos modelos?

Métrica: Esforço de identificação necessário para identificação da ordem para execução do teste de integração dos modelos.

4.4.2 – Planejamento do Estudo

4.4.2.1 – Seleção do Contexto

Todos os participantes recebem autotreinamento do processo de aplicação das heurísticas (Anexo C.1) e os diagrama de classes UML com formulário de resposta dos modelos 1 e 2 (Anexos C.2 e C.3), onde é feita a ordenação das classes para execução dos testes de integração destas, bem como anotação do tempo gasto (em minutos) para conclusão desta tarefa.

O processo utilizado é *off-line* porque as heurísticas não são aplicadas durante um ciclo de desenvolvimento real do projeto de um software.

4.4.2.2 – Definição das Hipóteses

4.4.2.2.1 – Modelo 1: Esforço de Teste

Hipótese nula (H0): o esforço de teste obtido com a aplicação do processo das heurísticas no mesmo modelo 1 pelos participantes é diferente do esforço de teste encontrado pelo pesquisador com o mesmo processo para o mesmo modelo?

$S1(P)_{heurísticas}$: número de *stubs* específicos – participantes – heurísticas – modelo 1; e

$S1(G)_{heurísticas}$: número de *stubs* específicos – pesquisador – heurísticas – modelo 1.

$H0: S1(P)_{heurísticas} - S1(G)_{heurísticas} \neq \varnothing$

Hipótese alternativa (H1): o aprendizado sobre o processo de aplicação das heurísticas permitiu aos participantes estabelecer para o modelo um esforço de teste igual ao esforço de teste encontrado pelo pesquisador para o modelo 1.

$H1: S1(P)_{heurísticas} - S1(G)_{heurísticas} = \varnothing$

4.4.2.2.2 – Modelo 2: Esforço de teste

Hipótese nula (H0): o esforço de teste obtido com a aplicação do processo das heurísticas no mesmo modelo 2 pelos participantes é diferente do esforço de teste encontrado pelo pesquisador com o mesmo processo para o mesmo modelo?
$S2(P)_{heurísticas}$: número de *stubs* específicos – participantes – heurísticas – modelo 2; e
$S2(G)_{heurísticas}$: número de *stubs* específicos – pesquisador – heurísticas – modelo 2.
H0: $S2(P)_{heurísticas} - S2(G)_{heurísticas} \neq \varnothing$

Hipótese alternativa (H1): o aprendizado sobre o processo de aplicação das heurísticas permitiu aos participantes estabelecer para o modelo um esforço de teste igual ao esforço de teste encontrado pelo pesquisador para o modelo 2.
H1: $S2(P)_{heurísticas} - S2(G)_{heurísticas} = \varnothing$

4.4.2.3 – Seleção dos Indivíduos

A mesma do 2º. Estudo de Caso. O participante com identificador 17 desistiu de participar dos estudos devido a necessidade de atender a outras prioridades.

4.4.2.4 – Variáveis

A variável independente é: o processo de aplicação das heurísticas.

As variáveis dependentes são:

1. A lista ordenada de classes para execução dos testes de integração.
2. O esforço de teste (número de *stubs* específicos).
3. O esforço de identificação (tempo em minutos).

4.4.2.5 – Validade do Estudo

As validades interna, externa e de construção são as mesmas do 2º. Estudo de Caso. A validade de conclusão aumenta, pois o tratamento (aplicação das heurísticas pelos participantes) é aplicado a dois modelos e com distintos graus de complexidade.

O autotreinamento foi o procedimento escolhido para informação do processo de aplicação das heurísticas para os participantes, com o propósito de tentar eliminar o viés existente na forma e na transmissão do conteúdo pelo instrutor nos treinamentos necessários aos 20 participantes.

4.4.3 – Operação do Estudo

4.4.3.1 – Preparação

O autotreinamento sobre o processo de aplicação das heurísticas (Anexo C.1) constituí-se de uma Apresentação em PowerPoint, com propósito de apresentar aos participantes todas as informações conceituais disponíveis sobre as heurísticas e demonstrar o processo por meio da aplicação detalhada em dois exemplos.

Os participantes não tiveram limite de tempo pré-estabelecido para conclusão do autotreinamento. Não foi fornecido nenhum outro material com informações complementares. Durante o autotreinamento, não foi permitido qualquer interação entre participantes ou pesquisador até a conclusão deste estudo.

4.4.3.2 – Modelo 1: Aplicação das Heurísticas pelo Pesquisador

O processo de *Pesquisar o Fator de Influência* é iniciado pela atividade "Aplicar critérios de precedência". Observa-se que, no modelo 1 (Figura 4.8), a classe A tem precedência sobre a classe C (critério da dependência); a classe B tem precedência sobre as classes A (critério da associação unidirecional) e C (critério da herança); a classe C tem precedência sobre as classes D e E; a classe D tem precedência sobre as classes C e E; e a classe E não tem precedência sobre nenhuma outra.

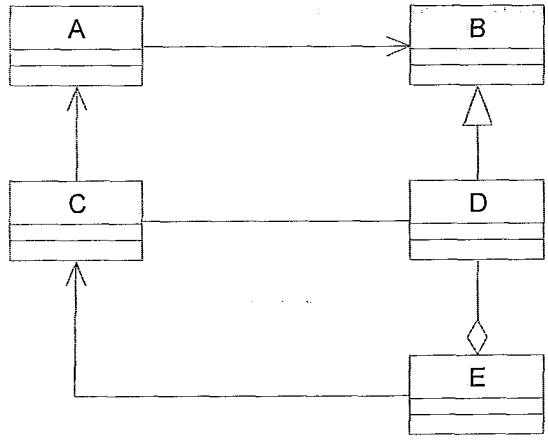


Figura 4.8 – Modelo 1: Diagrama de Classes.

Existe uma classe que deve ser integrada após a classe A, então o valor um deve ser associado ao *fator de influência* (FI) da classe A durante a atividade "Atribuir FI". Os valores de FI apresentados na Tabela 4.9 são resultantes da execução das atividades descritas anteriormente para todas as classes do modelo, analogamente ao demonstrado para a classe A. A lista LCD é igual a {E}, pois FI da classe E é nulo.

Classes	FI
A	1
B	2
C	2
D	2
E	0

Tabela 4.9 – Pesquisador: Fatores de Influência do Modelo 1.

A seguir é iniciado o processo *Tratar Fator de Integração Tardia* por meio da atividade de "Identificar precedências". A classe A depende somente da classe B para ser integrada; B não depende de nenhuma classe; C depende de A e D; e D depende de B e C. Durante a atividade "Calcular FIT", o valor do FI da classe B será atribuído ao FIT da classe A. O resultado do FIT_1 (primeira iteração) calculado para a lista LCNO = {A, B, C, D} é mostrado na Tabela 4.10.

Classes	FI	FIT_1	FIT_2	FIT_3
A	1	2	0	-
B	2	0	-	-
C	2	3	3	2
D	2	4	2	2

Tabela 4.10 – Pesquisador: FIT do Modelo 1.

A classe B que possui FIT nulo será incluída na lista LCOTI. Antes de passar para a próxima iteração, é necessário executar a atividade "Reduzir o fator de influência das classes ordenadas" para a classe B. O mesmo acontece com a classe A na segunda iteração. Na terceira iteração (FIT_3), não existem classes cujo valor de FIT seja nulo. Então, a próxima atividade a ser executada será "Priorizar as classes com menor FIT". São elas: C e D, com FIT_3 igual a 2. A lista de classes ordenadas para teste de integração seria LCOTI={ B, A, C, D, E}, após o processo *Gerar Lista Ordenada de Classes*.

O resultado da aplicação das heurísticas pelo pesquisador para o modelo 1 é mostrado na Tabela 4.11.

Ordem Estabelecida	$S1(G)_{heurísticas}$	$T1(G)_{heurísticas}$
B – A – C – D – E	1	3

Tabela 4.11 – Pesquisador: Resultados do Modelo 1.

4.4.3.3 – Modelo 1: Aplicação das Heurísticas pelos Participantes

Após a identificação da ordem das classes para execução dos testes de integração pelos participantes, foram obtidos: **S1(P)$_{heurísticas}$** (número de *stubs* específicos encontrados pelos participantes com a aplicação das heurísticas para o modelo 1) e **T1(P) $_{heurísticas}$** (tempo de identificação - em minutos - heurísticas, aplicados pelos participantes) apresentados nas Figuras 4.9 e 4.10, onde os resultados obtidos pelo pesquisador na seção 4.4.3.2 estão marcados na cor vermelha.

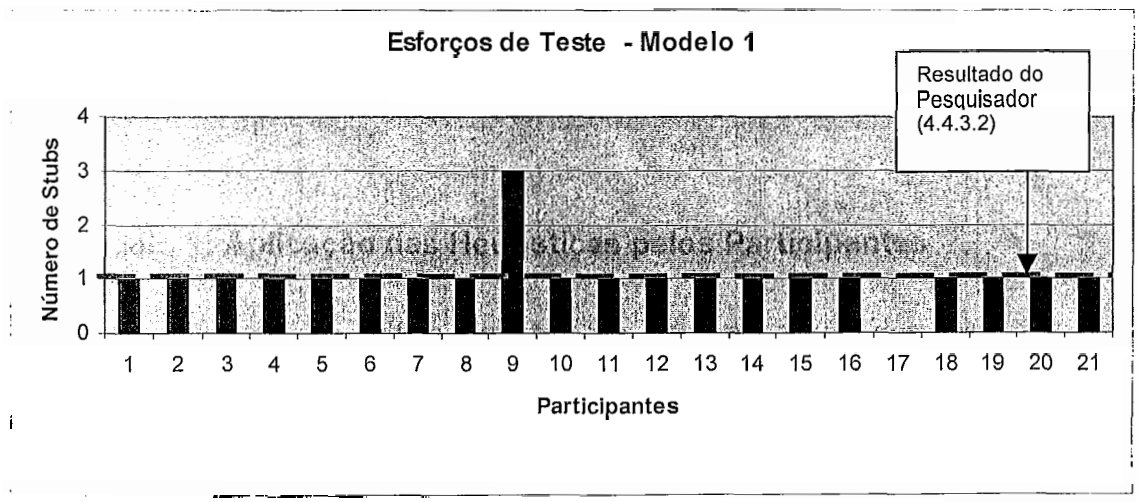


Figura 4.9 – Modelo 1: Esforços de Teste dos Participantes.

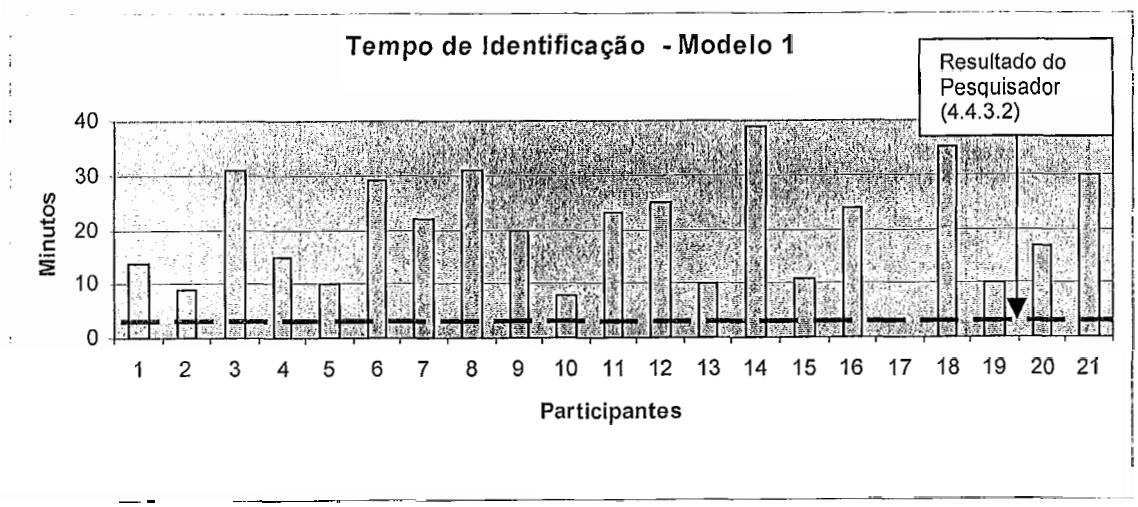


Figura 4.10 – Modelo 1: Tempo de Identificação dos Participantes.

4.4.3.4 – Modelo 2: Aplicação das Heurísticas pelo Pesquisador

O processo de *Pesquisar o Fator de Influência* é iniciado pela atividade "Aplicar critérios de precedência". Observa-se que, no modelo 2 (Figura 4.11), a classe A tem precedência sobre as classes B (critério da agregação) e C (critério da dependência); a classe B tem precedência sobre as classes C (critério da associação bidirecional), D (critério da herança) e E (critério da associação bidirecional); a classe C tem precedência somente sobre a classe B; a classe D não tem precedência sobre outras classes; a classe E tem precedência sobre as classes B e F; e a classe F tem precedência sobre as classes C e E.

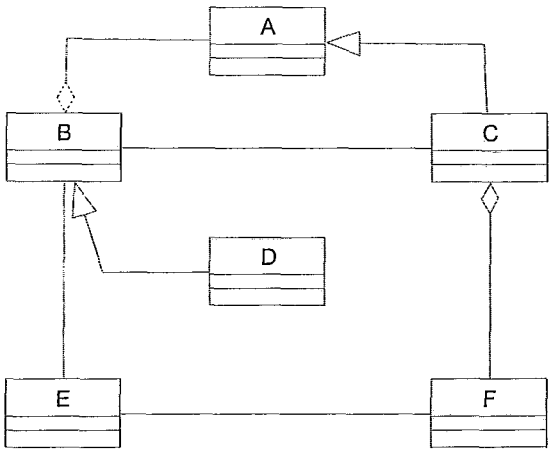


Figura 4.11 – Modelo 2: Diagrama de Classes.

Existem duas classes que devem ser integradas após a classe A, então o valor dois deve ser associado ao *fator de influência* (FI) da classe A durante a atividade "Atribuir FI". Os valores de FI apresentados na Tabela 4.12 são resultantes da execução das atividades descritas anteriormente para todas as classes do modelo, analogamente ao demonstrado para a classe A. A lista LCD é igual a {D} pois FI da classe D é nulo.

Classes	FI
A	2
B	3
C	1
D	0
E	2
F	2

Tabela 4.12 – Pesquisador: Fatores de Influência do Modelo 2.

A seguir é iniciado o processo *Tratar Fator de Integração Tardia* por meio da atividade de "Identificar precedências". A classe A não depende de outras classes; a classe B depende das classes A, C e E para ser integrada; C depende das classes A, B e F; E depende de B e F; e F depende de E. Durante a atividade "Calcular FIT", o valor zero é atribuído ao FIT da classe A. O resultado do FIT_1 (primeira iteração) calculado para a lista LCNO = {A, B, C, E, F} é mostrado na Tabela 4.13.

Classes	FI	FIT_1	FIT_2	FIT_3
A	2	0	-	-
B	3	5	3	3
C	1	7	5	3
E	2	5	5	3
F	2	2	2	-

Tabela 4.13 – Pesquisador: FIT do Modelo 2.

A classe A que possui FIT nulo será incluída na lista LCOTI. Antes de passar para a próxima iteração, é necessário executar a atividade "Reduzir o fator de influência das classes ordenadas" para a classe A. O mesmo acontece com a classe F na segunda iteração, pois possui menor FIT. Na terceira iteração (FIT_3), não existem classes cujo valor de FIT seja nulo. Então, a próxima atividade a ser executada será "Priorizar as classes com menor FIT". São elas: B, C e E, com FIT_3 igual a 3. A lista de classes ordenadas para teste de integração seria LCOTI={ A, F, C, E, B, D}, após o processo *Gerar Lista Ordenada de Classes.*

O resultado da aplicação das heurísticas pelo pesquisador para o modelo 2 é mostrado na Tabela 4.14.

Ordem Estabelecida	$S1(G)_{heurísticas}$	$T1(G)_{heurísticas}$
A – F – C – E – B - D	3	5

Tabela 4.14 – Pesquisador: Resultado do Modelo 2.

4.4.3.5 – Modelo 2: Aplicação das Heurísticas pelos Participantes

Após a identificação da ordem das classes para execução dos testes de integração pelos participantes, foram obtidos: $\mathbf{S2(P)_{heurísticas}}$ (número de *stubs*

específicos encontrados pelos participantes com a aplicação das heurísticas para o modelo 2) e **T2(P)** $_{heurísticas}$ (tempo de identificação - em minutos - heurísticas, aplicados pelos participantes) apresentados nas Figuras 4.12 e 4.13, onde os resultados obtidos pelo pesquisador na seção 4.4.3.4 estão marcados na cor vermelha.

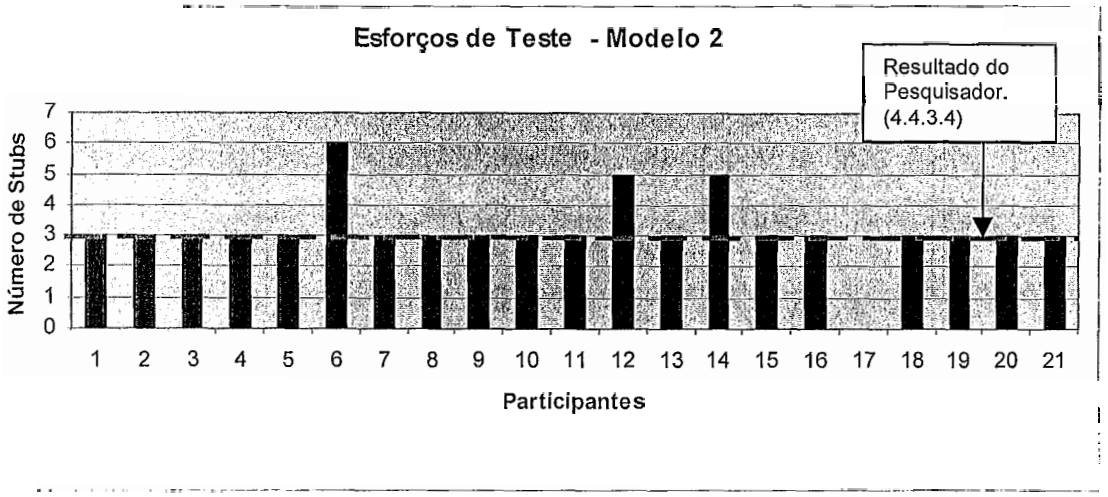


Figura 4.12 – Modelo 2: Esforços de Teste dos Participantes.

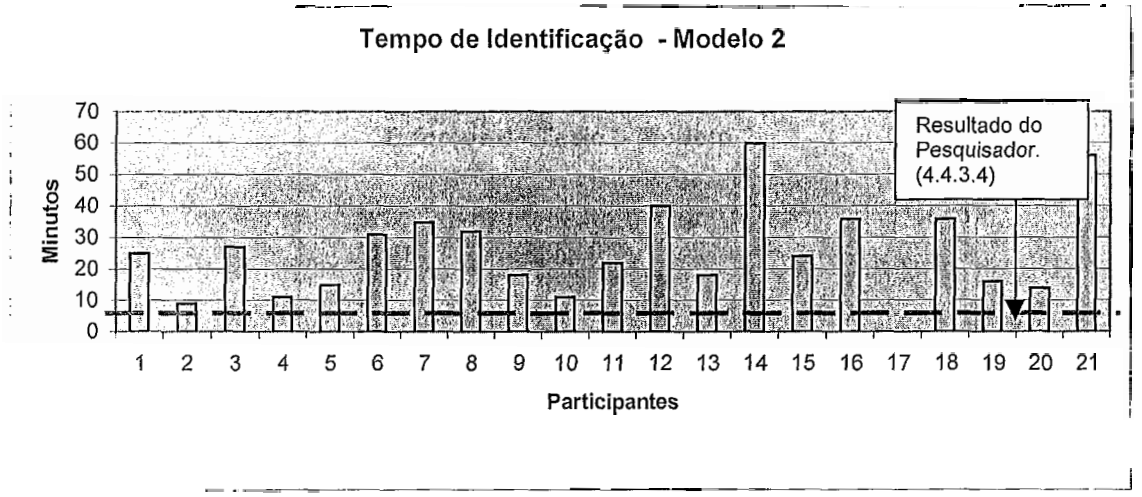


Figura 4.13 – Modelo 2: Tempo de Identificação dos Participantes.

4.4.4 – Análise dos Resultados

4.4.4.1 – Autotreinamento dos Participantes

O tempo médio e desvio padrão, calculados para o autotreinamento dos grupos, são apresentados na Tabela 4.15. Existe um acréscimo de 25% do tempo médio necessário para o autotreinamento do Grupo 2 em relação ao Grupo 1.

Grupo	Tempo Médio (minutos)	Desvio Padrão (minutos)	Coeficiente de Variação (%)
Academia (1)	78,73	28,50	36,20
Indústria (2)	98,67	28,77	29,15
Geral (1 e 2)	87,70	29,66	33,11

Tabela 4.15 – Tempos do Autotreinamento.

A análise qualitativa feita das informações referentes ao treinamento, obtidas no formulário de avaliação (Anexo C.4), permite observar que 45% dos participantes informaram apresentar algum tipo de dúvida na forma de apresentação. Entretanto, nenhum participante considerou o treinamento difícil.

4.4.4.2 – Teste das Hipóteses

Os esforços de teste (Mediana) calculados para os modelos 1 e 2 usando T-Test são apresentados na Tabela 4.16. A partir destes resultados pode-se perceber que:

- a hipótese nula (H0) – relativa ao modelo 1 – foi refutada uma vez que os participantes encontraram o mesmo esforço de teste que o pesquisador. Por conseqüência, a hipótese alternativa 1 (H1) na qual temos que **H1: $S1(P)_{heurísticas} - S1(G)_{heurísticas} = \varnothing$** foi confirmada.
- a hipótese nula (H0) – relativa ao modelo 2 – foi refutada uma vez que os participantes encontraram o mesmo esforço de teste que o pesquisador. Por conseqüência, a hipótese alternativa 1 (H1) na qual temos que **H1: $S2(P)_{heurísticas} - S2(G)_{heurísticas} = \varnothing$** foi confirmada.

	S1(G) heurísticas	S1(P) heurísticas	S2(G) heurísticas	S2(P) heurísticas
Valor testado (Mediana)	1	1	3	3
Número de casos =< Mediana	19	-	17	-
Número de casos > Mediana	1	-	3	-
Total de Casos	20	-	20	-
p-value	0	-	0,002	-

Tabela 4.16 – Resultados do 3º. Estudo de Caso.

Outra análise pode ser feita em relação aos tempos de identificação, considerando separadamente os valores dos Grupos 1 e 2, como apresentados na Tabela 4.17 e 4.18, referentes aos modelos 1 e 2. O Grupo 1 (Academia) reduziu a dispersão dos tempos de aplicação entre os modelos 1 e 2, indicando uma maior fixação das heurísticas.

Grupo	T1(P) Média	T1(P) Desvio Padrão	Coeficiente de Variação
Academia (1)	18,90	10,607	56,12
Indústria (2)	22,33	11,000	49,25

Tabela 4.17 – 3º. Estudo de Caso – Modelo 1: Tempo de Identificação.

Grupo	T2(P) Média	T2(P) Desvio Padrão	Coeficiente de Variação
Academia (1)	21,45	9,256	43,14
Indústria (2)	33,33	16,882	50,65

Tabela 4.18 – 3º. Estudo de Caso – Modelo 2: Tempo de Identificação.

4.4.4.3 – Análise Qualitativa

A análise qualitativa feita das informações referentes ao conjunto de heurísticas e seu processo de aplicação, obtidas no formulário de avaliação (Anexo C.4), revela que todos os participantes informaram que utilizariam as heurísticas em projetos futuros, entretanto 25% ressaltaram a importância de automatizar o processo. Não houve anotações de críticas às heurísticas.

4.4.5 – Lições Aprendidas

O estudo foi avaliado para identificar se o aprendizado sobre o processo de aplicação das heurísticas pelos participantes permite aos participantes estabelecerem esforços de teste iguais ou menores que o pesquisador. As Figuras 4.14 e 4.15 mostram que 95% dos participantes (Modelo 1) e 85% dos participantes (Modelo 2) obtiveram esforços de teste iguais aos esforços de testes do pesquisador para os mesmos modelos. Ressalta-se que nenhum participante encontrou esforços de teste menores do que o pesquisador.

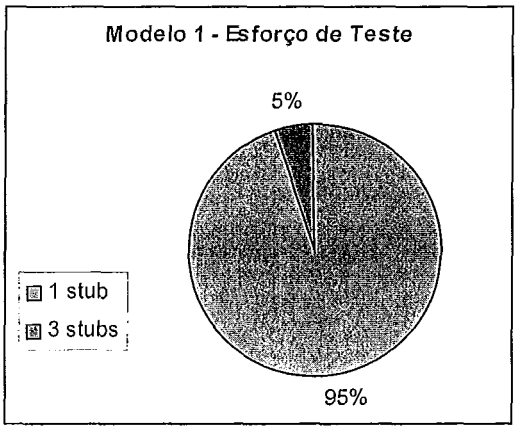


Figura 4.14 – Modelo 1: Esforço de Teste dos Participantes.

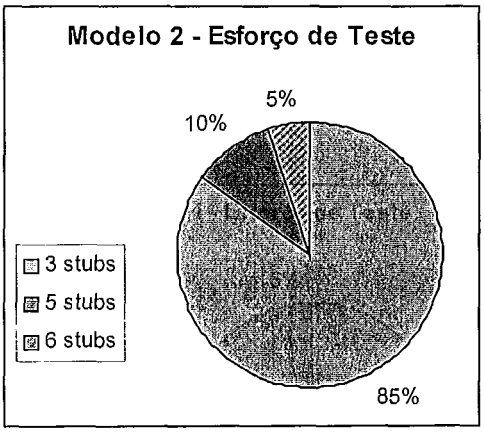


Figura 4.15 – Modelo 2: Esforço de Teste dos Participantes.

Foi identificada a necessidade de elaborar outro estudo de caso para identificar se o processo de aplicação das heurísticas é mais eficaz (em relação ao esforço de teste) do que os procedimentos *ad-hoc* conhecidos pelos participantes, conhecido o nível de partida (2^{O}. Estudo de Caso – pré-teste).

4.5 – 4^{O}. Estudo de Caso: Efetividade das Heurísticas (pós-teste)

4.5.1 – Definição dos Objetivos

4.5.1.1 – Objetivo Global

Caracterizar a efetividade de utilização das heurísticas para identificação da ordem das classes para execução dos testes de integração de um modelo. Para tal será feita uma comparação do esforço de teste necessário para a ordem das classes estabelecida utilizando o processo de aplicação das heurísticas (pós-teste) e o esforço

de teste utilizando processos *ad-hoc* em um mesmo modelo de software da área financeira (pré-teste – 2º. Estudo de Caso).

4.5.1.2 – Objetivo da Medição

Tendo como base o 2º. estudo de caso (pré-teste), caracterizar se o esforço de teste diminuiu quando empregado o processo de aplicação das heurísticas pelos participantes (pós-teste) para comprovar a efetividade das heurísticas.

4.5.1.3 – Objetivo do Estudo

Analisar o processo de aplicação das heurísticas para identificação da ordem das classes para execução de testes de integração OO

Com o propósito de caracterizar

Com respeito à aderência das heurísticas no que tange ao esforço de teste e ao esforço de identificação

Do ponto de vista de gerentes de software e engenheiros de software

No contexto do mesmo modelo do 2º. estudo de caso (Modelo 0).

4.5.1.4 – Questões

Q1: O processo de aplicação das heurísticas reduziu o esforço de teste para identificação da ordem das classes para execução dos testes de integração quando comparado ao esforço de teste necessário para a ordem estabelecida pelos participantes utilizando outros procedimentos para o mesmo modelo 0?

Métrica: Esforço de teste necessário para execução do teste de integração.

Q2: O processo de aplicação das heurísticas reduziu o esforço de identificação (tempo em minutos) da ordem das classes para execução dos testes de integração quando comparado ao esforço de identificação necessário para a ordem estabelecida pelos participantes utilizando outros procedimentos para o mesmo modelo 0?

Métrica: Esforço de identificação necessário para identificação de uma ordem.

4.5.2 – Planejamento do Estudo

4.5.2.1 – Seleção do Contexto

Para tentar eliminar o viés referente à aplicação das heurísticas sobre o mesmo modelo de projeto da área financeira utilizado no 2^{o}. Estudo de Caso foram adotadas as medidas:

- descaracterização do diagrama de classes – todas as classes foram renomeadas, com seus nomes originais substituídos por letras (A, B, C, etc);
- espaço de tempo – o 4^{o}. Estudo de Caso foi executado seis meses depois do 2^{o}., tempo significativo considerando que não se trata de domínio de uso pelos participantes.

4.5.2.2 – Definição das Hipóteses

4.5.2.2.1 – Em relação ao esforço de teste

Hipótese nula (H0): o esforço de teste obtido no pré-teste ('tratamento – procedimentos *ad-hoc*) não se altera no pós-teste (tratamento – heurísticas) pelos participantes no modelo 0?

$S0(P)_{ad\text{-}hoc}$: número de *stubs* específicos – tratamento procedimentos *ad-hoc*; e

$S0(P)_{heurísticas}$: número de *stubs* específicos – tratamento com heurísticas.

H0: $S0(P)_{ad\text{-}hoc} - S0(P)_{heurísticas} = \varnothing$

Hipótese alternativa (H1): no pós-teste o esforço de teste reduziu se comparado ao esforço de teste no pré-teste.

H1: $S0(P)_{ad\text{-}hoc} - S0(P)_{heurísticas} > \varnothing$

4.5.2.2.2 – Em relação ao Esforço de Identificação

Hipótese nula (H0): o esforço de identificação obtido no pré-teste ('tratamento – procedimentos *ad-hoc*) não se altera no pós-teste (tratamento – heurísticas) pelos participantes no modelo 0?

$T0(P)_{ad\text{-}hoc}$: tempo em minutos – tratamento procedimentos *ad-hoc*; e

$T0(P)_{heurísticas}$: tempo em minutos – tratamento com heurísticas.

H0: $T0(P)_{ad\text{-}hoc} - T0(P)_{heurísticas} = \varnothing$

Hipótese alternativa (H1): no pós-teste o esforço de identificação aumentou se comparado ao esforço de teste no pré-teste.

H1: $T0(P)_{heurísticas} - T0(P)_{ad\text{-}hoc} > \varnothing$

4.5.2.3 – Seleção dos Indivíduos

A mesma do 2º. Estudo de Caso. O participante com identificador 17 desistiu de participar dos estudos devido a necessidade de atender a outras prioridades.

4.5.2.4 – Variáveis

A variável independente é: o processo de aplicação das heurísticas utilizado pelo pesquisador.

As variáveis dependentes são:

1. A lista ordenada de classes para execução dos testes de integração.
2. O esforço de teste (número de *stubs* específicos).
3. O esforço de identificação (tempo em minutos).

4.5.2.5 – Validade do Estudo

A mesma do 2º. Estudo de Caso.

4.5.3 – Operação do Estudo

Após a identificação da ordem das classes para execução dos testes de integração pelos participantes utilizando o processo de aplicação das heurísticas, foram obtidos: $S0(P)_{heurísticas}$ (número de *stubs* específicos) e $T0(P)_{heurísticas}$ (tempo em minutos) apresentados nas Figuras 4.16 e 4.17, onde o resultado obtido pelo pesquisador na seção 4.3.3.2 está marcado na cor vermelha.

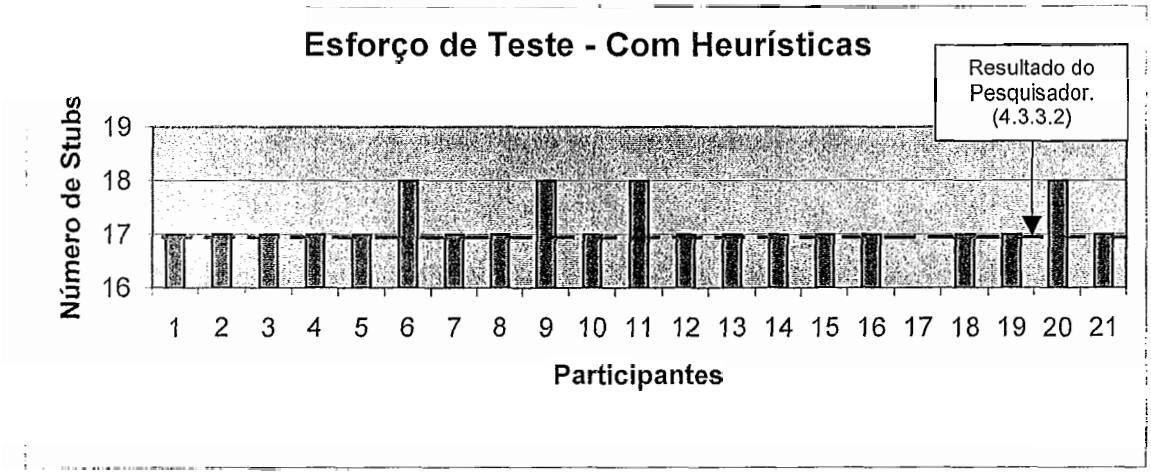


Figura 4.16 – Modelo 0: Esforço de Teste dos Participantes com Heurísticas.

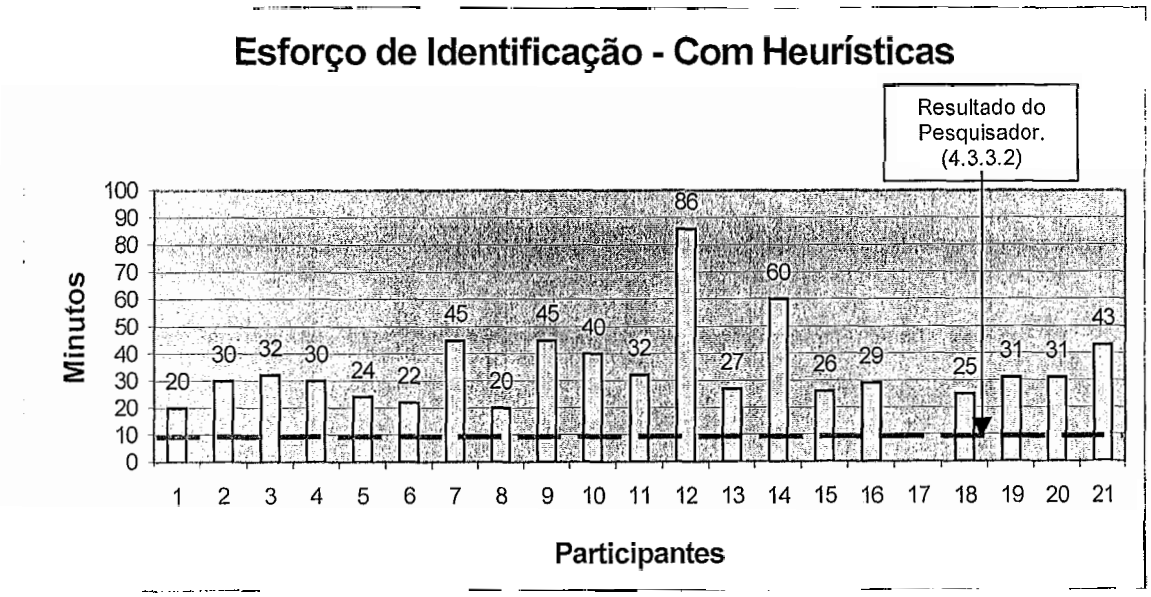


Figura 4.17 – Modelo 0: Esforço de Identificação dos Participantes com heurísticas.

4.5.4 – Análise dos Resultados

4.5.4.1 – Teste das Hipóteses

Os esforços de teste (medianas) e os esforços de identificação (médio e desvio padrão) calculados usando *Paired T-Test* (amostras emparelhadas) são apresentados nas Tabelas 4.19. A partir destes resultados pode-se perceber que:

- a hipótese nula (H0) – relativa ao esforço de teste – refutada uma vez que o esforço de teste no pós-teste foi menor. Por conseqüência, a hipótese alternativa 1 (H1) na qual temos que **H1: $S0(P)_{ad\text{-}hoc} - S0(P)_{heurísticas} > \varnothing$** foi confirmada.
- a hipótese nula (H0) – relativa ao esforço de identificação – refutada uma vez que o esforço de identificação foi menor no pós-teste. Por conseqüência, a hipótese alternativa 1 (H1) na qual temos que **H1: $T0(P)_{heurísticas} - T0(P)_{ad\text{-}hoc} > \varnothing$** foi confirmada.

Wilcoxon Signed Test	Número de Casos
$S0(P)_{ad\text{-}hoc} > S0(P)_{heurísticas}$	5
$S0(P)_{ad\text{-}hoc} = S0(P)_{heurísticas}$	15
$S0(P)_{ad\text{-}hoc} < S0(P)_{heurísticas}$	0
p-value	0,000

Tabela 4.19a – Resultados do Pré e Pós-teste: Esforço de Teste.

	$T0(P)_{ad\text{-}hoc}$ (minutos)	**$T0(P)_{heurísticas}$ (minutos)**
Média	18,60	34,90
Desvio Padrão	8,970	15,620

Tabela 4.19b – Resultados do Pré e Pós-teste: Esforço de Identificação.

Outra análise dos tempos de identificação pode ser feita, considerando separadamente os valores dos Grupos 1 e 2, como mostrados na Tabela 4.20. Os coeficientes de variações dos tempos de identificação para os grupos diminuíram no pós-teste, indicando uma menor dispersão dos tempos.

Processo	***Esforço de Identificação* T0(P)**	**Academia**	**Indústria**
Ad-hoc	Média	20,91	15,78
	Desvio Padrão	7,11	10,57
	Coeficiente de Variação	33,98	66,98
Heurísticas	Média	30,91	39,78
	Desvio Padrão	9,21	20,61
	Coeficiente de Variação	29,81	51,81

Tabela 4.20 – 4º. Estudo de Caso: Esforço de Identificação por Grupo.

4.5.5 – Lições Aprendidas

O estudo foi avaliado somente na efetividade do processo de aplicação das heurísticas. A Figura 4.18 mostra que 80% (16) dos participantes obtiveram um esforço de teste igual ao esforço de teste do pesquisador com o processo de aplicação das heurísticas para o mesmo modelo.

Analisando os 20% restantes (4 participantes), foi observado que, apesar de não terem encontrado o mesmo valor do pesquisador, não tiveram seus esforços de teste aumentados em relação ao pré-teste. Dois destes participantes também não apresentaram bons desempenhos durante o 3º. Estudo de Caso (Autotreinamento das Heurísticas), o que pode estar refletindo algum mau entendimento das heurísticas.

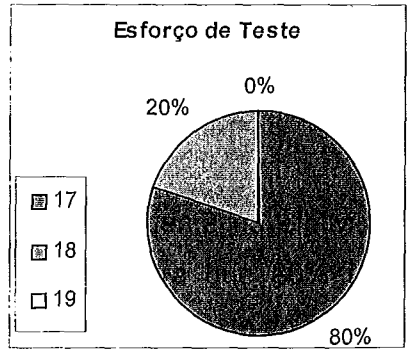


Figura 4.18 – 4º. Estudo de Caso – Esforço de Teste com Heurísticas.

Foi identificado que os outros dois participantes partiram de valores de FI e FIT_1 (primeira iteração) corretos, entretanto não produziram o resultado esperado com o processo de aplicação das heurísticas devido a erros de cálculo durante o processo e não por falta de entendimento do mesmo.

Não houve participante cujo valor do esforço de teste tenha sido aumentado com o processo de aplicação das heurísticas (pós-teste) em relação ao pré-teste.

Foi observado que houve uma redução no coeficiente de variação do esforço de identificação de ambos os grupos (Academia e Indústria) no pós-teste. Indica que apesar do aumento do esforço de identificação médio com o processo de aplicação das heurísticas houve uma menor dispersão entre os participantes dos grupos. A diferença média dos esforços de identificação foi de 15 minutos (Figura 4.19).

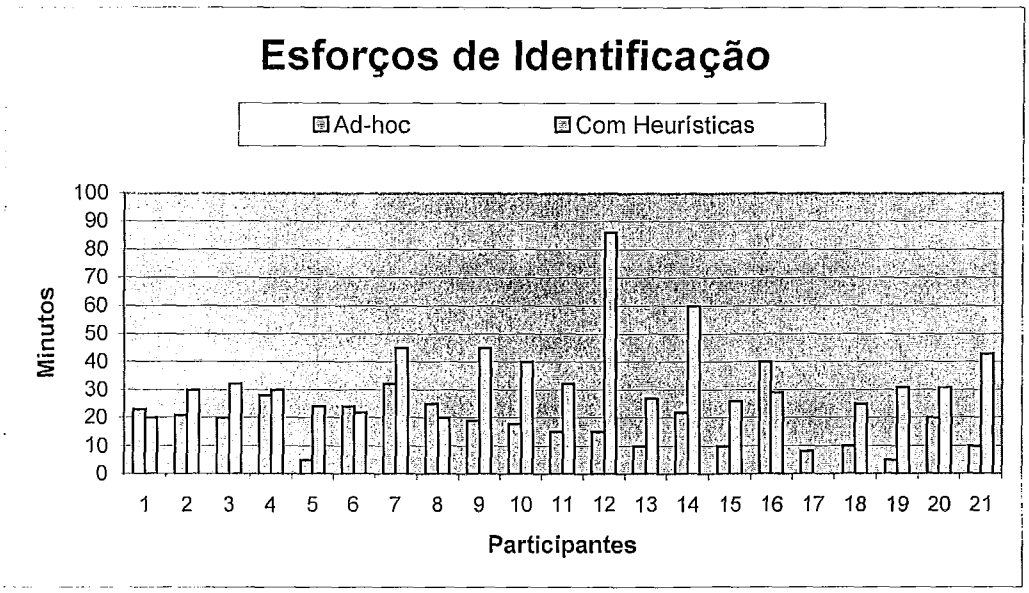


Figura 4.19 – Resultados do Pré e Pós-teste: Diferença de Esforços.

4.6 – Considerações Finais

Este capítulo fez uma breve introdução à experimentação destacando alguns benefícios no campo da Engenharia de Software e ressaltando sua aplicabilidade nesta tese para reduzir incertezas sobre a efetividade do conjunto de heurísticas e seu processo de aplicação.

Feito isto, foram apresentados quatro estudos de caso definidos e planejados para caracterizar a viabilidade do processo de aplicação das heurísticas na identificação da ordem das classes de um modelo orientado a objetos (diagrama de classes UML) com um esforço de teste mínimo, apresentada no Capítulo 3. Por meio de suas execuções e da análise dos resultados, foi possível identificar gradualmente a efetividade das heurísticas.

A estratégia proposta por BRIAND *et al.* (2001) obteve resultados melhores que outras estratégias – KUNG *et al.* (1995a); TAI e Daniels (1999); e LE TRAON *et al.* (2000). Portanto, obter uma evidência de um resultado melhor (menor esforço de teste) com o uso do processo de aplicação das heurísticas (1°. estudo de caso) foi fundamental para a viabilidade destas e continuidade deste trabalho.

O 2°. estudo de caso permitiu conhecer que, no contexto utilizado, não foram aplicados e/ou relatados outros procedimentos, pelos participantes, que permitam encontrar resultados iguais ou melhores do que os obtidos pelo pesquisador utilizando as heurísticas propostas. Além disto, serviu de base (pré-teste) na comparação destes

resultados (procedimentos *ad-hoc*) com aqueles obtidos pelos desenvolvedores de software e gerentes de projetos com a aplicação do processo das heurísticas. Reforçado pelos desempenhos obtidos no 3º. estudo de caso (outros dois modelos) pelos participantes, pode-se perceber que o processo de aplicação das heurísticas é facilmente compreendido e permite aos usuários uma convergência para um número mínimo de *stubs*, diminuindo o esforço de teste.

Os estudos, como um todo, demonstraram que a utilização do conjunto de heurísticas e seu processo de aplicação permitem a identificação da ordem de integração das classes em testes aplicados a software OO, a partir do diagrama de classes UML, que minimize o esforço de teste. Além disso, outras contribuições relevantes foram obtidas com análises qualitativas dos estudos, como: o interesse dos participantes no uso das heurísticas em projetos futuros e sua facilidade de assimilação (considerando os percentuais de acertos nos três modelos em que foram aplicadas após o autotreinamento: 95%, 85% e 80%).

Entretanto, não foi possível identificar qualquer indício em relação ao desempenho individual no aprendizado e utilização das heurísticas com algum fator de caracterização do participante, como: experiência prática em software ou conhecimento prévio dos conceitos de orientação a objetos e UML. Este problema pode estar relacionado a possível imprecisão das informações prestadas pelos participantes em relação ao próprio conhecimento, feita de maneira subjetiva. Entretanto, a possibilidade de submeter os participantes a um teste objetivo para avaliar conhecimentos não foi considerada inicialmente.

Outro problema detectado foram os erros de cálculo cometidos por alguns participantes durante o 4º. Estudo de Caso, em virtude do aumento do número de classes e complexidade dos diagramas de classes. Percebeu-se que há necessidade da automatização do processo. Uma ferramenta poderia evitar tais erros, bem como, motivar a utilização do conjunto das heurísticas para sugerir a seqüência de classes para os testes de integração, o que atenderia também às solicitações dos participantes, identificadas durante a análise qualitativa dos estudos.

Cabe ressaltar que, embora os estudos realizados tenham demonstrado que o conjunto de heurísticas e o processo de aplicação são viáveis, é necessário que outros estudos sejam executados com propósito de analisar a abrangência destas em outros contextos, como a aplicação em projetos de maior escala (grande número de classes) e maior número de participantes.

Por fim, como proposto por SHULL *et al.* (2001), a utilização de estudos observacionais permitiu ao pesquisador não somente evidências que justificassem a continuidade de caracterização do processo de aplicação das heurísticas, mas

também a possibilidade de identificar novas questões, como, por exemplo, a influência da complexidade dos *stubs* em relação ao esforço de teste. Esta contribuição pôde, então, ser modelada como requisito da ferramenta que automatizou o processo, apresentada no Capítulo 4.

Capítulo 5

FAROL: Automação do Processo de Aplicação das Heurísticas.

Neste capítulo é apresentado o protótipo da ferramenta construída para identificar uma ordem de integração de classes de um projeto orientado a objetos empregando o processo de aplicação das heurísticas, formalizado no Capítulo 3.

5.1 – Introdução

Buscando atender a necessidade apontada por engenheiros de software e gerentes de projeto durante os estudos experimentais apresentados no Capítulo 4, a ferramenta FAROL[2] foi definida e implementada. FAROL baseia-se fundamentalmente nas heurísticas e no respectivo processo de aplicação apresentados no Capítulo 3, permitindo identificar uma ordem de integração das classes de um projeto de software orientado a objetos, a partir do seu diagrama de classes UML.

Devido a grande variedade de ferramentas CASE para representação de diagrama de classes (UML) – da estrutura lógica estática da coleção de elementos de um modelo (classes, tipos e seus respectivos conteúdos e relações) – existentes no mercado: ROSE (www.ibm.com/software/rational/), Together (www.borland.com.br/together/), ArgoUML (www.tigris.org/argouml/) e Poseidon (www.gentleware.com), entre outras, optou-se pela construção de uma solução independente destas aplicações.

Como XMI (*XML Metadata Interchange*) é um padrão XML que provê um formato para descrição de elementos do modelo UML – conceitos de orientação a objetos integrados ao XML – e está disponível para muitas ferramentas UML, diretamente ou com auxílio de *plug-ins*, XMI foi adotado como solução para o intercâmbio de dados entre ferramentas CASE e FAROL.

[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.

O diagrama apresentado na Figura 5.1 mostra, de forma simplificada, os componentes que compõem a ferramenta. FAROL é constituída pelos componentes Tradutor XMI; Ordenador; Avaliador; e de Exteriorização.

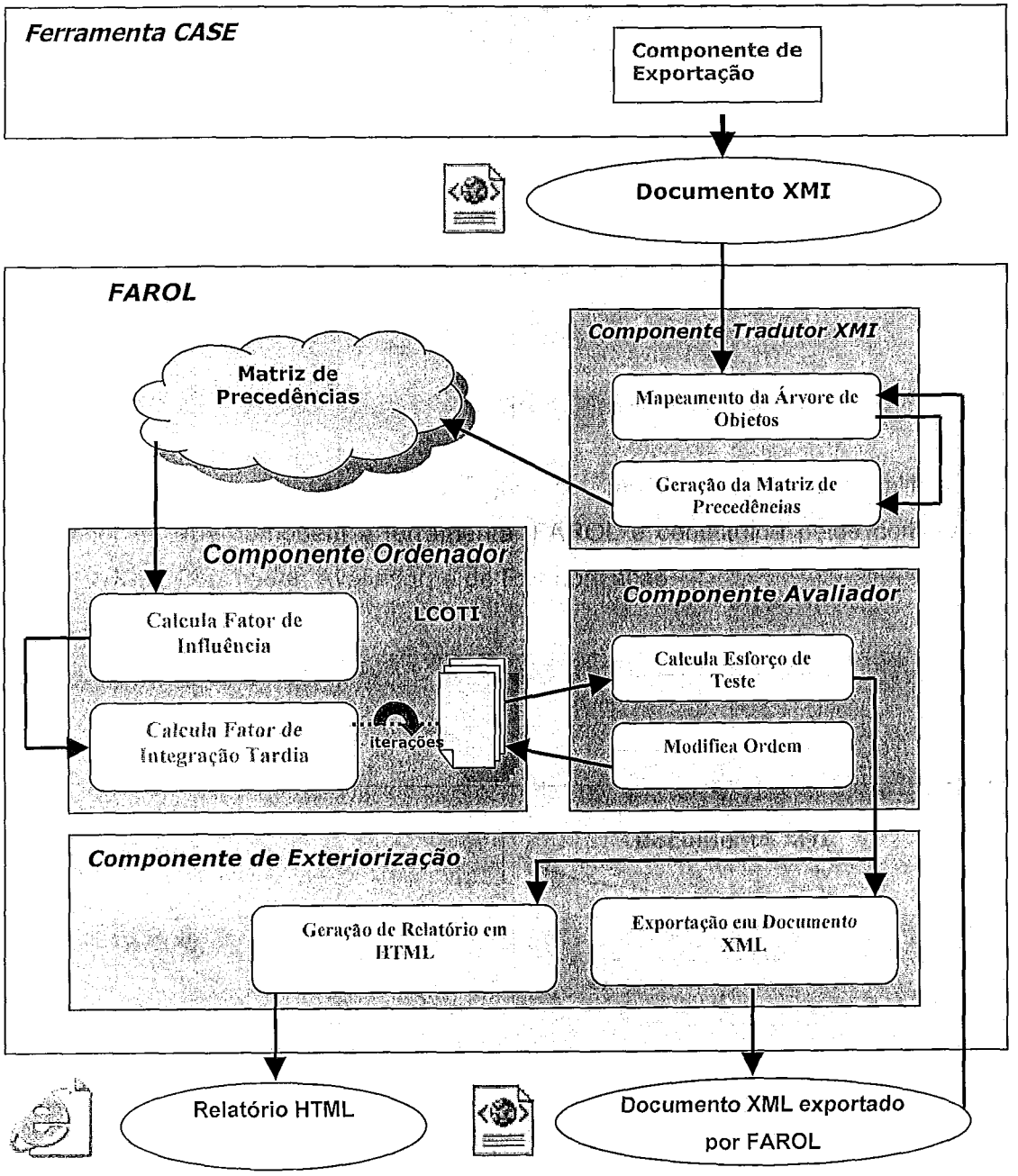


Figura 5.1 – Componentes Implementados por FAROL.

O componente Tradutor XMI por trabalhar diretamente sobre arquivos com sintaxe XMI, necessita de um *parser* que disponibilize uma interface para manipulação dos elementos do documento. Devido à simplicidade das informações a serem traduzidas do arquivo XMI de entrada foi implementado um *parser* especialmente para obter estas informações. O diagrama apresentado na Figura 5.2 mostra, de forma

simplificada, as classes adotadas, quais métodos foram implementados em cada uma delas e de qual componente externo elas dependem para execução de suas funcionalidades.

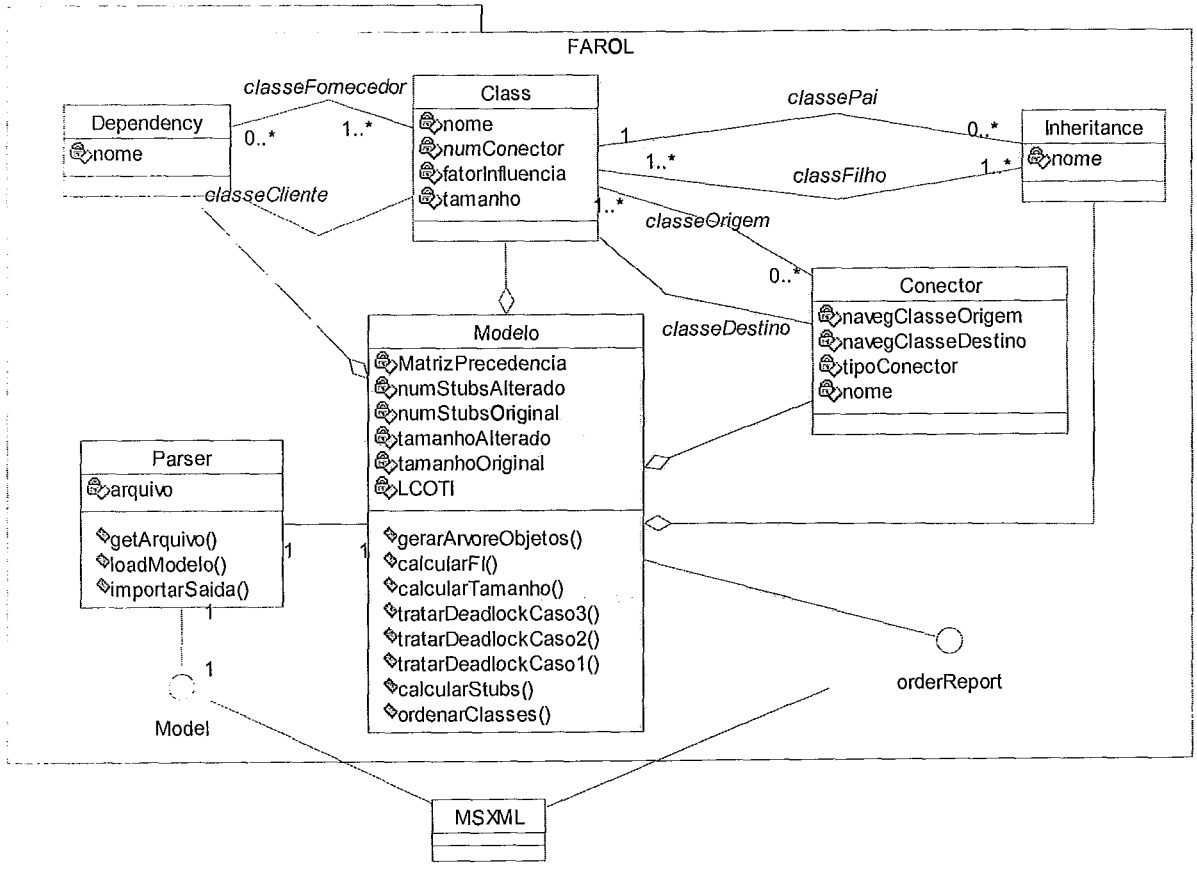


Figura 5.2 – Diagrama de Classes do FAROL

Além desta introdução, na qual foram apresentados os componentes implementados no protótipo para identificar uma ordem de integração de classes usando as heurísticas, este capítulo também descreve uma pequena revisão da tecnologia XMI, detalhes técnicos da implementação do FAROL, as atribuições de cada componente junto com um exemplo de utilização do protótipo implementado.

A ferramenta FAROL foi desenvolvida na linguagem C++ utilizando o ambiente de programação Borland C++ Builder 6.0 na plataforma Windows. A escolha deste ambiente de programação se deu por ele ser bastante intuitivo para a criação de interfaces com o usuário e, principalmente, por prover diversos componentes para manipulação de arquivos no formato XML. Outro fator preponderante na escolha do ambiente C++ Builder foi a disponibilidade da licença acadêmica do produto para LENS/COPPE/UFRJ.

5.2 – Mapeando UML para XMI

O XMI é um formato padrão recomendado pela OMG (*Object Management Group*) que tem como objetivo o intercâmbio de dados possibilitando o compartilhamento de modelos entre ferramentas diferentes, sendo utilizado para integrar três padrões: XML, UML e MOF (*Meta Objects Facility)* (BRYAN, 1992) (CARLSON, 2001).

A abordagem tradicional de metamodelagem está baseada numa arquitetura de quatro níveis. A primeira camada (M0) é conhecida como camada dos dados. A segunda (M1), conhecida como metadados, refere-se aos esquemas, possuindo dados que contém informações dos dados que descrevem as informações da camada de dados imediatamente abaixo. A terceira camada (M2) é conhecida como camada dos meta-metadados e refere-se aos modelos, contendo as descrições que definem a estrutura e semântica dos metadados. Nesta camada está o modelo UML. A quarta camada (M3) contém as descrições que definem a estrutura e semântica dos meta-metadados. Conhecida como meta-meta-metadados, contém o modelo que descreve todos os outros metamodelos. A Figura 5.3 mostra uma representação da arquitetura.

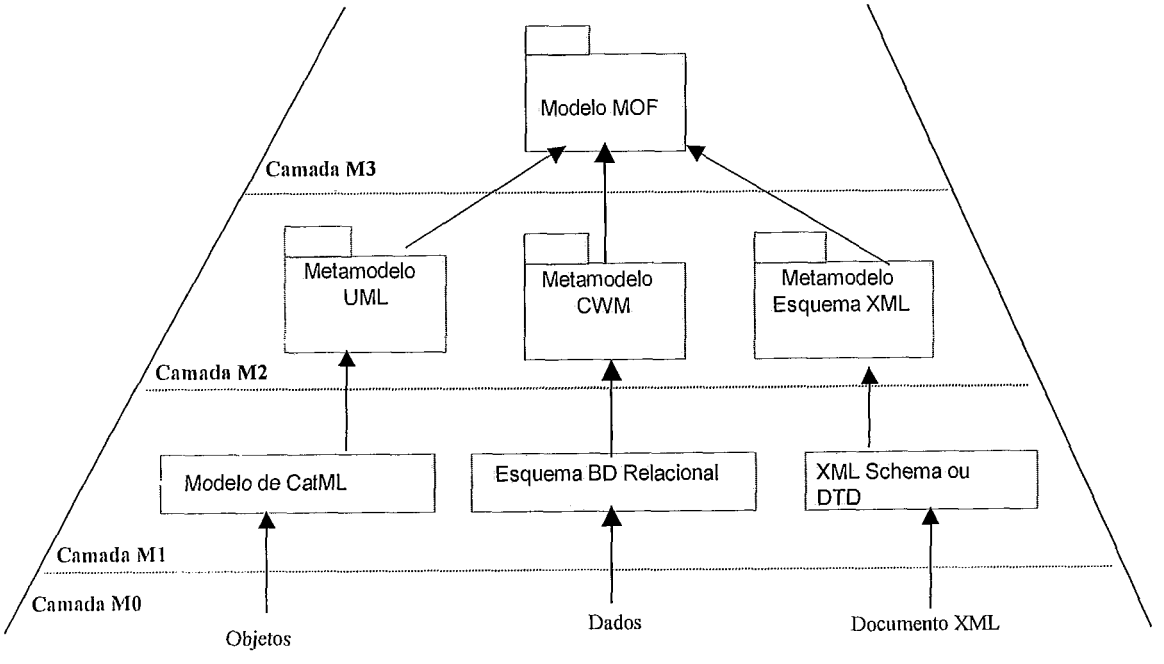


Figura 5.3 – Camadas da Arquitetura MOF [CARLSON, 2001].

Nesta última camada está o MOF (*Meta Object Facility*), criado pela OMG, que especifica um modelo capaz de descrever qualquer outro metamodelo. Como esta camada é capaz de entender todas as outras camadas será necessário somente implementar uma interpretação para o MOF. Segundo esta concepção, o metamodelo de cada ferramenta seria uma instância do metamodelo MOF.

A principal função do XMI é realizar o intercâmbio entre objetos e meta-objetos modelados em MOF, utilizando o formato do XML. Ele utiliza o MOF como metamodelo e o XML como linguagem padrão para formatação dos dados a serem transferidos e faz o mapeamento de MOF para XML. Além de definir um padrão para a representação dos elementos do MOF em XML, o XMI define também um conjunto de metadados específicos para a descrição dos meta-objetos MOF. Todos os elementos do XML definidos pelo XMI estão descritos em "http://www.omg.org/xmi".

O padrão XML proporciona uma sintaxe genérica usada para marcação de dados através de *tags* simples e legíveis a humanos. Em documentos XML, os dados são incluídos como *strings* de texto entre marcações que descrevem esses mesmos dados. A unidade básica de dados é chamada de elemento. A especificação XML define exatamente a sintaxe que as marcações devem seguir: como os elementos são delimitados pelas *tags*, quais nomes são aceitos para os elementos, onde os atributos são colocados e assim por diante. Documentos que satisfazem a essa gramática são chamados bem-formados e aqueles que não o fazem, não são permitidos.

A marcação, em um documento XML, descreve a estrutura do documento mostrando as associações entre os elementos. Se este for bem-formado, a marcação também pode descrever a semântica do documento. Por exemplo, uma marcação pode indicar se o elemento é uma data, uma pessoa ou um código de barras. Uma das principais características do XML é a de ser uma linguagem de meta marcação, o que significa que não existe um conjunto pré-definido de *tags* e elementos que deve ser utilizado por todos e em todas as áreas. XML é uma linguagem de marcação estrutural e semântica, e não uma linguagem de apresentação.

DTD (*Data Type Definition*) consiste em um documento no formato de uma gramática que contem a descrição formal da estrutura válida para um documento XML, ou seja, um arquivo XML associado a um DTD possui uma estrutura de nós (tags) de acordo com o que foi especificado no DTD associado. Para a implementação do componente Tradutor XMI, foi escolhido o XMI DTD 1.0 para os diagramas da UML na sua versão 1.3 (disponível em http://www.uml.org) devido ao fato desta versão estar disponível na maioria das ferramentas CASE de modelagem por meio do diagrama UML. Um DTD completo da UML para todos os diagramas na versão 1.3 pode ser obtido no endereço http://www.omg.org/cgi-bin/doc?formal/02-07-01.

Para exemplificar, o modelo contendo um relacionamento de dependência entre a classe A e a classe B, representado no diagrama de classes UML da Figura 5.4, foi exportado para um arquivo XMI conforme ilustrado pela Figura 5.5.

Figura 5.4 – Representação UML para Dependência entre classes.

```
<!-- ========== Dependencia [Model] ========== -->
 <UML:Model xmi.id="G.0" name="Dependencia"
      visibility="public" isSpecification="false" isRoot="false"
      isLeaf="false" isAbstract="false">
   <UML:Namespace.ownedElement>
     - <!-- ========== Dependencia::A [Class] ========== -->
      <UML:Class xmi.id="S.130.1102.05.1" name="A"
          visibility="public" isSpecification="false" isRoot="true"
          isLeaf="true" isAbstract="false" isActive="false"
          namespace="G.0" clientDependency="G.1" />
     <!-- ========== Dependencia::B [Class] ========== -->
      <UML:Class xmi.id="S.130.1102.05.2" name="B"
          visibility="public" isSpecification="false" isRoot="true"
          isLeaf="true" isAbstract="false" isActive="false"
          namespace="G.0" supplierDependency="G.1" />
     <!-- ========== Dependencia::{A?B}{409F8B440013} [Dependency] --
       >
      <UML:Dependency xmi.id="G.1" name="" visibility="public"
          isSpecification="false" client="S.130.1102.05.1"
          supplier="S.130.1102.05.2" />
   </UML:Namespace.ownedElement>
 </UML:Model>
```

Figura 5.5 – Mapeamento de dependência entre classes para XMI.

A seguir será apresentado o mapeamento utilizado pelo FAROL na tradução das informações do arquivo XMI de entrada.

5.3 – Tela Principal de FAROL

A tela principal da ferramenta FAROL é composta basicamente por cinco áreas, conforme apresentado na Figura 5.6. As áreas são:

Figura 5.6 – FAROL: Tela Principal.

A) Menu Principal e Barra de ferramentas:

O Menu Principal é composto por cinco principais itens: Funções, Opções, Configurações, Ajuda e Sair.

- **Funções:** Contém as funcionalidades referentes à identificação da ordem de integração de classes. É composta pelos subitens: Abrir arquivo XMI; Gerar ordenação; e Fechar arquivo.
- **Opções:** Contém as funcionalidades que permitem interagir com um modelo de classes, porém não fazem parte das heurísticas. É composta pelos subitens: Salvar Resultado; Importar Resultado; e Imprimir Resultado.
- **Configurações:** Contém as funcionalidades de configuração da ferramenta Farol. É composta pelos subitens: Painel de Modelo de Classes (permite habilitar ou desabilitar o painel); Painel de Seqüência de Ordenação (permite habilitar ou desabilitar o painel); e Mudar Idioma (permite escolher entre Português e Inglês).
- **Ajuda:** Contém informações que auxiliem os usuário nas suas tarefas. É composto pelos subitens: Conteúdo e Sobre, apresentado na Figura 5.7.

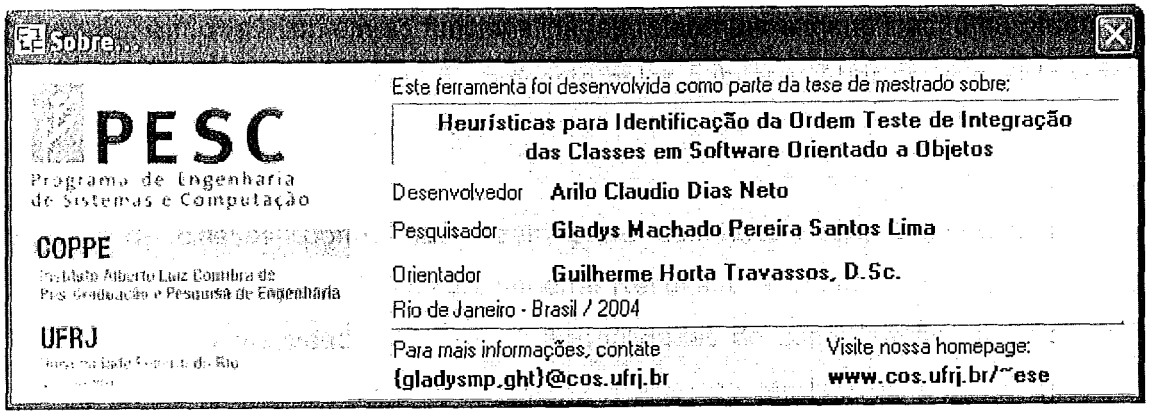


Figura 5.7 – FAROL: Formulário Sobre.

- **Sair:** Finaliza a execução da ferramenta.

A Barra de Ferramentas é composta por alguns elementos do menu, funcionando como atalho para as principais funcionalidades do sistema, apresentada na Figura 5.8.

Figura 5.8 – FAROL: Barra de Ferramentas.

(B) Painel de Modelo de Classes:

O Painel de Modelo de Classes, mostrado na Figura 5.9, contém as informações extraídas de um diagrama de classes UML que são relevantes para o cálculo das 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.

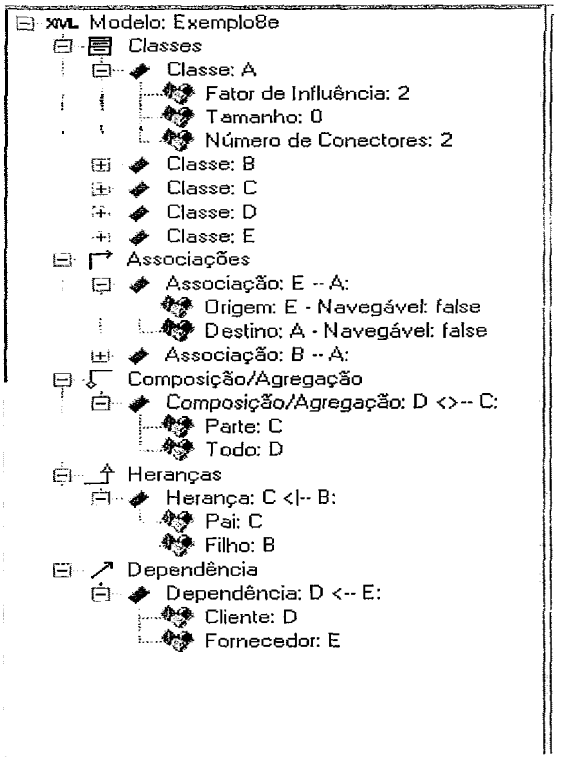


Figura 5.9 – FAROL: Painel de Modelo de Classes.

(C) Painel de Seqüência de Ordenação:

Painel de Seqüência de Ordenação mostra os valores de FI e FIT calculados durante o processo de aplicação das heurísticas para a lista de classes não ordenadas (LCNO), conforme apresentado na Figura 5.10.

Sequência de Ordenação

Classe / Iteração	Valor de FI	FIT - Iteração 1	FIT - Iteração 2	FIT - Iteração 3
	2	2	2	2
	1	4	2	2
	2	0	-	-
	1	2	0	-
	1	3	3	2

Figura 5.10 – FAROL: Painel de Seqüência de Ordenação.

(D) Lista Ordenada de Classes:

A Lista Ordenada de Classes é obtida pelo processo de aplicação das heurísticas. A Figura 5.11 apresenta uma Lista Ordenada de Classes para teste de integração (LCOTI).

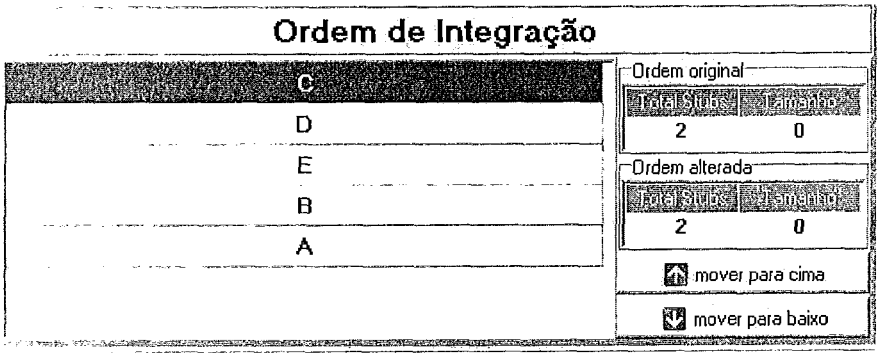


Figura 5.11 – FAROL: Lista Ordenada de Classes.

(E) Esforço de Teste e Modificação da Lista Ordenada:

A ferramenta apresenta o número de *stubs* e tamanho que representam o esforço de teste original. Esforço de teste é o número de *stubs* necessário para integrar e testar as classes na ordem de integração sugerida. A Figura 5.12 destaca a área da tela que exibe o esforço de teste e permite a modificação na Lista Ordenada de Classes (LCOTI).

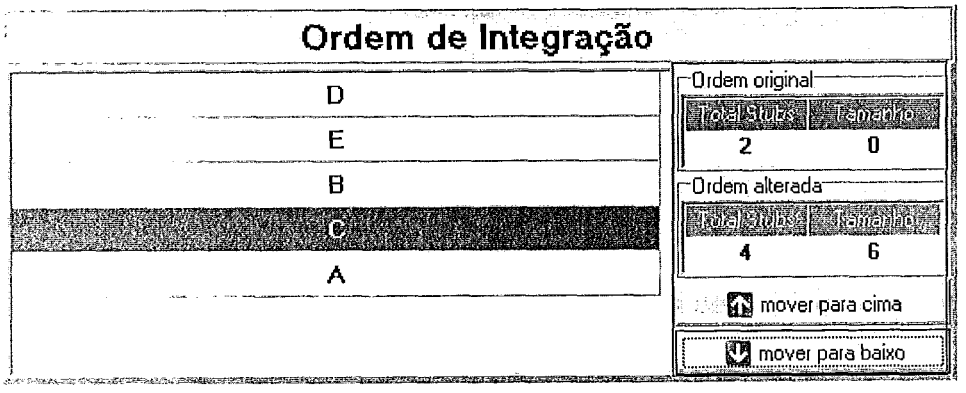


Figura 5.12 – FAROL: Esforço de Teste.

5.4 – Componente Tradutor XMI

Este componente é responsável pela tradução do arquivo de entrada, reconhecendo os objetos do diagrama de classes UML de acordo com o XMI DTD 1.0 para UML versão 1.3. O resultado do mapeamento é apresentado no Painel de Modelo de Classes (Figura 5.13). Será utilizado o diagrama de classes apresentado na Figura 5.11 para descrever as funcionalidades de componente *Tradutor XMI*.

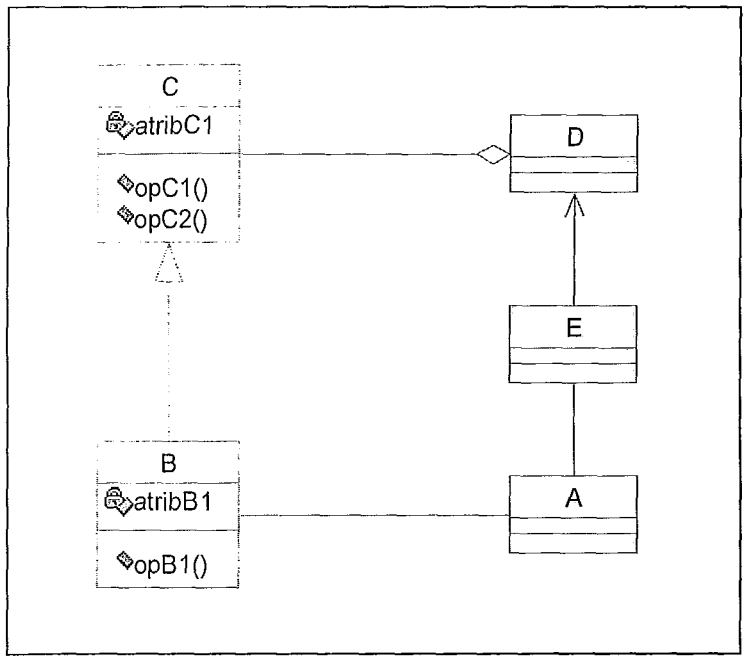


Figura 5.13 – Diagrama de Classes Exemplo.

Os objetivos deste componente são basicamente dois:

1. Reconhecer os elementos relevantes à aplicação das heurísticas e que compõem o diagrama de classe selecionado por meio da ferramenta, e;
2. Gerar uma matriz de precedência entre as classes que compõem o diagrama para que possam ser aplicadas às heurísticas de ordenação das classes descritas na seção 5.4.2.

5.4.1 – Reconhecendo elementos do diagrama UML

A partir do diagrama de classe selecionado por meio da ferramenta serão identificados os principais elementos que o compõem e que serão aplicados nos cálculos das heurísticas descritas no Capitulo 3. Estes elementos são:

- **Classes:** serão identificadas as classes que compõem o diagrama selecionado, e para cada classe serão identificadas as seguintes informações: nome da classe, tamanho (número de atributos + número de operações) e quantidade de conectores associados à classe (estes conectores podem ser associações, heranças, agregação/composição ou dependência). Neste momento é construída a Lista de Classes Não Ordenadas (LCNO).

- **Associações:** serão identificadas as associações que compõem o diagrama selecionado, e para cada associação serão identificadas as seguintes informações: classe de origem, classe de destino, navegabilidade da classe de origem e navegabilidade da classe de destino.
- **Composição/agregação:** serão identificadas as composições/agregações que compõem o diagrama selecionado, e para cada composição/agregação serão identificadas as seguintes informações: classe que representa o "todo", classe que representa a "parte".
- **Dependências:** serão identificadas as dependências que compõem o diagrama selecionado, e para cada dependência serão identificadas as seguintes informações: classe fornecedora, classe cliente.

A partir dessas informações, o passo seguinte consiste na criação da matriz de precedência.

5.4.2 – Criação da Matriz de Precedências

A matriz de precedência tem como objetivo principal a transformação de um modelo em uma simples matriz que indique as precedências de cada classe sobre as demais. A matriz de precedências é uma matriz bidimensional (NxN), onde N é o número de classes no diagrama. 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; caso contrário, seu valor será 0 (zero). Desta forma, numa coluna estão representadas todas classes que têm precedência sobre a classe representada na coluna.

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

Tabela 5.1 – Matriz de precedências para o diagrama da Figura 5.13.

A matriz de precedências correspondente ao diagrama de classes apresentado na Figura 5.11 está descrita na Tabela 5.1. 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.

A Figura 5.12 apresenta a tela de FAROL após a execução do componente *Tradutor XMI* para o exemplo descrito na Figura 5.14.

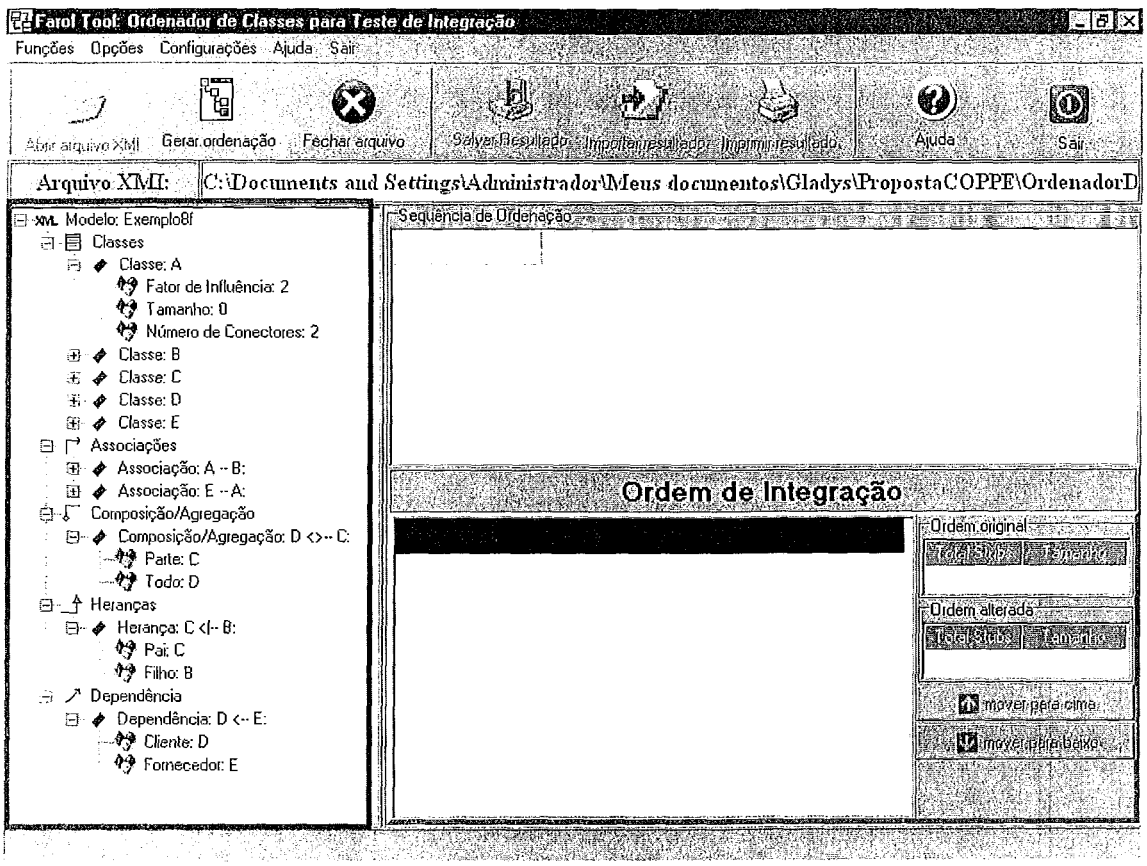


Figura 5.14 – FAROL: Traduzindo o XMI do Exemplo.

5.5 – Componente Ordenador

Este componente é responsável pelo processo de aplicação das heurísticas. Todos os cálculos relativos às heurísticas são obtidos a partir da matriz de precedências descrita na seção anterior. Os cálculos a serem obtidos são:

- **Fator de Influência (FI):** o FI para uma determinada classe consiste, simplesmente, na quantidade de colunas marcadas na matriz de precedências para uma determinada classe. Por exemplo, o FI da classe C na Tabela 5.1 é 2, pois as colunas B e D estão marcadas. O mesmo deve ser 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 (FIT):** o FIT para uma classe consiste, simplesmente, no somatório do FI das classes marcadas na matriz de precedências na coluna referente à classe em questão. Por exemplo, o FIT para a classe B na Tabela 5.1 é 4, pois na coluna da classe B estão marcadas as classes A, cujo FI é 2, e C, cujo FI também é 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.

Ao final dessas iterações será obtida a Lista de Classes Ordenadas para Teste de Integração (LCOTI). A tela de FAROL após a execução do componente Ordenador (e conseqüentemente do componente *Avaliador* que ainda será descrito) é apresentada na Figura 5.15, destacando a área associada a esse componente.

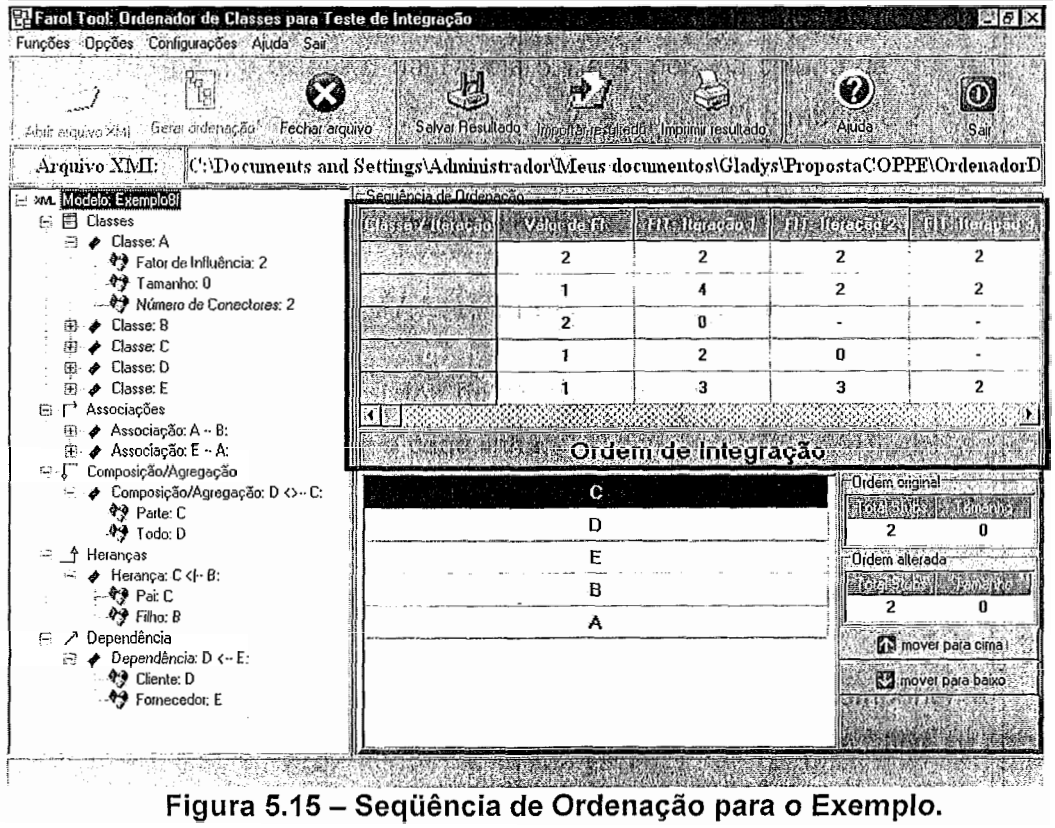


Figura 5.15 – Seqüência de Ordenação para o Exemplo.

5.6 – Componente Avaliador

Este componente permite o cálculo do esforço de teste para uma lista de classes em uma ordem específica. O esforço de teste é calculado de duas formas distintas:

(1) É calculado o número de *stubs* necessário para o teste de integração na ordem especificada pela lista; e

(2) É calculado o tamanho total dos *stubs* a serem construídos. O tamanho de um *stubs* consiste no tamanho da classe (número de atributos + número de métodos) para a qual está sendo construído o *stub*.

A Figura 5.16 apresenta a área destinada ao componente *Avaliador* na ferramenta FAROL.

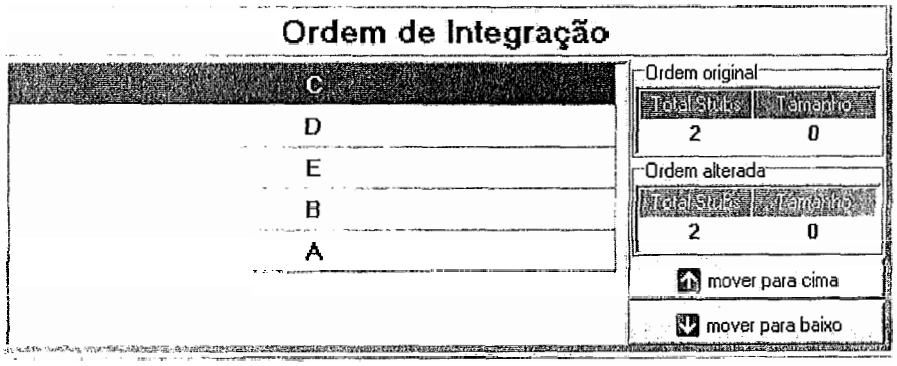


Figura 5.16 – LCOTI calculada através das heurísticas para o Exemplo.

FAROL permite que a ordem das classes obtidas pelas heurísticas seja modificada (Figura 5.17) por meio dos botões "mover para cima" ou "mover para baixo" a classe selecionada. Após cada modificação, a ferramenta recalcula o novo esforço de teste correspondente à nova ordem estabelecida.

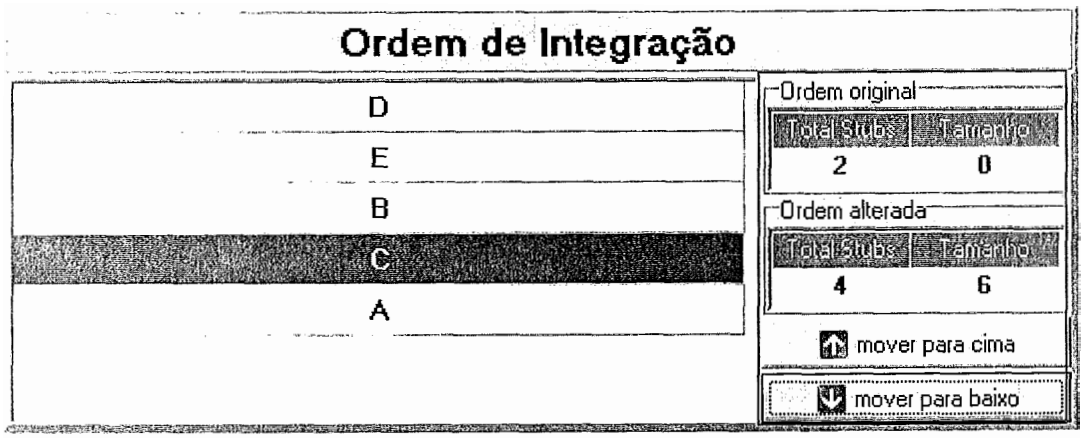


Figura 5.17 – Alterando LCOTI.

5.7 – Componente de Exteriorização

FAROL disponibiliza algumas funcionalidades que visam fornecer informações para entidades externas à ferramenta. As funcionalidades são as seguintes: salvar, importar e imprimir resultados.

5.7.1 – Opção de Salvar os Resultados

FAROL permite que os resultados obtidos pela ferramenta sejam salvos em um formato XML, possibilitando que estes resultados possam ser reutilizados pela ferramenta posteriormente. Esta opção somente está disponível após a geração da lista com a ordem das classes a serem integradas. Para salvar os resultados, o usuário deve clicar no botão *Salvar Resultados* na tela principal.

5.7.2 – Opção de Importar os Resultados

FAROL permite que os resultados salvos pela ferramenta (descrito na seção 5.7.1) sejam reutilizados. Neste momento FAROL simula todo o processo para obtenção da ordem de integração de acordo com os resultados salvos no arquivo XML e ao final apresenta a mesma ordem especificada durante a exportação desses resultados. Somente os arquivos que foram salvos pela ferramenta poderão ser importados, ou seja, os arquivos que não estejam na estrutura definida durante a operação de salvar não poderão ser importados pela ferramenta. Para salvar os resultados, o usuário deve clicar no botão *Importar Resultados* na tela principal.

5.7.3 – Opções de Impressão

FAROL disponibiliza a opção de visualização e impressão dos resultados obtidos em um formato HTML. Esta opção só está disponível após a lista LCOTI ter sido gerada. Para visualizar o relatório, o usuário deve clicar no botão *Imprimir Resultados* na tela principal.

O relatório gerado por FAROL apresenta informações sobre o modelo e as classes utilizados pela ferramenta. Apresenta ainda informações sobre a ordem de integração obtida pela ferramenta e os *stubs* necessários para cada classe do modelo, assim como o número de *stubs* total para a ordem especificada. A Figura 5.18 descreve um exemplo de relatório para o exemplo descrito ao longo deste capítulo.

Modelo: *Exemplo8f*

Criado em: 27/10/2004

Descrição das Classes

Nome	Tamanho	Fator de Influência
A	0	2
B	2	1
C	3	2
D	0	1
E	0	1

Ordem de Integração

Nome da Classe	Quantidade de Stubs
C	(0)
D	(0)
E	(1,A)
B	(1,A)
A	(0)
Ordem alterada	
Número de Stubs	2
Tamanho	0
Ordem original	
Número de Stubs	2
Tamanho	0

Figura 5.18 – Impressão dos resultados em HTML.

5.8 – Outras Facilidades de FAROL

FAROL disponibiliza, ainda, algumas facilidades aos seus usuários a fim de simplificar as atividades cotidianas de seus usuários, apresentadas a seguir.

5.8.1 – Opções de Localização

FAROL possui uma opção que permite a alteração do idioma corrente durante a sua utilização. O usuário pode optar entre os idiomas disponíveis, Português e Inglês, acessando o menu Configurações → Mudar Idioma (Crtl+M), conforme apresentado na Figura 5.19. Novos idiomas podem ser inseridos por meio da DLL instalada em conjunto à ferramenta.

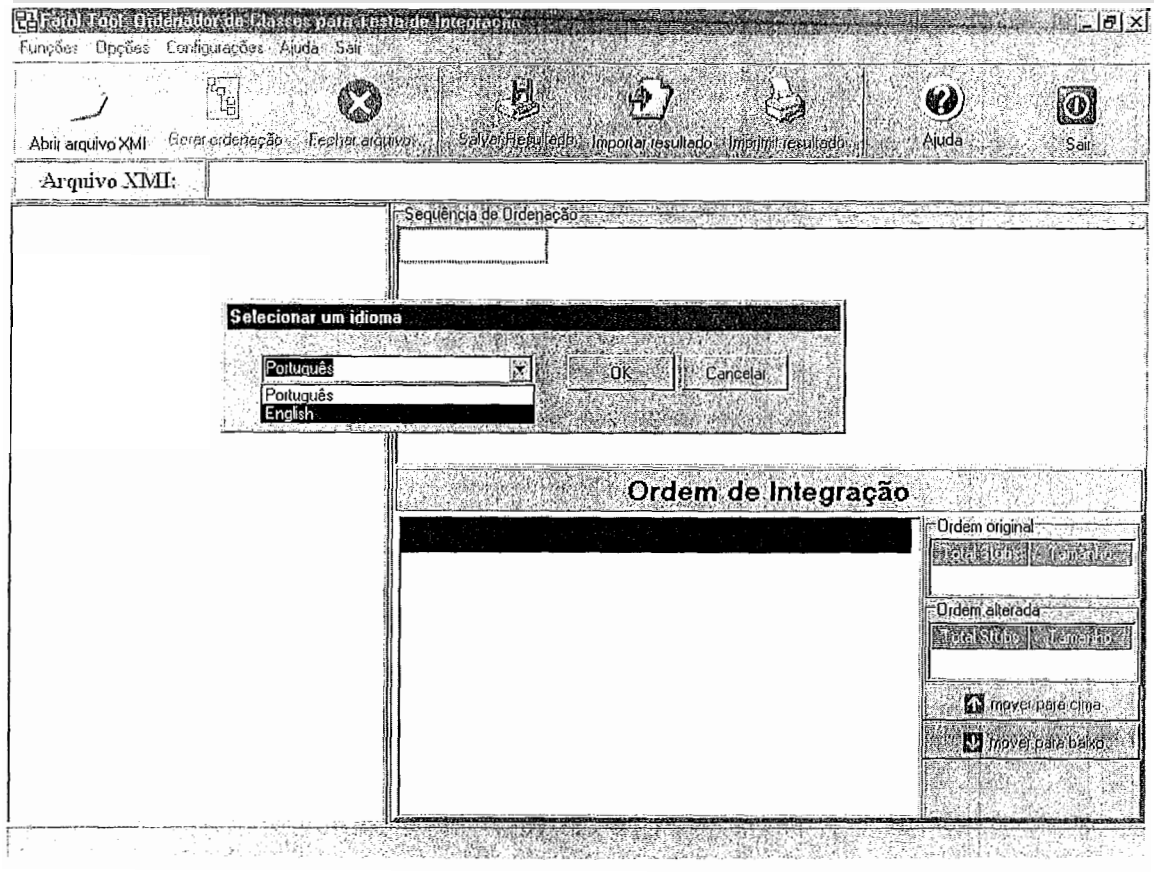


Figura 5.19 – Alterando Idioma.

5.8.2 – Opção de Ajuda

FAROL disponibiliza uma ajuda *On-line* aos seus usuários. A ajuda disponibilizada descreve principalmente dois tipos de informações:

(1) sobre a ferramenta FAROL e sua utilização; e
(2) sobre as heurísticas descritas no Capítulo 3.

Um exemplo de página de ajuda disponibilizada por FAROL é apresentado na Figura 5.20. Para obter a ajuda da ferramenta, o usuário deve clicar no botão *Ajuda* na tela principal.

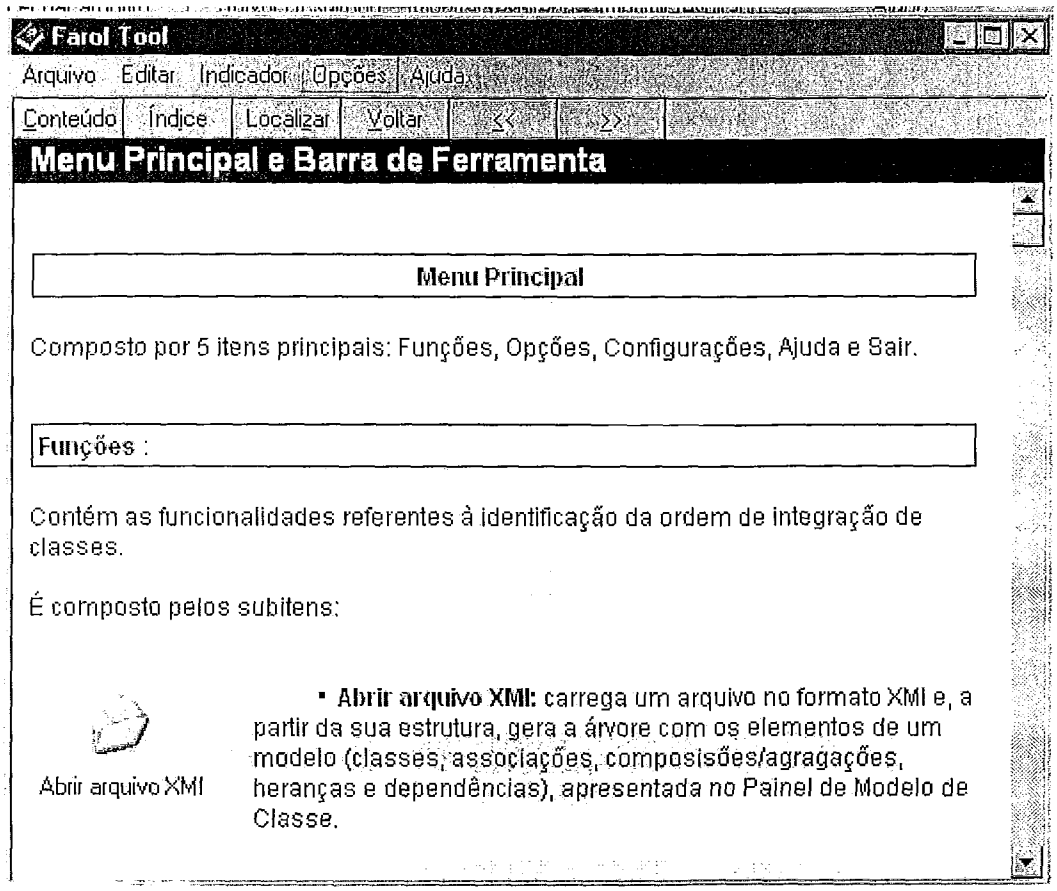


Figura 5.20 – FAROL: Ajuda disponível.

5.9 – Considerações Finais

Este capítulo apresentou a ferramenta FAROL que foi implementada para apoiar o emprego do conjunto de heurísticas e seu processo de aplicação para identificar a ordem de integração e teste das classes a partir de um diagrama de classes UML, apresentados no Capítulo 3. A importância deste processo está principalmente no fato de prover ao usuário uma noção de como FAROL funciona e quais os passos devem ser seguidos para sua correta utilização.

Pelo fato de FAROL reconhecer arquivos em um formato XMI que estejam de acordo com o DTD da UML versão 1.3, é possível utilizar os modelos gerados por diversas ferramentas CASE para modelagem de diagramas UML. No entanto, os modelos UML e, conseqüentemente, suas especificações tendem a evoluir com o tempo, o que tornaria necessário um esforço para adaptar FAROL a essa evolução.

Por fim, vale ressaltar que a implementação do mecanismo de ordenação tenta automatizar as tarefas objetivando eliminar a interferência do usuário no processo de aplicação das heurísticas, evitando possíveis erros de cálculos, como observados durante os estudos de casos apresentados no Capítulo 4, que poderiam levar a uma

lista de classes ordenadas para teste de integração (LCOTI) sem, entretanto, atingir o esforço mínimo de teste. O usuário poderá interagir com o protótipo alterando a ordem das classes sugerida pelas heurísticas e obtendo um novo esforço de teste (número de *stubs*) para a nova ordem informada.

No próximo capítulo serão discutidas algumas considerações finais e direções futuras para dar continuidade a este trabalho.

Capítulo 6

Considerações Finais

Neste capítulo são apresentadas as conclusões, contribuições, limitações, e as perspectivas futuras de trabalhos que poderão ser realizados a partir desta tese.

6.1 – Conclusões

O paradigma de desenvolvimento orientado a objetos trouxe novos desafios aos engenheiros de software, sendo o teste de integração um deles, como discutido durante esta tese. A necessidade de integrar e testar classes implica na necessidade de identificar a ordem de integração destes componentes.

Os diversos relacionamentos entre classes, que representam os serviços disponibilizados entre objetos para implementação das funcionalidades, podem gerar os ciclos de dependências entre estas. Durante a integração e teste das classes, estes ciclos precisam ser quebrados, gerando a necessidade de construir mais software – os *stubs*. A ordem das classes para testes de integração afeta diretamente no número de *stubs* necessários para sua execução, pois estes simulam o comportamento de classes ainda não testadas, mas que prestam serviços às classes que estão sendo testadas. O número de *stubs* necessário representa o esforço de teste correspondente a uma determinada ordem de integração.

Na engenharia de software, algumas estratégias de teste de integração (*Strategy for Integration Testing* – SIT) têm sido desenvolvidas para ordenar as classes de um projeto OO visando minimizar o esforço de teste. O principal problema dessas abordagens está na necessidade de construção de outros artefatos, diferentes daqueles existentes num projeto OO, representando custo adicional no planejamento dos testes.

Neste contexto, esta tese apresentou um processo para aplicação das heurísticas formalizadas para apoiar à atividade de teste de integração aplicados a software orientado a objetos na identificação da ordem de integração e testes das classes de um projeto a partir do diagrama de classes UML. Devido ao fato de serem aplicadas num artefato de abstração de alto nível, as heurísticas podem auxiliar no planejamento dos testes de integração, antes mesmo de iniciar a atividade de

implementação das classes, podendo inclusive servir de guia no planejamento e construção das classes.

A execução de quatro estudos de caso permitiu avaliar a efetividade das heurísticas. O primeiro estudo caracterizou a efetividade comparando o esforço de teste para um mesmo modelo utilizado por outra estratégia. As heurísticas mostraram ser viáveis também nos outros estudos, quando após um autotreinamento alguns engenheiros de software e gerentes de projeto obtiveram um desempenho melhor ao aplicarem as heurísticas do que aplicando procedimentos *ad-hoc*.

Uma análise qualitativa dos estudos anteriores possibilitou identificar a necessidade de automatizar o processo de aplicação das heurísticas. As heurísticas e seu processo de aplicação foram implementados na ferramenta FAROL. Com FAROL é possível obter a ordem de integração e o esforço de teste correspondente a partir de um arquivo XMI contendo as informações exportadas de um diagrama de classes UML. A FAROL permite alterar a ordem sugerida com as heurísticas, sendo recalculado automaticamente o novo esforço de teste.

A partir de agora serão descritas as contribuições deste trabalho, suas limitações e por último, as perspectivas futuras.

6.2 – Contribuições

Dentre as principais contribuições deste trabalho, destacamos:

1. Aperfeiçoamento e complemento do conjunto de heurísticas para testes de integração aplicados a software OO a serem empregadas em um artefato de abstração de alto nível do projeto;
2. Definição de um processo de aplicação das heurísticas com o qual é possível identificar a ordem de integração das classes para um projeto OO resultando em esforço mínimo de teste (modelos pequenos), a partir de um diagrama de classes UML;
3. Definição e execução de quatro estudos de caso para caracterizar a viabilidade das heurísticas;
4. Elaboração de um autotreinamento sobre o processo de aplicação das heurísticas, na forma de uma apresentação PowerPoint, contendo as informações conceituais e a demonstração do processo por meio de exemplos; e
5. A construção da ferramenta FAROL que automatiza o processo de aplicação das heurísticas.

6.3 – Limitações

Algumas limitações foram consideradas durante o desenvolvimento da abordagem proposta nesta tese. Os estudos experimentais executados possuem as seguintes limitações:

1. O esforço de teste medido considerou somente o número de *stubs* específicos envolvidos, sendo assumido 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*;
2. A validade das conclusões obtidas não podem ser estendida para projetos de maior escala, pois os resultados apresentados referem-se a modelos de pequenos projetos utilizados (máximo de 14 classes); e
3. As inferências estatísticas (teste das hipóteses) foram obtidas para uma população pequena composta de 20 participantes.

Outra limitação refere ao emprego da versão 1.3 da DTD (*Data Type Definition*) durante a implementação da ferramenta FAROL, que representará na necessidade de uma adaptação do Tradutor XMI da FAROL no caso de alteração deste padrão pelas ferramentas CASE.

É importante ressaltar que as contribuições e conclusões, referentes ao conjunto de heurísticas e ao processo de aplicação, apresentadas neste trabalho foram obtidas durante a execução dos estudos de casos, portanto, estão limitadas as validades destes.

6.4 – Perspectivas Futuras

Buscando-se melhorar e expandir a abordagem de integração proposta, algumas perspectivas de trabalhos futuros são destacadas.

1. Planejar e executar estudos de caso para estender o conjunto de heurísticas e o seu processo de aplicação, para tratar as situações:
    - quantidade e complexidade dos *stubs*;

- dependências cíclicas como *clusters*, sem necessidade de quebras das dependências; e
- integração entre subsistemas;

2. Definir novos requisitos para a ferramenta FAROL a partir das conclusões e lições aprendidas com os estudos propostos no item anterior; e
3. Estudar a aplicabilidade das heurísticas em paradigmas de desenvolvimento alternativos, como:
    - sistemas multi-agentes (*Multi-agent Systems – MAS*), tomando por base os aspectos estáticos do metamodelo TAO (*Taming Agents and Objets*), descrito pela MAS-ML (LUCENA, 2004); e
    - desenvolvimento baseado em componentes (DBC).
4. Determinar a relação entre o menor esforço de teste (obtido com o processo de aplicação das heurísticas) e o esforço de teste mínimo dos modelos estudados.
5. Avaliar a estratégia proposta para teste de integração, no que tange a ordem de integração das classes de um diagrama de classes UML, sua aplicabilidade e limitações quando da necessidade de aplicação de testes de regressão, conforme a técnica selecionada (GRAVES *et al.*, 2001)

REFERÊNCIAS BIBLIOGRÁFICAS

ABNT, NBR ISO/IEC 12207 – Tecnologia de informação - Processos de ciclo de vida de software. Rio de Janeiro, 1998.

AMARAL, E.A.G., Empacotamento de Experimentos em Engenharia de Software, Tese de Mestrado, COPPE/UFRJ, Programa de Engenharia de Sistemas e Computação, Rio de Janeiro, 2003.

ANDERSSON, C., THELIN, T., RUNESON, P., DZAMASHVILI, N., 2003, "An Experimental Evaluation of Inspection and Testing for Detection of Design Faults". *Proc. 2003 International Symposium on Experimental Software Engineering*, Setembro, pp 174-184.

ANTONIOL, G., BRIAND, L.C., PENTA, M.D., LABICHE, Y., 2002, "A Case Study Using the Round-Trip Strategy for State-Based Class Testing". *Proc 13th International Symposium on Software Reliability Engineering*, IEEE Computer Society.

ARMOUR, P.G., 2004, "Not-Defect: the Mature Discipline of Testing". *Communications of the ACM*, vol. 47, issue 10, Outubro, pp 15-18.

BARBEY, S., STROHMEIER, A., 1994, "The Problematics of Testing Object-Oriented Software". *Proc. 2nd Conference on Software Quality Management*, vol. 2, pp 411-426, Julho, Edinburgh, UK.

BASILI, V.R., 1996, "The Role of Experimentation in Software Engineering: Past, Current, and Future". *IEEE Proceedings of ICSE-18*, pp 442-449.

BASILI, V.R., SHULL, F., LANUBILE, F., 1999, "Building Knowledge Through Families of Experiments". *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, Julho/Agosto, pp 458-473.

BEIZER, B., *Software System Test and Quality Assurance*. Van Norstrand Reinhold Company Inc, 1984.

BEIZER, B., *Software Testing Techniques*. Van Norstrand Reinhold Company Inc, 1990.

BERK, K.N., CAREY, P., *Data Analysis with Microsoft Excel*. Duxbury Press, 1995.

BERLING, t., HÖST, M., 2003, "A Case Study Investigating the Characteristics of Verification and Validation Activities in the Software Development Process". *Proc. 29th EUROMICRO Conference*, Setembro, pp 405-408.

BERTOLINO, A., INVERARDI, P., MUCCINI, H., ROSETTI, A., 1997, "An Approach to Integration Testing Based on Architectural Descriptions". *Proc. 3rd International Conference on Engineering of Complex Computer Systems*, Setembro, pp 77-84.

BEYDEDA, S., GRUHN, V., STACHORSKI, M., 2001, "A Graphical Class Representation for Integrated Black- and White-Box Testing". *IEEE International Conference on Software Maintenance*, IEEE Computer Society, Novembro, pp 706-715.

BINDER, R.V., 1994, "Testing Object-Oriented Systems: a Status Report". *American Programmer*, vol. 7, no. 4, Abril, pp 23-28.

BINDER, R.V., *Testing Object-Oriented Systems: models, patterns, and tools*. Addison-Wesley Publishing Company, 2000.

BOMPANI, L., CIANCARINI, P., VITALI, F., 2000, "Software Engineering and the Internet: a Roadmap". Pp. 261-277, ACM Press.

BOOCH, G., *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Publishing Company, 1994.

BOOCH, G., RUMBAUGH, J., JACOBSON, I., *UML – Guia do Usuário*. Editora Campus, 2000.

BRIAND, L.C., LABICHE, Y., WANG, Y., 2001, "Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles". *Proc. 12th International Symposium on Software Reliability Engineering (ISSRE)*, Hong Kong, Novembro 27-30, pp 287-296.

BRIAND, L.C., LABICHE, Y., 2002a, "A UML-Based Approach to System Test". In: Technical Report TR SCE-01-01-Version 4, Junho, Carleton University, Ottawa, Canada.

BRIAND, L.C., FENG, J., LABICHE, Y., 2002b, "Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders". In: Technical Report TR SCE-02-03-Version 3, Outubro, Carleton University, Ottawa, Canada.

BRIAND, L.C., LABICHE, Y., WANG, Y., 2003, "A Comprehensive and Systematic Methodology for Client-Server Class Integration Testing". *Proc. 14th International Symposium on Software Reliability Engineering*, Novembro, pp 14-25.

BRIAND, L.C., PENTA, M.D., LABICHE, Y., 2004, "Assessing and Improving State-Based Class Testing: a Series of Experiments". *IEEE Transactions on Software Engineering*, vol. 30, no. 11, Novembro, pp 770-793.

BROOKS, A., DALY, J., MILLER, J., ROPER, M., WOOD, M., 1996, "Replication of Experimental Results in Software Engineering". ISERN Technical Report ISERN 96-10.

BRYAN,M. 1992. An introduction to the standard generalization markup language (SGML). Disponível em http://www.personal.u-net.com/sgml/sgml.htm. Último acesso em 15/11/2003.

CARLSON, D., *Modeling XML Applications with UML*. Addison Wesley, 2001.

CRAIG, R.D., JASKIEL, S.P., *Sistematic Software Testing*. Artech House Inc, 2002.

DALY, J., BROOKS, A., MILLER, J., ROPE, M., WOOD, M., 1995, "The Effect of Inheritance on the Maintainability of Object-Oriented Software: an Empirical

Study". *Proc. of the International Conference of Software Maintenance*, Outubro, pp 20-29.

DELAMARO, M.E., MALDONADO, J.C., MATHUR, A.P., 2001, "Interface Mutation: an Approach for Integration Testing". *IEEE Transactions on Software Engineering*, vol. 27, no. 3, Março, pp 228-247.

DUSTIN, E., Effective Software Testing: 50 Specific Ways to Improve Your Test. Pearson Education, Inc, 2003.

EGYED, A., 2002, "Automated Abstraction of Class Diagrams". *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 4, Outubro, pp 449-491.

EMMERICH, W., 2000, Software Engineering and Middleware: A Roadmap. In A. Finkelstein, editor, *Future of Software Engineering*, pages 117–129. ACM Press.

FEWSTER, M., GRAHAM, D., *Software Test Automation*. ACM Press Books, 1999.

FOWLER, M., SCOTT, K., *UML Distilled: a Brief Guide to the Standard Object Modeling Language*. 2 ed. Addison-Wesley Publishing Company, 1999.

FREUND, R.J., WILSON, W.J., *Statistical Methods*. 2 ed., Academic Press, 1997.

FUHRMAN, C., DJLIVE, F., PALZA, E., 2003, "Software Verification and Validation within the (Rational) Unified Process". *Proc. 28th Annual NASA Goddard Software Engineering Workshop*, Dezembro, pp 216-220.

FURLAN, J.D., *Modelagem de Objetos através da UML*. Makron Books, 1998.

GRAVES, T.L., HARROLD, M.J., KIM, J.M., PORTER, A., ROTHERMEL, G., 2001, "An Empirical Study of Regression Test Selection Techniques". *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 2, Abril, pp 184-208.

HANH, V.L., AKIF, K.., TRAON, Y.L., JÉZÉQUEL, J-M., 2001, "Selecting an Efficient OO Integration Testing Strategy: an Experimental Comparison of Actual

Strategies". *Proc. European Conference of Object-Oriented Programming 2001*, J. Lindskov Knudsen (Ed): ECOOP2001, LNCS 2072, pp 381-401.

HARMON, P., WATSON, M., *Understanding UML: the Developer's Guide: with a Web-based Application in Java.* Morgan Kaufmann Publishers, 1997.

HARRISON, W., OSSHER, H., TARR, P., 2000, *The Future of Software Engineering , Software Engineering Tools and Environments: a Roadmap.* Pp. 261-277, ACM Press.

HOUDEK, F., 2003, "External Experiments – a Workable Paradigm for Collaboration Between Industry and Academia". Lecture Notes On Empirical Software Engineering, Capítulo 4, pp 133-166, World Scientific.

HUTCHESON, L.M., *Software Testing Fundamentals: Methods and Metrics.* John Wiley & Sons, 2003.

IEEE, IEEE standard glossary of software engineering terminology. Standard 610.12, IEEE Press, 1990.

ISO / IEC 12.207:1995. Tecnologia da informação – processo de ciclo de vida de software, 1997.

JACOBSON, I., *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley, 2 ed., 1992.

JALOTE, P., *An Integrated Approach to Software Engineering.* 2 ed. Springer, 1997.

JANERT, P.K., 2004, "Introducing Test-Driven Software Development". *IEEE Software*, vol. 21, no. 6, Novembro/Dezembro, pp 100-101.

JURISTO, N., MORENO, A.M., *Basics of Software Engineering Experimentation.* Kluwer Academic Publishers, 2000.

JURISTO, N., MORENO, A.M., VEGAS, S., 2004, "Reviewing 25 Years of Testing Technique Experiments". *Empirical Software Engineering*, Kluwer Academic Publishers, 9, Março, pp 7-44.

KIRSOPP, C., SHEPPERD, M., WEBSTER, S., 1999, "An Empirical Study into the Use of Measurement to Support OO Design Evaluation". *Proc. 6th IEEE International Symposium on Software Metrics*, Novembro, pp 230-241.

KITCHENHAM, B.A., PFLEEGER, S.L., PICKARD, L.M., JONES, P.W., HOAGLIN, D.C., EMAM, K.E., ROSENBERG, J., 2002, "Preliminary Guidelines for Empirical Research in Software Engineering". *IEEE Transactions on Software Engineering*, Agosto, Vol. 28, no. 8, pp 721-734.

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

KUNG, D., GAO, J., HSIA, P., TOYOSHYMA, Y., CHEN, C., KIM, YS., SONG, Y-K, 1995b, "Developing an Object-Oriented Software Testing and Maintenance Environment". *Communications of the ACM*, vol. 38, no. 10, Outubro, pp 75-87.

LABICHE, Y., THÉVENOD-FOSSE, P., WAESELYNCK, H., DURAND, M.-H., 2000, "Testing Levels for Object-Oriented Software". *22nd IEEE International Conference on Software Engineering*, Limerick, Ireland, Junho, pp 136-145.

LIMA, G.M.P.S., PINHEIRO, L.N.P, 2004, "Certificação e Teste de Componentes de Software". In: *Reutilização de Software e Desenvolvimento Baseado em Componentes*, Publicações Técnicas 4/2004, COPPE/UFRJ, Fevereiro, 2004.

LIMA, G.M.P.S., TRAVASSOS, G.H., 2004, Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes. In: Relatório Técnico ES-632/04, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, disponível em http://www.cos.ufrj.br/publicacoes.

LIMA, G.M.P.S., TRAVASSOS, G.H., 2004, "Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes". In: *Anais do III Simpósio Brasileiro de Qualidade de Software*, Brasília, Junho, pp 221-233 e Revista Pesquisa Naval, no. 17, Novembro, pp 61-69.

LIMA, G.M.P.S., NETO, A.C.D., TRAVASSOS, G.H., 2004, "Empirical Study of Class Integration Testing Order Heuristics". *1st Experimental Software Engineering Latin American Workshop (ESELAW'04)*, Position Paper, Brasília, Brasil, Outubro.

LIMA, G.M.P.S., TRAVASSOS, G.H., 2005, "A Strategy for Object-Oriented Software Integration Testing". *6th IEEE Latin-American Test Workshop (LATW2005)*, Salvador, Brasil.

LORENZ, M., KIDD, J., *Object-Oriented Metrics: a Practical Guide*. Prentice Hall, USA, 1994.

LOWRY, M., BOYD, M., KULKAMI, D., 1998, "Towards a Theory for Integration of Mathematical Verification and Empirical Testing". *Proc. 13th IEEE Conference on Automated Software Engineering*, Outubro, pp 322-331.

LUCENA, C., 2004, "Ongoing Research on the Software Engineering of Multi-Agent Systems". *18º. Simpósio Brasileiro de Engenharia de Software*, Outubro, pp 02-09.

McGREGOR, J.D., KORSON, T.D., 1994, "Integrated Object-Oriented Testing and Development Processes". *Communications of the ACM*, Setembro, Vol. 37, no. 9, pp 59-77.

McGREGOR, J.D., SYKES, D.A., *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley Publishing Company, 2001.

MILLER, J., 2000, "Applying Meta-Analytical Procedures to Software Engineering Experiments". *Journal of Systems and Software*, 54: pp 29-39.

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

OLIVEIRA, H., TRAVASSOS, G. H., "Construção de um Componente Genérico Baseado em Heurísticas para Ordenação das Classes em Ordem de Prioridade para Testes de Integração", Estudo do Laboratório de Engenharia de Software, COPPE/UFRJ, 2003.

PAUL, R., 2001, "End-to-End Integration Test". *Proc. 2nd Asia Pacific Conference on Quality Software*, Dezembro, pp 211-230.

PERRY, W.E., *Effective Methods for Software Testing*. Wiley-QED Publication, 1995.

PEZZE, M., YOUNG, M., 2004, "Testing Object Oriented Software". *Proc. 26th International Conference on Software Engineering*, Maio, pp 739-740.

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

PFLEEGER, S.L. *Engenharia de Software: Teoria e Prática*. 2 ed. Prentice Hall, 2004.

PORWAL, R., GURSARAN, 2002, "An Experimental Evaluation of Weak-Branch Criterion for Class Testing". The Journal of Systems and Software, 70 (2004), pp 209-224.

PRESSMAN, R.S., *Engenharia de Software*, 5 ed. McGrawHill, 2001.

QUATRANI, T.,*Visual Modeling with Rational Rose and UML*. Addison-Wesley Publishing Company, 1998.

RAKITIN, S., *Software Verification and Validation for Practitioners and Managers*. Artech House Inc, 2 ed., 2001.

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

RUMBAUGH, J., *Modelagem e Projetos Baseados em Objetos*. Editora Campus, 1994.

SCHACH, S.R., 1996, "Testing: Principles and Practice". *ACM Computing Surveys*, vol. 28, no. 1, pp 277-279.

SEAMAN, C.B., 1999, "Qualitative Methods in Empirical Studies of Software Engineering". *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, Julho/Agosto, pp 557-572.

SHULL, F., CARVER, J., TRAVASSOS, G.H., 2001, "An Empirical Methodology for Introducing Software Process". *Proc. 8th European Software Engineering Conference*, pp 288-296, Vienna, Austria.

SHULL, F., MENDONÇA, M.G., BASILI, V., CARVER, J., MALDONADO, J.C., FABBRI, S., TRAVASSOS, G.H, FERREIRA, M.C., 2004, "Knowlegde-Sharing Issues in Experimental Software Engineering". *Empirical Software Engineering*, 9, pp 111-137, Kluwer Academic Publisher.

SHULL, F., CARVER, J., TRAVASSOS, G.H., MALDONADO, J.C., CONRADI, R., BASILI, V.R., 2003, "Replicated Studies: Building a Body of Knowledge about Software Reading Techniques". *Lecture Notes On Empirical Software Engineering*, Capítulo 2, pp 39-84, World Scientific.

SOLHEIM, J.A., ROWLAND, J.H., 1993, "An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems". *IEEE Transactions on Software Engineering*, vol. 19, no. 10, Outubro, pp 941-949.

SOLINGEN, R., BERGHOUT, E., *The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development.* McGrawHill Companies, 1999.

SOUTER, A.L., POLLOCK, L.L., 2002, "Putting Escape Analysis to Work for Software Testing". *Proc. of International Conference of Software Maintenance (ICSM'02)*, Outubro, pp 430-439.

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

TARJAN, R., 1972, "Depth-First Search and Linear Graph Algorithms". *SIAM Journal on Computing*, vol. 1 (2), pp 146-160.

TASSEY, G., "The Economic Impacts of Inadequate Infrastructure for Software Testing", National Institute of Standards & Technology, U.S. Department of Commerce, Maio, 2002.

TICHY, W.F., 1998, "Should Computer Scientists Experiment More?". *IEEE Computer*, 31(5), pp. 32-39.

TORII, K., NAKAKOJI, K., TAKADA, Y., TAKADA, S., SHIMA, K., 1999, "Ginger2: an environment for Computer-Aided Empirical Software Engineering". *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, Julho/Agosto, pp 474-491.

TRAON, Y.L., JÉRON, T., JÉZÉQUEL, J.M., MOREL, P., 2000, "Efficient Object-Oriented Integration and Regression Testing". *IEEE Transaction on Reliability*, vol. 49(1), pp 12-25.

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

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

USCHOLD, M.E., 1998. Knowledge level modeling: Concepts and Terminology. Knowledge Engineering Review, 12(1). Also available as AIAI-T R-196 from AIAI, The University of Edinburgh.

VIEIRA, M.E.R., Abordagem para Apoio ao Teste Baseado no Comportamento de Sistemas Orientados a Objetos, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 1998.

VIEIRA, M.E.R., TRAVASSOS, G.H., 1998, "An Approach to Perform Behavior Testing in Object-Oriented Systems". *Technology of Object-Oriented Languages and Systems*, Setembro, pp 318-328.

VILLELA, K., "Ambientes de Desenvolvimento de Software Orientados à Organização", Tese de D. Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2004.

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

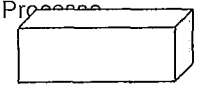
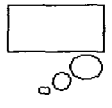
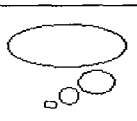
WONG, S.K, DILLON, T.S., HANISH, A., CHANG, E., 1999, "Software Testing of the Behavioral Aspects of Objects". *Proc. 5th International Workshop on Object-Oriented Real-Time Dependable Systems*, Novembro, pp 51-55.

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

Anexo A – Linguagem para Modelagem de Processos

Neste anexo é apresenta a notação utilizada para representar o Processo de Aplicação das Heurísticas descrito no capítulo 3.

A linguagem proposta para modelagem de processos em (VILELA, 2004) é composta de elementos gráficos que podem ser do tipo área, objeto ou ligação, onde uma ligação estabelece uma relação entre dois objetos e uma área agrupa objetos, definindo um contexto para os mesmos. Objetos ainda permitem adornos, utilizados para representar explicitamente características dos objetos. A seguir, cada elemento da linguagem é brevemente apresentado.

Objeto	Notação	Definição
Processo 		Objeto referente ao conceito de mesmo nome definido na ontologia de organização (seção 5.7). *Atributos Especiais:* Origem (Interno, Externo)
Evento 		Objeto que representa um acontecimento no ambiente que provoca o início ou fim de um processo. A notação é proveniente do produto comercial de *workflow* ARIS *ToolSet*.
Ator 		Objeto que representa um pessoa, agente ou unidade organizacional. Estes conceitos encontram-se definidos na ontologia de organização. A notação foi utilizada por KRUCHTEN [179] para representação dos *workflows* básicos do *Rational Unified Process*.
Atividade 		Objeto referente ao conceito de mesmo nome definido na ontologia de organização. A notação foi utilizada por KRUCHTEN [179] para representação dos *workflows* básicos do *Rational Unified Process*. *Atributos Especiais:* Origem (Interna, Externa) Granularidade (Elementar ou Composta)
Estado Inicial		Objeto puramente notacional, proveniente dos diagramas de estado e que indica onde é iniciado o fluxo de atividades que definem um processo ou uma atividade composta
Estado Final 		Objeto puramente notacional, proveniente dos diagramas de estado e que indica onde é encerrado o fluxo de atividades que definem um processo ou uma atividade composta
Conhecimento Explícito		Objeto que representa um conhecimento que pode ser expresso em palavras e números e ser facilmente transmitido e compartilhado. A notação foi proposta por ALLWEYER [10].
Conhecimento Implícito		Objeto que representa um conhecimento que é altamente pessoal e difícil de formalizar, o que o torna também difícil de ser compartilhado. A notação foi proposta por ALLWEYER [10].

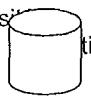
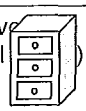
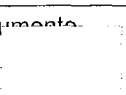
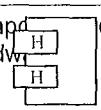
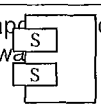
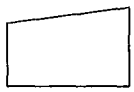
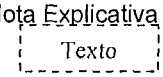
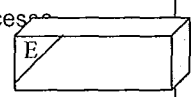
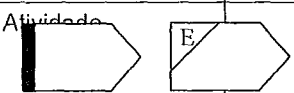
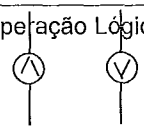
Com 		Objeto que representa a comunicação de dados ou informações a partir da, ou para a, execução de uma atividade. A comunicação pode ser verbal ou escrita e exemplos são e-mail e fax.
Repos (Meio  tico)		Objeto que representa um meio magnético para o armazenamento de dados e informações. A notação é proveniente do produto comercial de *workflow* ARIS *ToolSet*.
Arquiv (Local )		Objeto que representa um local físico para armazenamento de documentos e comunicações escritas.
Doc 		Objeto referente ao conceito de mesmo nome definido na ontologia de organização. A notação é proveniente do produto comercial de *workflow* ARIS *ToolSet*.
Comp  de Hardw		Objeto referente ao conceito de mesmo nome definido na ontologia de organização. A notação é baseada na notação de componente da UML.
Comp  de Softw		Objeto referente ao conceito de mesmo nome definido na ontologia de organização. A notação é baseada na notação de componente da UML.
Peça 		Objeto referente ao conceito de mesmo nome definido na ontologia de organização.
Matéria-Prima 		Objeto referente ao conceito de mesmo nome definido na ontologia de organização.
Bem 		Objeto referente ao conceito de mesmo nome definido na ontologia de organização. A notação fornecida pode ser substituída por uma mais significativa para o objeto específico do modelo como, por exemplo, o logotipo do software. *Atributos Especiais:* Tipo (Usufruto, Software, Hardware e Equipamento de Produção)
Nota Explicativa 		Objeto que permite que notas explicativas sejam adicionadas ao modelo. *Atributos Especiais:* Texto

Tabela C.1 – Definição e Notação dos Objetos

Objeto	Notação com Adornos	Definição dos Adornos
Proces 		Adorno que indica que o processo é externo, ou seja, que é executado por outra organização.
Atividade 	(a) (b)	(a) Adorno que indica que a atividade é composta, o que significa que ela pode ser decomposta em sub-atividades; (b) Adorno que indica que a atividade é externa, ou seja, que é executada por outra organização.
Operação Lógica 	 (a) (b) (c)	(a) Adorno que indica a operação lógica *E*; (b) Adorno que indica a operação lógica OU; (c) Adorno que indica a operação lógica OU Exclusivo.

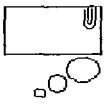
Conhecimento Explícito		Adorno que indica que foi especificado um caminho para acesso ao conhecimento disponível em meio magnético. Este adorno só deve ser utilizado se a visualização do modelo for apoiada por uma ferramenta de software que permita o acesso ao conhecimento. *Atributos Especiais:* Localização do Arquivo

Tabela C.2 – Definição e Notação dos Adornos

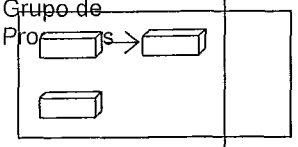
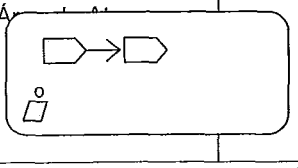
Objeto	Notação	Definição
Grupo de Processos		Área que agrupa processos relacionados.
Área de Atividades		Área que agrupa atividades executadas por um ator ou grupo de atores. O ator ou o grupo de atores também precisa estar contido na área.

Tabela C.3 – Definição e Notação das Áreas

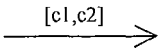
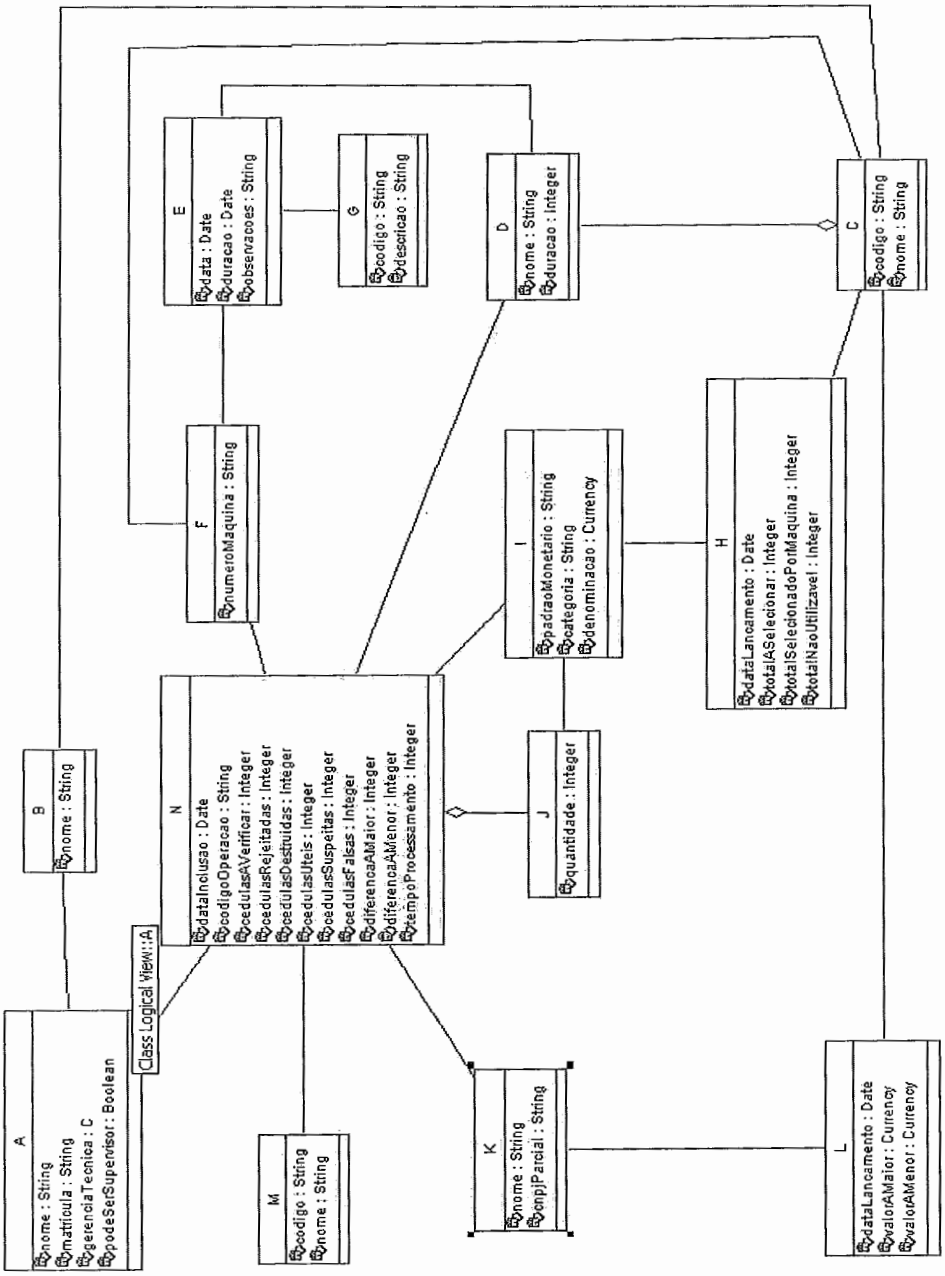
Objeto	Notação	Definição
Fluxo de Controle 		Ligação que indica a passagem de controle do objeto origem para o objeto destino. O c1 e o c2 indicados na notação são os rótulos das condições estabelecidas para que a passagem de controle ocorra. *Atributo Especial:* Condição, formada por rótulo e descrição
Fluxo de Entrada/Saída 		Ligação que estabelece um insumo (se o fluxo é de entrada) ou um produto de uma atividade (se o fluxo é de saída). Quando o objeto de origem ou destino é um armazenador (repositório ou arquivo), a notação pode incluir os rótulos das informações trafegadas, existindo, então, um atributo especial. *Atributo Especial:* Informação, formada por rótulo e descrição
Ligação Não Direcionada		Ligação que não indica passagem de controle nem estabelece insumos e produtos para uma atividade, sendo utilizada para conectar bens de produção (software, hardware e equipamentos) utilizados como recursos para execução das atividades e para conectar eventos que atuam sobre processos, provocando o seu início ou fim. No segundo caso, um atributo especial é definido. *Atributo Especial:* Papel do Evento (Iniciador, Terminador)
Ligação para Nota Explicativa		Ligação que estabelece que uma nota explicativa é referente a um elemento do modelo.

Tabela C.4 – Definição e Notação das Ligações

Anexo B – Artefatos do 2°. Estudo de Caso

Neste anexo são apresentados os artefatos preparados e utilizados durante o 2°. estudo de caso apresentado no capítulo 4.

B.1 – Diagrama de Classes UML e Formulário de Resposta



FORMULÁRIO RESPOSTA

Nome: _____ Tempo: _____ (min)

Ordem de Prioridade de Testes de Integração de Classes

Ordem	Classe
	DiferençaLançamentoEstoque
	DiferençaLançamentoInstituição
	Funcionário
	GerênciaTécnica
	Inserção
	InstituiçãoFinanceira
	MáquinaProcessamento
	MotivoParadaMáquina
	OperaçãoProcessamento
	ParadaMáquina
	TipoCédula
	Transportadora
	Turno
	Usuário

B.2 – Modelo de Consentimento

TESTES DE INTEGRAÇÃO EM SOFTWARE OO

Eu declaro ter mais de 18 anos de idade e que concordo em participar de um estudo conduzido pelo Prof. Guilherme Horta Travassos, como parte das atividades do curso de Engenharia de Software Orientada a Objetos no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. Este estudo visa compreender a viabilidade de aplicação de heurísticas para estabelecer uma ordem de integração das classes de um software orientado a objetos, representadas por meio de um diagrama de classes (UML), tentando entender sob que circunstâncias estas heuristicas podem ser efetivamente utilizadas.

PROCEDIMENTO

Este estudo acontecerá em duas etapas. Na primeira etapa serão levantados os procedimentos atualmente em uso por mim e, na segunda etapa, serei ensinado sobre um processo de aplicação de heurísticas para determinação da ordem de integração de classes e serei solicitado a aplica-las nos exemplos distribuídos. Nestes exercícios alguns métodos experimentais serão aplicados por mim visando permitir pensar sobre seu uso e avaliá-los. Eu entendo que, uma vez o experimento tenha terminado, os trabalhos que desenvolvi, representando alguns dos exercícios, serão estudados visando entender a eficiência dos procedimentos e as técnicas que me foram ensinadas.

CONFIDENCIALIDADE

Toda informação coletada neste estudo é confidencial, e meu nome não será identificado em momento algum. Da mesma forma, me comprometo a não comunicar os meus resultados enquanto não terminar o estudo, bem como manter sigilo das técnicas e documentos apresentados e que fazem parte do experimento.

BENEFÍCIOS, LIBERDADE DE DESISTÊNCIA

Eu entendo que os benefícios que receberei deste estudo são limitados ao aprendizado do material que é distribuído e ensinado, independente de participar ou não deste estudo, mas que os pesquisadores esperam aprender mais sobre quão eficiente é a identificação do processo de determinação da ordem em que as classes deverão ser testadas e os benefícios trazidos por este estudo para o contexto da Engenharia de Software.

Eu entendo que sou livre para realizar perguntas a qualquer momento ou solicitar que qualquer informação relacionada a minha pessoa não seja incluída no estudo. Eu entendo que participo de livre e expontânea vontade com o único intuito de contribuir para o avanço e desenvolvimento de técnicas e processos para a Engenharia de Software.

EQUIPE:

PESQUISADOR RESPONSÁVEL
Gladys Machado Pereira Santos Lima
CASNAV - Centro de Análises de Sistemas Navais - Marinha do Brasil

PROFESSOR RESPONSÁVEL
Prof. Guilherme Horta Travassos
Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ

Nome (em letra de forma):_____

Assinatura: _____ *Data:*_____

B.3 – Modelo de Caracterização

CARACTERIZAÇÃO DO DESENVOLVEDOR

Nome _____

Formação Acadêmica:

☐ Doutorado ☐ Mestrado ☐ Especialização ☐ Graduação
☐ Técnico ☐ Outra: _____

Formação Geral:

Por favor, estime sua habilidade em utilizar material de trabalho em Inglês:

__ Eu falo, leio e escrevo fluentemente.
__ Considero o Inglês como sendo uma linguagem onde :

Minhas habilidades de leitura e compreensão de textos:
__ poderiam ser melhores
__ são moderadas
__ são altas
__ são muito altas

Minha capacidade de trabalhar/seguir instruções escritas em Inglês:
__ poderiam ser melhores
__ são moderadas
__ são altas
__ são muito altas

Qual é sua experiência anterior com desenvolvimento de software na prática ?
(marque aqueles itens que melhor se aplicam)
__ nunca desenvolvi software.
__ tenho desenvolvido software para uso próprio.
__ tenho desenvolvido software como parte de uma equipe, relacionado a um curso.
__ tenho desenvolvido software como parte de uma equipe, na indústria.

Por favor, explique sua resposta. Inclua o número de semestres ou número de anos de experiência relevante em desenvolvimento. (E.g. "Eu trabalhei por 10 anos como programador na indústria")

_____

_____

_____

1/2

Experiência em Desenvolvimento de Software

Por favor, indique o grau de sua experiência nesta seção seguindo a escala de 5 pontos abaixo:

1 = nenhum
2 = estudei em aula ou em livro
3 = pratiquei em 1 projeto em sala de aula
4 = usei em 1 projeto na indústria
5 = usei em vários projetos na indústria

Experiência em projeto de sistemas	1	2	3	4	5
Experiência em desenvolver projetos a partir de requisitos e casos de uso	1	2	3	4	5
Domínio de Conceitos de Orientação a Objetos (herança, Encapsulamento e polimorfismo)	1	2	3	4	5
Experiência criando diagrama de classes	1	2	3	4	5
Experiência lendo diagrama de classes	1	2	3	4	5
Experiência com Unified Modeling Language (UML)	1	2	3	4	5
Experiência executando testes de unidades (classes)	1	2	3	4	5
Experiência executando testes de integração	1	2	3	4	5
Experiência executando testes de sistema	1	2	3	4	5
Experiência gerando casos de teste de projetos OO	1	2	3	4	5
Experiência planejando teste de projetos OO	1	2	3	4	5
Outras Experiências:					
Experiência com gerenciamento de projeto de software?	1	2	3	4	5
Experiência com testes de software?	1	2	3	4	5

Experiência em Contextos Diferentes:

Nós usaremos esta seção para compreender quão familiar voce está com vários sistemas que poderão ser utilizados como exemplos ou para exercícios durante o experimento.

Por favor, indique o grau de experiência nesta seção seguindo a escala de 3 pontos abaixo:
1 = Eu não tenho familiriadade com a área. Eu nunca fiz isto.
3 = Eu utilizo isto algumas vezes, mas não sou um especialista.
5 = Eu sou muito familiar com esta área. Eu me sentiria confortável fazendo isto.

Quanto voce sabe sobre...			
Utilizar um posto de gasolina?	1	3	5
Utilizar um estacionamento?	1	3	5
Solicitar um financiamento ou linha de crédito?	1	3	5

2/2

B.4 – Questionário de Avaliação

Questionário de Avaliação

O propósito único deste questionário é apoiar o pesquisador no levantamento de dados para melhor entendimento do estudo experimental em questão. Não será utilizado para avaliar o participante.

1. Você fez uso de alguma técnica conhecida e que tenha sido publicada?

2. Quais as características do modelo que representaram maior dificuldade durante o procedimento de determinação da ordem de integração?

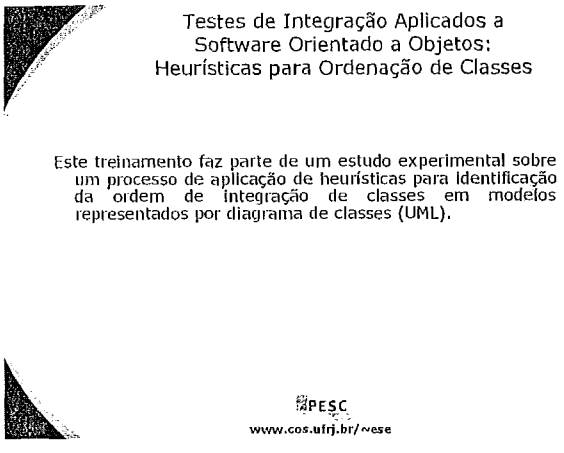
3. Quais critérios de decisão utilizados durante o procedimento?

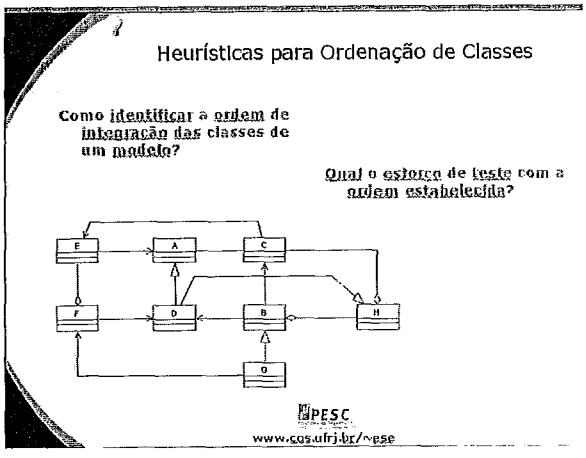
4. Poderia sugerir possíveis soluções que melhorariam o procedimento adotado?

Anexo C – Artefatos do 3°. Estudo de Caso

Neste anexo são apresentados os artefatos preparados e utilizados durante o 3°. estudo de caso apresentado no capítulo 4.

C.1 – Autotreinamento sobre as heurísticas





Heurísticas para Ordenação de Classes

A partir do estudo das dependências entre classes, representadas em diagramas de classes, foram observadas algumas características determinantes com base na semântica estabelecida pela UML para definir a ordem dos testes de integração destas classes.

Foram, então, estabelecidos critérios de precedência entre classes, que são:

- herança;
- assinatura dos métodos de uma classe;
- agregação;
- navegabilidade;
- classes de associação;
- dependência; e
- cardinalidade.

Heurísticas para Ordenação de Classes

Critério de Precedência
Herança

Considerando que as subclasses herdam as características e, principalmente, o comportamento das classes-base, garantir que a subclasse funcione de forma adequada significa **garantir, primeiro, que a superclasse tenha *sido devidamente* testada.**

Ordem de Teste: A
B

Heurísticas para Ordenação de Classes

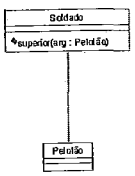
Quando a superclasse for abstrata, deve-se testar primeiro a classe-filha que seja menos acoplada.

A análise das dependências em relação à classe-base propicia uma análise indireta das dependências da subclasse, na medida que a subclasse somente será testada após as classes das quais a classe-base depende terem sido testadas.

Heurísticas para Ordenação de Classes

Critério de Precedência:
Assinatura dos métodos de uma classe

Se uma classe (cliente) utiliza serviços de outra classe (servidora), deve-se primeiro avaliar se estes serviços estão corretamente implementados. Então, ***testar, primeiro, a classe servidora e depois a classe cliente.***

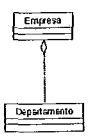


Ordem de Teste:
Pelotão
Soldado

Heurísticas para Ordenação de Classes

Neste contexto, agregações simples e composta possuem a mesma semântica, e a classe "*todo*" depende dos serviços fornecidos pelas classes "*partes*".

Então **a classe "*parte*" na agregação terá precedência para teste de integração sobre a classe que representa o "*todo*".**



Ordem de Teste:
Departamento
Empresa

Heurísticas para Ordenação de Classes

Critério de Precedência:
Navegabilidade

A navegabilidade indica que uma classe torna-se atributo de outra.

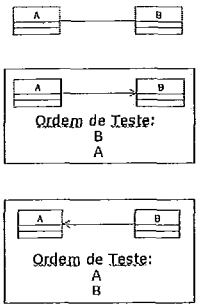
Sendo assim, **será utilizada a navegabilidade para definir o critério de precedência quando a ligação entre as duas classes ocorrer através de relação de associação.**

Ordem de Teste:
B
A

Heurísticas para Ordenação de Classes

A menos que seja declarado explicitamente o contrário, uma associação implica navegação bidirecional.

Sendo assim, **deverá ser analisada a navegabilidade dos dois sentidos.**

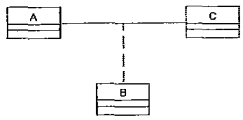


Heurísticas para Ordenação de Classes

Critério de Precedência:
Classe de Associação

Uma classe de associação surge a partir da necessidade de representar as informações do relacionamento entre duas outras classes.

As classes que deram origem à classe de associação terão precedência de teste de integração sobre ela.



Ordem de Teste:
A e C
B

Heurísticas para Ordenação de Classes

Uma dependência indica a ocorrência de um relacionamento semântico *entre dois ou mais elementos* do modelo.

Uma classe cliente é dependente de alguns serviços da classe fornecedora, mas não tem uma dependência estrutural interna com esse fornecedor.

No teste de integração, **a classe fornecedora terá precedência para ser testada.**

Ordem de Teste:
B
A

Heurísticas para Ordenação de Classes

Critério de Precedência: Cardinalidade

Quando se analisa a navegabilidade bidirecional, esta também pode expressar um critério de precedência. Um dos aspectos chave em associações é a cardinalidade de uma associação, também chamada multiplicidade.

A *cardinalidade* representa o número de objetos que participam em cada lado da associação, correspondendo à noção de obrigatório (1), opcional (0..1), um-para-muitos (1..N), muitos-para-muitos (N..M) ou outras variações desta possibilidade, sendo especificada para cada extremidade da associação.

Será utilizada a cardinalidade para definir o critério de precedência quando a cardinalidade representar a noção de *opcionalidade* (zero ou zero-para-muitos). Neste caso, **a classe com cardinalidade opcional deverá ser testada após a outra classe da associação.**

Heurísticas para Ordenação de Classes

Para viabilizar a aplicação dos critérios de precedência e estabelecer a ordem de integração foram definidas duas propriedades:

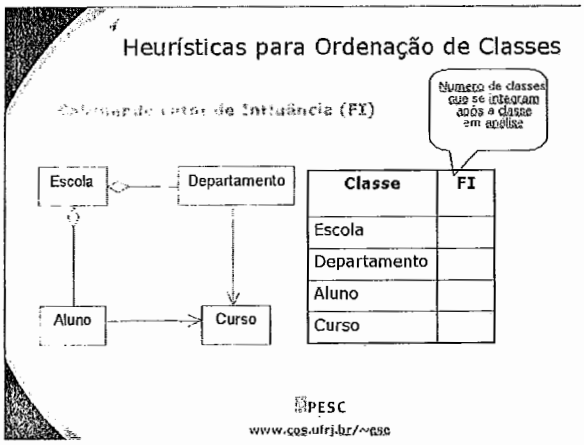
- Fator de Influência (FI); e
- Fator de Integração Tardia (FIT).

Heurísticas para Ordenação de Classes

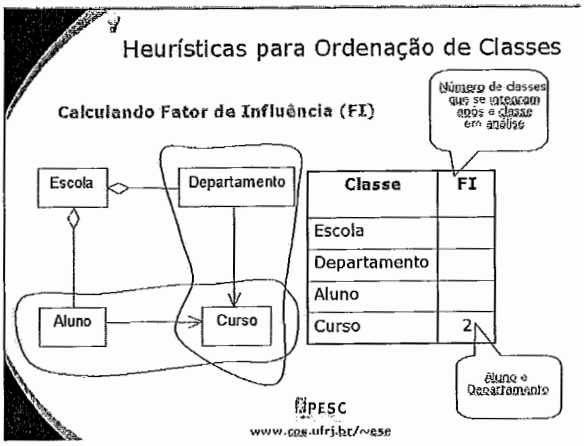
Fator de Influência (FI):

é o número que quantifica a relação de precedência entre as classes.

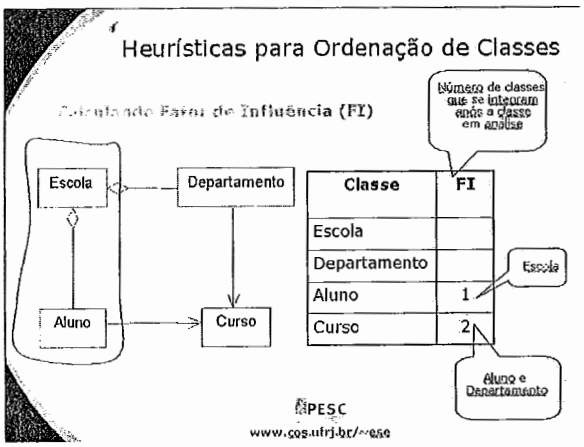
É diretamente proporcional ao número de classes que precisam ser integradas posteriormente à classe em questão.



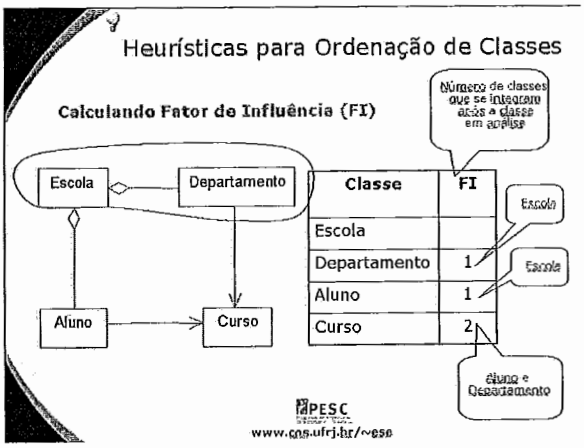
16



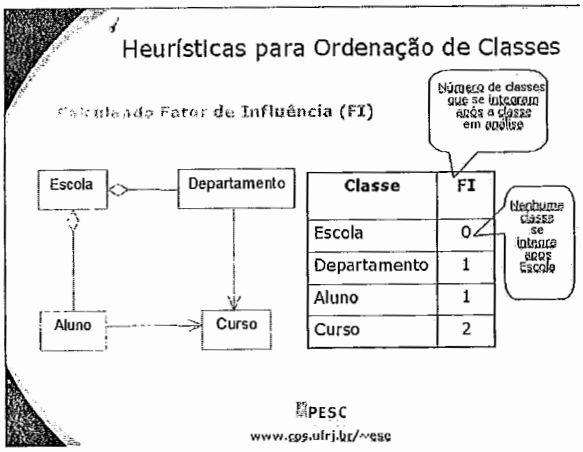
17



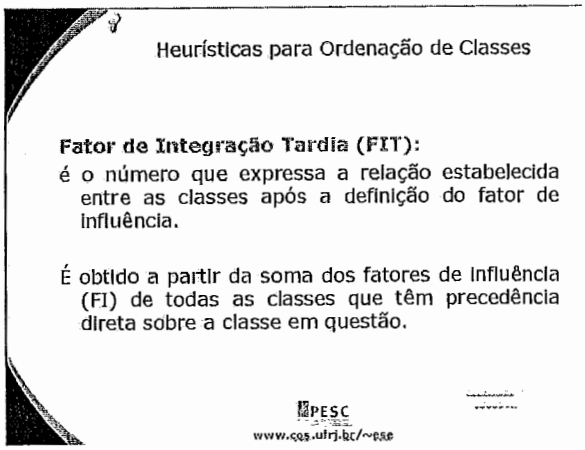
18



19



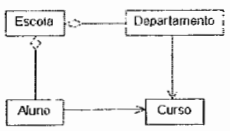
20



21

Heurísticas para Ordenação de Classes

Calculando Fator de Integração Tardia (FIT)



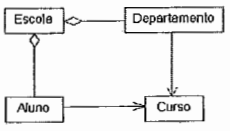
Classe	FI	FIT
Escola	0	
Departamento	1	
Aluno	1	
Curso	2	0

Soma dos FI das classes com precedência

Curso não depende de outra classe para ser integrado

Heurísticas para Ordenação de Classes

Calculando Fator de Integração Tardia (FIT)



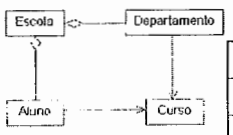
Classe	FI	FIT
Escola	0	
Departamento	1	
Aluno	1	2
Curso	2	0

Soma dos FI das classes com precedência

Aluno depende de Curso

Heurísticas para Ordenação de Classes

Calculando Fator de Integração Tardia (FIT)



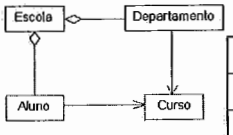
Classe	FI	FIT
Escola	0	
Departamento	1	2
Aluno	1	2
Curso	2	0

Soma dos FI das classes com precedência

Departamento depende de Curso

Heurísticas para Ordenação de Classes

Calculando Fator de Integração Tardia (FIT)



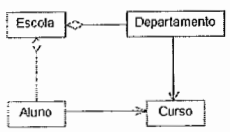
Classe	FI	FIT
Escola	0	2
Departamento	1	2
Aluno	1	2
Curso	2	0

Soma dos FI das classes com precedência

Escola depende de Departamento e Aluno

Heurísticas para Ordenação de Classes

Calculando Fator de Integração Tardia (FIT)



Classe	FI	FIT
Escola	0	2
Departamento	1	2
Aluno	1	2
Curso	2	0

Soma dos FI das classes com precedência

Heurísticas para Ordenação de Classes

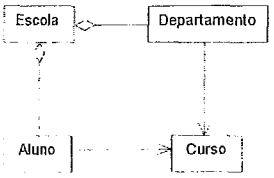
Como o Fator de Influência (FI) e o Fator de Integração Tardia (FIT) ajudam a encontrar a ordem de integração das classes?

Quanto maior for o valor do FIT de uma classe, mais tarde deve ser integrada esta classe.

Ou seja, **as classes que apresentarem FIT nulo deverão ser integradas primeiro.**

Heurísticas para Ordenação de Classes

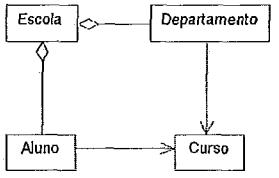
Qual a primeira classe a ser integrada?



Classe	FI	FIT
Escola	0	2
Departamento	1	2
Aluno	1	2
Curso	2	0

Heurísticas para Ordenação de Classes

Qual a primeira classe a ser integrada?



Classe	FI	FIT
Escola	0	2
Departamento	1	2
Aluno	1	2
Curso	2	0

Será a classe Curso pois possui FIT igual a zero.

Heurísticas para Ordenação de Classes

Como identificar as próximas classes a serem integradas?

Será necessário reduzir a influência da classe que acabou de ser selecionada.

Heurísticas para Ordenação de Classes

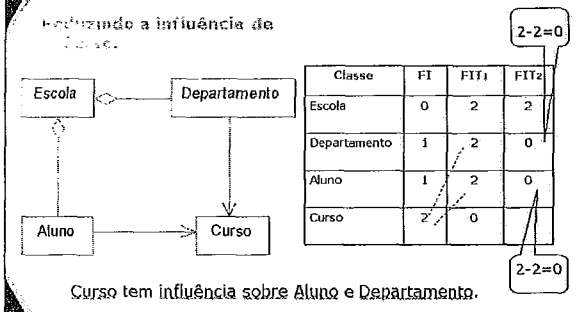
Como reduzir esta influência?

Recalculando os valores de FIT das classes ainda não integradas.

Os valores FIT nesta iteração serão calculados a partir do valor de FIT da iteração anterior menos o FI das classes que acabaram de ser integradas e que possuam precedência sobre a classe em questão.

Heurísticas para Ordenação de Classes

Reduzindo a influência de Curso.

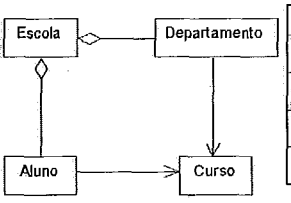


Classe	FI	FIT1	FIT2
Escola	0	2	2
Departamento	1	2	0
Aluno	1	2	0
Curso	2	0	

Curso tem influência sobre Aluno e Departamento.

Heurísticas para Ordenação de Classes

Quais as próximas classes a serem testadas?



Classe	FI	FIT1	FIT2
Escola	0	2	2
Departamento	1	2	0
Aluno	1	2	0
Curso	2	0	

Departamento e Aluno pois possuem FIT = 0.

Heurísticas para Ordenação de Classes

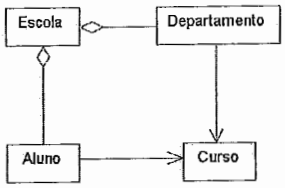
Será necessário reduzir a influência das classes que acabaram de ser selecionadas:
Aluno e Departamento.

Para tal é preciso recalcular os valores de FIT reduzindo os Fatores de Influência das classes Aluno e Departamento.

Classe	FI	FIT1	FIT2	FIT3
Escola	0	2	2	0
Departamento	1	2	0	2-1-1=0
Aluno	1	2	0	
Curso	2	0		

Heurísticas para Ordenação de Classes

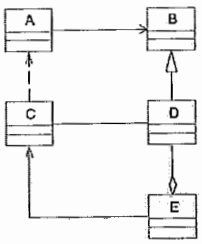
A ordem de integração para este modelo será:



Curso
Aluno ou Departamento
Escola

Heurísticas para Ordenação de Classes

Estabeleça uma ordem de integração para as classes do modelo no. 1, usando os conhecimentos adquiridos.



Heurísticas para Ordenação de Classes

Revendo alguns conceitos:

Agregação
Navegabilidade
Fator de Influência (FI)
Fator de Integração Tardia (FIT)
Reduzindo Influências

Heurísticas para Ordenação de Classes

Vamos usar outro exemplo para mostrar algumas situações especiais.

Veja como tratá-las !!

Heurísticas para Ordenação de Classes

Existem situações especiais que durante o processo de ordenação das classes devem ser tratadas de maneira diferenciada.

Que situações são estas?

- Classes com Fator de Influência (FI) nulo;
- Modelos que não possuem classes com FIT nulo;
- Classes com mesmo FIT.

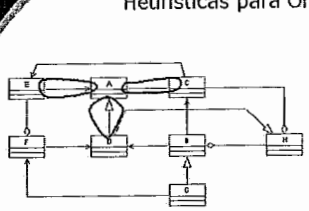
Heurísticas para Ordenação de Classes

Análise do Fator de Influência Nulo:

Expressa que a referida classe deverá ter seu teste de integração executado posteriormente à execução dos testes das demais classes com fatores de influência não nulos do modelo.

Estas classes têm seu teste de integração totalmente dependente da integração das demais classes do modelo.

Heurísticas para Ordenação de Classes



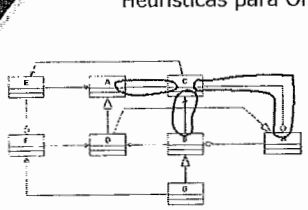
Número de classes que se integram após a classe em análise

Classes	FI
A	3
B	
C	
D	
E	
F	
G	
H	

C, E e D

Vamos calcular o Fator de Influência para as classes do modelo.

Heurísticas para Ordenação de Classes



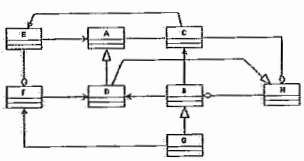
Número de classes que se integram após a classe em análise

Classes	FI
A	3
B	
C	3
D	
E	
F	
G	
H	

C, E e D

A, B e H

Heurísticas para Ordenação de Classes



Número de classes que se integram após a classe em análise

Classes	FI
A	3
B	1
C	3
D	2
E	2
F	1
G	0
H	2

C, E e D

G

A, B e H

B e F

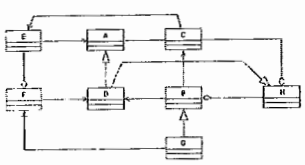
C e F

G

B e D

Assim teríamos:

Heurísticas para Ordenação de Classes

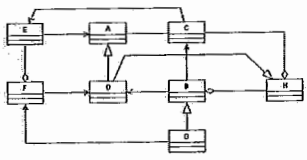


Classes	FI
A	3
B	1
C	3
D	2
E	2
F	1
G	0
H	2

A classe "G" por ter FI = 0 (totalmente dependente).

deverá ser incluída ao final da lista das classes ordenadas.

Heurísticas para Ordenação de Classes



Número de classes que se integram após a classe em análise

Classes	FI
A	3
B	1
C	3
D	2
E	2
F	1
H	2

A classe G será excluída das próximas etapas do procedimento de ordenação.

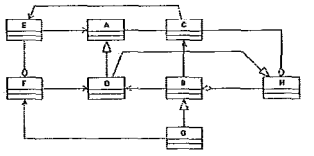
Heurísticas para Ordenação de Classes

Fator de integração tardia Nulo:

Modelos podem ser representados somente por classes com forte acoplamento, não existindo inicialmente classes com *fator de integração tardia* nulo.

Nestes casos, a seqüência ao teste de integração deve ser feita por meio das classes que possuam o menor FIT calculado.

Heurísticas para Ordenação de Classes



Soma dos FI das classes com precedência

Classes	FI	FIT
A	3	3
B	1	7
C	3	5
D	2	5
E	2	13
F	1	4
H	2	3

O modelo não apresenta classes com FIT nulo.

As classes A, E e H devem ser selecionadas para serem testadas antes das demais por possuirem menor FIT.

Heurísticas para Ordenação de Classes

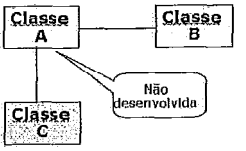
Classes com mesmo valor de FIT:
como estabelecer uma ordem entre elas?

Estas situações representam ciclos de dependências entre as classes com mesmo valor de FIT.

Para dar continuidade ao teste de integração é necessário a construção de *stubs*.

Heurísticas para Ordenação de Classes

Precisamos saber o que é um stub.



Stub é componente fachada escrito para simular o comportamento de uma classe ainda não desenvolvida ou não testada.

Um ***stub específico*** é escrito para simular o comportamento de A **em relação à classe** B.

Um ***stub realístico*** é escrito para simular o comportamento de A em relação às classes B e C.

Heurísticas para Ordenação de Classes

Tratamento de *Deadlock*

Para estabelecer a prioridade de integração entre as classes **com mesmo valor de FIT**, deve-se respeitar os seguintes critérios:

Primeiro: A classe selecionada deve gerar o menor número de *stubs* específicos em comparação com o número de *stubs* específicos gerados pelas outras classes com mesmo valor de FIT.

Segundo: No caso de alguma dessas classes a serem testadas contribuir para diminuir o número de *stubs* específicos para testar as outras de mesmo FIT, indicando uma dependência interna, esta deverá ser testada primeiro.

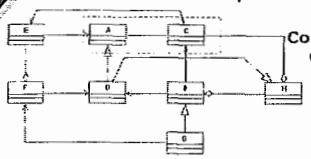
Heurísticas para Ordenação de Classes

Tratamento de *Deadlock*: (continuação)

Terceiro: No caso de empate do primeiro critério e nenhuma classe atender ao segundo critério, deverá ser testada aquela classe de menor tamanho (soma do número de atributos e métodos da classe).

Quarto: No caso dos critérios anteriores não permitirem a definição, e caso exista alguma associação com navegabilidade obrigatória, a classe deverá ser testada primeiro, preferencialmente.

Heurísticas para Ordenação de Classes



Como estabelecer a ordem de integração entre A, E e H ?

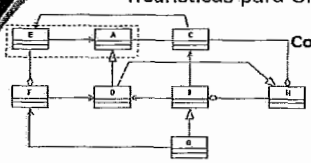
Se testássemos a Classe A primeiro, seria necessário um stub de C para A.

Classes	FI	FIT
A	2	3
B	1	7
C	3	5
D	2	5
E	2	3
F	1	4
H	2	3

PESC
www.cos.ufrj.br/~ese

52

Heurísticas para Ordenação de Classes



Como estabelecer a ordem de integração entre A, E e H ?

Se testássemos a Classe A primeiro, seria necessário um stub de C para A.

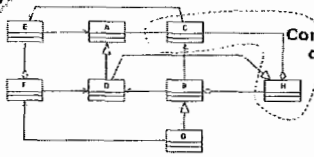
Se testássemos a Classe E primeiro, seria necessário um stub de A para E.

Classes	FI	FIT
A	2	3
B	1	7
C	3	5
D	2	5
E	2	3
F	1	4
H	2	3

PESC
www.cos.ufrj.br/~ese

53

Heurísticas para Ordenação de Classes



Como estabelecer a ordem de integração entre A, E e H ?

Se testássemos a Classe A primeiro, seria necessário um stub de C para A.

Se testássemos a Classe E primeiro, seria necessário um stub de A para E.

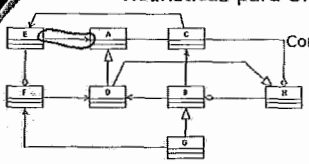
Se testássemos a Classe H primeiro, seria necessário um stub de C para H.

Classes	FI	FIT
A	2	3
B	1	7
C	3	5
D	2	5
E	2	3
F	1	4
H	2	3

PESC
www.cos.ufrj.br/~ese

54

Heurísticas para Ordenação de Classes



Como estabelecer a ordem de integração entre A, E e H ?

Pelo primeiro critério do tratamento de *deadlock*, ambas as classes precisam de 1 stub.

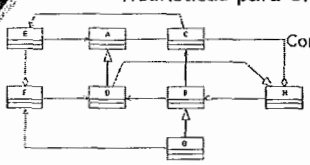
Pelo segundo critério do tratamento de *deadlock*, ao testar a classe A primeiramente não precisamos de stubs para testar a classe E.

Classes	FI	FIT
A	2	3
B	1	7
C	3	5
D	2	5
E	2	3
F	1	4
H	2	3

PESC
www.cos.ufrj.br/~ese

55

Heurísticas para Ordenação de Classes



Como estabelecer a ordem de integração entre A, E e H ?

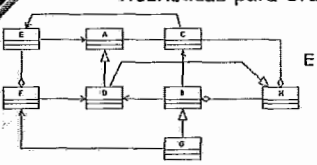
Portanto, a seqüência que necessitaria do menor número de stubs seria: A, E e H.

Classes	FI	FIT
A	2	3
B	1	7
C	3	5
D	2	5
E	2	3
F	1	4
H	2	3

PESC
www.cos.ufrj.br/~ese

56

Heurísticas para Ordenação de Classes



E agora? Como estabelecer a ordem entre B, C, D e F ?

É preciso reduzir a influência das classes testadas até o momento (A, E e H).

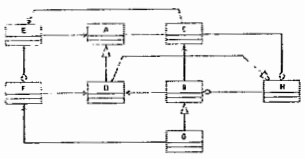
Atenção! É preciso reduzir simultaneamente a influência das três classes que tinham o mesmo valor de FIT.

Classes	FI	FIT
A	2	3
B	1	7
C	3	5
D	2	5
E	2	3
F	1	4
H	2	3

PESC
www.cos.ufrj.br/~ese

57

Heurísticas para Ordenação de Classes

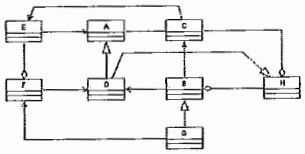


Classes	FI	FIT 1	FIT 2
A	3	3	
B	1	7	5
C	3	5	0
D	2	5	0
E	2	3	
F	1	4	2
H	2	3	

Recalculando os valores de FIT **das classes que ainda não foram ordenadas**, subtraindo o valor do FI das classes já integradas do último valor de FIT (A, E e H).

Heurísticas para Ordenação de Classes

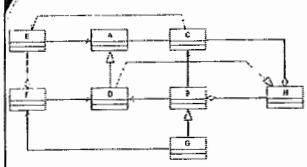
Quais as próximas classes a serem testadas?



Classes	FI	FIT 1	FIT 2
A	3	3	-
B	1	7	5
C	3	5	0
D	2	5	0
E	2	3	-
F	1	4	2
H	2	3	-

Seriam C e D, em qualquer ordem, pois possuem FIT = 0.

Heurísticas para Ordenação de Classes

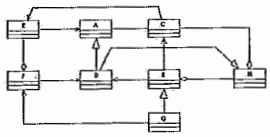


Classes	FI	FIT 1	FIT 2	FIT 3
A	3	3		
B	1	7	5	0
C	3	5	0	
D	2	5	0	
E	2	3		
F	1	4	2	0
H	2	3		

Novamente recalculamos os valores de FIT, subtraindo os fatores de influência das classes ordenadas na interação anterior (C e D).

Heurísticas para Ordenação de Classes

Quais as próximas classes?

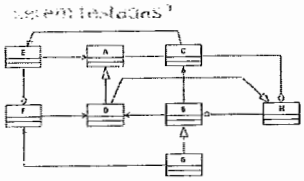


Classes	FI	FIT 1	FIT 2	FIT 3
A	3	3	-	-
B	1	7	5	0
C	3	5	0	-
D	2	5	0	-
E	2	3	-	-
F	1	4	2	0
H	2	3	-	-

Seriam B e F, em qualquer ordem, pois possuem FIT = 0.

Heurísticas para Ordenação de Classes

Quais as próximas classes a serem testadas?



Classes	FI	FIT 1	FIT 2
A	3	3	-
B	1	7	5
C	3	5	0
D	2	5	0
E	2	3	-
F	1	4	2
H	2	3	-

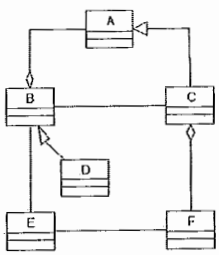
Seriam C e D, em qualquer ordem, pois possuem FIT = 0.

Heurísticas para Ordenação de Classes

Novamente vamos exercitar seu conhecimento sobre as heurísticas!

Estabeleça uma ordem de integração para as classes do modelo no. 2, usando os conhecimentos adquiridos.

Revisão rápida:



Heurísticas para Ordenação de Classes

conceitos:

PESC
www.cos.ufrj.br/~pesc

64

Heurísticas para Ordenação de Classes

Agradecemos sua participação neste treinamento. Outras informações sobre estas heurísticas e seu processo de aplicação poderão ser obtidas em

Qualquer sugestão poderá ser enviada para a caixa postal:

PESC
www.cos.ufrj.br/~pesc

65

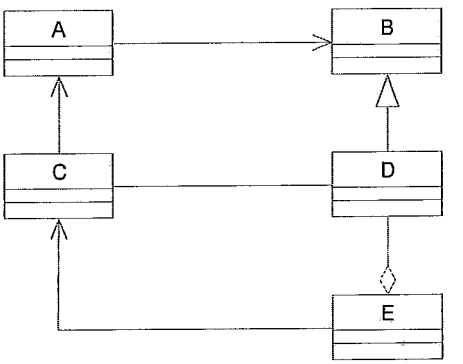
C.2 – Formulário de Respostas

"Testes de Integração em Software Orientado a Objetos"

Formulário de Resposta

Identificador: _____ Tempo: _____ (min) Data: ___ / ___ / _____

Exercício 1



Ordem: _____

Classes	FI	FIT	FIT	FIT	FIT	FIT
A						
B						
C						
D						
E						

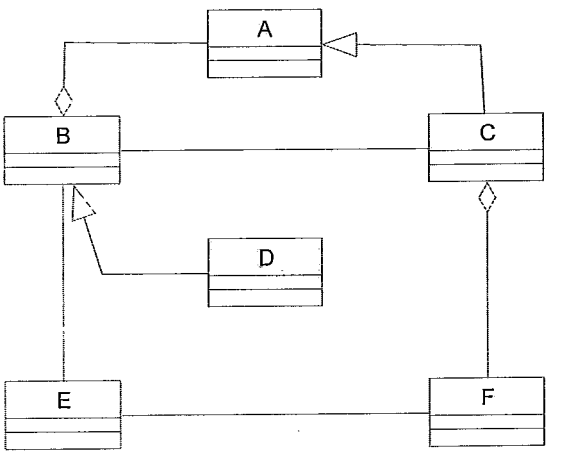
Equipe de Engenharia de Software Experimental
www.cos.ufrj.br/~ese

"Testes de Integração em Software Orientado a Objetos"

Formulário de Resposta

Identificador: _____ Tempo: _____ (min) Data: ___ / ___ / _____

Exercício 2



Ordem: _____

Classes	FI	FIT	FIT	FIT	FIT	FIT
A						
B						
C						
D						
E						
F						

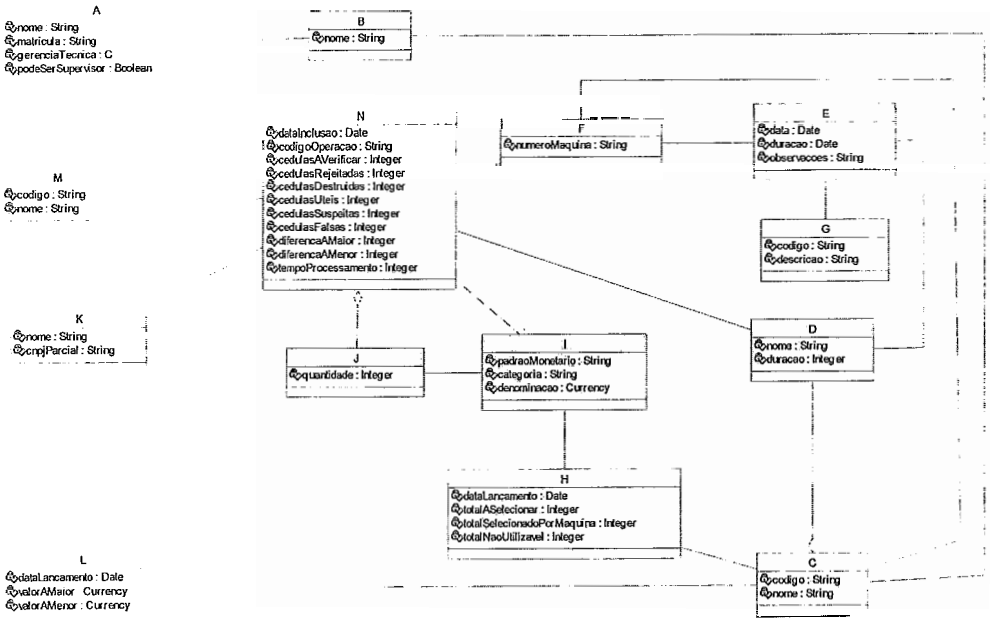
Equipe de Engenharia de Software Experimental
www.cos.ufrj.br/~ese

"Testes de Integração em Software Orientado a Objetos"

Formulário de Resposta

Identificador: _____ Tempo: _____ (min) Data: ___ / ___ / _____

Exercício 3



Ordem: _____

Classes	FI	FIT	FIT	FIT	FIT	FIT	FIT
A							
B							
C							
D							
E							
F							
G							
H							
I							
J							
K							
L							
M							
N							

Equipe de Engenharia de Software Experimental
www.cos.ufrj.br/~ese

C.3 – Questionário de Avaliação

"Testes de Integração em Software Orientado a Objetos"

Questionário de Avaliação

O propósito único deste questionário é apoiar o pesquisador no levantamento de dados para melhor entendimento do estudo experimental em questão. Não será utilizado para avaliar o participante.

Identificador: _____

1. Qual sua opinião sobre o autotreinamento em relação às heurísticas e processo de aplicação?

2. Existiu alguma informação que você não compreendeu devido à forma utilizada para apresentá-la?

3. Você considera que o tempo empregado no autotreinamento foi suficiente?

4. Quais críticas e /ou sugestões você teria sobre as heurísticas?

5. Você utilizaria estas heurísticas em projetos futuros? Por favor, justifique.

6. Poderia sugerir possíveis alterações que melhorariam o procedimento adotado?

Equipe de Engenharia de Software Experimental
www.cos.ufrj.br/~ese