Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes

Gladys Machado Pereira Santos Lima
gladysmpsl@yahoo.com.br
Centro de Análises de Sistemas Navais
Marinha do Brasil

Guilherme Horta Travassos
ght@cos.ufrj.br
Programa de Engenharia e Sistemas de Computação
COPPE / UFRJ

Resumo

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* [7] 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. Este trabalho apresenta um conjunto de heurísticas aplicadas aos diagramas de classes UML que permitem estabelecer uma ordem de prioridade para o teste de integração das classes que compõem o software, utilizando o número de *stubs* necessários para o teste como medida do esforço requerido. Quando comparada às abordagens existentes, estas heurísticas se aplicam em nível mais alto de abstração (projeto), facilitando sua utilização e permitindo antecipar a tomada de decisão no planejamento do teste de integração. Um estudo de caso demonstra sua aplicação e realiza uma comparação dos resultados com estudos realizados encontrados na literatura.

Palavras-chave: Teste de Integração, Ordem de Integração, Engenharia de Software Orientada a Objetos, Experimentação.

Abstract

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 [7], 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 classes. This paper describes a set of heuristics applied to UML class diagrams that allows classes' integration priority order identification. The number of stubs needed for testing is used as an effort measure. Such heuristics, when compared with the other existent approaches, can be applied in higher abstraction level (design) making possible decisions anticipation regarding the integration testing planning. A case study demonstrates its use and shows the results comparison with other related studies found in technical literature.

Key-Words: Integration Test, Integration Order, Object-Oriented Software Engineering, Experimentation.

1. Introdução

O planejamento cuidadoso ajuda a projetar e organizar testes de um sistema de forma a garantir sua adequação. Muitos aspectos estão envolvidos no teste e na integração de componentes para construir um sistema. Podemos dizer que conhecidos os objetivos dos testes, devemos decidir como integrar os componentes de um sistema. Entretanto, 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 (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, no qual esse trabalho encontra-se inserido. Segundo Vieira [7], os principais fatores que diferenciam testes em software OO de testes em software convencionais são: encapsulamento; classes abstratas; herança; e polimorfismo. Estes fatores adicionam complexidade que não existe quando se desenvolve software utilizando o paradigma convencional.

Graham resume as diferenças entre os testes orientados a objetos e os testes tradicionais, de duas maneiras [14]. Por exemplo, os objetos tendem a ser pequenos, e a complexidade, que pode geralmente residir no componente, é freqüentemente transferida para a interface entre os componentes. Essa diferença quer dizer que o teste de unidade é menos difícil, ao passo que o teste de integração deve ser muito mais extensivo. O encapsulamento é considerado um atributo positivo do projeto OO, mas também requer um teste de integração mais extensivo. De maneira semelhante, a herança introduz a necessidade de mais testes.

O teste de integração constitui-se em uma atividade de se descobrir erros associados às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto, conforme Maldonado [16]. Segundo Pfleeger, é o processo de verificar se os componentes do sistema, juntos, trabalham conforme descrito nas especificações do sistema e do projeto do programa [14]. Portanto, uma estratégia de integração deve responder a três questões: quais componentes são focados pelos testes de integração, em que seqüência as interfaces de componentes deverão ser exercitadas e qual técnica de teste será empregada para exercitar a interface?

Devido aos problemas associados com a integração *big-bang,* as classes precisam ser integradas uma de cada vez ou, em alguns casos, em pequenos *clusters* [7]. Dentro deste contexto, o problema pode ser descrito em como identificar uma ordem de prioridade das classes para testes de integração, ou seja, estabelecer critérios de precedência entre as classes para verificar o funcionamento conjunto das mesmas.

Alguns pesquisadores apresentam estratégias baseadas em grafos que representam as interações entre as classes do sistema, obtidos a partir da codificação destas. Entretanto, estas soluções não focam esforços para facilitar decisões de projeto em nível de abstração mais alto, apesar de buscarem um esforço de teste reduzido, em função do número de *stubs* necessários para a integração conforme a ordem estabelecida. Segundo Pfleeger, a ordem em que os componentes são testados afeta a escolha dos casos de teste e das ferramentas [14]. A partir desta motivação, o objetivo deste estudo é identificar heurísticas que possam ser aplicadas aos diagramas de classes (UML – *Unified Modeling Language*), a fim de estabelecer uma ordem de prioridade das classes para teste de integração.

2. Teste de Integração: *Stubs* e Esforço de Teste

Independentemente da estratégia de teste de integração escolhida (*bottom-up* ou *top-dow* ou, as duas combinadas), os componentes são integrados com outros que, por definição, já se encontram desenvolvidos e com seus testes de unidades executados. Ou seja, o código do componente foi examinado, revisado e defeitos, porventura existentes nos dados ou na sintaxe, já foram retirados. Além disto, cada componente é integrado para teste somente uma vez e, em nenhum momento, um componente pode ser modificado para simplificar o teste, segundo Pfleeger [14].

Entretanto, 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). Assim, para dar prosseguimento aos testes quando existirem componentes necessários à integração ainda não codificados e testados é preciso que sejam criados os *stubs* de testes. Os *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 as classes dependentes deles.

Um s*tub* é a implementação parcial de um componente [3]. Um componente usado como fachada para simular o comportamento de um componente real [2]. O teste de um componente A, que chama um componente B, mas B ainda não foi testado, implica na substituição de B por um componente chamado *stub*. Um *stub específico* é escrito para simular o comportamento de B em relação ao componente A. *Stubs* são confiáveis, apesar de se tornarem obsoletos. Entretanto, alguns componentes precisam ter seus *stubs* implementados (*stubbed component* ou *stubbed class*). Um *stub realístico* é escrito para simular o comportamento do componente B em qualquer caminho de teste [10].

A estratégia de testes explica por que e como os componentes são combinados para testar o sistema [14]. A ordem de integração dos componentes (classes) pode afetar o custo e a eficácia do teste. O custo da integração pode ser majorado caso o número de *stubs* necessários para sua conclusão também aumente. Dentro deste contexto, o esforço de teste pode ser medido pelo número de *stubs* que precisam ser escritos conforme a estratégia de integração. O número de *stubs* é calculado dependendo do tipo que foi escrito. Se um *stub realístico* é usado, o esforço é 1. Se dois *stubs específicos* de um mesmo componente são escritos para testar outros dois componentes, então, o esforço é contabilizado em 2 [10].

O *stub* representa despesa indireta. É software que precisa ser escrito, mas que não é entregue com o produto final do software. Se os *stubs* são mantidos bem simples, as despesas indiretas reais são relativamente baixas. [15].

2.1. Análise de Dependência

Segundo Binder, os componentes, tipicamente, dependem uns dos outros de diversas maneiras [3]. 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 1, 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*.

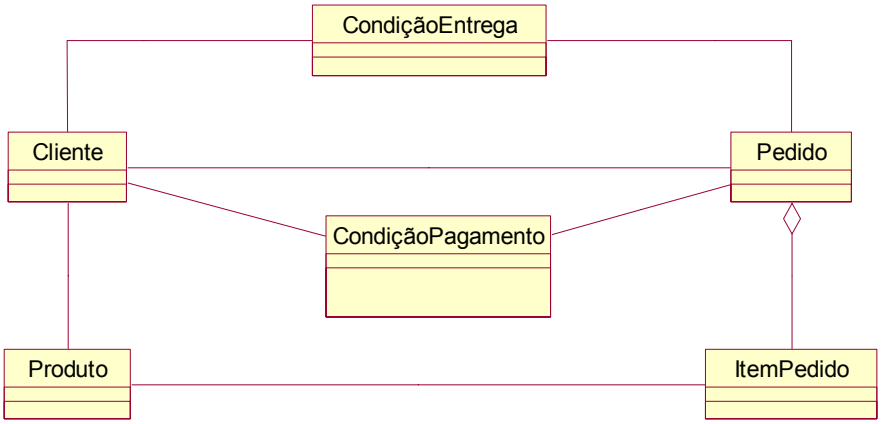

Figura 1 – Diagrama de Classe: Sistema de Controle de Vendas

De maneira semelhante, as dependências ocorrem entre componentes no escopo de um sistema. Algumas abordagens usam a análise de dependências para apoiar o teste de integração *bottom-up*. As dependências explícitas entre componentes correspondem às interfaces que necessitam serem exercitadas pelos testes de integração adequados [3].

3. Estratégias Existentes Baseadas em Grafos

Alguns pesquisadores apresentaram trabalhos relacionados com a busca de uma solução (estratégias ou algoritmos) para determinar a ordem de integração das classes e com o objetivo de minimizar o número de *stubs* a serem produzidos durante os testes de integração das classes, com a finalidade de reduzir o esforço de teste. Estas abordagens empregam a análise das dependências para quebrar alguns ciclos de dependências e, então, criar os respectivos *stubs*. Segundo Briand [8], Kung *et al*, por exemplo, identificam componentes fortemente conectados (SCC- *strongly connected components*) e removem as associações entre classes até não existir mais ciclos, mas não apresentando solução para os casos de mais de uma associação candidata. Outros pesquisadores, como Tai e Daniels apresentaram uma solução para tratamento dos ciclos que, na situação particular onde não existem associações no ciclo, os resultados não eram ótimos em termos do número de *stubs* gerados.

Le Traon *et al* propuseram uma estratégia alternativa baseada em algoritmos de busca em profundidade para identificação de componentes fortemente acoplados em grafos de dependências [10]. Entretanto, este algoritmo depende de algumas decisões arbitrárias para a pesquisa, o que poderia levar a resultados significativamente diferentes. Além disto, ao quebrar os ciclos de dependências, esta abordagem remove relacionamentos de herança ou agregação, o que pode levar a *stubs* necessários mais complexos.

Briand *et al,* assim como Le Traon e seus colegas, identificam os SCCs recursivamente usando o algoritmo de busca baseado em chamadas recursivas para o algoritmo de Tarjan. Pode-se dizer que Briand *et al* aliou esta estratégia com uma solução modificada para o conceito de peso de Tai e Daniels, pois em cada etapa, isto é, dentro de cada SCC não trivial, calcula-se o peso de cada dependência da associação, como critério para quebrar a dependência da associação, escolhendo aquela de maior peso [8]. Briand *et al* realizaram cinco estudos de caso, para analisar o uso das três técnicas (Tai e Daniels; Le Traon *et al*; e Briand *et al*) de ordenação baseadas em grafos (ORD) para execução de testes de integração, no contexto de ambientes de desenvolvimento orientado a objetos utilizando

modelos representados em ORD, obtidos por engenharia reversa de cinco sistemas desenvolvidas em Java [8].

Outra abordagem, não baseada em grafos, mas em algoritmos genéticos, também propõe uma solução para o problema da ordenação de classes para testes de integração, por meio de uma técnica de otimização global baseada em heurísticas [7]. Entretanto, neste caso, o propósito é minimizar a complexidade dos *stubs* necessários à integração.

4. Heurísticas Alternativas

As heurísticas apresentadas por Briand *et al* foram realizadas com base em diagramas de relacionamento entre objetos, obtidos a partir da própria implementação das classes em estudo [8]. Diferentemente das propostas anteriores, as heurísticas apresentadas neste artigo visam o estudo das dependências entre classes em um nível de abstração mais elevado, empregando como modelo o diagrama de classes de um projeto, descrito pela UML (*Unified Modeling Language*), como base de entrada para todas as informações necessárias ao seu emprego.

A partir da aplicação de um conjunto de heurísticas que determinam critérios de precedência entre classes, poderá ser estabelecida uma lista ordenada das classes para execução de testes de integração, anteriormente ao detalhamento de projeto e implementação do modelo em estudo.

4.1. Versão inicial: Travassos e Oliveira

As heurísticas apresentadas a seguir foram retiradas de Oliveira [13] e são fruto de observação e aplicação desta abordagem na academia e na indústria, discutidas no curso de Engenharia de Software Orientada a Objetos da COPPE / UFRJ por Travassos.

4.1.1. Critérios de Precedência

Os critérios de precedência foram definidos com o propósito de verificar, com base na semântica estabelecida pela UML, quais são as características determinantes para que certas classes sejam testadas antes de outras, de modo a realizar satisfatoriamente os testes de integração, minimizando o número de *stubs* específicos a serem gerados. Os critérios de precedência, descritos a seguir, são: herança; assinatura dos métodos de uma classe; agregação; navegabilidade; classes de associação; e dependência. As definições para estes conceitos podem ser encontradas em Booch [4].

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, primeiramente, que a superclasse tenha sido devidamente testada. 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.

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, primeiramente, a classe servidora e depois a classe cliente.

Agregação – 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*.

Navegabilidade – A navegabilidade indica que uma classe torna-se atributo de outra. Segundo Booch, a menos que seja declarado explicitamente o contrário, uma associação implica navegação bidirecional [4]. Sendo assim, será utilizada a navegabilidade para definir critério de precedência quando a ligação entre as duas classes ocorrer através da associação. Nos casos em que houver navegabilidade bidirecional, serão analisadas as possibilidades de propiciar a escolha da classe a ser ordenada primeiramente pelo desenvolvedor, configurando um processo semi-automático.

Classes de Associação – Uma classe de associação surge a partir da necessidade de colaboração entre duas outras classes. As classes que deram origem à classe de associação terão precedência de teste de integração sobre a classe derivada.

Dependência – 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 cliente terá precedência para ser testada.

4.1.2. Fator de Integração e Fator de Integração Tardia

Para viabilizar a aplicação dos critérios de precedência e estabelecer a ordem de prioridade, foram definidas duas propriedades: *Fator de Influência* (FI) e *Fator de Integração Tardia* (FIT).

O *fator de influência* (FI) de uma classe é um valor que quantifica a relação de precedência entre as classes, sendo, portanto, diretamente proporcional ao número de classes que precisam ser integradas posteriormente à classe em questão. Deve ser definido considerando os relacionamentos diretos da classe em questão. Quanto maior o número de classes que possuam relação de precedência com a classe sob análise, maior será seu *fator de influência*.

O *fator de integração tardia* (FIT) de uma classe expressa a relação que é estabelecida entre as classes após a definição do fator de influência e é obtido a partir da soma dos fatores de influência de todas as classes que têm precedência direta sobre a classe em questão. Quanto maior o *fator de integração tardia* de uma classe, mais tarde deve ser realizado o teste de integração para a classe em questão.

As propriedades fator de influência e fator de integração tardia são utilizadas para possibilitar a implementação da ordenação das classes por ordem de prioridade para teste de integração.

Exemplificando.

A partir da combinação dos critérios de precedência, por meio dos fatores *de influência* (FI) e *de integração tardia* (FIT), que serão calculados para todas as classes existentes do modelo, será possível estabelecer uma lista ordenada das classes para execução de testes de integração.

Em análise à Figura 2, observa-se que a classe *Escola* não tem precedência sobre nenhuma outra classe do modelo, sendo seu *fator de influência* igual a zero. *Departamento* tem precedência em relação à classe *Escola* (parte-todo), o que significa que seu *fator de influência* para a classe *Departamento* é igual a um, pois influencia apenas uma classe do modelo. Da mesma forma que a classe *Departamento*, a classe *Aluno* possui *fator de influência* igual a um, uma vez que influencia apenas uma classe do modelo, a classe *Escola*.

Por outro lado, a classe *Curso* tem precedência sobre as classes *Departamento* e *Aluno*, seu *fator de influência*, então, é igual a dois.

A ordem de preenchimento da lista de classes surgirá, inicialmente, a partir da seleção daquelas classes que apresentarem *fatores de integração tardia* iguais a zero (Tabela 1), como calculado para a classe *Curso* do exemplo da Figura 2.

Tabela 1. Valores para FI e FIT iniciais

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

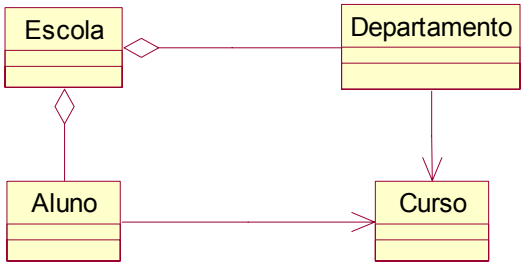

Figura 2. Modelo de Classes Inicial

Para selecionar as próximas classes, deverão ser recalculados os *fatores de integração tardia* das demais classes desprezando o *fator de influência* das classes anteriormente selecionadas (Figura 2) e, então, incluir na lista aquelas que apresentarem novo fator igual a zero (Tabela 2). Neste exemplo, o resultado da lista ordenada para teste de integração seria {*Curso*; *Departamento* ou *Aluno*; e *Escola*}.

Tabela 2. Novos Valores para FI e FIT

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

4.2. Evoluindo as Heurísticas

A aplicação das heurísticas em diagramas que não continham classes fortemente acopladas mostrou-se eficiente. Entretanto, para diagramas contendo mais de uma classe com mesmo *fator de integração tardia*, representando ciclos de dependências entre classes, as heurísticas não resultaram um esforço de teste satisfatório, demandando um tratamento especial para situações de *deadlock* [11]. Outras situações de melhoria das heurísticas foram observadas, em cursos, durante estudos de casos, bem como na observação direta de alguns aspectos, como a análise do *fator de influência* nulos, levando à necessidade de complementação das heurísticas, inicialmente sugeridas por Travassos e Oliveira [13], as quais apresentamos a seguir.

4.2.1. Novo critério de precedência: a Cardinalidade

A cardinalidade, quando se analisa a navegabilidade bidirecional, também pode expressar um critério de precedência. Um dos aspectos chaves 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, opcional, um-para-muitos, muitos-para-muitos ou outras variações desta possibilidade, sendo especificada para cada extremidade da associação. Será utilizada a cardinalidade para definir 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.

4.2.2. Análise do *Fator de Influência* Nulo

O resultado de valor nulo para o cálculo do fator de influência (FI) de alguma classe de um modelo é bastante significativo no processo de ordenação das classes. 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. Conceitualmente, a classe com FI nulo representa a generalização de outras classes do modelo.

4.2.3. Inexistência de *Fatores de Integração Tardia* Nulos

Conforme exposto por Oliveira [13], busca-se, primeiramente, para execução dos testes de integração, as classes que se encontram com *fator de integração tardia* nulos. Entretanto, alguns modelos podem ser representados somente por classes com forte acoplamento, não existindo inicialmente classes com *fator de integração tardia* nulos. Nestes casos, a seqüência ao teste de integração deve ser feita por meio das classes que possuam o menor FIT calculado.

4.2.4. Tratamento de *Deadlock*

Existem situações em que mais de uma classe do modelo apresenta o mesmo valor de FIT, significando que de alguma forma estas classes possuem uma dependência, existindo um ciclo. 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.

O tratamento dos ciclos será realizado por meio da integração de todas as classes que apresentarem o mesmo valor de *fator de integração tardia* (FIT), com a conseqüente geração de *stubs* específicos necessários, antes de subtrair o valor da influência destas classes dos valores do FIT das demais classes ainda não integradas, ou seja, antes de outra iteração para o cálculo do FIT.

Para estabelecer prioridade entre as classes com mesmo valor de FIT, deve-se respeitar os seguintes critérios:

- A classe selecionada deve gerar o menor número de *stubs* específicos necessários em comparação com o número de *stubs* gerados pelas outras classes com mesmo valor de FIT.

- No caso das classes necessitarem da implementação do mesmo número de *stubs*, deverá ser testada aquela que os *stubs* apresentarem a menor complexidade (medida pelo tamanho da classe, ou seja, pelo somatório do número de atributos e do número de métodos de cada *stubs,* [12].
- No caso de alguma dessas classes a serem testadas contribuir para diminuir o número de *stubs* necessários para testar as outras de mesmo FIT, indicando uma dependência interna, esta deverá ser testada primeiramente.
- Caso apresente alguma associação com navegabilidade obrigatória, a classe deverá ser testada primeira, preferencialmente.

4.3. Estudo de Caso: o modelo de Briand

Para ilustrar as técnicas estudadas e compará-las com sua própria proposta, Briand [8] utilizou como exemplo o diagrama de relacionamento entre objetos (ORD) da Figura 3, onde: heranças são identificadas por ***I***; agregações por ***Ag***; associações por ***As;*** e as classes por letras maiúsculas.

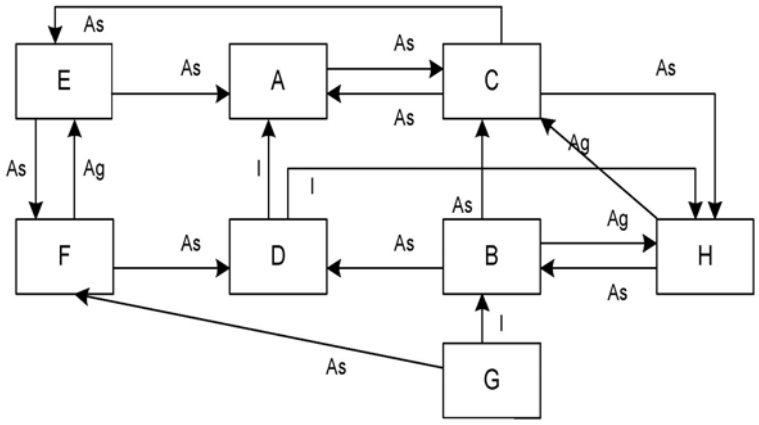

Figura 3. Exemplo extraído de Briand [8].

Segundo Briand [7] um critério de avaliação para comparar ordens é o esforço de construção de *stubs* (esforço de teste) requeridos para integrar classes conforme a ordem específica. Se uma ordem de integração de classes necessita de outras classes que ainda não foram integradas, faz-se necessário o desenvolvimento de *stubs* para continuidade dos testes. A tabela 3 resume os resultados obtidos com a aplicação das estratégias relatadas por Briand [8] sobre o modelo da Figura 3. Podemos observar que Le Traon *et al* apresentam melhores resultados do que Tai e Daniels. Como o resultado de Briand *et al* é determinístico, seu esforço de teste pode ser considerado melhor que o apresentado por Le Traon *et al*, pois este último somente apresenta resultado igual ao de Briand quando o vértice escolhido inicialmente for G.

Tabela 3. Resultados do Estudo de Briand [8]

Proposta	Seqüência de Teste	*Stubs* Específicos
Tai e Daniels	{A, E, C, F, H, D, B, G}	Stub (C, A) Stub (F, E) Stub (H, C) Stub (D, F) e Stub (B, H)
Le Traon *et al* (começando com o nó G)	{A, H, D, E, F, C, B, G}	Stub (C,A) Stub (B, H) e Stub(C, H) Stub (F,E)
Briand *et al*	{A, E, C, H, D, F, B, G}	Stub (C,A) Stub (F,E) Stub (H, C) Stub (B, H)

Tendo por base os conceitos utilizados em [6] para efetuar a relação entre um diagrama de classes e seu correspondente ORD, foi gerado o diagrama de classes da Figura 4, que será usado para demonstrar uma aplicação das heurísticas alternativas proposta neste trabalho, considerando que todas as classes são concretas.

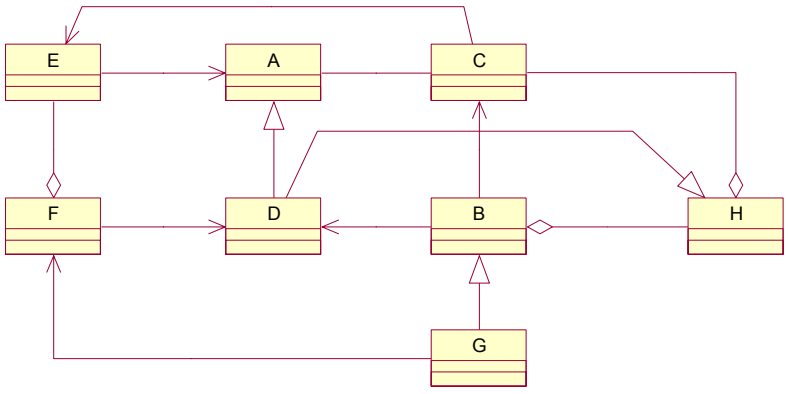

Figura 4. Diagrama de Classes correspondente a Figura 3.

Tabela 4. Valores de FI

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

Com base nos critérios de precedência (seção 4.1.1.), veremos que a classe A tem precedência sobre as classes: E (pelo critério da *navegabilidade*); D (pelo critério de *herança*); e C (pelo critério de *navegabilidade*). Ou seja, existem três classes que devem ser integradas após a classe A em questão, portanto, o valor três deve ser atribuído ao *fator de influência* (FI) da classe A. O valor dois deve ser atribuído ao FI da classe E, pois a classe C (pelo critério de *navegabilidade*) e a classe F (pelo critério da *agregação*) devem ser integradas após a classe E.

Ao calcular o valor do *fator de influência* para a lista de classes não ordenadas, composta por LCNO={A, B, C, D, E, F, G, H,}, obteremos os valores de *fatores de influência* apresentados na Tabela 4. A classe G, por possuir *fator de influência* nulo, será retirada da LCNO por ser totalmente dependente dos testes de integração das classes B e H, que por sua vez são dependentes das demais classes, sendo, então, incluída posteriormente ao final da lista ordenada a ser gerada.

Tabela 5. Valores de FI e FIT nas diferentes interações

Classes	FI	FIT 1ª. Interação	FIT 2ª. interação	FIT 3ª. interação
A	3	**3**		
B	1	7	2	**0**
C	3	5	**0**	
D	2	5	**0**	
E	2	**3**		
F	1	4	2	**0**
H	2	**3**		

O *fator de integração tardia* para nova lista LCNO={A, B, C, D, E, F, H} inicia-se com os valores mostrados pela Tabela 5. Observando a inexistência de *fatores de integração tardia* nulos, as classes A, E e H serão priorizadas conforme a necessidade da dependência interna entre E e A, pois ambas necessitam de 1 *stub* específico. A lista de classes ordenadas para teste de integração será, neste momento, LCOTI={A, E, H}.

Devemos reduzir a influência das classes A, E e H, e recalcular o FIT, numa segunda interação, para a LCNO={B, C, D, F}, composta pelas demais classes a ordenar. Neste momento, selecionamos as classes C e D, por serem folhas (FIT=0), que atualizam a LCOTI. Como os valores de FIT para as classes F e B na terceira interação são nulos, a LCOTI={A, E, H, D, C, F, B, G} é finalmente estabelecida com a inclusão da classe G ao final mesma.

A utilização da seqüência de ordenação das classes para os testes de integração encontrada seguindo as heurísticas apresentadas neste trabalho, implicará na necessidade de implementação de dois *stubs* específicos: um *stub* de C para testar A e um outro *stub* de C para testar H, determinando um esforço de teste igual a dois, ou ainda, na implementação de um *stub* realístico de C para ambas as classes A e C. Comparando este resultado com o valor de quatro *stubs* para a ordem estabelecida por Briand *et al* [8], podemos observar que houve uma redução substancial do esforço de teste. Alie-se a isto, a facilidade de entendimento das

abstrações representadas num diagrama de classe UML comparativamente ao manuseio de um ORD.

5. Conclusões e Trabalho Futuro

A garantia de resultados corretos em modelos com grande número de classes envolvidas e com diversos ciclos de dependências, a serem tratados durante a aplicação das heurísticas na determinação da ordem de integração das diversas classes, muitas vezes é obtida por meio de uma aplicação criteriosa e organizada. Esta observação mostra a necessidade da formalização do processo de aplicação das heurísticas.

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, identificamos a necessidade de elaborar estudo experimental visando à caracterização destas heurísticas.

Devido ao processo manual de cálculo dos fatores de influência (FI) e integração tardia (FIT) para diagramas com grande número de classes e relacionamentos ser dispendioso, deve-se preparar, anteriormente à aplicação do estudo experimental de caracterização das heurísticas, uma ferramenta para automatizar o processo de ordenação das classes para aplicação dos testes de integração. A possibilidade de automatizar, também, a geração da lista de *stubs* específicos necessários ao teste será estudada.

A melhoria nos testes pode reduzir os custos de desenvolvimento de software ou ainda melhorar o desempenho. Desta maneira, podemos pensar na ordem de integração das classes como guia na determinação da ordem de implementação das classes, o que poderá ajudar na redução do tempo requerido para o desenvolvimento e teste de sistemas.

Agradecimentos

Ao Hamilton Oliveira pelo trabalho inicial com as heurísticas para integração de classes. À Marinha do Brasil pela oportunidade de participação desta Oficial no Curso de Mestrado em Engenharia de Sistemas e Computação do Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia - COPPE / UFRJ.

Este trabalho está sendo realizado no contexto do projeto CNPq – 475407/2003-2.

Referências Bibliográficas

[1] ANTONIOL, G.; BRIAND, L.C.; DI PENTA, M.; LABICHE, Y.; **A case study using the round-trip strategy for state-based class testing**, Proceedings of the 13th International Symposium on Software Reliability Engineering, 12-15 Nov. 2002, Page(s): 269 –279

[2] BEIZER, B.; **Software System Testing and Quality Assurance**; Van Nostrand Reinhold Company Inc, 1984.

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

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

[5] BRIAND, L.C.; LABICHE, Y.; YIHONG, W; **Revisiting strategies for ordering class integration testing in the presence of dependency cycles**, ISSRE 2001. Proceedings of

the. 12th International Symposium on Software Reliability Engineering, Nov. 2001, Page(s): 287 -296

[6] BRIAND, L.C.; LABICHE, Y.; SOCCAR, G.; **Automating impact analysis and regression test selection based on UML designs**, Software Maintenance, 2002. Proceedings. International Conference on, 3-6 Oct, 2002.

[7] BRIAND, L.C.; FENG, J. ; LABICHE, Y.; **Experimenting with Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders**, Carleton University, Technical Report SCE-02-03, Version 3, Oct, 2002.

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

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

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

[11] LIMA, G.M.P.S.; TRAVASSOS, G.H.; **Testes de Integração Aplicados a Software Orientado a Objetos: Heurísticas para Ordenação de Classes**, Relatórios Técnicos do Programa de Engenharia de Sistemas e Computação, ES-632/04, COPPE, UFRJ, 2004.

[12] LORENZ, M., KIDD, J.; **Object-Oriented Metrics: A Pratical Guide**, Prentice Hall, USA, 1994.

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

[14] PFLEEGER, S. L., **Engenharia de Software: Teoria e Prática***,* Prentice Hall, 2ª. Edição, 2004.

[15] PRESSMAN, R. S., **Engenharia de Software***,* Mc Graw Hill, 5ª. Edição, 2001.

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

[17] TRAVASSOS, G.H.; GUROV, D.; AMARAL, E.A.G.G., **Introdução à Engenharia de Software Experimental**, Relatório Técnico ES-590/02-Abril, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ.

[18] VIEIRA, M.E.R.; **Abordagem para Apoio ao Teste Baseado no Comportamento de Sistemas Orientados a Objetos,** Tese de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE, UFRJ, Rio de Janeiro, 1998.